ABSTRACTING GRADUAL TYPING: METATHEORY AND APPLICATIONS

TESIS PARA OPTAR AL GRADO DE
DOCTOR EN CIENCIAS, MENCION COMPUTACIÓN

MATÍAS TORO IPINZA

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
FEDERICO OLMEDO BERON
ISMAEL FIGUEROA PALET
JEREMY SIEK

Este trabajo ha sido financiado por CONICYT-PCHA/Doctorado Nacional/2015-21150510

SANTIAGO DE CHILE
2019
Resumen

Han habido muchos enfoques para integrar tipado estático y dinámico. Uno de los enfoques más notables es el del tipado gradual. El enfoque clásico para diseñar lenguajes graduales es usualmente ad-hoc, pero existen metodologías que sistematizan este proceso. Una de ellas es la Abstracting Gradual Typing (AGT), que ayuda a construir sistemáticamente lenguajes graduales a partir de lenguajes estáticamente tipados usando interpretación abstracta al nivel de tipos. A pesar que se a mostrado que AGT a sido efectiva en diferentes contextos, hay aún muchas preguntas abiertas: ¿AGT escala a mecanismos de lenguaje y disciplinas de tipos complejos? ¿Qué lenguajes obtienen al usar abstracciones más ricas, o al introducir imprecisión de una manera poco convencional? ¿Cómo se compara el lenguaje gradual resultante con lo existente en la literatura? ¿Qué propiedades AGT garantiza de preservar por construcción? ¿Podemos aplicar AGT a un lenguaje gradual derivado con AGT?

En esta tesis se trata de responder a estas preguntas, aplicando AGT a disciplinas de tipos y mecanismos de lenguaje complejos. Primero, se aplica AGT a un cálculo lambda con tipado simple y referencias mutables, donde se muestra que una directa aplicación de AGT no garantiza una semántica eficiente respecto al espacio. Se prueba equivalencia contextual con uno de los lenguajes graduales con referencias encontrados en la literatura.

Segundo, se aplica AGT a un lenguaje con tipado de seguridad y referencias, introduciendo inexactitud solo en las etiquetas de seguridad de los tipos. Se aprende que una aplicación directa de AGT sólo garantiza conservar por construcción la seguridad de tipos y los criterios refinados de lenguajes graduales. En orden de satisfacer no-interferencia, la propiedad semántica crucial del lenguaje estático, se deben refinar las abstracciones usadas en la semántica dinámica. Pero debido a las referencias mutables, se agrega un chequeo extra en la regla de reducción de asignaciones para prevenir flujos implícitos de información a través de la memoria. Este chequeo extra rompe la garantía gradual dinámica, la cual es parte de los criterios refinados de los lenguajes graduales.

Tercero, se aplica AGT para introducir una nueva forma de imprecisión en los tipos, llamada unión gradual, un diseño original de tipos de unión que combina ambos beneficios de uniones etiquetadas y no etiquetadas. Se descubre que las uniones graduales interactúan con el tipo desconocido en una forma que exige un enfoque estratificado para AGT, dependiendo de la composición de dos interpretaciones de abstracción distintas en orden de recuperar optimalidad.

Cuarto, se aplica AGT a System F, un lenguaje que soporta polimorfismo paramétrico. Se descubre que una aplicación directa de AGT rompe parametricidad, una propiedad semántica crucial de System F. En orden de recuperar parametricidad, se refinan las abstracciones (y se personalizan ciertas operaciones) usadas en la semántica dinámica. Esta personalización ayuda a preservar parametricidad pero a costa de violar la garantía gradual dinámica. Esta garantía fue dejada como una conjetura en todos los trabajos previos; aquí se prueba que es simplemente incompatible con la noción clásica de parametricidad. Sin embargo, se establece una propiedad más débil que permite refutar varias afirmaciones acerca de teoremas graduales gratis, clarificando el tipo de razonamiento soportado por la parametricidad gradual.
Abstract

There have been many approaches to integrate static and dynamic typing. One of the most notable approaches is gradual typing. The classical approach to designing gradual languages is usually ad-hoc, but there are some methodologies that systematize the process. One of them is the Abstracting Gradual Typing (AGT) methodology, which systematically constructs gradually-typed languages from statically-typed languages, using abstract interpretation at the type level. Although AGT has been shown to be effective in different contexts, there are still many open questions: does AGT scale to complex language constructs and type disciplines? What kind of gradual languages do we obtain by using richer abstractions, or by introducing imprecision in a unconventional way? How do the resulting gradual language compares with the literature? What properties does AGT guarantee to preserve by construction? Can we apply AGT to a gradual language derived with AGT?

In this thesis we try to answer these questions, by applying AGT to complex type disciplines and language constructs. First, we apply AGT to a simply-typed lambda calculus with mutable references, where we show that the direct application of AGT does not guarantee space efficient semantics. We compare the resulting language with other gradual languages with references in the literature, and we show contextual equivalence with one of them.

Second, we apply AGT to a security-typed language with references by introducing imprecision only in the security labels of types. We learn that a direct application of AGT, only guarantees to preserve by construction type safety and the refined criteria of gradual languages. In order to satisfy noninterference, the crucial semantic property of the static language, we need to refine the abstractions used in the dynamic semantics. But due to mutable reference, we add an extra check in the assignment reduction rule to prevent implicit flows through the heap. This extra check brakes the dynamic gradual guarantee, part of the refined criteria of gradual languages.

Third, we apply AGT to introduce a new form of imprecision on types, named gradual unions, a novel design of union types that combines benefits of both tagged and untagged unions. We uncover that gradual unions interact with the unknown type in a way that mandates a stratified approach to AGT, relying on a composition of two distinct abstract interpretations in order to retain optimality. We also show how to compile such a language to a threesome cast calculus, and prove that the compilation preserves the semantics and properties of the language.

Fourth, we apply AGT to System F, a language with support for parametric polymorphism. We discover that a direct application of AGT breaks parametricity, a rich semantic property of System F. In order to recover parametricity we refine the abstractions (and customize some operations) used in the dynamic semantics. This customization helps preserve parametricity but at the cost of violating the dynamic gradual guarantee. This guarantee was left as conjecture in all prior work; here it is proven to be simply incompatible with the classical notion of parametricity. We nevertheless establish a weaker property that allows us to disprove several claims about gradual free theorems, clarifying the kind of reasoning supported by gradual parametricity.
# Contents

1 Introduction ................................................. 1

2 Background .................................................. 11
   2.1 Gradual Typing ........................................ 11
   2.2 Abstracting Gradual Typing ............................ 15
   2.3 Complex Type Disciplines and Language Constructs of Interest ............................ 16
      2.3.1 References ..................................... 17
      2.3.2 Information-Flow Security Typing ............... 19
      2.3.3 Parametric Polymorphism ........................ 22

3 First step: Gradualizing References ....................... 25
   3.1 Gradual Typing with References ....................... 25
   3.2 Preliminary: The Static Language $\lambda^{\text{REF}}$ ............... 27
   3.3 Gradualizing $\lambda^{\text{REF}}$ ........................ 29
      3.3.1 Syntax and Meaning of Gradual Types ............ 29
      3.3.2 Lifting the Type System ........................ 31
      3.3.3 Static Semantics .............................. 32
      3.3.4 Dynamic Semantics ............................ 33
      3.3.5 Elaboration of $\lambda^{\text{REF}}_e$ terms ........... 38
      3.3.6 Properties .................................. 40
      3.3.7 $\lambda^{\text{REF}}_e$ in Action ..................... 41
   3.4 Comparing $\lambda^{\text{REF}}_e$ and HCC ..................... 42
      3.4.1 The Coercion Calculus ........................... 42
      3.4.2 Relating $\lambda^{\text{REF}}_e$ and HCC$^+$ ............. 47
      3.4.3 Recovering Space Efficiency in $\lambda^{\text{REF}}_e$ .......... 50
   3.5 Encoding Permissive and Monotonic References in $\lambda^{\text{REF}}_e$ ...................... 52
      3.5.1 Static Semantics .............................. 53
      3.5.2 Dynamic Semantics ............................ 53
      3.5.3 Properties .................................. 54
   3.6 Related work ....................................... 55
   3.7 Conclusion ......................................... 56

4 Type-driven Gradual Security Typing ................... 58
   4.1 Introduction ...................................... 59
   4.2 Type-Driven Gradual Security Typing in Action .......... 61
   4.3 Static Security Typing with References ................ 65
4.4  GSL\textsubscript{Ref}: Type-Driven Gradual Security Typing ........................................ 69
    4.4.1  Static Semantics ........................................ 72
    4.4.2  Dynamic Semantics ........................................ 73
    4.4.3  Examples of Reduction ........................................ 83
    4.4.4  GSL\textsubscript{Ref}: Safety and Graduality ................. 84
    4.4.5  Prototype Implementation ................................. 86
4.5  GSL\textsubscript{Ref}: Noninterference ........................................ 86
4.6  Deriving GSL\textsubscript{Ref} with AGT (almost) ......................... 91
    4.6.1  Deriving the Statics ........................................ 92
    4.6.2  Deriving the Dynamics ........................................ 93
    4.6.3  Policing Dynamic Heap Updates ......................... 97
4.7  Related Work .................................................. 99
4.8  Conclusion ..................................................... 105

5  A Gradual Interpretation of Union Types ........................................ 106
    5.1  Introduction .................................................. 106
    5.2  Background and Motivation ..................................... 108
        5.2.1  Tagged Unions ........................................ 108
        5.2.2  Untagged Unions ........................................ 109
        5.2.3  Gradual Unions ........................................ 110
        5.2.4  Comparing Unions ........................................ 111
        5.2.5  Gradual Unions vs. Standard Gradual Types ................. 113
    5.3  GTFL\textsuperscript{⊕}: Static Semantics ......................... 114
        5.3.1  The Static Language: STFL .................................. 114
        5.3.2  Defining Gradual Types Separately ......................... 114
        5.3.3  Combining Gradual Types: Take 1 ......................... 116
        5.3.4  Combining Gradual Types: Take 2 ......................... 117
        5.3.5  Static Semantics of GTFL\textsuperscript{⊕} ..................... 120
    5.4  GTFL\textsuperscript{⊕}: Dynamic Semantics and Properties ................. 122
    5.5  Correctness of the Translational Semantics ..................... 125
    5.6  Related Work .................................................. 129
    5.7  Conclusion ..................................................... 130

6  Gradual Parametricity, Revisited ........................................ 132
    6.1  Introduction .................................................. 132
    6.2  The Need to Revisit Gradual Parametricity ..................... 133
        6.2.1  Static Semantics Issues .................................. 134
        6.2.2  Dynamic Semantics Issues ................................. 135
    6.3  GSF, Informally ............................................. 137
        6.3.1  Design Principles, Goals and Non-Goals .................... 137
        6.3.2  GSF in Action ............................................. 138
    6.4  Preliminary: The Static Language SF .......................... 139
    6.5  GSF: Statics .................................................. 141
        6.5.1  Syntax and Syntactic Meaning of Gradual Types ......... 141
        6.5.2  Lifting the Static Semantics .............................. 143
        6.5.3  Static Properties of GSF .................................. 145
    6.6  GSF: Evidence-Based Dynamics .................................. 145
6.7 Evidence for Gradual Parametricity .................. 148
  6.7.1 Simple Evidence, and Why It Fails ............... 148
  6.7.2 Refining Evidence .................................. 149
  6.7.3 Basic Properties of GSF Evaluation .............. 152
6.8 GSF: Parametricity ................................... 153
6.9 Parametricity vs. Dynamic Gradual Guarantee ....... 155
6.10 Gradual Free Theorems in GSF ....................... 157
6.11 Related Work ........................................ 159
6.12 Conclusion ........................................... 160

7 Conclusions and Future Work .......................... 162

Bibliography .................................................. 165

Appendices ...................................................... 176

Appendix A First step: Gradualizing References ...... 177
  A.1 Gradualizing λREF, Elaborating λREF .............. 177
  A.2 Type Safety ........................................... 180
  A.3 Gradual Guarantee .................................... 184
    A.3.1 Conservative Extensions of the Static Discipline
    A.3.2 Static Gradual Guarantee ....................... 193
    A.3.3 Dynamic Gradual Guarantee .................... 196
  A.4 Relation to the coercion calculus .................. 201
  A.5 Encoding Permissive and Monotonic References in
       λREF ................................................. 215

Appendix B Type-driven Gradual Security Typing ...... 219
  B.1 Additional Definitions ................................. 219
    B.1.1 SSLRef: Static Semantics ........................ 219
    B.1.2 SSLRef: Noninterference Definitions .......... 219
    B.1.3 GSLRef: Static Semantics ....................... 222
    B.1.4 GSLRef: Static Semantics ....................... 223
    B.1.5 GSLRef: Dynamic Semantics ..................... 224
    B.1.6 GSLRef: Translation to GSLεRef ............... 227
  B.2 Static Security Typing with References ............ 229
    B.2.1 SSLRef: Static type safety ..................... 229
    B.2.2 SSLRef: Noninterference ......................... 240
  B.3 Gradualizing the Static Semantics .................. 251
    B.3.1 From Gradual Labels to Gradual Types .......... 251
    B.3.2 Static Criteria for Gradual Typing ............ 253
  B.4 Gradualizing the Dynamic Semantics ................. 259
    B.4.1 Precise Evidence for Consistent Security Judgments
    B.4.2 Initial evidence .................................. 261
    B.4.3 Evolving evidence: Consistent Transitivity ... 262
    B.4.4 Algorithmic definitions ......................... 263
    B.4.5 Proofs ............................................ 266
  B.5 GSLεRef: Dynamic properties ........................ 271
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>$\lambda^{\text{REF}}$: Syntax and Type System</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>$\lambda^{\text{REF}}$: Dynamic Semantics</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>$\lambda^{\text{REF}}$: Syntax and Typing Rules</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>$\lambda^{\text{REF}}$: Syntax and Typing Rules</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>$\lambda^{\text{REF}}$: Dynamic semantics</td>
<td>38</td>
</tr>
<tr>
<td>3.6</td>
<td>Elaboration of $\lambda^{\text{REF}}$ from $\lambda^{\text{REF}}$</td>
<td>39</td>
</tr>
<tr>
<td>3.7</td>
<td>HCC$^+$: Static semantics</td>
<td>43</td>
</tr>
<tr>
<td>3.8</td>
<td>HCC$^+$: Dynamic semantics</td>
<td>45</td>
</tr>
<tr>
<td>3.9</td>
<td>HCC$^+$: Coercion normal forms</td>
<td>46</td>
</tr>
<tr>
<td>3.10</td>
<td>$\lambda^{\text{REF}}$ to HCC$^+$ translation rules</td>
<td>46</td>
</tr>
<tr>
<td>3.11</td>
<td>Coercion insertion function</td>
<td>47</td>
</tr>
<tr>
<td>3.12</td>
<td>Map from evidence augmented consistent judgments to coercions</td>
<td>48</td>
</tr>
<tr>
<td>3.13</td>
<td>Bisimulation relation between intrinsic terms and the coercion calculus</td>
<td>49</td>
</tr>
<tr>
<td>3.14</td>
<td>$\lambda_c^{\text{REF}}$: Space-efficient dynamic semantics</td>
<td>51</td>
</tr>
<tr>
<td>3.15</td>
<td>Changes to the static semantics</td>
<td>53</td>
</tr>
<tr>
<td>3.16</td>
<td>$\lambda_{\text{pm}}^{\text{REF}}$: Dynamic semantics (selected rules)</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>SSL$^{\text{Ref}}$: Syntax and Static Semantics</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>SSL$^{\text{Ref}}$: Label Tracking Dynamic Semantics</td>
<td>68</td>
</tr>
<tr>
<td>4.3</td>
<td>Index of terms, operations and relations used in this chapter, along with their notation, and reference to corresponding Figure (F) or Page (P)</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>GSL$^{\text{Ref}}$: Static Semantics</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>GSL$^{\text{Ref}}$: elaboration to GSL$^{\text{Ref}}$ terms</td>
<td>79</td>
</tr>
<tr>
<td>4.6</td>
<td>GSL$^{\text{Ref}}$: Dynamic semantics</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>GSL$^{\text{Ref}}$: Evaluation frames and reduction</td>
<td>82</td>
</tr>
<tr>
<td>4.8</td>
<td>GSL$^{\text{Ref}}$: Example reduction</td>
<td>85</td>
</tr>
<tr>
<td>4.9</td>
<td>Related values</td>
<td>89</td>
</tr>
<tr>
<td>4.10</td>
<td>Related computations</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>Stratified interpretation of UTYPE</td>
<td>117</td>
</tr>
<tr>
<td>5.2</td>
<td>GTFL$^\uparrow$: Syntax and typing rules</td>
<td>120</td>
</tr>
<tr>
<td>5.3</td>
<td>GTFL$^\uparrow$: Syntax and typing rules</td>
<td>123</td>
</tr>
<tr>
<td>5.4</td>
<td>GTFL$^\uparrow$: Dynamic Semantics of GTFL$^\uparrow$</td>
<td>124</td>
</tr>
<tr>
<td>5.5</td>
<td>Cast Insertion: From GTFL$^\uparrow$ to GTFL$^{\uparrow\downarrow}$</td>
<td>125</td>
</tr>
<tr>
<td>5.6</td>
<td>Proof of dynamic properties approach</td>
<td>126</td>
</tr>
<tr>
<td>5.7</td>
<td>Gradual Intrinsic Terms for GTFL$^{\uparrow\downarrow}$</td>
<td>126</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>C.8</td>
<td>Syntax and notions of Reduction</td>
<td>339</td>
</tr>
<tr>
<td>C.9</td>
<td>Intrinsic Reduction</td>
<td>340</td>
</tr>
<tr>
<td>C.10</td>
<td>Intrinsic term precision</td>
<td>341</td>
</tr>
<tr>
<td>C.11</td>
<td>Syntax of the intermediate language</td>
<td>345</td>
</tr>
<tr>
<td>C.12</td>
<td>GTFL ⊕ ⇒: Type system for the intermediate language</td>
<td>345</td>
</tr>
<tr>
<td>C.13</td>
<td>GTFL ⊕ ⇒: Dynamic Semantics of GTFL ⊕ ⇒</td>
<td>346</td>
</tr>
<tr>
<td>C.14</td>
<td>Cast insertion rules</td>
<td>347</td>
</tr>
<tr>
<td>C.15</td>
<td>Logical relations between intrinsic terms and cast calculus terms.</td>
<td>348</td>
</tr>
<tr>
<td>D.1</td>
<td>Evidence Type Precision</td>
<td>369</td>
</tr>
<tr>
<td>D.2</td>
<td>GSF: Computing Initial Evidence</td>
<td>370</td>
</tr>
<tr>
<td>D.3</td>
<td>GSF: Consistent Transitivity part 1</td>
<td>371</td>
</tr>
<tr>
<td>D.4</td>
<td>GSF: Consistent Transitivity part 2</td>
<td>372</td>
</tr>
<tr>
<td>D.5</td>
<td>GSF to GSFε translation</td>
<td>373</td>
</tr>
<tr>
<td>D.6</td>
<td>GSFε: Syntax and Static Semantics - Contexts</td>
<td>418</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Most programming languages today can be classified as statically or dynamically typed languages. Statically-typed languages bring the benefits of early detection of errors, and less dynamic checks which translates into faster runtime execution, but at the expense of rejecting program that may go right, due to the conservative nature of the static type analysis. Dually, dynamically-typed languages are more flexible, but at the cost of slower runtime execution due to extra dynamic checks, and defer many errors to runtime.

There have been many approaches to integrate static and dynamic typing, such as optional typing [19], hybrid typing [41], and gradual typing [109]. Optional typing allow programmers to partially introduce type annotations to capture some errors statically, but at runtime it performs the same dynamic checks that a dynamic language would do. This approach is used by several languages such as TypeScript [28], Flow [71], and Dart [30]. Hybrid typing [75, 108, 24, 133, 20, 81, 104] combines static analysis and runtime monitoring techniques to make analyses more precise, driven by concerns such as checking of the (un)decidability of a static predicate, or the need to pre-compute information for enhancing runtime checking. Gradual typing, first introduced by Siek and Taha [109], is about the smooth transition between static and dynamic checking based on the precision of type annotations. Type precision is a relation between types, where we say that one type is more precise than the other, if the former represents less static types than the latter. Most gradually-typed languages introduce imprecision via the unknown type ?, which may represents any type whatsoever, e.g. Int is more precise than ?. Type precision helps us define other relations between types, such as consistency , which may be interpreted as the gradual counterpart of type equality, e.g. \( \text{Int} \rightarrow \text{Int} \sim ? \rightarrow \text{Int} \), but \( \text{Int} \rightarrow \text{Int} \not\sim ? \rightarrow \text{Bool} \).

A statically-typed language is designed by specifying both the static semantics (syntax and typing rules) and dynamic semantics (reduction rules). The classical approach to design a gradually-typed language consists in first defining its static semantics, but then the dynamic semantics are defined via a typed-driven translation to a cast calculus, inserting casts (runtime type checks) at the boundaries between static and dynamic typing, ensuring at runtime that no static assumptions are violated. Finally, the cast calculus is designed by specifying its own static and dynamic semantics.
After the introduction of gradual typing, to characterize what it means for a language to be gradually-typed, Siek et al. [113] proposed a **refined criteria** for gradual typing presenting mainly four desirable properties:

- **The conservative extension of the static discipline**: consider a gradual language and its statically-typed language counterpart. Then the static and dynamic semantics of the statically-typed language and the gradual language behave equivalently for fully precise terms.

- **The embedding of the dynamic discipline**: consider a gradual language and its dynamically-typed language counterpart, then any term of the dynamically-typed language can be encoded into a term of the gradual language, where all types are annotated as the unknown type. Additionally, the dynamic semantics of the dynamically-typed language and the gradual language are equivalent for fully imprecise terms.

- **Type safety**: Well-typed programs do not get stuck. They either reduce to values, diverge or fail due to a runtime error.

- **The gradual guarantees**: These properties relate programs with different precision. We say that a program is less or equally precise than another if every type annotation in the first program is less or equally precise than the other.

  - The **static gradual guarantee**: reducing precision on a program does not introduce new type errors.

  - The **dynamic gradual guarantee**: reducing precision on a program does not introduce new runtime errors.

The classical approach to designing gradual languages is questionable: why is it not possible to define the dynamic semantics directly over the gradual source language? How should unknown information be dealt with in the presence of complex type relations, such as subtype polymorphism or subtyping with effects? Also, is the resulting gradual language the “right” counterpart? How to derive the cast calculus? Is it unique? What properties of the static or dynamic language are guaranteed to be preserved in the gradual counterpart? These questions generate the need for a solid foundation for gradual typing, which does not appeal to ad hoc justifications.

There are some approaches to try to answer some of these questions. Cimini and Siek present the **Gradualizer** [26], an algorithm that generates the static semantics of a gradual language and the transformation rules to a cast calculus, from a well-formed static type system. Recently, they extended the gradualizer [27] to also automatically generate the semantics of the cast calculus. Garcia et al. propose a methodology named Abstracting Gradual Typing (AGT) [44], which systematically constructs gradually-typed languages, using abstract interpretation [29] at the type level. The derived gradual language consists of a static and dynamic semantics, without the need of an intermediate cast calculus. Also, the derived language satisfies by construction the refined criteria for gradual typing previously described.
The AGT methodology helps derive a gradual language in the following way:

- Deriving the static semantics
  - Start from a statically-typed language and its type safety proof.
  - Define the syntax of gradual types, and give them meaning via a concretization function: a function from gradual types to sets of static types. Then define its corresponding most precise abstraction function: a partial function from sets of static types to gradual types, forming a Galois connection [29].
  - Existentially lift type predicates and functions used in the type system of the static language through the Galois connection. Starting with the static type system, replace static types with gradual types, and static predicates and type functions with their corresponding gradual counterparts.

- Deriving the dynamic semantics
  - Define the structure of evidence for consistent judgments, usually a pair of types that justify why such a judgment holds.
  - Define the dynamic semantics of the gradual language, mirroring reasoning steps of the static language type safety proof, exploiting the correspondence between proof normalization and term reduction [63]. The static language type safety proof relies on transitivity of types relations, but in a gradual setting this is not always true, so evidence is combined to justify transitivity steps. When such combination of evidence fails, a runtime error is produced.

The AGT methodology has been shown to be effective in different contexts: records and subtyping [44], pure security typing [45], static semantics of gradual effects [12, 14], as well as refinement types [76] and set-theoretic types [22]. But still the scope of AGT is not clear, we need to apply AGT to more language constructs and type disciplines. Some of the benefits of doing this are the following. First, it would be beneficial to know if AGT scales to more complex language constructs and type disciplines. Second, as far as we know, AGT has mostly been applied using abstractions that only add support for the unknown type. There is still work to be done exploring applications of AGT that use different or richer abstractions, for example in a language with subtyping, we could add support for type intervals instead of the unknown type. Third, we know that one of the inputs of AGT is the Galois connection used in the representation of evidence. But all previous work have been using the same Galois connections as the one used for deriving the static semantics. We can answer questions such as: what is the role of this design space? What happens if we choose a different Galois connection for evidence? How does the resulting language behave in comparison? Fourth, we would like to know how the resulting gradual languages compare to related proposals (if they have been independently developed in the literature). Fifth, we can also answer questions regarding properties of the static and the gradual language, such as: can we guarantee by construction properties other than type safety and the gradual guarantees? What conditions

1Bañados Schwerter et al. [12] [14] introduced imprecision on effect sets via the statically unknown privileges ¿.
must hold in order to preserve these other properties? And finally, answer other questions regarding composability of AGT: can we independently apply AGT to different aspects of a static language and then combine the results? Instead of applying AGT to a static language, can we apply AGT to a gradual language to add more expressiveness?

In this thesis we explore the boundaries of AGT, stressing the application of this methodology to complex type disciplines and languages constructs. In particular we first apply AGT to a simply-typed lambda calculus with mutable references, then to a security-typed language with references, and finally to System F, a language with support for *parametric polymorphism*. From these three applications, we learn that the straightforward application of AGT is always possible, but only guarantees to derive gradual languages that satisfies type safety and the refined criteria of Siek *et al.* In order to satisfy other properties, in particular hyper-properties such as *noninterference* [50], and *relational parametricity* [100], first we have to use refined abstractions in the representation of evidence, and second, the dynamic semantics have to be customized in some way. Although with these customizations we are able to recover these properties, the dynamic gradual guarantee is repeatedly lost in the process. In particular, we formally prove that there is a trade-off between the standard interpretation of parametricity and the dynamic gradual guarantee. We also show that AGT can be applied in a stratified way. We notice that for a particular combination of different imprecise gradual types, we cannot obtain an optimal abstraction function (and thus establish a Galois connection). To address this, we apply AGT to a gradual language already derived using AGT, adding support for different types of gradual types in each application in a stratified manner. The following chapters are organized as follows.

Chapter 2 introduces the background necessary to understand the rest of this document. First, we present what is gradual typing and the desired properties of gradual typing. Second, we present a lightweight explanation about the AGT methodology; an explanation in details and an example of application is postponed to Chapter 3. Third, we present background about the three static languages we will apply AGT to, along with related work about corresponding current gradual approaches.

Chapter 3 presents the first application of AGT to a simply-typed lambda calculus with references. This first application has the main objective to explain AGT in detail. Mutable references are a form of *computational effect* that provides three operations: *reference*, *dereference*, and *assignment* that allocates, read, and changes the content of a value in a *store* or *heap* [92] respectively. For instance, consider the following program:

```plaintext
1 let x = ref 4
2 !x
3 x := 10
4 !x
```

Line 1 creates a new reference and returns a new location o pointing to a mutable cell in the store whose content is 4. Line 2 reads 4 from the current stored value of o. Line 3 updates the stored value of o to 10. And finally, line 4 reads again the current stored value of o, which is now 10. Reference operations and locations are typed using reference types Ref T, e.g. the expression ref 4 (and thus location o) is typed as Ref Int.

There are currently four major approaches to gradual typing with references: invariant
references [109], guarded references Herman et al. [60], monotonic references [114], and permissive references [114]. Invariant references are a form of references where, contrary to other approaches, reference types are invariant with respect to type consistency. Guarded references are presented in the coercion calculus (HCC) of Herman et al., which is a space-efficient approach based on coercions, where the runtime type of an allocated cell never changes during execution. Monotonic references favor efficiency over flexibility by only allowing reference cells to vary only towards more precise types. Permissive references, which favors flexibility over efficiency, are developed on top of monotonic references. A permissive reference can be initialized and updated to any value of any type at any time.

In this chapter we present $\lambda^{REF}$, a gradual language derived with AGT with support for mutable references. To illustrate $\lambda^{REF}$ in action, consider the following programs:

```
1 let x = ref (4 :: ?)
2 let y: Ref Bool = x
3 !y ← runtime error
```

Example 1

```
1 let x = ref (4 :: ?)
2 let y: Ref Bool = x
3 y := true
4 !y
```

Example 2

```
1 let x = ref 4
2 let y: ? = true
3 x := y ← runtime error
```

Example 3

Example 1 raises a runtime error at line 3 because it is trying to read a $\text{Bool}$ where an $\text{Int}$ is stored. Example 2 fixes example 1, by updating the location with an actual boolean value before the dereference operation. This is possible because the location is created at type $\text{Ref } ?$, meaning that it can store any value of any type (any type is consistent with $\text{?}$). Example 3 type checks, but a runtime error is raised at line 3, because the assignment of a boolean is incompatible with the type of the reference.

We show that $\lambda^{REF}$ behaves identical to HCC in regards to references. The only difference between both languages is about the order of combination of coercions/evidences. HCC is space efficient whereas $\lambda^{REF}$ is not: we can write programs in $\lambda^{REF}$ that accumulate casts. We present the changes needed in the runtime semantics of $\lambda^{REF}$ to regain space efficiency.

We also present $\lambda^{REF}_{pm}$, an extension of $\lambda^{REF}$ with added support for both permissive and monotonic references. For instance, consider the following $\lambda^{REF}_{pm}$ programs:

```
1 let x = mref (4 :: ?)
2 let y: Ref Bool = x ← runtime error
3 y := true
4 !y
```

Example 4

```
1 let x = mref (4 :: ?)
2 let y: Ref Int = x
3 x := true ← runtime error
```

Example 5

We use the $\text{mref}$ constructor to create monotonic references. In example 4, when variable $x$ is cast to $\text{Ref Bool}$ at line 2, the cast is performed directly on the heap: the cast fails as the stored value as type $\text{Int}$ instead of $\text{Bool}$. In example 5, when variable $x$ is cast to $\text{Ref Int}$, the runtime type of the heap cell is updated to the more precise type $\text{Int}$. Therefore, the subsequent assignment to $\text{true}$ at line 3, triggers a runtime error because $\text{Bool}$ is not consistent with $\text{Int}$.
To summarize, this chapter makes the following contributions:

- We present $\lambda^{\text{REF}}$, a gradual language with support for mutable references. We present and explain all the steps needed to derive this language using AGT.

- We formalize the relation between $\lambda^{\text{REF}}$ and HCC. Given a $\lambda^{\text{REF}}$ term and its compilation to HCC, we prove that both terms are contextually equivalent.

- We formalize the changes needed in the dynamic semantics of $\lambda^{\text{REF}}$ to recover space efficiency.

- We present $\lambda^{\text{REF}}_{\text{pm}}$, an extension of $\lambda^{\text{REF}}$, with support for both permissive and monotonic references.

**Type-driven Gradual Security Typing.** Chapter 4 presents an application of AGT to a complex type discipline: a simply-typed lambda calculus with security types and references. We report on our experience with a number of important considerations that complement the original presentation of AGT. In addition, we highlight the limitation of AGT when applied to semantically-rich type disciplines.

Security typing allows us to classify program entities using security labels that belong to a lattice, such as $\bot \leq L \leq H \leq \top$. Security typing enforces a property called noninterference: high-security inputs do not affect low-security results [50]. For instance the following program

```plaintext
let mix : Int \rightarrow L \rightarrow L Int \rightarrow L Int \rightarrow L =
  fun pub priv => if pub < priv then 1 else 2
```

is rejected statically as the public output (1\text{L} or 2\text{L}) depends on a private argument (\text{priv}). In this chapter we present GSLRef, a gradual security language with references, that adds flexibility by supporting the ? security label. The previous example can be type-checked in GSLRef, by introducing imprecision as follows:

```plaintext
let mix : Int \rightarrow L \rightarrow L Int \rightarrow L Int \rightarrow L =
  fun pub priv => if pub < priv then 1 else 2
```

GSLRef enforces noninterference dynamically: the program mix 1\text{L} 5\text{L} reduces to 1\text{L}, but mix 1\text{L} 5\text{H} reduces to an error as the output would depend on the second private argument.

Noninterference is a modular reasoning principle and as such, allows us to reason about open terms. Just by looking at the signature of a function we can derive free noninterference theorems. For instance, if a function has type Int \rightarrow L \rightarrow L Int \rightarrow L Int \rightarrow L, then we know as a free noninterference theorem, that the result cannot depend on the second argument (otherwise a low security output would depend on a high security value, thus breaking noninterference). GSLRef preserves such type-based reasoning about security. Consider a function smix which wraps mix as follows.

```plaintext
let smix : Int \rightarrow L \rightarrow L Int \rightarrow L Int \rightarrow L =
  fun pub priv => mix pub priv
```
Both programs smix 1_L 5_L and smix 1_L 5_H fail at runtime, because smix cannot reveal any information about its second argument (regardless of the actual security level of the argument), if it is to respect the free noninterference theorem derived from its type.

To summarize, this chapter reports the following contributions:

- We present GSL_{Ref}, a gradual security-typed higher-order language with references, that supports seamless transition between simply-typed and security-typed programming. Security typing annotations alone drive the balance between static and dynamic information flow checking.

- We prove that GSL_{Ref}'s type discipline enforces termination-insensitive noninterference: GSL_{Ref}'s types reflect strong information-flow invariants that hold even in code that contains gradually-typed subexpressions.

- We prove the static gradual guarantee. Interestingly, in order to ensure noninterference in presence of references (and hence implicit flows through the heap), GSL_{Ref} sacrifices the dynamic gradual guarantee.

- Finally, we contribute more generally to the foundations of gradual typing for advanced type disciplines. We find that GSL_{Ref}'s security invariants require separate consideration of syntactic type safety and semantic type soundness, each of which constrains the design of the gradual language.

A Gradual Interpretation of Union Types. Chapter 5 presents a stratified application of AGT to support both a novel design of union types, called gradual union types, and the traditional unknown type. In the literature, two approaches have been developed to safely, and fully statically, deal with the possibility of an expression to have possibly different types: disjoint (or tagged) union types, such as sum types \( T_1 + T_2 \) and variant types, and untagged union types, usually noted \( T_1 \lor T_2 \) [92]. Both forms of union types have complementary pros and cons when viewed from a pragmatic angle. Following the abstract interpretation of gradual types put forth in AGT, a gradual union \( T_1 \oplus T_2 \) is a gradual type that abstracts both \( T_1 \) and \( T_2 \). Seen in this light, a gradual union is a gradual type that is more precise than the unknown type \( ? \). For instance, consider the following program.

```ocaml
let f : Bool \to ? =
  fun x => if x then 1 :: ? else false
```

Function \( f \), given a \( \text{Bool} \), it may return anything. Therefore the program \(( f \ false) 1\) is statically accepted, but will always fail on runtime. Using gradual unions, the same program can be written as follows.

```ocaml
let f : Bool \to \text{Int} \oplus \text{Bool} =
  fun x => if x then 1 :: \text{Int} \oplus \text{Bool} else false
```

Now \( f \) represents a function that given an \( \text{Int} \), \( f \) may return specifically either an \( \text{Int} \) or a \( \text{Bool} \). Therefore, the program \(( f \ false) 1\) is statically rejected; the program \(( f \ false) + 1\) is statically accepted but fails at runtime, and \(( f \ true) + 1\) runs successfully without errors.
To summarize, this chapter presents the following contributions:

- A novel design of union types that combines benefits of both tagged and untagged unions, with added static flexibility backed by runtime checks. Compared to a standard gradually-typed language with only the totally-unknown type \( \Box \), the resulting design is stricter, allowing more blatantly wrong programs to be statically rejected.

- A first example of a stratified approach to AGT. To derive the static semantics of a gradual language, AGT requires a Galois connection between gradual types and sets of static types, which then guides the lifting of functions and predicates on static types to their gradual counterparts \[14\]. We observe that applying AGT directly to introduce both the unknown type and gradual unions breaks optimality of the abstraction, thereby weakening the meaning of type information, both statically and dynamically. To address this, we develop a stratified approach to AGT that allows us to recover optimality. More specifically, we first apply AGT to support only the unknown type, and then we apply AGT once more to introduce support for gradual unions. We prove that the composed abstraction is optimal. We conjecture that this technique might prove helpful in integrating other gradualization efforts.

- The formalization and meta theory of the proposed language, including type safety and the gradual guarantees of Siek et al. \[113\].

- A compilation scheme to an internal language with threesomes, a space-efficient representation for casts \[112\]. We prove the correctness of the compilation with respect to the reference semantics derived by AGT using logical relations.

**Revisiting Gradual Parametricity.** Finally, Chapter \[6\] presents an application of AGT to another complex type discipline: parametric polymorphism, as provided in System F \[49, 100\]. System F, is an extension of the simply-typed lambda calculus (STLC) adding support for terms to depend on types. This is achieved by using explicit type abstractions, type variables, and universal quantification over types. This mechanism of type abstraction is called parametric polymorphism. This allows programmers to reuse code on data of different types. For instance, consider the following program

```plaintext
let first: \forall X. \forall Y. X -> Y -> X = \Lambda X. \Lambda Y. (\lambda x: X. (\lambda y: Y. x))
```

`first` is a type abstraction that given two types, and two values (of the corresponding types), returns the first argument. For instance, `first [Int] [String] 1 'one'` reduces to 1, and `first [Bool] [Int] true 1` reduces to `true`. System F satisfies an important property called parametricity, which captures the intuitions that terms of abstract types must behave uniformly for any type instantiation. This property provides guarantees about the behavior of programs. In other words, static types induce free theorems \[127\] about programs. For instance, consider a function `f` typed `\forall X.X \rightarrow X`. Parametricity guarantees\[2\] that `f` must be the identity function.

In this chapter we present GSF, a gradual counterpart of System F with support for explicit

\[2\]For any pure strongly-normalizing language.
polymorphism, which satisfies parametricity. Consider the following program written in GSF:

```plaintext
let f = λg:(∀ X.X -> X).g [Int] 10
```

And consider three different scenarios where `f` is applied to a function `h` of unknown type.

```plaintext
let h : ? = ΛX.λx:X.x in f h ----> 10
let h : ? = ΛX.λx:?.x in f h ----> 10
let h : ? = ΛX.λx:?.x+1 in f h ----> error
```

In the first case, `h` is the standard System F identity function, and in the second case, `h` is a less precise version, which behaves identically. Therefore, using either of these functions in the program above produces the result 10. Conversely, in the last case function, `h` is not a proper identity function. A runtime error is raised when the body of the function attempts to perform an addition, since this type-specific operation is a violation of parametricity.

Current gradual approaches [5, 6, 7, 130] suffer from some design issues, such as signaling parametric errors in unexpected situations, and improperly handling type instantiations when imprecise types are involved. To illustrate excess of failure, consider the following example:

```plaintext
let f : ∀ X.X -> \? = ΛX.λx:X.x in (f [Int] 1) + 1
```

While the annotated return type of `f` is unknown, the function itself is the identity function. We expect that this program reduce to 2 as a result, but in current gradual approaches, the above program fails with a runtime error. In GSF this issue is addressed, returning value 2 as expected.

To illustrate lack of failure, consider the following example:

```plaintext
let g : ? = ΛX.λx:X.x in g [Int] true
```

We expect this program to fail at runtime, because the identity function is instantiated with `Int` and later applied with a value of an incompatible type `Bool`. Current approaches do not respect type instantiations that involve the unknown type, as this program reduce successfully to `true`. In GSF this issue is addressed, raising a runtime error as expected.

To summarize, this chapter makes the following contributions:

- We introduce GSF, a gradual counterpart of System F that addresses the design issues identified in prior work and satisfies parametricity.
- We uncover that a direct application of AGT breaks parametricity. To recover parametricity we have to take multiple extra considerations in the dynamic semantics.
- We prove properties of the static semantics such as the static gradual guarantee, and the static equivalence for static terms.
- We prove properties of the dynamic semantics such as type safety, a weak version of the conservative extension of the dynamic semantics, and last but not least, parametricity.
- We show that the standard notion of parametricity, is incompatible with the dynamic gradual guarantee.
• We establish a novel property of GSF regarding the wrapping of System F terms into less precise types, which allows us to disprove some claims from the literature about gradual free theorems.

Publications. Chapters 4, 5, and 6 present adaptations of published results. In particular, Chapter 4 titled “Type-driven Gradual Security Typing” was published in ACM Transactions on Programming Languages and Systems 40, number 4, pp.16:1-16:55, November 2018, and presented at POPL 2019 [120]. Chapter 5 titled “A Gradual Interpretation of Union Types” was published and presented in the Proceedings of the 24th Static Analysis Symposium (SAS 2017), volume 10422, pp.382-404, 2017 [124]. Finally, Chapter 6 titled “Gradual Parametricity, Revisited” was published and presented in the Proceedings of the ACM on Programming Languages, issue POPL, volume 3, pp.17:1-17:30, 2019 [122].

We thank Amal Ahmed, Alison M. Clark, Dominique Devriese, Joshua Dunfield, Ronald Garcia, Kenji Maillard, Chris Martens, Gabriel Scherer, Jeremy Siek, the attendees of various oral presentations of these works, and the anonymous reviewers of these publications and previous submissions for their helpful comments, questions, and detailed readings.

To summarize, in this thesis we show that AGT can be applied to complex type disciplines and language constructs. The resulting language is always a gradual version of the static language. A direct application of AGT does not guarantee space efficient operational semantics, but we show how to recover space efficiency by tuning the operational semantics. We also show that the blind application of AGT only guarantees type safety and the refined criteria of gradual languages. In order to preserve other properties, such as noninterference and parametricity, either the application of AGT must be tuned by adjusting the Galois connections, and/or the operational semantics must be customized at the risk of breaking the dynamic gradual guarantee. Finally, we show that it is possible to add expressiveness and extend gradual types by applying AGT to a gradual language already derived with AGT; we call this a stratified approach to AGT.

In the next chapter we present some basic background needed to understand the rest of the chapter of this thesis.
Chapter 2

Background

In this chapter we present basic background needed to understand this thesis. First, we present gradual typing and its desired properties (§2.1). Second, we present a brief explanation of the AGT methodology (§2.1). Third, we present background related to the three static languages we will apply AGT to, along with related work about corresponding current gradual approaches: mutable references (§2.3.1), information-flow security-typing (§2.3.2), and parametric polymorphism (§2.3.3). Detailed formalisms about AGT and the involved static languages are presented in the subsequent chapters.

2.1 Gradual Typing

Most programming languages are either statically or dynamically typed. Static typing and dynamic typing have dual advantages and limitations. For instance, adopting a static discipline provides early detection of errors at the expense of conservatively rejecting some programs that may go right. On the other hand, adopting a dynamic discipline provides flexibility at the cost extra checks (and errors!) at runtime. Therefore there have been a lot of effort to combine static and dynamic typing [31, 23, 79, 51, 115, 37]. One of the most notable contributions in this area is gradual typing, first introduced by Siek and Taha [109].

The key specificity of gradual typing is to support the smooth transition between static and dynamic checking based on the (programmer-controlled) precision of type annotations [109, 113]. Type precision $\subseteq$ is an ordering relation between gradual types, where we say that the gradual type $G_1$ is more precise than $G_2$, notation $G_1 \subseteq G_2$, if $G_1$ represents less static types than $G_2$. In most gradual languages, imprecision is introduced by adding the notion of an unknown type $?$, which represents any static type whatsoever, i.e. $G \subseteq ?$ for any $G$. Type precision also helps us to define consistent type relations. The most common one is type consistency, which represents the gradual counterpart of the equality relation between static types. For instance the $?$ type is consistent with any gradual type and, viceversa, any gradual type is consistent with the $?$ type and itself. This relation can be more complex depending on the static type system counterpart of the gradual language (e.g. consistent subtyping [110] is the gradual counterpart of the static subtyping).
For instance, consider a simple-typed lambda calculus (STLC) extended with the \( \wedge \) type, called \( \lambda^{\wedge} \). The syntax of gradual types can be defined as follows:

\[
G ::= B \mid G \rightarrow G \mid ?
\]

where a gradual type \( G \) may be a base type \( B \), a function type \( G \rightarrow G \), or the unknown type \( ? \). The precision relation between types is defined inductively as follows:

\[
\begin{align*}
B \subseteq B & \\
G_1 \subseteq G'_1 & G_2 \subseteq G'_2 \quad \frac{G_1 \rightarrow G_2 \subseteq G'_1 \rightarrow G'_2}{G \subseteq ?}
\end{align*}
\]

Note that for functions the relation does not flip the order for the argument types: the definition of type precision coincides with the naive subtyping relation [128]. Type consistency is formally defined as:

\[
\begin{align*}
B \sim B & \\
G_1 \sim G_1 & G_2 \sim G_2 \quad \frac{G_1 \rightarrow G_2 \sim G'_1 \rightarrow G'_2}{G \sim ?}
\end{align*}
\]

**Classic design of a gradually-typed language**

A programming language is designed by specifying both its static semantics and runtime semantics. Similarly, a gradually-typed language is designed using almost the same recipe. First, the set of gradual typing rules that account for unknown information are specified. For instance, for \( \lambda^{\wedge} \), the syntax of terms is defined as:

\[
t ::= b \mid x \mid (\lambda x : G.t) \mid t t
\]

where a term \( t \) can be a constant \( b \), a variable \( x \), a function \( (\lambda : G.t) \), or an application \( t t \). The type rules for function application are defined as follows:

\[
\begin{align*}
(G\text{App1}) & \quad \Gamma \vdash t_1 : ? & \Gamma \vdash t_2 : G_2 \\
(G\text{App2}) & \quad \Gamma \vdash t_1 : G_{11} \rightarrow G_{12} & \Gamma \vdash t_2 : G_2
\end{align*}
\]

Judgment \( \Gamma \vdash t : G \) says that term \( t \) has type \( G \) under type environment \( \Gamma \). A type environment \( \Gamma \) is a finite map from variables to types. Rule \( (G\text{App1}) \) is used when the type of \( t_1 \) is unknown (and therefore consistent with anything). Rule \( (G\text{App2}) \) uses type consistency (instead of equality) between the argument type of the function, and the actual type of the argument.

But then the runtime semantics are not defined over the gradual source language. The program is translated into a cast calculus program, inserting casts at the boundaries between static and dynamic typing, ensuring at runtime that no static assumptions are violated. If a static assumption is violated, then a runtime error is raised. The cast calculus is formalized by defining a new type system and its runtime semantics.

For instance, terms of \( \lambda^{\wedge} \) are translated into terms of a cast calculus named \( \lambda^{(G)} \). The syntax of \( \lambda^{(G)} \) is defined by extending the syntax of \( \lambda^{\wedge} \) with casted terms:

\[
v ::= b \mid (\lambda x : G.t) \quad \text{(values)}
\]
\[
t ::= v \mid x \mid t t \mid \langle G_2 \Leftarrow G_1 \rangle t \quad \text{(terms)}
\]

12
where \( G_2 \leftarrow G_1 \) represents a cast from type \( G_1 \) to \( G_2 \). Type \( G_1 \) is called the source type and \( G_2 \) the target type of a cast. The type rules for function application and casts are defined now as follows:

\[
\frac{\Gamma \vdash t_1 : G_{11} \to G_{12} \quad \Gamma \vdash t_2 : G_{11}}{\Gamma \vdash t_1 \ t_2 : G_{12}} \quad \text{(TApp)}
\]

\[
\frac{\Gamma \vdash t_1 : G_1 \quad G_1 \sim G_2}{\Gamma \vdash G_2 \leftarrow G_1 \ t_1 : G_2} \quad \text{(TCast)}
\]

Note that now there is only one rule for application (TApp), and the type of \( t_2 \) must be exactly the same as the domain of \( t_1 \). The only place where consistency is used is when type checking casts (TCast). A selection of reduction rules for the cast calculus is presented as follows:

\[
\langle G \leftarrow G \rangle v \mapsto v \quad (1)
\]

\[
\langle G_2 \leftarrow ? \rangle (\langle ? \leftarrow G_1 \rangle v) \mapsto \langle G_2 \leftarrow G_1 \rangle v \quad (2)
\]

\[
\langle G_2 \leftarrow G_1 \rangle v \mapsto \text{error} \quad \text{if } G_1 \# G_2
\]

\[
\langle (G_1 \rightarrow G_2 \leftarrow G_1 \rightarrow G_2) \rangle v_1 v_2 \mapsto \langle G_2 \leftarrow G_2 \rangle v_1 (\langle G_1 \leftarrow G_1' \rangle v_2) \quad (4)
\]

if not \( G_1 \rightarrow G_2 = G'_1 \rightarrow G'_2 = ? \rightarrow ? \)

Rule (1) reduces trivial casts from and to the same type. Rule (2) combines two casts that go through unknown. Rule (3) produces a runtime error when the types of a cast are incompatible (\( B_1 \neq B_2 \Rightarrow B_1 \# B_2, B \# G_1 \rightarrow G_2, \text{ and } G_1 \rightarrow G_2 \# B \)). Rule (4) reduces the application of a casted function, by casting the argument to the expected domain type, and the resulting value to the codomain type.

Finally, the translation between \( \lambda \) and the cast calculus is done inductively via cast insertion rules. Judgment \( \Gamma \vdash t \Rightarrow t' : G \) says that term \( t \) is translated into term \( t' \), both typed at \( G \) under type environment \( \Gamma \). These rules mimic the structure of the typeing rules of \( \lambda \). For function application, the cast insertion rules are defined as:

\[
\frac{\Gamma \vdash t_1 \Rightarrow t'_1 : ? \quad \Gamma \vdash t_2 \Rightarrow t'_2 : G_2}{\Gamma \vdash t_1 \ t_2 \Rightarrow (\langle G_2 \rightarrow ? \leftarrow ? \rangle t'_1) t'_2 : ?} \quad \text{(CApp1)}
\]

\[
\frac{G_{11} \sim G_2}{\Gamma \vdash t_1 \Rightarrow (G_1 \leftarrow G_2) t_2 : G_{12}} \quad \text{(CApp2)}
\]

\[
\frac{\Gamma \vdash t_1 \Rightarrow t'_1 : G_{11} \rightarrow G_{12} \quad \Gamma \vdash t_2 \Rightarrow t'_2 : G_{11}}{\Gamma \vdash t_1 \ t_2 \Rightarrow t'_1 t'_2 : G_{12}} \quad \text{(CApp3)}
\]

Rule (CApp1) gains a bit of information about the domain of the function term, so the function term is cast to a function that takes as argument something of type \( G_2 \). Rule (CApp2), knows that the argument type of the function is consistent with the argument, but they are not the same, so the argument is casted into the domain type of the function. Finally, Rule (CApp3) does not introduce unnecessary casts as they are going to be trivially eliminated during reduction. Observe that for function application, we have one rule in the static language, two rules in the gradual source language, and three rules in the translation semantics. This raises the question of how to come up with these rules.
Desirable properties of gradual languages

There are many desirable properties that a gradual language should satisfy. In the context of giving a formal characterization of what it means to be gradually-typed, Siek et al. [113] proposed a refined criteria for gradual typing proposing four fundamental properties:

- Conservative extension of the static discipline. A gradual type system is equivalent to its static type system counterpart on fully annotated programs. Formally,

\[ \Gamma \vdash_S t : T \iff \Gamma \vdash t : T \]

where \( \Gamma \vdash_S t : T \) is the judgment used to type check terms in the static language (for instance in a simple-typed λ-calculus). In addition, both reductions behave equivalently:

\[ t \Downarrow_S v \iff t \Downarrow v \]

where \( \Downarrow_S \) and \( \Downarrow \) denote big step reduction relations of the static and gradual language respectively.

- Embedding of the dynamic discipline. Terms from the corresponding dynamic language can be encoded into terms of the gradual language, where all types are annotated as \( ? \). Let \( t \) be a term of the dynamic language, then

\[ \vdash [t] : ? \]

where \( [\cdot] \) is a function that introduces \( ? \) type annotations everywhere. Also, the reductions of a term of the dynamic language and its corresponding encoding in the gradual language behave equivalently. Formally,

\[ t \Downarrow_D v \iff [t] \Downarrow [v] \]

where \( \Downarrow_D \) denotes the reduction relation of the corresponding dynamic language.

- Type Safety. First, well-typed programs do not get stuck, and second, either well-typed programs reduce to values, diverge, or fail due to a runtime check error.

Formally, if \( \vdash t : G \) then either \( t \Downarrow v \) and \( \vdash v : G, t \uparrow \), or \( t \Downarrow \text{error} \), where \( t \uparrow \) indicates that \( t \) diverges.

- Gradual guarantees. These guarantees relate terms with different precision. Term precision is the natural lifting of precision on gradual types (e.g. a term typed \( ? \) can be typed more precisely than \( ? \rightarrow \text{Int} \)).

  - Static gradual guarantee: typeability is monotone with respect to imprecision, i.e. reducing precision does not introduce new type errors. Formally, let \( t \) and \( t' \) terms such \( t \sqsubseteq t' \), where \( t \sqsubseteq t' \) denotes the natural lifting of precision to terms.

\[ \vdash t : G \Rightarrow (\vdash t' : G' \land G \sqsubseteq G') \]

---

1It is important to note that this proposal leaves out all discussions related to blame (knowing what parts of the code are to be blamed when an error occurs). Blame is out of scope of this thesis work.
Dynamic gradual guarantee: reducibility is monotone with respect to imprecision, i.e. reducing precision does not introduce new runtime errors. Let $t$ and $t'$ such that $t \sqsubseteq t'$, then

$$t \Downarrow v \Rightarrow (t' \Downarrow v' \land v \sqsubseteq v')$$

Ad hoc gradual typing. The classical approach to designing gradual languages, followed by most gradual systems, presents some challenges: how to come up with the typing rules? why is it not possible to define the runtime semantics directly over the gradual source language? How should unknown information be dealt with? The last question may seem straightforward to answer for relations such as the consistency relation, but in reality, it gets complicated if we consider to other type relations such as subtype polymorphism or subtyping with effects. Also, is the resulting gradual language the “right” definition of its static language counterpart? And with respect to the cast calculus: how is it derived from the source gradual language? is it unique? These questions generate the need for a solid foundation for gradual typing, which does not appeal to ad hoc justifications.

2.2 Abstracting Gradual Typing

Garcia et al propose a methodology named Abstracting Gradual Typing (AGT) [44], which uses abstract interpretation [29] at the type level to systematically constructs gradually-typed languages from pre-existing statically-typed ones. The derived gradual language consists of a gradual type system and runtime semantics, without the need for an intermediate cast calculus. A gradual language built with the AGT approach fulfills, by construction, the design goals of gradual typing and satisfies the refined criteria for gradual typing formulated by Siek et al [113]. Detailed formalism about AGT is presented in Chapter 3.

The AGT methodology proposes to derive the static and dynamic semantics of a gradual language in the following manner.

1. Deriving the statics.

   (a) Start from a language with a fully-static typing discipline, including the particulars of its type safety proof.

   (b) Define the syntax of gradual types, and give them meaning via a concretization function, which maps gradual types to sets of static types; then define the corresponding most precise abstraction function, forming a Galois connection. For example, a gradual type may be the unknown type $\textit{?}$, which represents any type. Therefore, in this case, the concretization of the unknown type is the set of all possible static types. But the unknown type can be restricted to represent selected types instead, for instance $\texttt{Bool}$ or $\texttt{Int}$, in which case the concretization of unknown would be the set of both types.

   (c) Existentially lift type predicates and functions used in the type system of the static language through the Galois connection to obtain the gradual type system. For
instance, equality between static types is lifted into consistent equality between gradual types. The lifting is based on plausibility: a lifted predicate between gradual types holds if and only if there is a concretization of the involved gradual types that satisfies the static predicate. Similarly, in a language with subtyping, the join function calculates the least upper bound between two static types. That function is then lifted into the consistent join function that calculates the consistent least upper bound between two gradual types by abstracting the result of applying the function to all possible concretizations.

2. Deriving the dynamics.

(a) Define the structure of evidence for consistent judgments, which represents justification for why such a judgment holds; this representation depends on a Galois connection—usually the same as the one used for deriving the static semantics, but not necessarily.

(b) Reduce gradual programs by reducing gradual typing derivations decorated with evidence, mirroring reasoning steps of the static language type safety proof, hence exploiting the correspondence between proof normalization and term reduction. But the reduction of a type safety proof relies on the transitivity of type relations, which does not always hold in a gradual context. For instance, equality is a transitive type relation, whereas type consistency is not always transitive: \( \text{Int} \sim ? \) and \( ? \sim \text{Bool} \) but \( \text{Int} \not\sim \text{Bool} \). Therefore, given a pair of consistent judgements, their evidences may justify the transitivity judgement. To justify the transitivity judgement, a consistent transitivity operator is defined on evidences, which may fail producing a runtime error.

Therefore, the “inputs” to AGT are only the static language, and the Galois connection(s) that give meaning to gradual types and evidences. As “output”, one obtains the static and dynamic semantics of the gradual language, together with the guarantee that it is type safe, it satisfies the gradual guarantees, and depending on the syntax and meaning of gradual types, it is a conservative extension of the static and dynamic disciplines.

Note that in order to achieve an implementation one must also provide algorithmic characterizations of the operators obtained through the abstract interpretation methodology. Often these algorithms can be calculated by induction on types, but sometimes it requires trial-and-error. In any case, the AI-based definition provides the baseline against which to formally validate such characterizations.

2.3 Complex Type Disciplines and Language Constructs of Interest

Deriving gradual languages from complex type disciplines and language constructs is hard. In particular enforcing “equivalent” properties of the static language may be challenging. In many cases derivations appeal to ad-hoc definitions and relations. How do we know that the
resulting gradual language is a “right” derivation? In this section we introduce three different type disciplines and language constructs of interest that we will gradualize using the AGT methodology. We start by presenting mutable references in § 2.3.1 then information-flow security typing in § 2.3.2 and finally parametric polymorphism in § 2.3.3.

### 2.3.1 References

We will apply AGT to a simply-typed lambda calculus with mutable references ($\lambda^{REF}$). Mutable references are a form of computational effects that provides an assignment operation that changes the content of a value in a store or heap [92]. $\lambda^{REF}$ adds three basic operations on references: allocation $\text{ref}\ t$ (which allows to create new references, returning a new location or pointer), dereferencing $!t$ (which allows to read the referenced value of a location), and assignment $t := t$ (which allows to update the referenced value of a location). For instance:

```plaintext
1 let x = ref 4
2 !x
3 x := 10
4 !x
```

Line 1 creates a new reference and returns a new location $o$ pointing to a mutable cell in the store whose content is 4. Line 2 reads 4 from the current stored value of $o$. Line 3 updates the stored value of $o$ to 10. And finally, line 4 reads again the current stored value of $o$, which is now 10. An allocation term $\text{ref}\ t$ has type $\text{Ref}\ T$ where $T$ is the type of the subterm $t$. Locations $o$ are not part of the source language; they are introduced during reduction. To type locations we use a store type $\Sigma$, a finite map from locations to types, such that $o$ has type $T$ if $\Sigma(o) = T$. One interesting particularity of reference types is that they are invariant with respect to subtyping, i.e. $\text{Ref}\ T_1 \sim \text{Ref}\ T_2$ if and only if $T_1 = T_2$. This observation is key in a gradual language when considering the type consistency relation as we explain next.

#### Existing gradual approaches.

We now briefly review the four major formulations of gradual typing with references that have been proposed in the literature.

**Invariant references.** Siek and Taha [109] include a treatment of references in their original gradual typing work. However, based on the observation that “allowing variance under reference types compromises type safety”, they impose reference types to be invariant with respect to type consistency. In other words, $T_1 \sim T_2$ does not imply that $\text{Ref}\ T_1 \sim \text{Ref}\ T_2$. Consider example 1, which presents a program being rejected at line 2 because $\text{Ref}\ ?$ is not consistent with $\text{Ref}\ Int$:

```plaintext
1 let x = ref (4 :: ?)
2 let y: Ref Int = x ← type error
3 y := 10
```

**Example 1**

**Guarded references.** Herman et al. [60] develop a space-efficient approach to gradual typing
based on coercions \[59\]. We call this language HCC for future references. Their language
includes references, albeit with a different semantics from the one proposed by Siek and
Taha \[109\]. In particular, the type system allows consistency variance for reference types,
i.e. \( T_1 \sim T_2 \Rightarrow \text{Ref } T_1 \sim \text{Ref } T_2 \). The dynamic semantics of the language is given by
translation to a language with coercions. Coercions can be normalized in order to avoid
accumulation of wrappers that compromise space efficiency. This normalization allows to
eagerly combine coercions, allowing to detect some errors immediately—e.g. at function
application—in contrast to lazier approaches that accumulate wrappers and errors are not
detected until the casted function is applied. The resulting semantics is called “guarded”
because each reference cell assignment (resp. dereference) is guarded with a coercion from
the static type of the assigned values (resp. expected type of the read value) to/from the
actual type of the reference cell. In other words the runtime type of an allocated cell never
changes during execution. The approach is intuitively justified by analogy with how first-class
functions can be used at different (consistent) types, provided that the appropriate guards
check arguments and return values.

Examples 2, 3 and 4 illustrate the use of guarded references:

1. let \( x \) = ref (4 :: ?)
2. let \( y : \text{Ref } \text{Bool} = x \)
3. \( !y \leftarrow \text{runtime error} \)

Example 2

1. let \( x \) = ref (4 :: ?)
2. let \( y : \text{Ref } \text{Bool} = x \)
3. \( y := \text{true} \)
4. \( !y \)

Example 3

Example 2 raises a runtime error at line 3 because it is trying to read a \text{Bool} where an \text{Int}
is stored. Example 3 fixes example 2 by updating the location with an actual boolean before
the dereference operation. This is possible because the location is created at type \text{Ref } ?,
meaning that it can store any value of any type (any type is consistent with ?). In example
4 a runtime error is raised at line 3 because the coercion from \text{Bool} to ? cannot be combined
with the coercion from ? to \text{Int}.

Monotonic references. Siek et al. \[114\] propose a design for gradually-typed references
called monotonic references. The design is driven by efficiency considerations, namely allowing
statically-typed code to be compiled with direct memory access instructions—without
coercions or wrappers. Like guarded references, monotonic references are variant under con-
sistency. In order to avoid using reference wrappers in statically-typed code, the runtime
type of reference cells is allowed to vary—casts on references are performed directly on the
heap—but only towards more precise types. The monotonicity restriction ensures that direct
reference accesses from statically-typed code are safe.

Examples 3 and 5 illustrate the use of monotonic references:

1. let \( x \) = ref (4 :: ?)
2. let \( y : \text{Ref } \text{Bool} = x \leftarrow \text{runtime error} \)
3. \( y := \text{true} \)
4. \( !y \)

Example 3

Example 4

Example 5
In example 3, when variable \( x \) is cast to \( \text{Ref Bool} \) at line 2, the cast is performed directly on the heap: the cast fails as the stored value as type \( \text{Int} \) instead of \( \text{Bool} \). In example 5, when variable \( x \) is cast to \( \text{Ref Int} \), the runtime type of the heap cell is updated to the more precise type \( \text{Int} \). Therefore, the subsequent assignment of \( \text{true} \) to \( x \) triggers a runtime error at line 3 because \( \text{Bool} \) is not consistent with \( \text{Int} \). Note that under the guarded semantics this program runs without errors. The difference is that accesses to \( y \) in the monotonic semantics are ensured that the value on the heap is of type \( \text{Int} \), while under the guarded semantics a coercion will be necessary.

**Permissive references.** The monotonicity discipline favors efficiency over flexibility. Siek et al. [114] also develop a flexible notion of permissive references on top of the language with monotonic references. In essence, permissive references consist in treating the type of all heap cells as \( ? \). A source-level translation then adds the necessary ascriptions on dereferences and assignments. Note that the transformation would work equivalently using the guarded semantics as target (but not with the invariant semantics).

The following example shows the previous example, which is rejected at runtime by the monotonic system, this time written using permissive references (left) and the program once transformed to the monotonic language (right):

```
1 let x = ref* 4
2 let y: Ref* Bool = x
3 x := true
4 !y
```

```
1 let x = ref (4 :: ?)
2 let y = x
3 x := (true :: ?)
4 (!y) :: Bool
```

Example 6  
Example 7

With the permissive semantics, the program does not produce any runtime error. The first line of the transformed program creates a new reference of type \( \text{Ref} \ ? \). The third line shows that every assigned value is first ascribed the \( ? \) type. Therefore the runtime type of the heap cell does not change: it stays \( ? \). Finally, in the last line, since the variable \( y \) originally had type \( \text{Ref}^* \text{Bool} \), the dereference is ascribed the type \( \text{Bool} \).

Note that the permissive semantics is even more flexible than the guarded semantics, which allows programmers to fix the type of heap cells at more precise types than \( ? \).

### 2.3.2 Information-Flow Security Typing

This is the first complex type discipline that we want to consider to apply the AGT methodology. To explain what information-flow security typing is, consider a program that processes employee data:

```
1 let intToString : Int → String = ...  
2 let print : String → Unit = ...
```

As in [114], we use \( \text{ref}^* \ t \) to denote the permissive reference constructor, and \( \text{Ref}^* \ T \) to denote a permissive reference type.

\[^2\text{Adapted from 35.}\]

\[^3\text{Adapted from 35.}\]
The program is well-typed, but it has a significant error that simple types do not catch: if salaries are confidential and printing is publicly observable, then this program leaks confidential data.

Information-flow security typing lets a programmer statically classify program entities according to a lattice of security labels [32] and rely on type-checking to prevent information leaks. One exemplar security lattice, which we use as a running example, is the U.S. Dept of Defence classification scheme: Unclassified \( \preceq \) Confidential \( \preceq \) Secret \( \preceq \) Top Secret, which we simplify to \( \bot \preceq L \preceq H \preceq \top \), denoting minimum, low, high, and maximum security respectively [131]. To inform static type checking, each type constructor is statically annotated with a security label (e.g. \( \text{Int}_L \) denotes a low security integer, and \( \text{Int}_L \rightarrow_H \text{Int}_H \) denotes a high security function, that given a low security integer it produces a high security integer); source program values are also annotated to unambiguously determine their static security (e.g. \( 58000_H \) has type \( \text{Int}_H \)). Security label ordering induces a natural subtyping relation (e.g. \( \text{Int}_L \ll \text{Int}_H \) and \( \text{Int}_H \rightarrow_L \text{String}_L \ll \text{Int}_L \rightarrow_H \text{String}_H \)), which denotes security-respecting substitutability. Armed with security types and subtyping, an information-flow security type system statically ensures that high-confidence data may not flow directly or indirectly to low-confidence channels [126].

In the example above, if we annotate the salary as high-security data (of type \( \text{Int}_H \)), and specify that print takes a low-security argument (of type \( \text{String}_L \)), then our operational intuition tells us that the program cannot satisfy these directives: it should be rejected. To combat this, information-flow security languages enforce a general property called noninterference, which guarantees that high-security inputs do not affect low-security results [50]. Noninterference clearly subsumes our simple security leak, but before the type system can reject this program, we must determine the security levels of every type in the program. In a static language, this means that every type and value must be annotated. While security label inference and polymorphism [86] can reduce this burden, one cannot experiment with some security levels without first determining all security levels. Once all security types are assigned, the static type system forbids passing a high-security value to a function that expects a low-security argument, so the type checker rejects the above program.

Security types induce free noninterference theorems. The employee data example demonstrates a simple security leak, where high-security data flows directly to a low-security channel. But security types must also contend with sophisticated leaks, where low-security variables may change control-flow through high-security code and mutable state can enable implicit security leaks [32]. Noninterference also prevents implicit and control-based leaks, where an attacker attempts to use low-security inputs and outputs to learn about high-security data.

In security-typed languages, higher-order security types denote modular guarantees about noninterference [57]. For example, consider a hypothetical function:

\[
\text{let mix} : \text{Int}_L \rightarrow_L \text{Int}_H \rightarrow_L \text{Int}_L = \text{fun pub priv} \Rightarrow ...
\]
Based on its name, it appears to “mix” its arguments \texttt{pub} and \texttt{priv} to produce some result. However, the security annotations on its type guarantee that the integer result \textit{cannot} leak information about \texttt{priv}, no matter what value is given to \texttt{pub}. The key to this result is how the relevant typing judgment is interpreted. The body of the \texttt{mix} function, \( t \), must satisfy the typing judgment \( \texttt{pub} : \texttt{Int}_L, \texttt{priv} : \texttt{Int}_H \vdash t : \texttt{Int}_L \). To endow this judgment with meaning, a logical relation-based semantic model \cite{101} is defined directly in terms of the language dynamic semantics. According to this \textit{semantic typing judgment}, changing the value of \texttt{priv} has no effect on the final value of \( t \). This guarantee holds even if \texttt{mix} uses mutable state \cite{131}. The end result is that an attacker with no direct access to a high-security channel cannot manipulate the value of \texttt{pub} to uncover the value of \texttt{priv}.

In a static security language, these noninterference guarantees follow from the type structure of the language. No runtime checks are required, and the security labels applied to values and types are simply static annotations. Like type annotations, security labels appear in dynamic semantics solely to prove type safety and noninterference: they are erased in a practical runtime. In essence, static security types induce \textit{free theorems} about the noninterference behaviors of computations, just as parametric polymorphic types induce free theorems about data abstraction \cite{127}. Free noninterference theorems provide enormous benefits to programmers. First, they support \textit{modular} reasoning about noninterference: a programmer who implements a higher-order function with type \((\texttt{Int}_L \rightarrow \texttt{Int}_H \rightarrow \texttt{Int}_L) \rightarrow \texttt{Bool}_H\) knows that the function body can safely call its argument with high-security data as the second argument: the provided function cannot leak that data. Second, type-based reasoning is \textit{compositional}: the syntactic typing rules precisely specify how the security properties of subprograms (\textit{e.g.} a function-typed expression and a potential argument) compose to determine security properties of a larger program (\textit{e.g.} via function application). Finally, this reasoning is \textit{static}: one need not reason directly about operational behavior or data flow to understand security. That reasoning can be done once-and-for-all in a type-driven noninterference proof: type structure guides reasoning. These properties are especially useful for partial programs like software libraries.

\textbf{Existing gradual approaches.} Like any static type discipline, security typing has its downsides. As discussed above, security typing cannot be checked until all types are given a security level, through ascription, polymorphism, or inference. One cannot incrementally add security levels and observe the consequences. In addition, verifying noninterference is in general undecidable, so static security checking is necessarily conservative, and as a result programmers must sometimes refactor perfectly safe and clear code simply to appease the type checker.

To address these shortcomings, researchers have explored ways to combine static and dynamic security checking. These approaches can be classified roughly as hybrid \cite{108, 24, 133, 20, 81, 104} or gradual \cite{35, 39, 40}. It is important to clarify that the prior work in this space, hybrid and gradual alike, take a \textit{check-driven} approach to analysis: the core of the security model is based on associating a security level to each \textit{value} in a program and managing security levels using two distinct operations: security \textit{upgrades} and \textit{checks}. A security upgrade elevates a value’s security label, \textit{e.g.} \((\texttt{Int}_H)5_L \rightarrow 5_H\). A security check signals an error if the checked label is not at least as high as the value’s tag, \textit{e.g.} \((\texttt{Int}_H?)5_L \rightarrow 5_L\), but
(\text{\texttt{Int}}_L ?)_{\delta H} \rightarrow \text{error}. \text{ Upgrades and checks have different dynamic behavior, but with help from static typing, gradual security languages combine them into type-based upcasts and downcasts, e.g. } (\text{\texttt{Int}}_L) t, \text{ which checks if } L \text{ is lower than } t's \text{ static security and upgrades } t \text{ otherwise. This approach easily detects direct flows of high-security values to low-security channels, but preventing implicit flows through control transfer requires extra care, including prophylactic upgrades to program values } [21] \text{ and policies to restrict upgrades } [39]. \text{ Disney and Flanagan } [35] \text{ and Fennell and Thieman } [39] \text{ pioneered a check-driven approach to gradual security typing, starting from dynamic checking. Both develop notions of blame tracking and prove blame theorems for their semantics. Later, Fennell and Thiemann } [40] \text{ extend their prior work on gradual security typing with references to the object-oriented setting, in a language called LJGS. LJGS performs local inference of security labels, and supports polymorphic security signatures. It is also important to recall that these approaches, while dubbed "gradual", are based on explicit security casts, and are therefore more akin to cast calculi than to gradual languages. In particular, this means that these languages do not respect the gradual guarantees by design, specially the static one, because changing the precision of type annotations requires manually adding/removing explicit casts.}

\subsection*{2.3.3 Parametric Polymorphism}

Parametric polymorphism is the second complex typing discipline that we will consider to apply the AGT methodology. In the simply-typed lambda calculus (STLC), function definitions allows to abstract terms out of terms, and later instantiate these abstractions by applying them with some argument. The polymorphic lambda calculus [49, 100] (System F), is an extension of STLC adding support for terms to depend on types. This is achieved by using type variables in place of actual types, introduced by type abstraction $\Lambda X.t$. And, similarly to STLC, type abstractions can be instantiated to any type, using type application $t[T]$. Type abstraction is supported by the type system by adding universal quantification over types, e.g. term $\Lambda X.t$ has type $\forall X.T$, where $t$ has type $T$ (which may contain $X$ as a free type variable). This mechanism of type abstraction is called parametric polymorphism. This allows programmers to reuse code on data of different types. For instance, consider the following program

\begin{verbatim}
1 let first: \forall X. \forall Y. X \rightarrow Y \rightarrow X = \Lambda X. \Lambda Y. (\lambda x: X. (\lambda y: Y. x))
2 first [Int] [String] 1 'one'
3 first [Bool] [Int] true 1
\end{verbatim}

The \texttt{first} function abstracts over the types used in the nested lambdas. This means that \texttt{first} can be instantiated with any types $T_1$ and $T_2$, returning a new function that takes two arguments of those types, and that always returns the first argument of type $T_1$. Lines 2 and 3 instantiate \texttt{first} with \texttt{Int} and \texttt{String}, and \texttt{Bool} and \texttt{Int} respectively, and then apply the resulting functions with arguments of their corresponding types; returning 1 and \texttt{true} respectively.

As type abstractions must be able to be instantiated with any type whatsoever, it is unnatural to manipulate a term of some abstract type in a way that depends on its current type. For instance, the following program is rejected by the type system because it potentially tries to use a term of an abstract type as a number:
\begin{verbatim}
  let plus1 : \forall X. X \to X = \Lambda X. (\lambda x : X. x + 1)
\end{verbatim}

If we accepted this program, then program `plus1 [Bool] true` would get stuck, because addition is not defined on booleans.

**Relational Parametricity.** This last restriction which establishes that terms of abstract types must behave uniformly for any type instantiated, is formalized in a modular reasoning principal called *relational parametricity* \[101\]. This property relates two terms of the same type if they have the same behavior, which is commonly expressed using logical relations. Informally, we say that two values whose type is a *base type* (e.g. `Bool`, `Int`, `Unit`) are related if the values are equal. Two functions are related if their application to two related argument values are related computations. We say that two terms are related computations if both terms can be reduced to two related values. Finally, two type abstractions are related if their instantiation to two—not necessarily equal—types, are related. This is the key aspect of relational parametricity, as pairs of values typed as the instantiated abstract types are related under any arbitrary relation. This implicitly imposes the restriction that values of abstract types cannot be manipulated according to their actual type.

The *parametricity* property, or fundamental property \[100\], states that if a term is well-typed, then the term must be related with itself at its own type. This simple, but powerful property provides guarantees about the behavior of programs. In other words, static types induce *free theorems* \[127\] about programs. For instance, consider a function `f` typed \( \forall X. X \to X \). The parametricity property for that type guarantees that, for any strongly normalizing language, `f` must be the identity function `id` which always return its argument, i.e. \( \forall v : T, f[T]v \downarrow v \). Similarly, a function `g` typed \( \forall X. \forall Y. X \to Y \to Y \) must be a function that always returns its second argument, i.e. \( \forall v_1 : T_1, \forall v_2 : T_2, g[T_1][T_2]v_1 v_2 \downarrow v_2 \).

**Explicit vs Implicit.** In a language with *explicit* polymorphism, such as System F, the term language includes explicit type abstraction \( \Lambda X.t \) and explicit type application \( t[T] \). In languages with *implicit* polymorphism, such as Haskell, type abstraction and type application are not written down by programmers, but inferred by the type system. Implicit polymorphism induces a notion of subtyping that relates polymorphic types to their instantiations \[80\] \[89\]; e.g. \( \forall X.X \to X <: \text{Int} \to \text{Int} \). Implicitly-polymorphic languages generally use an explicitly-polymorphic languages underneath (e.g. GHC Core), providing the convenience of implicitness through an inference phase that produces an explicitly-annotated program. In essence, the use of the subtyping judgment \( \forall X.X \to X <: \text{Int} \to \text{Int} \) is materialized by introducing an explicit instantiation \( \text{Int} \), and vice-versa, the use of the judgment \( \text{Int} \to \text{Int} <: \forall X.\text{Int} \to \text{Int} \) is materialized by inserting a type abstraction constructor \( \Lambda X \).

**Gradual parametric polymorphism** Gradual parametricity supports imprecise typing information, yet ensures that assumptions about parametricity are enforced at runtime whenever they are not provable statically. In the following program, function `f` expects a function `g` of type \( \forall X.X \to X \) as argument. It is applied to an argument `h` of the unknown type. By consistency, this program is well-typed; however the compliance of `h` with respect to its assumed parametric signature is unknown statically.
let f = \g:(\forall X.X \to X).g [\text{Int}] 10 in let h : ? = ... in f h

By parametricity, function \( f \) can deduce that \( g \) behaves like the identity function. In presence of gradual types, this conclusion should be relaxed to account for the fact that using \( g \) might raise a runtime error if \( g \) fails to comply with its signature. Therefore, as a consequence of parametricity, we can prove that if the program above terminates, it should either produce 10, or fail with a runtime error denoting that \( h \) was in fact not a proper identity function.

Let us consider three possible implementations of \( h \):

\[
\begin{align*}
h_1 &= \Lambda X.\lambda x:X.x \\
h_2 &= \Lambda X.\lambda x:? .x \\
h_3 &= \Lambda X.\lambda x:? .x+1
\end{align*}
\]

Function \( h_1 \) is the standard System F identity function, and function \( h_2 \) is a less precise version, which behaves identically. Therefore, using either of these functions in the program above produces the result 10. Conversely, function \( h_3 \) is not a proper identity function. Note that the function is well-typed, because \( x \) has type \(?\) in the body. Also, using \( h_3 \) in the program above is type safe, because \( f \) happens to instantiate its argument at type \text{Int}, so execution could proceed safely without errors and yield 11; this would however be a violation of parametricity, so an error should be raised.

Existing gradual approaches While the basics of gradual parametricity are well understood, the details are tricky. In particular, establishing that a gradual parametric language enforces parametricity has been a long-standing open issue: early work on the polymorphic blame calculus did not prove parametricity [5, 6]; only very recent work on a variant of that calculus, \( \lambda B \), has achieved this result [7]. In fact, \( \lambda B \) is a cast calculus, not a gradual source language, meaning that the program written above would not be valid; explicit casts should be sprinkled in different places to achieve the same result. Igarashi et al. recently developed a gradual source language, System F\(_G\), which does support the intended lightweight, cast-free syntax of gradual languages. Following the early tradition of gradual typing [109], the semantics of System F\(_G\) are given by translation to a cast calculus, System F\(_C\), which is a close cousin of \( \lambda B \). Igarashi et al. in fact do not prove parametricity, but conjecture that due to the similarity between System F\(_G\) and \( \lambda B \), parametricity should hold. Xie et al. [130] develop a language with implicit polymorphism (here referred to as CSA), which compiles to \( \lambda B \) and therefore satisfies parametricity. \( \lambda B \) is not a conservative extension of System F (§6.2.1), and the gradual guarantees are left as an open question. System F\(_G\) is a conservative extension of System F, and CSA of an implicit variant of System F. Both System F\(_G\) and CSA satisfy the static gradual guarantee, although System F\(_G\) uses an ad hoc notion of precision tuned to that effect (§6.2.1). The dynamic gradual guarantee for both System F\(_G\) and CSA are still open questions.

Finally, gradual free theorems about imprecise type signatures have not been formally studied, beyond a number of claims that we mention later (and disprove) in §6.10.

In the next chapter we start by applying AGT to a simply-typed lambda calculus with mutable references, explaining more in detail the AGT methodology along the way.

\footnote{The program might also diverge; indeed, gradual simple types admit non-termination [109].}
Chapter 3

First step: Gradualizing References

In this chapter we apply the AGT methodology to a simply-typed lambda calculus with references. This chapter not only serves to showcase the application of AGT in a context with references, but also serves as a stepping stone to understand the AGT methodology in detail. We present $\lambda^{\text{REF}}$, a gradual language with references (§ 3.3). We show that $\lambda^{\text{REF}}$ satisfies Siek et al.’s criteria for gradually-typed languages (§ 3.3.6). $\lambda^{\text{REF}}$ treats references as guarded references, presented initially in the coercion calculus of Herman et al. [60] (HCC). We present a translation semantics from $\lambda^{\text{REF}}$ to HCC+\textsuperscript{+}, an adapted version of HCC (§ 3.4.1). We formalize the relation between $\lambda^{\text{REF}}_\varepsilon$ (the intrinsic semantics of $\lambda^{\text{REF}}$) and HCC+\textsuperscript{+} showing that for any $\lambda^{\text{REF}}$ programs both semantics are behaviorally equivalent, save for space efficiency (§ 3.4). Later, we show the changes needed in $\lambda^{\text{REF}}$ to regain space efficiency (§ 3.4.3). Finally, we present the final changes to $\lambda^{\text{REF}}$ in order to support two other semantics of gradual references: permissive references and monotonic references (§ 3.5).

3.1 Gradual Typing with References

Several designs for gradually-type references have been proposed as shown in § 2.3.1. These four developments reflect different design decisions with respect to gradual references: is the reference type constructor variant under consistency? Can the programmer specify a precise bound on the static type of a reference, and hence on the corresponding heap cell type? Can the heap cell type evolve its precision at runtime, and if yes, how? There is obviously no absolute answer to these questions, as they reflect different tradeoffs. This chapter explores the semantics that results from the application of AGT, which has never been applied to mutable references in isolation. In this chapter we answer the following open questions: Which semantics for gradually-type references follows by systematically applying AGT? Does AGT justify one of the existing approaches, or does it suggest yet another design? Can we recover other semantics for gradual references, if yes, how?
Contributions  To summarize, this work makes the following contributions:

• We present $\lambda_{\text{REF}}$, a gradual language with support for mutable references ($\S$3.1). We derive $\lambda_{\text{REF}}$ by applying the AGT methodology to a simple language with references called $\lambda_{\text{REF}}$ ($\S$3.3). This is the first application of AGT that focuses on gradually-typed mutable references.

• We prove that $\lambda_{\text{REF}}$ satisfies the gradual guarantee of Siek et al. [113]. We also present the first formal statement and proof of the conservative extension of the dynamic semantics of the static language [113], for a gradual language derived using AGT ($\S$3.3.6).

• We prove that the derived language, $\lambda_{\text{REF}}$, corresponds to the semantics of guarded references from HCC ($\S$3.4). Formally, given a $\lambda_{\text{REF}}$ term and its compilation to HCC$^+$ (an adapted version of HCC) we prove that both terms are contextually equivalent via a bisimulation ($\S$3.4).

• Although contextually equivalent, $\lambda_{\text{REF}}$ and HCC$^+$ differentiate in the order of combination of runtime checks. As a result, HCC is space efficient whereas $\lambda_{\text{REF}}$ is not: we can write programs in $\lambda_{\text{REF}}$ that may accumulate an unbounded number of checks. We formalize the changes needed in the dynamic semantics of $\lambda_{\text{REF}}$ to achieve space efficiency ($\S$3.4.3).

• We formally describe how to support other gradual reference semantics in $\lambda_{\text{REF}}$ by presenting $\lambda_{\text{REF}}^{\text{pm}}$, an extension that additionally supports both permissive and monotonic references ($\S$3.5).

Note that the technique to recover space efficiency is in fact independent from mutable references, and can therefore be used to recover space efficiency in other gradual languages derived with AGT.

Complete definitions and proofs of the main results can be found in $\S$A. Additionally, we implemented $\lambda_{\text{REF}}$ as an interactive prototype that exhibits both typing derivations and reduction traces. All the examples mentioned in this chapter are readily available in the online prototype available at https://pleiad.cl/grefs.

In the next sections we proceed as follows. First we present $\lambda_{\text{REF}}$, a standard statically-typed language with references ($\S$3.2). Second, we systematically apply AGT to $\lambda_{\text{REF}}$ ($\S$3.3) and observe the resulting semantics, which we called $\lambda_{\text{REF}}$ ($\S$3.1). We observe that $\lambda_{\text{REF}}$ manifests the guarded references semantics of HCC. Third, we formalize this observation by relating $\lambda_{\text{REF}}$ with HCC ($\S$3.4). We present an extension of HCC, called HCC$^+$, and a type-driven translation from $\lambda_{\text{REF}}$ to HCC$^+$. We prove that a $\lambda_{\text{REF}}$ term and its translation to HCC$^+$ are contextually equivalent. Fourth, we show that, contrary to HCC$^+$, $\lambda_{\text{REF}}$ is not space-efficient. We then present the changes needed in the dynamic semantics to recover space efficiency ($\S$3.4). Finally, we present $\lambda_{\text{REF}}^{\text{pm}}$, an extension of $\lambda_{\text{REF}}$ to support other semantics both permissive and monotonic references ($\S$3.5).
3.2 Preliminary: The Static Language \( \lambda^{\text{REF}} \)

We now apply AGT to a simple language with references, called \( \lambda^{\text{REF}} \), whose static and dynamic semantics are defined in Figures 3.1 and 3.2 respectively.

**Static semantics** The definition of \( \lambda^{\text{REF}} \) is standard. We use the metavariable \( l \) to range over a countably infinite set \( \text{Loc} \) of locations. A store typing \( \Sigma \) is a partial function from locations to types. A term \( t \) can be a lambda abstraction, a constant \( b \), a variable, an application, a binary operation on constants \( \oplus \), a conditional expression, a type ascription, a reference, a dereference, an assignment, or a location. Types may be base types (we use \( B \) to abstract over all base types), functions, and references. \( \text{Ref } T \) represents a reference to a value of type \( T \).

To prepare for the application of AGT, the presentation of the type system follows the convention of Garcia et al. \[44\], in which the type of each sub-expression is kept opaque, the type relations are made explicit as side conditions, and (partial) type functions are used explicitly instead of relying on matching metavariables. In particular, the \( \text{dom} \) (resp. \( \text{cod} \)) partial function is used to obtain the domain (resp. codomain) of a function type; it is undefined otherwise. We similarly introduce the \( \text{tref} \) partial function to extract the underlying type of a reference type. Save for the use of \( \text{tref} \), rules \( \text{Tref}, \text{Tderef}, \text{Tasgn} \) and \( \text{TI} \) are all standard \[92\]. We use the \( \theta \) metafunction to determine the type of constants (e.g. \( \theta(\text{true}) = \text{Bool}, \theta(1) = \text{Int} \)).

**Dynamic semantics** The dynamic semantics of \( \lambda^{\text{REF}} \) are presented in Figure 3.2 and are standard as well. The semantics are straightforward using evaluation contexts to reduce terms. A store \( \mu \) maps locations \( o \) to values \( v \). Here \( \mu[o \mapsto v] \) stands for a new store in which the location \( o \) is mapped to the value \( v \). The domain of a store \( \mu \), written \( \text{dom}(\mu) \), is the set of locations for which the finite map is defined. The expression \( \text{ref } t \) is evaluated by reducing the term \( t \) to a value \( v \), obtaining a fresh location in memory and storing the value at that location. The result of \( \text{ref } v \) is the newly created location. A dereference expression \( !t \) first evaluates the term \( t \) to a location \( o \), then returns the value stored in memory at location \( o \). An assignment \( t_1 := t_2 \) evaluates term \( t_1 \) to a location \( o \) and evaluates term \( t_2 \) to a value \( v \). The expression \( o := v \) updates the store at location \( o \) with the new value \( v \), and returns unit.

**Properties** Type safety of \( \lambda^{\text{REF}} \) is established as usual: a well-typed closed term is either a value or it can take a step (along a well typed store) to a term of the same type (and a well-typed store that extends the original one).

**Proposition 1** (Type safety). Let \( \phi; \Sigma \vdash t : T \). Then one of the following is true:

1. \( t \) is a value \( v \);
2. if \( \Sigma \vdash \mu \) then \( t \mid \mu \mapsto t' \mid \mu' \), where \( \phi; \Sigma' \vdash t' : T \) and \( \Sigma' \vdash \mu' \) some \( \Sigma' \supseteq \Sigma \).

27
\[ T \in \text{TYPE}, \quad B \in \text{BASE\!TYPE}, \quad x \in \text{VAR}, \quad o \in \text{LOC}, \quad b, \in \text{CONST}, \]
\[ t \in \text{TERM}, \quad \oplus \in \text{OPERATOR}, \quad \Gamma \in \text{VAR}^{\text{fin}} \quad \text{TYPE}, \quad \Sigma \in \text{LOC}^{\text{fin}} \quad \text{TYPE} \]
\[ T ::= B \mid T \to T \mid \text{Ref} \ T \quad \text{(types)} \]
\[ t ::= v \mid x \mid t \mid t \oplus t \mid \text{if} \ t \ \text{then} \ t \ \text{else} \ t \mid t :: T \mid \text{ref} \ t \mid \text{!} t \mid t := t \quad \text{(terms)} \]

\[
\begin{align*}
(Tx) & \quad \frac{x : T \in \Gamma}{\Gamma; \Sigma \vdash s \ x : T} & \quad (Tc) & \quad \frac{\theta(b) = B}{\Gamma; \Sigma \vdash s \ b : B} \\
(Tapp) & \quad \frac{\Gamma; \Sigma \vdash s \ t_1 : T_1 \quad \Gamma; \Sigma \vdash s \ t_2 : T_2 \quad T_2 = \text{dom}(T_1)}{\Gamma; \Sigma \vdash s \ t_1 \ t_2 : \text{cod}(T_1)} & \quad (Ty) & \quad \frac{\text{ty}(\oplus) = B_1 \times B_2 \to B_3 \quad T_1 = B_1 \quad T_2 = B_2}{\Gamma; \Sigma \vdash s \ t_1 \oplus t_2 : B_3} \\
(Tif) & \quad \frac{\Gamma; \Sigma \vdash s \ t_1 : T_1 \quad \Gamma; \Sigma \vdash s \ t_2 : T_2 \quad T_1 = \text{Bool}}{\Gamma; \Sigma \vdash s \ \text{if} \ t_1 \ \text{then} \ t_2 \ \text{else} \ t_3 : \text{equate}(T_2, T_3)} & \quad (T\lambda) & \quad \frac{\Gamma, x : T_1; \Sigma \vdash s \ t : T_2}{\Gamma; \Sigma \vdash s \ (\lambda x : T_1. t) : T_1 \to T_2} \\
(T::) & \quad \frac{\Gamma; \Sigma \vdash s \ t : T' \quad T' = T}{\Gamma; \Sigma \vdash s \ t :: T : T} & \quad (Tref) & \quad \frac{\Gamma; \Sigma \vdash s \ t : T}{\Gamma; \Sigma \vdash s \ \text{ref} \ t : \text{Ref} \ T} \\
(Tderef) & \quad \frac{\Gamma; \Sigma \vdash s \ t : T}{\Gamma; \Sigma \vdash s \ !t : \text{tref}(T)} & \quad (Tasgn) & \quad \frac{\Gamma; \Sigma \vdash s \ t_1 : T_1 \quad \Gamma; \Sigma \vdash s \ t_2 : T_2 \quad T_2 = \text{tref}(T_1)}{\Gamma; \Sigma \vdash s \ t_1 := t_2 : \text{Unit}} \\
(To) & \quad \frac{o : T \in \Sigma}{\Gamma; \Sigma \vdash s \ o : \text{Ref} \ T} \\
\end{align*}
\]

\[ T = T \]

\[
\begin{array}{cccc}
B = B & T_1 \to T_2 = T'_1 \to T'_2 & T_1 = T_2 & \text{Ref} \ T_1 = \text{Ref} \ T_2
\end{array}
\]

\[
\begin{align*}
\text{dom} : \text{TYPE} \to \text{TYPE} & \quad \text{cod} : \text{TYPE} \to \text{TYPE} \\
\text{dom}(T_1 \to T_2) = T_1 & \quad \text{cod}(T_1 \to T_2) = T_2 \\
\text{dom}(T) \ \text{undef. otherwise} & \quad \text{cod}(T) \ \text{undef. otherwise} \\
\text{tref} : \text{TYPE} \to \text{TYPE} & \quad \text{equate} : \text{TYPE} \to \text{TYPE} \\
\text{tref}(\text{Ref} \ T) = T & \quad \text{equate}(T, T) = T \\
\text{tref}(T) \ \text{undef. otherwise} & \quad \text{equate}(T_1, T_2) \ \text{undef. otherwise}
\end{align*}
\]

Figure 3.1: \(\lambda^{\text{REF}}\): Syntax and Type System

**Proof.** The proof is standard and follows from progress and preservation \[92]. \[\square\]
\[ v ::= c \mid \lambda x.t \mid o \quad \text{(values)} \]
\[ E ::= E \mid E \oplus t \mid v \oplus E \mid E t \mid v E \mid \text{if } E \text{ then } t \text{ else } t \mid \text{ref } E \mid !E \mid E := t \mid v := E \mid E :: T \quad \text{(Contexts)} \]
\[ \mu ::= \mu, o \mapsto v \quad \text{(store)} \]

\[ t, \mu \rightarrow_{\tau_s} t, \mu \quad \text{Notions of Reduction} \]
\[ b_1 \oplus b_2 \mid \mu \rightarrow_{s} b_3 \mid \mu \quad \text{where } c_3 = b_1[\oplus]b_2 \]
\[ (\lambda x : T.t) v \mid \mu \rightarrow_{s} ([v/x]t) \mid \mu \]
\[ \text{if } b \text{ then } t_1 \text{ else } t_2 \mid \mu \rightarrow_{s} \begin{cases} t_1 \mid \mu & \text{if } b = \text{true} \\ t_2 \mid \mu & \text{if } b = \text{false} \end{cases} \]
\[ \text{ref } v \mid \mu \rightarrow_{s} o \mid \mu[o \mapsto v] \text{ where } o \not\in \text{dom} (\mu) \]
\[ !o \mid \mu \rightarrow_{s} v \mid \mu \text{ where } v = \mu(o) \]
\[ o := v \mid \mu \rightarrow_{s} \text{unit} \mid \mu[o \mapsto v] \quad v :: T \mid \mu \rightarrow_{s} v \mid \mu \]

\[ t, \mu \rightarrow_{s} t, \mu' \quad \text{Reduction} \]
\[ \frac{t_1 \mid \mu \rightarrow_{s} t_2 \mid \mu'}{E[t_1] \mid \mu \rightarrow_{s} E[t_2] \mid \mu'} \]

Figure 3.2: $\lambda^{\text{REF}}$: Dynamic Semantics

### 3.3 Gradualizing $\lambda^{\text{REF}}$

Once we have defined the static language $\lambda^{\text{REF}}$, the AGT methodology drives the derivation of its gradual counterpart, $\lambda^{\text{REF}}$, following three steps:

1. define the syntax of gradual types and give them meaning by concretization to sets of static types; consequently obtain the most precise abstraction, establishing a Galois connection.

2. derive the gradual type system by using lifted type predicates and type functions in the typing rules.

3. derive the runtime semantics of the gradual language by proof normalization of gradual typing derivations.

#### 3.3.1 Syntax and Meaning of Gradual Types

We start by defining the syntax of gradual types. We decide to allow references to gradual types:

\[ G \in \text{GTYPE} \]
\[ G ::= B \mid G \rightarrow G \mid \text{Ref } G \mid ? \quad \text{(gradual types)} \]

Terms $t$ are lifted to gradual terms $t \in \text{GTERM}$, i.e. terms with gradual type annotations.

We then give meaning to gradual types via a concretization function $\gamma$ from gradual types
to non-empty sets of static types. We start from the concretization function for GTFL given by Garcia et al. [44], adding an extra case to deal with reference types. This is the natural lifting of concretization to the reference type constructor: \texttt{Ref} \texttt{G} denotes the set of reference types \texttt{Ref} \texttt{T} for each \texttt{T} in the concretization of \texttt{G}:

**Definition 1** (Concretization). Let \( \gamma : \text{GType} \to \mathcal{P}^*(\text{Type}) \) be defined as follows:

\[
\gamma(B) = \{ B \}
\]
\[
\gamma(G_1 \to G_2) = \{ T_1 \to T_2 \mid T_1 \in \gamma(G_1) \land T_2 \in \gamma(G_2) \}
\]
\[
\gamma(\text{Ref } G) = \{ \text{Ref } T \mid T \in \gamma(G) \}
\]
\[
\gamma(?) = \text{Type}
\]

The notion of type precision between gradual types coincides with set inclusion of their concretizations:

**Definition 2** (Type Precision). \( G_1 \sqsubseteq G_2 \) if and only if \( \gamma(G_1) \subseteq \gamma(G_2) \).

**Proposition 2** (Precision, inductively). The following inductive definition of type precision is equivalent to Definition 2.

\[
\begin{align*}
B \sqsubseteq B & \quad \text{G}_1 \sqsubseteq \text{G}_1' \quad \text{G}_2 \sqsubseteq \text{G}_2' \quad \text{G}_1 \to \text{G}_2 \sqsubseteq \text{G}_1' \to \text{G}_2' \quad \text{Ref } \text{G}_1 \sqsubseteq \text{Ref } \text{G}_2 \quad \text{G} \sqsubseteq ?
\end{align*}
\]

Once \( \gamma \) is defined, we proceed to define its corresponding abstraction function:

**Definition 3** (Abstraction). Let the abstraction function \( \alpha : \mathcal{P}^*(\text{Type}) \to \text{GType} \) be defined as follows:

\[
\alpha(\{ B \}) = B
\]
\[
\alpha(\{ T_{i_1} \to T_{i_2} \}) = \alpha(\{ T_{i_1} \}) \to \alpha(\{ T_{i_2} \})
\]
\[
\alpha(\{ \text{Ref } T_i \}) = \text{Ref } \alpha(\{ T_i \})
\]
\[
\alpha(\overline{T}) = ? \text{ otherwise}
\]

The abstraction function preserves type constructors and falls back on the unknown type whenever a heterogeneous set is abstracted. As expected, abstraction preserves the \texttt{Ref} type constructor when all static types in the set are reference types. This abstraction function is both sound and optimal: it produces the most precise gradual type that over-approximates a given set of static types.

**Proposition 3** (Galois connection). \( \langle \gamma, \alpha \rangle \) is a Galois connection, i.e.:

a) (Soundness) for any non-empty set of static types \( S = \{ \overline{T} \} \), we have \( S \subseteq \gamma(\alpha(S)) \)

b) (Optimality) for any gradual type \( G \), we have \( \alpha(\gamma(G)) \subseteq G \).

Soundness (a) means that \( \alpha \) always produces a gradual type whose concretization over-approximates the original set. Optimality (b) means that \( \alpha \) always yields the best (i.e. least) sound approximation that gradual types can represent.
3.3.2 Lifting the Type System

In order to obtain the static semantics of $\lambda^{\text{REF}}$, we lift type relations (here, equality) and type functions ($\text{dom}$, $\text{cod}$, $\text{tref}$, $\text{equate}$). Recall from §2.2 that, following AGT, this lifting is obtained by exploiting the Galois connection we have just established through existential lifting.

**Definition 4** (Consistency). $G_1 \sim G_2$ if and only if $T_1 = T_2$ for some $(T_1, T_2) \in \gamma(G_1) \times \gamma(G_2)$. Inductively:

\[
\begin{align*}
G & \sim ? & \quad ? & \sim G & \quad G & \sim G \\
G_1 \sim G_2 & \quad G_2 \sim G_1 & \quad G_1 \sim G_2 & \quad \text{Ref } G_1 \sim \text{Ref } G_2 \\
G_{11} \sim G_{21} \quad G_{12} \sim G_{22} & \quad G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22} \\
\end{align*}
\]

As a first result, the concretization function justifies consistency variance for reference types—as adopted by all gradual reference systems, except the invariant semantics of Siek and Taha \[109\].

Lifting type functions follows abstract interpretation as well. For example, consider a partial function $F : \text{Type} \times \text{Type} \rightarrow \text{Type}$. The lifting of $F$, called $\hat{F}$, is defined as $\hat{F}(G_1, G_2) = \alpha(\hat{F}(\gamma(G_1), \gamma(G_2)))$. Note that as $F$ is partial, the collecting application of $F$ may be the empty set, which is not part of the domain of $\alpha$; this situation captures the notion of type errors \[44\].

For instance, the lifting of the $\text{equate}$ operator presented in Figure 3.1 is defined as follows:

**Definition 5.** $\hat{\text{equate}}(G_1, G_2) = \alpha(\{ \text{equate}(T_1, T_2) \mid (T_1, T_2) \in \gamma(G_1) \times \gamma(G_2) \})$.

and it comes as no surprise that this definition coincides with the meet operator in the precision order \[44\]:

**Proposition 4.** $\hat{\text{equate}}(G_1, G_2) = G_1 \sqcap G_2$.

The meet operator is defined as $G_1 \sqcap G_2 = \alpha(\gamma(G_1) \cap \gamma(G_2))$, and inductively as:

\[
\begin{align*}
B \sqcap B & = B & \quad G_1 \sqcap G_2 & = G_2 \sqcap G_1 & \quad G \sqcap ? & = ? \sqcap G & = G \\
(G_{11} \rightarrow G_{12}) \cap (G_{21} \rightarrow G_{22}) & = (G_{11} \cap G_{21}) \rightarrow (G_{12} \cap G_{22}) & \quad \text{Ref } G_1 \sqcap \text{Ref } G_2 & = \text{Ref } G_1 \sqcap G_2 \\
\end{align*}
\]

$G_1 \sqcap G_2$ is undefined otherwise

**Compositional lifting** As previously noted by Garcia et al. \[44\] we cannot always apply compositionally lifting to predicates that use both type relations and type functions. However, we justify that we can do it for application and assignment rules.

**Proposition 5.** Let $P_1(T_1, T_2) \triangleq T_1 = \text{dom}(T_2)$. Then $\hat{P}_1(G_1, G_2) \iff G_1 \sim \hat{\text{dom}}(G_2)$.

**Proposition 6.** Let $P_2(T_1, T_2) \triangleq T_1 = \text{tref}(T_2)$. Then $\hat{P}_2(G_1, G_2) \iff G_1 \sim \hat{\text{tref}}(G_2)$.
The type system of \( \lambda \text{REF} \), along with algorithmic definitions of consistent functions, is presented in Figure 3.3. As expected, it justifies the fact that a term of the unknown type \( ? \) can be dealt with as if it were a reference type \( \text{Ref} ? \), since \( \text{tref}(?) = ? \).

### 3.3.3 Static Semantics

The type system of \( \lambda \text{REF} \) is presented in Figure 3.3 along algorithmic definitions of consistent functions; the type rules are obtained by replacing type predicates and functions with their corresponding liftings.
3.3.4 Dynamic Semantics

One of the salient features of the AGT methodology is that it provides a direct dynamic semantics for gradual programs [44], instead of the typical translational semantics through an intermediate cast calculus [109]. The key idea is to apply proof reduction on gradual typing derivations [63]; by the Curry-Howard correspondence, this gives a notion of reduction for gradual terms. We call such semantics the intrinsic semantics $\lambda_{e}^{\text{REF}}$.

Static Semantics of $\lambda_{e}^{\text{REF}}$

The main insight of AGT is that gradual typing derivations need to be augmented with evidence to support consistent judgments. Evidence reflects the justification of why a given consistent judgment holds. Therefore, the dynamic semantics mirrors the type preservation argument of the static language, combining evidences at each reduction step in order to determine whether the program can reduce or should halt with a runtime error.

Consider the gradual typing derivation of $(\lambda x : ? . x + 1) \text{false}$. In the inner typing derivation of the function, the consistent judgment $? \sim \text{Int}$ supports the addition expression, and at the top-level, the judgment $\text{Bool} \sim ?$ supports the application of the function to $\text{false}$. When two types are involved in a consistent judgment, we learn something about each of these types, namely the justification of why the judgment holds. This justification can be captured by a pair of gradual types, $\varepsilon = (G_1, G_2)$, which are at least as precise as the types involved in the judgment [44]. We use notation $\varepsilon \vdash G_1 \sim G_2$ to denote that evidence $\varepsilon$ justifies judgment $G_1 \sim G_2$. For instance, by knowing that $? \sim \text{Int}$ holds, we learn that the first type can only possibly be $\text{Int}$, while we do not learn anything new about the right-hand side, which is already fully static. Therefore the evidence of that judgment is $\varepsilon_1 = \langle \text{Int}, \text{Int} \rangle$, i.e. $\langle \text{Int}, \text{Int} \rangle \vdash ? \sim \text{Int}$. Similarly, the evidence for the second judgment is $\varepsilon_2 = \langle \text{Bool}, \text{Bool} \rangle$. Types in evidence can be gradual, e.g. $\langle ? \rightarrow ?, ? \rightarrow ? \rangle$ justifies that $\langle ? \rightarrow ? \rangle \sim ?$. In a language with consistency, both components of an evidence coincide, therefore for simplicity we use notation $\langle G \rangle$ instead of $\langle G, G \rangle$.

Evidence is initially computed by a partial function called an initial evidence operator $\mathcal{G}_{=}$. An initial evidence operator computes the most precise evidence that can be deduced from a given judgment. For instance the initial evidence of consistent judgment $G_1 \sim G_2$ is $\varepsilon = \mathcal{G}_{=}(G_1, G_2)$, i.e. $\mathcal{G}_{=}(G_1, G_2) \vdash G_1 \sim G_2$. Formally the interior function is defined as:

**Definition 6 (Interior).**

$$\mathcal{G}_{=}(G_1, G_2) = \alpha^2(\{\langle T_1, T_2 \rangle \mid T_1 \in \gamma(G_1), T_2 \in \gamma(G_2), T_1 = T_2\})$$

Given two set of static types that belong to the concretization of both gradual types, this functions abstract the set of pairs of static types such that both types are equal. In this setting with only consistency, the interior function coincides with the meet between the two types.

$$\alpha^2(\{\langle T_{1i}, T_{1j} \rangle \}) = \langle \alpha(\{T_{1i}\}), \alpha(\{T_{1j}\}) \rangle$$
Proposition 7. If \( G_1 \sim G_2 \), then \( \exists (G_1, G_2) = (G_1 \cap G_2) \).

At runtime, reduction rules need to combine evidences in order to either justify or refute a use of transitivity in the type preservation argument. In our example, after the application and right before the addition, we need to combine \( \varepsilon_1 \) and \( \varepsilon_2 \) in order to (try to) obtain a justification for the transitive judgment, namely that \( \text{Bool} \sim \text{Int} \). The combination operation, called consistent transitivity \( \circ \), determines whether two evidences support the transitivity: here, \( \varepsilon_2 \circ \varepsilon_1 = (\text{Bool}) \circ (\text{Int}) \) is undefined, so a runtime error is raised.

The definition of consistent transitivity for a type predicate \( P \), \( \circ^P \), is given by the abstract interpretation framework \([44]\); in particular, for type equality it is defined as follows

\[
\text{Definition 7 (Consistent transitivity). Suppose } \varepsilon_{ab} \vdash G_a \sim G_b \text{ and } \varepsilon_{bc} \vdash G_b \sim G_c. \text{ Evidence for consistent transitivity is deduced as } (\varepsilon_{ab} \circ^P \varepsilon_{bc}) \vdash G_a \sim G_c, \text{ where: }
\]

\[
\langle G_1, G_{21} \rangle \circ^P \langle G_{22}, G_3 \rangle = \alpha^2(\{(T_1, T_3) \in \gamma(G_1) \times \gamma(G_3) \mid \exists T_2 \in \gamma(G_{21}) \cap \gamma(G_{22}), T_1 = T_2 \wedge T_2 = T_3\})
\]

As \( G_1 = G_{21} \) and \( G_{22} = G_3 \), the definition of consistent transitivity corresponds to the meet of gradual types \( \cap \):

\[
\text{Lemma 8. } \langle G_1 \rangle \circ^P \langle G_2 \rangle = \langle G_1 \cap G_2 \rangle.
\]

To formalize this approach while avoiding writing down reduction rules on actual (bi-dimensional) derivation trees, Garcia et al. adopt intrinsic terms \([25]\), which are a flat notation that is isomorphic to typing derivations. Specifically, the typing derivation for the judgment \( \Gamma \vdash t : G \) is represented by an intrinsic term \( t^G \in T[G] \). The syntax of intrinsic terms is presented in Figure 3.4. Intrinsically-typed terms \( t^G \) comprise a family \( T[G] \) of type-indexed sets, such that ill-typed terms do not exist. They are built up from disjoint families \( \mathcal{V}[G] \) and \( \mathcal{L}[G] \) of intrinsically typed variables and locations respectively. Note that intrinsic terms do not need explicit type environment \( \Gamma \) or store environments \( \Sigma \). We omit the type exponent on intrinsic terms when not needed, writing for instance \( t \in T[G] \).

The syntax and type rules for intrinsic terms is presented in Figure 3.4. We use notation \( \text{et} \) to refer to an evidence term, which are terms augmented with evidence. This evidence justifies why the type of the term is consistent with the corresponding statically determined type. For instance, in term \( \langle \text{Int} \rangle 1 :: ?, \) evidence \( \langle \text{Int} \rangle \) is the companion of the raw value 1 and justifies that \( \text{Int} \sim ? \). Intrinsic values \( v \) can either be simple values \( u \) or ascribed values \( \varepsilon u :: G \). A simple value \( u \) can be a variable \( x \), a constant \( b \), a lambda abstraction \( \lambda x.t \), or a location \( o^G \). Some terms carry extra type annotations purely to help prove type safety, such as \( G \) in \( \text{et} :=^G \text{et} \), and to ensure unicity of typing during reduction such as \( G \) in \( \text{et} @^G \text{et} \). The type rules mirror the type rules of \( \lambda^\text{REF} \) where each consistent judgment is justified by some evidence. The presentation may differ sometimes: for instance in Rule (IGasgn), its extrinsic counterpart has premise \( \text{tref}(G_1) \sim G_2 \) which is equivalent to both \( G_1 \sim \text{Ref} G_3 \) and \( G_2 \sim G_3 \). We choose the later representation because it allows us to track evidence for each of the subterms. Something similar occurs in rules (IGderef) and (IGref): extrinsic rules (Gderef) and (Gref) has no consistent judgment whatsoever. This judgment is justified as subterms may evolve during reduction into something of a different (but consistent) type.

34
et ∈ EvTerm, ⟨G⟩ ∈ Evidence, u ∈ SimpleValue, v ∈ Value, t ∈ T[*],

u ::= x | b | λx.t | o^G
v ::= u | ⟨G⟩u :: G
et ::= ⟨G⟩t
t ::= v | et ⊕ et | et @^G et | et :: G | if et then et else et | ref^G et | !^G et | et :=^G et

\[\begin{align*}
\text{(IGb)} & \quad \theta(b) = B \quad b ∈ T[B] \\
\text{(IGx)} & \quad x^G ∈ T[G] \\
\text{(IG+)−} & \quad t^G_1 ∈ T[G_1], t^G_2 ∈ T[G_2] \quad (B_1) ⊢ G_1 \sim B_1, (B_2) ⊢ G_2 \sim B_2 \\
& \quad ty(⊕) = B_1 \times B_2 → B_3 \\
& \quad (B_1) t^G_1 ⊕ (B_2) t^G_2 ∈ T[B_3] \\
\text{(IGλ)} & \quad \lambda x^G. t^G_2 ∈ T[G_2] \quad \lambda x^G. t^G_2 ∈ T[G_2] \\
\text{(IGref)} & \quad t^G_1 ∈ T[G_1], \quad (G') ⊢ G_1 \sim G_2, \quad \text{ref}^G_2 \quad \langle G' \rangle t^G_1 ∈ T[\text{Ref} \ G_2] \\
\text{(IGasgn)} & \quad t^G_1 ∈ T[G_1], \quad (G'_1) ⊢ G_1 \sim \text{Ref} \ G_3 \\
& \quad t^G_2 ∈ T[G_2], \quad (G'_2) ⊢ G_2 \sim G_3 \\
& \quad \langle G'_1 \rangle t^G_1 :=^G_3 \quad \langle G'_2 \rangle t^G_2 ∈ T[\text{Unit}] \\
\text{(IGl)} & \quad o^G ∈ T[\text{Ref} \ G]
\end{align*}\]

For instance, in rule (IGderef), evidence \(\langle \text{Ref} \ G_1 \rangle\) justifies that the type of subterm \(t^{G_2}\) is consistent with \(\text{Ref} \ G\), the type of the subterm during type checking. Alternatively, the elaboration of (IGderef) may also be seen as the lifting of the following equivalent rule:

\[\begin{align*}
& \frac{\Gamma; \Sigma ⊢ t : G_1 \quad G_1 \sim \text{Ref} \ G_2}{\Gamma; \Sigma ⊢ \!^G_1 t : G_2}
\end{align*}\]

where statically \(G_1 = \text{Ref} \ G_2\). The elaboration rules for intrinsic terms, i.e. from \(\lambda^{\text{REF}}\) to \(\lambda^{G^{\text{REF}}}_e\), is explained later in § 3.3.5.

**Reduction** Now we turn to the reduction rules of intrinsic terms, possibly failing with an error when combining evidences using consistent transitivity defined above.

All the combinations of evidence that we use in the reduction rules directly correspond to necessary combinations dictated by the static language and its static safety proof. To illustrate this, we now analyze a case of the \(\lambda^{\text{REF}}\) type safety proof and describe how to lift the reduction to \(\lambda_g^{\text{REF}}\). Consider the application case of \(\lambda^{\text{REF}}\)'s preservation proof, which in
essence reduces a type derivation \( D \) to a new one and updates the store \( \mu \).

\[
D = \frac{(T\lambda) \quad D_1 \quad (T\text{app}) \quad D_2}{\Gamma, x : T_1; \Sigma \vdash t : T_2 \quad \Gamma \vdash \lambda x : T_1.t : T_1 \rightarrow T_2}{\Gamma \vdash \text{cod}(T_1 \rightarrow T_2) \quad \Gamma \vdash \text{cod}(T_1 \rightarrow T_2) \quad \Gamma \vdash \text{cod}(T_1 \rightarrow T_2)}
\]

The relevant reduction rule (Fig 3.1) follows:

\[
(\lambda x : T_1.t) v \mid \mu \rightarrow_s ([v/x]t) \mid \mu
\]

As we are substituting \( x \) by \( v \), first we have to justify that \( v \) has the same type of \( x \), i.e. \( \Gamma; \Sigma \vdash v : T_1 \), but this is direct by combining \( \Gamma; \Sigma \vdash s t : T_2 \) and \( T_1' = \text{dom}(T_1 \rightarrow T_2) = T_1 \). Finally as \( \frac{D_1}{\Gamma, x : T_1; \Sigma \vdash s [v/x]t : T_2} \), then by substitution lemma we can conclude that

\[
\Gamma, x : T_1; \Sigma \vdash s [v/x]t : T_2.
\]

In the gradual setting the story is similar:

\[
D = \frac{(G\lambda) \quad D_1 \quad (G\text{app}) \quad D_2}{t \in T[G_{12}] \quad (\lambda x^{G_{11}}.t) \in T[G_{11} \rightarrow G_{12}] \quad u \in T[G'_1] \quad \varepsilon_1 \vdash G_{11} \rightarrow G_{12} \sim G_1 \rightarrow G_2 \quad \varepsilon_2 \vdash G'_1 \sim \text{dom}(G_1 \rightarrow G_2) \quad \varepsilon_1(\lambda x^{G_{11}}.t) \@^{G_1 \rightarrow G_2} \varepsilon_2 u \in T[\text{cod}(G_1 \rightarrow G_2)]}{\varepsilon_1(\lambda x^{G_{11}}.t) \vdash \text{cod}(G_1 \rightarrow G_2)}
\]

As explained before, we keep track of the function type computed during type checking \( (G_1 \rightarrow G_2) \), because during reduction the type of the function term may evolve into something consistent \( (G_{11} \rightarrow G_{12}, \text{justified by evidence } \varepsilon_1) \). Following \( \lambda^{\text{REF}} \)'s type safety proof, we have to substitute variable \( x \) by argument \( u \). So now we have to justify that the type of \( u \) \( (G'_1) \) is consistent with the type of variable \( x^{G_{11}} \) \( (G_{11}) \). By the inversion lemma on evidences we know that \( \text{idom}(\varepsilon_1) \vdash G_{11} \sim G_{11} \) and \( \text{icod}(\varepsilon_1) \vdash G_{11} \sim G_{12} \). Using consistent transitivity between \( \varepsilon_2 \) and \( \text{idom}(\varepsilon_1) \) (if defined) we can justify judgment \( \varepsilon_2 \circ \text{idom}(\varepsilon_1) \vdash G'_1 \sim G_{11} \). If consistent transitivity is defined, then we can now use ascriptions to construct a new value of type \( G_{11} \) and reduce the application substituting \( x^{G_{11}} \) as follows: \( t' = \left[ \left( \left( \varepsilon_2 \circ \text{idom}(\varepsilon_1) \right) u :: G_{11} \right) / x^{G_{11}} \right] t \).

But now this new expression has a problem: \( t' \) has type \( G_{12} \) and following \( \lambda^{\text{REF}} \)'s type safety proof, we have to prove that the resulting term has type \( \text{cod}(G_1 \rightarrow G_2) = G_2 \). We can address this by ascribing \( t' \) to \( G_2 \) justifying the judgment \( G_{12} \sim G_2 \) with evidence \( \text{icod}(\varepsilon_1) \). The final resulting term is presented next:

\[
\text{icod}(\varepsilon_1)[\left( \left( \varepsilon_2 \circ \text{idom}(\varepsilon_1) \right) u :: G_{11} \right) / x^{G_{11}}]t :: G_2
\]

Let us illustrate how this process scales to auxiliary structures like the heap. Consider now the assignment case of \( \lambda^{\text{REF}} \)'s preservation proof.

\[
D = \frac{(\text{Tasgn}) \quad o : T_1 \in \Sigma \quad D_1 \quad D_2 \quad T_2 = \text{tref}(\text{Ref} T_1)}{\Gamma; \Sigma \vdash o : \text{Ref} T_1 \quad \Gamma; \Sigma \vdash v : T_2}{\Gamma; \Sigma \vdash o := v : \text{Unit}}
\]
The relevant reduction rule (Fig 3.1) follows:

\[ o := v \mid \mu \rightarrow \text{unit} \mid \mu[o \mapsto v] \]

The fact that \( \mathcal{D} \) reduces to \( \Gamma; \Sigma \vdash_\nu \text{unit} : \text{Unit} \) is immediate, but we must also prove that the stored value \( v \) respect the store type, i.e., \( T_2 = T_1 \). But as \( tref(\text{Ref} \ T_1) = T_1 \) the result follows immediately from premise \( T_2 = tref(\text{Ref} \ T_1) \).

In \( \lambda^\text{REF}_\varepsilon \) the last step of reduction of an assignment has the form

\[ \varepsilon^{G_1} \in T[\text{Ref} \ G_1] \]  
\[ u \in T[G_2] \]  
\[ \varepsilon_1 \vdash \text{Ref} \ G_1 \sim \text{Ref} \ G_3 \]  
\[ \varepsilon_2 \vdash G_2 \sim G_3 \]  
\[ (\text{IGasgn}) \quad \frac{}{\varepsilon_1 o^{G_1} := G_3 \varepsilon_2 u \in T[\text{Unit}]} \]

Following \( \lambda^\text{REF}_\varepsilon \)'s type safety proof, we have to reduce this term to \text{unit} of type Unit (which is immediate), and update the store at location \( o^{G_1} \) with value \( u \). As \( o^{G_1} \) stores values of type \( G_1 \), we cannot just update the store to \( \mu[o^{G_1} \mapsto u] \) because \( u \) has type \( G_2 \). We need to ascribe \( u \) to \( G_1 \), but first we have to build evidence to justify that the type of \( u \) is consistent with \( G_1 \), i.e., \( G_2 \sim G_1 \). We know that \( \varepsilon_2 \vdash G_2 \sim G_3 \) and by inversion lemma \( i\text{ref}(\varepsilon_1) \vdash G_1 \sim G_3 \). In this setting evidence is symmetric, so we also know that \( i\text{ref}(\varepsilon_1) \vdash G_3 \sim G_1 \).

Using consistent transitivity between \( \varepsilon_2 \) and \( i\text{ref}(\varepsilon_1) \), we can justify (if defined) \( G_2 \sim G_1 \), i.e., \( (\varepsilon_2 \circ i\text{ref}(\varepsilon_1))u :: G_1 \).

Finally, the reduction rules for assignment is defined as follows

\[ \varepsilon_1 o^{G_1} := G_3 \varepsilon_2 u \mid \mu \quad \rightarrow \quad \text{unit} \mid \mu[o^{G_1} \mapsto (\varepsilon_2 \circ i\text{ref}(\varepsilon_1))u :: G_1] \]

The reduction rules are presented in Figure 3.5. They are defined over configurations \( \text{CONFIG}_G \) which consist of a pair of a term and a store. Contrary to [14], instead of using evaluation frames, we define the reduction semantics by using an equivalent representation using evaluation contexts [38].

Rules (r1), (r2), and (r3), present no novelty with respect to the original presentation of Garcia et al. [14].

Rule (r4) reduces a reference to a new location \( o^G \) not already present in the domain of store \( \mu \). The store is extended mapping \( o^G \) to the evidence value ascribed to \( G_2 \), the type of the reference determined statically.

Rule (r5) reduces a dereference to the underlying value \( v \) of location \( o^{G_2} \), ascribed to the statically determined type \( G \), where evidence \( \langle G_1 \rangle \) justifies that \( G_2 \) (the type of \( v \)), is consistent with \( G \).

Rule (r6) updates the corresponding value on the heap of location \( o^{G_1} \), with raw value \( u \) ascribed to \( G \). As \( \langle G_2 \rangle \) justifies that the type of \( u \) is consistent with \( G_3 \), and by inversion lemmas, \( \langle G_1 \rangle \) justifies that \( G_3 \sim G \), then evidence \( \langle G_2 \rangle \circ \sim \langle G_1 \rangle = \langle G_2 \cap G_1 \rangle \) (if defined) justifies that the type of \( u \) is consistent with \( G \). If \( G_2 \cap G_1 \) is not defined then a runtime error is signaled.

\[ ^3 \text{This reasoning also scales to subtyping as references are invariant} \]
\(ev \in \text{EvValue}, \ t \in \mathbb{T} [e], \ F \in \text{EvCtx}, \ E \in \text{TmFrame}\)

\[
\begin{align*}
ev & ::= \langle G \rangle u \\
F & ::= \square \mid E \oplus E \mid ev \oplus E \mid E \oplus G \mid ev \oplus G \mid E : G \\
& \quad \text{if } E \text{ then } et \text{ else } et \mid ref^G E \mid t^G E \mid E := G \mid ev := G E \\
E & ::= F \mid \langle G \rangle F \\
\mu & ::= \mu, o^G \rightarrow v
\end{align*}
\]

**Notions of Reduction**

\[\text{Config}_G = \mathbb{T}[G] \times \text{Store}\]

\[\rightarrow: \text{Config}_G \times (\text{Config}_G \cup \{\text{error}\})\]

\[\rightarrow_c: \text{EvTerm} \times (\text{EvTerm} \cup \{\text{error}\})\]

\[\begin{align*}
(r1) & \quad \langle B_1 \rangle b_1 \oplus \langle B_2 \rangle b_2 | \mu \rightarrow b_3 | \mu \quad \text{where } b_3 = b_1 [\oplus] b_2 \\
(r2) & \quad (G_{11} \rightarrow G_{12}) (\lambda x^{G_{11}}. t)@G_{1} \rightarrow G_{2} \ (\langle G_{2}' \rangle u) | \mu \rightarrow \begin{cases} 
\langle G_{12}' \rangle (\langle [G_{11} \cap G_{11}'] u : G_{11} \rangle / x^{G_{11}} t) : G_{2} | \mu \\
\text{error} & \text{if } G_{2} \cap G_{11} \text{ is not defined}
\end{cases} \\
(r3) & \quad \text{if } \langle \text{Bool} \rangle b \text{ then } \langle G \rangle t^{G_2} \text{ else } \langle G \rangle t^{G_3} | \mu \rightarrow \begin{cases} 
\langle G \rangle t^{G_2} : G | \mu \quad b = \text{true} \\
\langle G \rangle t^{G_3} : G | \mu \quad b = \text{false}
\end{cases} \quad \text{where } G = G_2 \cap G_3 \\
(r4) & \quad \text{ref}^{G_2} (\langle G_1 \rangle u | \mu \rightarrow o^{G_2} | \mu[o^{G_2} \rightarrow \langle G_1 \rangle u : G_2] \quad \text{where } o \notin \text{dom}(\mu) \\
(r5) & \quad !^{G} (\text{Ref} \ G_1) o^{G_2} | \mu \rightarrow \langle G_1 \rangle v : G | \mu \quad \text{where } v = \mu(o^{G_2}) \\
(r6) & \quad \text{Ref} \ G_1 o^{G} := G_3 (\langle G_2 \rangle u | \mu \rightarrow \begin{cases} 
\text{unit} | \mu[o^{G} \rightarrow \langle G_2 \cap G_1 \rangle u : G] \\
\text{error} & \text{if } G_2 \cap G_1 \text{ is not defined}
\end{cases} \\
(r7) & \quad \langle G_2 \rangle (\langle G_1 \rangle u : G) \rightarrow_c \begin{cases} 
\langle G_1 \cap G_2 \rangle u \\
\text{error} & \text{if } G_1 \cap G_2 \text{ is not defined}
\end{cases}
\end{align*}\]

**Figure 3.5:** \(\lambda^{\text{REF}}_{\varepsilon} \): Dynamic semantics

### 3.3.5 Elaboration of \(\lambda^{\text{REF}}_{\varepsilon} \) terms

So far we have presented intrinsic terms without formally explaining how to derive them. Figure 3.6 present the type-driven elaboration rules from \(\lambda^{\text{REF}} \) to \(\lambda^{\text{REF}}_{\varepsilon} \). Judgment \(\Gamma; \Sigma \vdash t \rightarrow_{\varepsilon} t^{G} : G \uparrow\) denotes the elaboration of term \(t^{G} \) from term \(t \), where \(t \) is typed \(G \) under environments \(\Gamma \) and \(\Sigma \). Basically each consistent type judgment is replaced by the initial evidence operator \(G_{\varepsilon} \).

\(^{4}\)We use the \(\varepsilon \) subindex to differentiate different translations presented in this chapter.
The same is applied in rules (TR): elaboration rules follow this same recipe.

Rule (TR) recurses on the subterm $t$, and the consistent subtyping judgment $G' \sim G$ from $(G ::)$ is replaced with $\varepsilon = \varepsilon(G', G)$, which computes evidence $\varepsilon$ for consistent subtyping. This evidence is eventually placed next to the translated term $\varepsilon t^{G'}$. Most of the elaboration rules follow this same recipe. Rule (TRapp), uses metafunctions $\overline{\text{dom}}$ and $\overline{\text{cod}}$ to avoid writing three different elaboration rules, e.g. when $t_1$ is typed $? \Rightarrow ?$ then $\varepsilon_1 = \varepsilon_2 = \varepsilon_3 = ?(?, ? \Rightarrow ?)$. The same is applied in rules (TRref) and (TRasgn) where we use $\overline{\text{tref}}$ instead.

Note that the elaboration rules only enrich derivations with evidence (by using the interior function), and such resulting derivations are represented as intrinsic terms. Then by construction, the elaboration rules trivially preserve typing:

**Proposition 9** (Elaboration preserves typing). If $\Gamma; \Sigma \vdash t : G$ and $\Gamma; \Sigma \vdash t \sim_\varepsilon t^G : G$, then $t^G \in T[G]$.
3.3.6 Properties

In order to establish type safety we first have to define well-typedness of the store \( \mu \). Well-typedness of the store is usually defined with respect to a store environment, i.e. \( \Sigma \vdash \mu \). Here, as we can see in Figure 3.4, intrinsically-typed locations \( o^G \in T[\text{Ref } G] \) obviate the need for store environment \( \Sigma \): the store environment of a term \( t \) is simply the set of intrinsically-typed free locations of the term, \( \text{freeLocs}(t) \). Therefore, contrary to standard reference type systems, well-typedness of the store is defined with respect to an intrinsic term:

**Definition 8** (\( \mu \) is well typed). A store \( \mu \) is said to be well typed with respect to an intrinsic term \( t^G \), written \( t^G \vdash \mu \), if

1. \( \text{freeLocs}(t^G) \subseteq \text{dom}(\mu) \), and
2. \( \forall v \in \text{cod}(\mu), v \vdash \mu \), and
3. \( \forall o^G \in \text{dom}(\mu), \mu(o^G) \in T[G] \).

A store \( \mu \) is well typed if all the free locations of a term and all the free locations of values in the store, are part of the domain of the store. Also for each of the intrinsic locations \( o^G \in T[G] \) that are part of the domain of the store, then all the underlying values \( v \in T[G] \).

Now we can establish type safety: closed terms do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

**Proposition 10** (Type safety). Let \( t^G \) a closed intrinsic term. If \( t^G \in T[G] \) then one of the following is true:

1. \( t^G \) is a value \( v \);
2. if \( t^G \vdash \mu \) then \( t^G \mid \mu \mapsto t'^G \mid \mu' \) for some term \( t'^G \in T[G] \) and some \( \mu' \) such that \( t'^G \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \);
3. \( t^G \mid \mu \mapsto \text{error} \).

Also, the gradual type system is a conservative extension of the static type system; i.e. both systems coincide on fully-annotated terms.

**Proposition 11** (Equivalence for fully-annotated terms (statics)). For any \( t \in \text{TERM}, . \vdash_s t : T \) if and only if . \( \vdash t : T \)

The equivalence of the dynamic semantics for fully-annotated terms is more subtle. We cannot rely on a syntactic comparison of values because during reduction \( \lambda^\text{REF} \) inserts (possibly redundant) ascriptions. For instance, \( (\lambda x : \text{int}.1) \) is syntactically different, but equivalent to \( (\lambda x^{\text{int}}.\langle \text{int} \rangle 1 :: \text{int}) \). To capture this relation, we formally connect both languages using logical relations between pair of terms and stores.

We use notation \( \langle t, \mu \rangle \approx \langle t', \mu' \rangle : T \) to denote that the pair of term \( t \) and store \( \mu \) is related to the pair of term \( t' \) and store \( \mu' \) at type \( T \). Two tuples of values and stores are related
values at type $T$, if first, both stores are related. Two stores are related if for all locations that are common to both stores, the stored values are related. Second, if $T$ is a constant $B$ or a reference $\text{Ref} \ T$, then both values must be equal. Third, if $T$ is a function, then both functions applied to related arguments yield two related computations. Two configurations (i.e. term-store pairs) are related computations if both configurations reduce to related values and stores. The complete definition and proofs are presented in A.3.1.

**Proposition 12** (Equivalence for fully-annotated terms (dynamics)). For any $t \in \text{TERM}$, $\vdash_s t : T$, $\vdash t \rightsquigarrow t^T : T$, then $t \mid \cdot \mapsto^* v \mid \mu \iff t^T \mid \cdot \mapsto^* v' \mid \mu'$, for some $\mu, \mu'$ such that $\langle v, \mu \rangle \approx \langle v', \mu' \rangle : T$.

Precision on terms, noted $t_1 \sqsubseteq t_2$, is the natural lifting of type precision to terms. The gradual type system satisfies the static gradual guarantee of Siek et al. [113], i.e. losing precision preserves typeability: if a program is well-typed, then a less precise version of it also type checks, at a less precise type.

**Proposition 13** (Static gradual guarantee). If $\vdash t_1 : G_1$ and $t_1 \sqsubseteq t_2$, then $\vdash t_2 : G_2$, for some $G_2$ such that $G_1 \sqsubseteq G_2$.

We also prove that $\overline{\lambda}^{\text{REF}}$ satisfies the dynamic component of the gradual guarantee: “any program that runs without error would continue to do so if it were given less precise types”. For this we must also extend the notion of precision over stores: intuitively a store is more precise than another store if its labels and values are more precise than the labels and values of the other.

**Proposition 14** (Dynamic gradual guarantee). Suppose $t^{G_1}_1 \sqsubseteq t^{G_2}_1$ and $\mu_1 \sqsubseteq \mu_2$. Then if $t^{G_1}_1 \mid \mu_1 \mapsto t^{G_1}_2 \mid \mu'_1$ then $t^{G_2}_1 \mid \mu_2 \mapsto t^{G_2}_2 \mid \mu'_2$ where $t^{G_1}_2 \sqsubseteq t^{G_2}_2$ and $\mu'_1 \sqsubseteq \mu'_2$.

### 3.3.7 $\overline{\lambda}^{\text{REF}}$ in Action

$\overline{\lambda}^{\text{REF}}$ is semantically equivalent to HCC  The resulting language $\overline{\lambda}^{\text{REF}}$ behaves exactly as HCC. Recall the examples from § 2.3.1  $\overline{\lambda}^{\text{REF}}$, like HCC, rejects examples 2 and 4, and accepts examples 3 and 5.

For instance, consider example 2. The corresponding $\overline{\lambda}^{\text{REF}}$ term is $!(\text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool})$. Its elaboration reduces as follows:

$$
\begin{align*}
!(\text{Bool}) & (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \\
& \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto !\text{Bool} \ (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{Ref} \ \text{Bool} \rangle (\langle \text{ref} \ 4 :: ? :: \text{Ref} \ \text{Bool} \rangle) \mid \cdot \mapsto \text{error}
\end{align*}
$$

because $\langle \text{Int} \rangle \circ \langle \text{Bool} \rangle$ is not defined. Of course, this is just an example reduction; formally establishing that both languages are semantically equivalent is the subject of § 3.4.
\( \lambda^{\text{REF}} \) is not space efficient Even though semantically equivalent, contrary to HCC, \( \lambda^{\text{REF}} \) is not space efficient. We can write programs in \( \lambda^{\text{REF}} \) that accumulate an unbounded number of evidences during reduction.

To illustrate, consider \( \lambda^{\text{REF}} \) term \( \Omega = (\lambda x : ?.x.x)(\lambda x : ?.x.x) \). Its elaboration to \( \lambda^{\text{REF}}_{\epsilon} \) is

\[
\Omega_{\epsilon} = (\Rightarrow ? \Rightarrow (\lambda x ?.(\Rightarrow ?)x ?) (\Rightarrow ?) (\lambda x ?.(\Rightarrow ?)x ?) x ?)
\]

After multiple steps of reduction, the resulting term accumulates ascriptions as follows:

\[
\langle ? \rangle (\langle ? \rangle (\langle ? \rangle \ldots (\langle ? \rangle \Omega ? :: ? \ldots :: ?) :: ?) :: ?) :: ?
\]

This example illustrates how the order of combination of evidences impacts space efficiency and destroys tail recursion. Note that this problem applies to any language derived with AGT.

In contrast, with HCC, the same program reduces as follows (we omit stores for simplicity):

\[
\Omega_{\epsilon} = (\lambda x ?.c_1 x x) c_2 (\lambda x ?.c_1 x x) \slice c_1 (c_2 (\lambda x ?.c_1 x x)) c_2 (\lambda x ?.c_1 x x) \slice \Omega_{\epsilon} \slice ...
\]

where \( c_1 \) is a coercion from \(? \) to \( ? \Rightarrow ? \), and \( c_2 \) from \( \Rightarrow ? \) to \( ? \).

Even though \( \lambda^{\text{REF}} \) and HCC are different regarding space efficiency, they are semantically equivalent: given a term and its compilations to to \( \lambda^{\text{REF}}_{\epsilon} \) and HCC (and adapted version of HCC), either both terms reduce to related values, both terms diverge, or both terms reduce to an error. In the following, we formalize the relation between \( \lambda^{\text{REF}} \) and HCC (§3.4.3), along with the changes needed to recover space efficiency in \( \lambda^{\text{REF}}_{\epsilon} \) (§3.4.4).

### 3.4 Comparing \( \lambda^{\text{REF}} \) and HCC

In this section we compare \( \lambda^{\text{REF}} \) and HCC, the space-efficient coercion calculus of Herman et al. [60]. We start by presenting the static and dynamic semantics of HCC+, an adapted version of HCC extended with conditionals and binary operations. Then we formalize the relation between both semantics as follows: given a \( \lambda^{\text{REF}} \) term and its corresponding elaboration to \( \lambda^{\text{REF}}_{\epsilon} \) and translation to HCC+, we prove that the resulting terms are in a bisimulation relation, and furthermore, the terms are contextually equivalent. Although we establish contextual equivalence, contrary to HCC the dynamic semantics of \( \lambda^{\text{REF}}_{\epsilon} \) are not space-efficient, i.e. ascriptions can be repeatedly accumulated during reduction. We finalize this section by presenting the changes needed in the dynamic semantics of \( \lambda^{\text{REF}}_{\epsilon} \), so we can recover space efficiency.

#### 3.4.1 The Coercion Calculus

In this section we present HCC+, an adaptation of HCC extended with conditionals and binary operations. This language is designed as a cast calculus for \( \lambda^{\text{REF}}_{\epsilon} \). The following
presentation of this language is closely related to the coercion calculus presented by Siek et al. [112].

Usually, the operational semantics of gradual languages generate proxies when reducing function applications which involve casts. This approach may result in an unbounded growth in the number of proxies, which impacts space efficiency and destroys tail recursion [60]. HCC was designed to represent and compress sequences of casts, by using coercions instead of casts (and function proxies). HCC recovers space efficiency by combining and normalizing adjacent coercions to limit their space consumption to a constant factor.

Static semantics  Figure 3.7 present the static semantics of HCC++. The syntax includes gradual types $G$, ground types $R$, coercions $c$, and terms $t$. Ground types $R$ are the only

\[ R \in \text{GROUNDTYPE}, \quad c \in \text{COERCION}, \quad t \in \text{CTERM}, \]

\[
\begin{align*}
G & ::= \, ? \mid B \mid G \rightarrow G \mid \text{Ref} \, G \\
R & ::= \, ? \rightarrow ? \mid \text{Ref} \, ? \mid B \\
c & ::= \, i_G \mid \text{Fail} \mid \text{R!} \mid \text{R?} \mid c \rightarrow c \mid \text{Ref} \, c \, c \mid c ; c \\
t & ::= \, b \mid (\lambda x : G.t) \mid o \mid t \mid t \oplus t \mid \text{if} \, t \, \text{then} \, t \, \text{else} \, t \mid c \, t \mid \text{ref} \, t \mid !t \mid t := t
\end{align*}
\]

Figure 3.7: HCC++: Static semantics
Dynamic semantics  Figure 3.8 presents the dynamic semantics of HCC+. A value v can be a raw value u, or a coerced value c u, where coercion c is in normal form. We say a coercion is in normal form if it is irreducible, denoted $\text{nmc}$; the predicate is defined in Figure 3.9. To reduce programs we use three different evaluation contexts: H to reduce coercions, and F and E to reduce terms. Coercions are combined using the coerced term reduction rule $\rightarrow_c$. Coercions are maintained in normal form throughout evaluation using big step semantics of the coercion reduction rule $\rightarrow$, and the normal form predicate $\text{nmc}$. The coercion reduction rule combine coercions using the notion of coercion reduction rule $\rightarrow$. A failure coercion is produced when a tagging and a check-and-untag coercions are combined, and the ground types involved are different. When the types are the same, then an identity coercion is produced. The combination of an identity coercion with another coercion c produces the same coercion c. On the contrary, the combination of a failure coercion with another coercion c propagates the failure coercion. Reduction of combination of function coercions and reference coercions are defined inductively. Notice the contravariant combination order for the argument of functions, and in the coercions for write values in the heap respectively. Note that in the reduction of coerced terms, a failure coercion does not trigger a runtime error immediately (i.e. $\text{Fail} t \rightarrow_c \text{error}$), but after the subterm is reduced to a raw value (i.e. $\text{Fail} u \rightarrow_c \text{error}$). The reason for this is that HCC aims to regain space efficiency without changing the behavior of standard cast calculus/coercion semantics, which combines casts when the subterm is a value. The rest of the dynamic semantics are standard to cast calculi. The reduction of coerced dereferences coerces the value on the heap with the second component of the coercion, and dually the reduction of coerced assignments coerces the updated value using the first component of the coercion.

Translation semantics  Figure 3.10 presents the translation rules from $\lambda^{\text{REF}}$ to HCC+. The translation is a type-driven coercion insertion. The key idea is to insert coercions where consistency is used in the typing derivation. The translation judgment has the form $\Gamma; \Sigma \vdash t \rightarrow_c t' : G$ which represent translation from $\lambda^{\text{REF}}$ term t of type G, to HCC+ term t', under environments $\Gamma$ and $\Sigma$. Coercions are introduced using the $(G_1 \rightarrow G_2)t$ metafunction, which represents the insertion of a coercion from $G_1$ to $G_2$. This metafunction avoids the
\[ u := b \mid (\lambda x : G_1.t) \mid o \quad \text{(raw values)} \]
\[ v := u \mid cu \text{ where } nm c \]
\[ H := \Box \mid \Box ; c \mid c :: \Box \mid \Box \rightarrow c \mid c \rightarrow \Box \mid \Box \rightarrow \Box \mid \text{Ref} \Box c \mid \text{Ref} c \Box \quad \text{(Coercion contexts)} \]
\[ F := \Box \mid E + t \mid v + E \mid E t \mid v E \mid E :: G \]
\[ \text{if } E \text{ then } t \text{ else } t' \mid \text{ref } E \mid !E \mid E := t \mid v := E \quad \text{(Cast-free contexts)} \]
\[ E := F \mid c F \]

\[ \rightarrow: \text{Coercion} \times \text{Coercion} \]

**Notion of coercion reduction**

\[ R!; R? \rightarrow i_R \]
\[ R_1!; R_2? \rightarrow \text{Fail} \]
\[ \text{if } R_1 \neq R_2 \]
\[ (c_11 \rightarrow c_12); (c_21 \rightarrow c_22) \rightarrow \]
\[ (c_11; c_12) \rightarrow (c_12; c_22) \]
\[ (\text{Ref } c_11 \ c_12); (\text{Ref } c_21 \ c_22) \rightarrow \]
\[ \text{Ref } (c_21; c_11) \ (c_12; c_22) \]
\[ i_G; c \rightarrow c \]
\[ c; i_G \rightarrow c \]
\[ \text{Fail}; c \rightarrow \text{Fail} \]
\[ c; \text{Fail} \rightarrow \text{Fail} \]

\[ \text{CONFIG} = \text{CTerm} \times \text{STORE} \]

\[ \rightarrow: \text{CONFIG} \times \text{CONFIG} \]

**Notion of term reduction**

\[ \lambda x : G.t \ v \mid \mu \rightarrow t[v/x] \mid \mu \]
\[ (c_1 \rightarrow c_2) u \ v \mid \mu \rightarrow c_1 \ u \ (c_2 \ v) \mid \mu \]
\[ \text{ref } v \mid \mu \rightarrow o \mid \mu \ o \rightarrow v \quad \text{o fresh} \]
\[ !o \mid \mu \rightarrow \mu(o) \mid \mu \]
\[ o := v \mid \mu \rightarrow \text{unit} \mid \mu \ o \rightarrow v \]
\[ !(\text{Ref } c_1 \ c_2) o \mid \mu \rightarrow c_2 \ o \mid \mu \]
\[ (\text{Ref } c_1 \ c_2) o := v \mid \mu \rightarrow (o := c_1) \ v \mid \mu \]
\[ \text{if } b \text{ then } t_2 \text{ else } t_3 \mid \mu \rightarrow \begin{cases} t_2 \mid \mu & \text{if } b \text{ = true} \\ t_3 \mid \mu & \text{if } b \text{ = false} \end{cases} \]
\[ u_1 \uplus u_2 \mid \mu \rightarrow u_1 \uplus u_2 \]

**Figure 3.8: HCC⁺: Dynamic semantics**

The insertion of a redundant coercion by checking if \( G_1 \) is syntactically equal to \( G_2 \). If both types are the same, then the coercion is not introduced. Otherwise we use the coercion function \( \llangle G_1 \Rightarrow G_2 \rrangle \) to elaborate the coercion from \( G_1 \) to \( G_2 \). The inductive definition of the coercion inserting function is presented in Figure [3.11] and follows closely the rules for coercion typing. For instance, as \( R? \vdash ? \Rightarrow R \), then \( \llangle ? \Rightarrow R \rrangle = R? \). There are some
Figure 3.9: HCC\textsuperscript{+}: Coercion normal forms

\[ \Gamma; \Sigma \vdash t \sim_c t' : G \]

Translation rules

\begin{align*}
\text{(HRx)} & \quad \frac{x : G \in \Gamma}{\Gamma; \Sigma \vdash x \sim_c x : G} \\
\text{(HRc)} & \quad \frac{\theta(b) = B}{\Gamma; \Sigma \vdash b \sim_c b : B} \\
\text{(HRapp)} & \quad \frac{\Gamma; \Sigma \vdash t_1 \sim_c t'_1 : G_1 \quad \Gamma; \Sigma \vdash t_2 \sim_c t'_2 : G_2}{\Gamma; \Sigma \vdash t_1 \oplus t_2 \sim_c (G_1 \Rightarrow dom(G_1) \rightarrow cod(G_2))t'_1 \oplus (G_2 \Rightarrow dom(G_1))t'_2 : \overline{cod(G_1)}} \\
\text{(HOp)} & \quad \frac{\Gamma; \Sigma \vdash t_1 \sim_c t'_1 : G_1 \quad \Gamma; \Sigma \vdash t_2 \sim_c t'_2 : G_2 \quad ty(\oplus) = B_1 \land B_2}{\Gamma; \Sigma \vdash t_1 \oplus t_2 \sim_c (G_1 \Rightarrow B_1)\oplus (G_2 \Rightarrow B_2) : B_3} \\
\text{(HRif)} & \quad \frac{\Gamma; \Sigma \vdash t \sim_c t_1 : G_1}{\Gamma; \Sigma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \sim_c \text{ if } (G_1 \Rightarrow \text{Bool})t_1 \text{ then } (G_2 \Rightarrow \overline{G})t_2 \text{ else } (G_3 \Rightarrow \overline{G})t_3 : G} \\
\text{(HRL)} & \quad \frac{\Gamma, x : G_1 \vdash t \sim_c t'_2 : G_2}{\Gamma; \Sigma \vdash \lambda x : G_1.t \sim_c (\lambda x : G_1.t'_2) : G_1 \Rightarrow G_2} \\
\text{(HR:)} & \quad \frac{\Gamma; \Sigma \vdash t \sim_c t' : G'}{\Gamma; \Sigma \vdash (t :: G) \sim_c (G' \Rightarrow \overline{G'})t' : G} \\
\text{(HRref)} & \quad \frac{\Gamma; \Sigma \vdash t \sim_c t' : G'}{\Gamma; \Sigma \vdash \text{ref } t \sim_c \text{ ref } (G' \Rightarrow \overline{G})t' : \overline{G'}} \\
\text{(HRderef)} & \quad \frac{\Gamma; \Sigma \vdash t \sim_c t' : G' \quad G = \overline{\text{tref}}(G')}{\Gamma; \Sigma \vdash \overline{t} \sim_c !G' \Rightarrow \overline{\text{Ref } G'}t' : \overline{G'}} \\
\text{(HRasgn)} & \quad \frac{\Gamma; \Sigma \vdash t_1 \sim_c t'_1 : G_1 \quad \Gamma; \Sigma \vdash t_2 \sim_c t'_2 : G_2 \quad G_3 \sim_c \overline{\text{tref}}(G_1)}{\Gamma; \Sigma \vdash t_1 := t_2 \sim_c (G_1 \Rightarrow \overline{\text{Ref } G_3})t'_1 := (G_2 \Rightarrow G_3)t'_2 : \text{Unit}} \\
\text{(HRI)} & \quad \frac{o : G \in \Sigma}{\Gamma; \Sigma \vdash o \sim_c o : \overline{\text{Ref } G}}
\end{align*}

where \( \langle G_1 \Rightarrow G_2 \rangle \) = \( \begin{cases} 
\text{t} & \text{if } G_1 = G_2 \\
\llangle G_1 \Rightarrow G_2 \rrangle \text{t} & \text{otherwise}
\end{cases} \)

Figure 3.10: \( \overline{\text{Ref}} \) to HCC\textsuperscript{+} translation rules
\[ \{? \Rightarrow R\} = R? \quad \{R \Rightarrow ?\} = R! \quad \{? \Rightarrow G_1 \rightarrow G_2\} = (? \rightarrow ?); \{? \rightarrow ? \Rightarrow G_1 \rightarrow G_2\} \]

\[ \{G_1 \rightarrow G_2 \Rightarrow ?\} = \{G_1 \rightarrow G_2 \Rightarrow ?\}; (? \rightarrow ?)! \]

\[ \{G_{11} \rightarrow G_{12} \Rightarrow G_{21} \rightarrow G_{22}\} = \{G_{21} \Rightarrow G_{11}\} \rightarrow \{G_{12} \Rightarrow G_{22}\} \]

\[ \{? \Rightarrow \text{Ref } G\} = (\text{Ref } ?); \{\text{Ref } ? \Rightarrow \text{Ref } G\} \quad \{\text{Ref } G \Rightarrow ?\} = \{\text{Ref } G \Rightarrow ?\}; (\text{Ref } ?)! \]

\[ \{\text{Ref } G_2 \Rightarrow \text{Ref } G_1\} = \text{Ref } \{G_2 \Rightarrow G_1\} \{G_1 \Rightarrow G_2\} \]

Figure 3.11: Coercion insertion function

subtleties worth mentioning, such as the definition of \(\{? \Rightarrow \text{Ref } G\}\). This should result in

a coercion from \(?\) to \(\text{Ref } G\), but there are no direct coercion from unknown to any given
type \(\text{Ref } G\). Consequently \(\{? \Rightarrow \text{Ref } G\}\) is defined as the composition
of a coercion from \(?\) to \(\text{Ref } ?\) (to test if the value is actually a reference), with a coercion from \(\text{Ref } ?\) to

\(\text{Ref } G\): \(\{\text{Ref } ? \Rightarrow \text{Ref } G\}\), which follows the inductive definition. Analogously, \(\{\text{Ref } G \Rightarrow ?\}\)
is defined as the composition of a coercion from \(\text{Ref } G\) to \(?\), with a coercion from \(\text{Ref } ?\)
to \(?\). Notice that by construction, we do not need definitions for \(\{G \Rightarrow G\}\) and \(\{? \Rightarrow ?\}\) as
this cases are avoided thanks to the \(\{. \Rightarrow .\}\) metamorphism.

3.4.2 Relating \(\lambda^\text{REF} \varepsilon\) and HCC\(^+\)

We now establish the equivalence of the \(\lambda^\text{REF} \varepsilon\) and HCC\(^+\) semantics only for elaborated
and translated \(\lambda^\text{REF}\) terms respectively, by using a bisimulation relation and contextual
equivalence.

Figure 3.12 presents function \(\llbracket \cdot \rrbracket\), which relates evidence augmented consistent judgments
with coercions during the definition of the bisimulation relation. A naive relation between
evidence and coercion is ambiguous unless one indicates the gradual types involved in the
judgment. For instance, evidence \(\langle \text{Int} \rangle\) corresponds to both coercion \(\text{Int}\)? or \(\text{Int}\!\), unless we
expose the judgment associated with the evidence, so \(\langle \text{Int} \rangle \vdash \text{Int} \sim ?\) correspond exactly to
the coercion from \(\text{Int}\) to \(?\), \(\text{Int}\!\). The definition follows the definition of the coercion insertion
function presented in Figure 3.10 (e.g. \(\langle ? \Rightarrow R \rangle = R?\), so \(\langle (R) \Rightarrow ? \sim R \rangle = R?\)), save for
a few extra cases described next. Reflexive judgments on ground types and the unknown
type, where evidence corresponds to the initial evidence, are mapped to identity coercions for
ground types and the unknown type respectively. The definition also takes into consideration
judgments where both types are unknown, and the evidence has become more precise. If the
evidence is some ground type \(\langle R \rangle\), then the judgment is mapped into the check-and-untag
coercion from \(?\) to \(R\), followed by the tagging coercion from \(R\) to \(?\). If the evidence is
a function type (resp. reference type), then the corresponding coercion is the check-and-
untag coercion from \(?\) to \(? \rightarrow ?\) (resp. \(\text{Ref } ?\)), followed by the corresponding coercion of
the consistent judgment between \(? \rightarrow ?\) and \(? \rightarrow ?\) (resp. \(\text{Ref } ? \sim \text{Ref } ?\)) using the same
\[
\begin{align*}
\langle \langle \langle R \rangle \vdash R \bowtie R \rangle \rangle &= i_R, \\
\langle \langle \langle ? \vdash ? \sim ? \rangle \rangle &= i_?, \\
\langle \langle \langle R \rangle \vdash ? \sim ? \rangle \rangle &= R?, \\
\langle \langle \langle R \rangle \vdash R \sim ? \rangle \rangle &= R!; \\
\langle \langle \langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22} \rangle \rangle &= \langle \langle \langle G_1 \rangle \vdash G_{21} \sim G_{11} \rangle \rangle \rightarrow \langle \langle \langle G_2 \rangle \vdash G_{12} \sim G_{22} \rangle \rangle, \\
\langle \langle \langle G_1 \rightarrow G_2 \rangle \vdash ? \sim ? \rightarrow G_{21} \rightarrow G_{22} \rangle \rangle &= (? \rightarrow ?); \langle \langle \langle G_1 \rightarrow G_2 \rangle \vdash ? \sim ? \rightarrow ? \sim ? \rightarrow G_{21} \rightarrow G_{22} \rangle \rangle, \\
\langle \langle \langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim ? \rangle \rangle &= \langle \langle \langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim ? \rightarrow ? \rangle \rangle; (? \rightarrow ?)!, \\
\langle \langle \langle \text{Ref } G \rangle \vdash \text{Ref } G_1 \sim \text{Ref } G_2 \rangle \rangle &= \text{Ref } \langle \langle \langle G \rangle \vdash G_2 \sim G_1 \rangle \rangle \rightarrow \langle \langle \langle G \rangle \vdash G_1 \sim G_2 \rangle \rangle, \\
\langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle &= \langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle; (\text{Ref } ?)!, \\
\langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle &= \langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle; (\text{Ref } ?)!, \\
\langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle &= \langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle; (\text{Ref } ?)!, \\
\langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle &= \langle \langle \langle \text{Ref } G_1 \rangle \vdash \text{Ref } G_2 \sim ? \rangle \rangle; (\text{Ref } ?)!
\end{align*}
\]

where \((G_i \neq R_i)\)

Figure 3.12: Map from evidence augmented consistent judgments to coercions.

evidence\[5\] finally followed by the tagging coercion from ? \rightarrow ? (resp. Ref ?) to ?.

The bisimulation relation is formally presented in Figure 3.13. This relation syntactically relates a \(\lambda^\text{REF}_e\) term and an HCC\(^+\) term. Rules (bconst), (bx), (b\lambda), and (bo), are straightforward. Rule (bapp) relates two application terms inductively, but notice that because evidence terms do not correspond to anything in HCC\(^+\), we build an ascribed term instead, e.g. to relate \(\varepsilon_1t_{11} \) with \(t_{21}\) we notice that the type of \(t_{21}\) has to be \(G_1 \rightarrow G_2\), therefore as \(\varepsilon \vdash G \sim G_1 \rightarrow G_2\) where \(t_{11} \in \text{T}[G]\), we can inductively relate \(\varepsilon_1t_{11} :: G_1 \rightarrow G_2\) with \(t_{21}\) instead. Similarly, we relate \(\varepsilon_2t_{12} :: G_1\) (as \(\varepsilon_2 :: G \sim G_1\), where \(t_2 \in \text{T}[G']\)) with \(t_2\). We use the same reasoning for rules (bref), (b!), and (b:=). Rules (b::eq), (b::id), and (b::leq) are the most important rules. Rule (b::eq) is the most intuitive rule; it relates an ascribed term with a coerced term, only if the underlying evidence of the ascription is mapped to the coercion. Rule (b::id) relates a redundant ascription with a term without a coercion. The reason is that terms like \(i_G u\) (which is related to \(\langle G \rangle u^G\) by (b::eq) if \(u\) is related to \(u^G :: G\) reduces to \(u\), whereas in \(\lambda^\text{REF}_e\) this redundant cast is not eliminated. Rule (b::leq) relates \(\lambda^\text{REF}_e\) terms with HCC\(^+\) terms that have eagerly combined coercions starting from the outermost pair of coercions. For instance, \(t = \varepsilon_1(\varepsilon_2t_1 :: G_2) :: G_1\) may be related to \(t = c_1(c_2t_2)\), if \(t_1 \approx t_2\), \(c_1 = \langle \varepsilon_1 \vdash G_2 \sim G_1 \rangle\), and \(c_2 = \langle \varepsilon_2 \vdash G_3 \sim G_1 \rangle\) for some \(G_3\). But \(t\) may take a step to \(c_2t_2\) where \(c_2 ; c_1 \vdash \varepsilon c_2\) and \(nm c_2c_1\). By using (b::leq) we can relate \(t\) and \(c_2t_2\), by decomposing \(c_2\) back into \(c_1\) and \(c_2\), as we know by (b::eq) that \(\varepsilon_2t_1 :: G_2\) is related to \(c_2t_2\). Finally rule (b\mu) relates two stores if for all related locations, their corresponding values in the stores are related.

\[5\] Note that if \(\langle G_1 \rightarrow G_2 \rangle \vdash ? \sim ?\) then \(\langle G_1 \rightarrow G_2 \rangle \vdash ? \rightarrow ? \sim ? \rightarrow ?\).
Figure 3.13: Bisimulation relation between intrinsic terms and the coercion calculus.

We can now state the bisimulation lemma between $\lambda^\text{REF}_x$ and HCC$^+$ as follows

**Lemma 15** (Bisimulation between $\lambda^\text{REF}_x$ and HCC$^+$). If $t_1 \in T[G]$, $\Gamma; \Sigma \vdash \mu_1 : G$, $\mu_2 \models \Sigma$, $\mu_1 \approx \mu_2$, and $t_1 \approx t_2$, then

1. If $t_1 | \mu_1 \to t'_1 | \mu'_1$, then $t_2 | \mu_2 \to t'_2 | \mu'_2$ such that $t'_1 \approx t'_2$ and $\mu'_1 \approx \mu'_2$.

2. $\exists j, 1 \leq j \leq 3$. if $t_2 | \mu_2 \to^j t'_2 | \mu'_2$, then $t_1 | \mu_1 \to^* t'_1 | \mu'_1$ such that $t'_1 \approx t'_2$ and $\mu'_1 \approx \mu'_2$.

Given a $\lambda^\text{REF}_x$ term, its elaboration to $\lambda^\text{REF}_x$ and its translation to HCC$^+$ are related.

**Proposition 16** (Translations are bisimilar). If $\phi; \phi \vdash t : G$, $\phi; \phi \vdash t \to^e t_1 : G$, and $\phi; \phi \vdash t \to^c t_2 : G$, then $t_1 \approx t_2$.

Next we proceed to state contextual equivalence between both transformations. A context $C$ is a program with a single hole $[]$. We use notation $\Gamma'; \Sigma' \vdash C[\Gamma]; \Sigma \vdash . : G' : G'$ to represent that filling the hole with term $t$ such that $\Gamma; \Sigma; t : G$, then $\Gamma'; \Sigma' \vdash C[t] : G'$. Similarly, $C$ represents a context for intrinsic terms, and we use notation $C[, \in T[G]] \in T[G']$ to represent that plugging $t \in T[G]$ in the hole, then $C[t] \in T[G']$. Finally, $C$ represents contexts for
terms of the coercion calculus, where $\Gamma';\Sigma' \vdash C[\Gamma;\Sigma \vdash \ldots : G] : G'$ represents that filling the hole with term $t$ such that $\Gamma;\Sigma; t : G$, then $\Gamma';\Sigma' \vdash C[t] : G'$. We write $t \mid \mu \Downarrow$ (resp. $t \mid \mu \Downarrow \text{error}$), if $t \mid \mu \longrightarrow^* v \mid \mu'$ (resp. $t \mid \mu \longrightarrow^* \text{error}$) for some resulting store $\mu'$. Similarly, we write $t \mid \mu \Downarrow$ (resp. $t \mid \mu \Downarrow \text{error}$), if $t \mid \mu \longrightarrow^* v \mid \mu'$ (resp. $t \mid \mu \longrightarrow^* \text{error}$) for some resulting store $\mu'$.

**Definition 9.** Consider $\phi;\Sigma \vdash C[\Gamma;\Sigma \vdash \ldots : G] : G'$, we use notation $C \sim (C, C)$, when $\forall \Gamma;\Sigma \vdash t : G$, $\Gamma;\Sigma \vdash t \rightsquigarrow t_1 : G$, $\Gamma;\Sigma \vdash t \rightsquigarrow t_2 : G$, $\phi;\Sigma \vdash C[t] \rightsquigarrow C[t_1] : G'$, and $\phi;\Sigma \vdash C[t] \rightsquigarrow C[t_2] : G'$.

**Definition 10 (Observational equivalence).** We use notation $\Gamma;\Sigma \models t_1 \approx_{\text{CTX}} t_2 : G$, read $t_1$ is observationally equivalent to $t_2$, when for any $\phi;\Sigma \vdash C[\Gamma;\Sigma \vdash \ldots : G] : G'$, $C \sim (C, C)$, stores $\mu_1 \models \Sigma$ and $\mu_2 \models \Sigma$, such that $\mu_1 \approx \mu_2$, then $C[t_1] \mid \mu \Downarrow \iff C[t_2] \mid \mu \Downarrow$, and $C[t_1] \mid \mu \Downarrow \text{error} \iff C[t_2] \mid \mu \Downarrow \text{error}$.

We then state that two bisimilar terms are observationally equivalent.

**Proposition 17.** If $t_1 \in T[G]$, $\Gamma;\Sigma \vdash t_2 : G$, and $t_1 \approx t_2$, then $\Gamma;\Sigma \vdash t_1 \approx_{\text{CTX}} t_2 : G$.

Finally we can state contextual equivalence between both semantics for elaborated and translated $\lambda^{\text{REF}}$ terms:

**Corollary 18 (Contextual equivalence).** If $\Gamma;\Sigma \vdash t : G$, $\Gamma;\Sigma \vdash t \rightsquigarrow t_1 : G$, and $\Gamma;\Sigma \vdash t \rightsquigarrow t_2 : G$, then $\Gamma;\Sigma \models t_1 \approx_{\text{CTX}} t_2 : G$.

### 3.4.3 Recovering Space Efficiency in $\lambda^{\text{REF}}$

Although we have established that given a $\lambda^{\text{REF}}$ term, its elaboration to $\lambda^{\text{REF}}_\varepsilon$ and its translation to HCC$^+$ are contextually equivalent, the dynamic semantics of $\lambda^{\text{REF}}_\varepsilon$ are not space-efficient, i.e. ascriptions can be repeatedly accumulated during reduction as illustrated in § 3.3.7. We now present the changes needed in the runtime semantics of $\lambda^{\text{REF}}_\varepsilon$ in order to adopt space efficient operational semantics. Changes are highlighted in gray.

The main problem with $\lambda^{\text{REF}}_\varepsilon$ is that the current definition of evaluation contexts allows ascriptions (and evidences) to accumulate until the corresponding subterm is reduced to a value. Figure 3.14 presents a space-efficient dynamic semantics variant with respect to the original dynamic semantics of Figure 3.5.

To achieve space efficiency, we eliminate the $E :: G$ evaluation context, as we do not want to allow reduction inside nested ascriptions anymore. Instead of combining evidences starting from the inner-most pair of evidences, rule ($r7$) now combines evidence eagerly starting from the outer-most pair of evidences by using the new $\square :: G$ evaluation context, before subterm $t$ is reduced to a value.
Preserving the failure behavior  To preserve the failure behavior of the original dynamic semantics (and fail at the same point of execution), \((r7)\) cannot simply reduce to an error when consistent transitivity is not defined. For instance, consider \(t = (\langle \text{Bool} \rangle (\langle \text{Int} \rangle (\langle \text{Int} \rangle 1 \oplus \langle \text{Int} \rangle 1) :: ?) :: ?)\). Using the original dynamic semantics, this term reduces to an error as follows

\[
\begin{array}{l}
t \cdot \mapsto \langle \text{Bool} \rangle (\langle \text{Int} \rangle 2 :: ?) :: ? \\
\cdot \mapsto \text{error}
\end{array}
\]

If we combine evidences starting from the outer-most pair of evidence, as \(\langle \text{Int} \rangle \circ (\langle \text{Bool} \rangle)\) is not defined, then \(t\) would reduce to an error immediately, i.e. \(t \cdot \mapsto \text{error}\).

If combination of evidences is not defined then instead of reducing directly to an error, we reduce to an evidence term using the pending error evidence \(\varepsilon_{\text{err}}\). The pending error evidence \(\varepsilon_{\text{err}}\) is defined such that \(\varepsilon_{\text{err}} \vdash G_1 \sim G_2\) for any \(G_1\) and \(G_2\). We also update the definition of intrinsic values \(v\) to raw values \(u\), or ascribed simple values \(\varepsilon u :: G\) where \(\varepsilon \neq \varepsilon_{\text{err}}\). Rule \((r8)\) just propagates evidence \(\varepsilon_{\text{err}}\) until rule \((r9)\) finally reduces a pending error evidence and a simple value to an error. Using the space-efficient variant, \(t\) now reduces as follows:

\[
\begin{array}{l}
t \cdot \mapsto \varepsilon_{\text{err}}(\langle \text{Int} \rangle 1 + \langle 1 \rangle 1) :: ? \\
\varepsilon_{\text{err}}2 :: ? \mapsto \text{error}
\end{array}
\]

Closing the relation with HCC\(^+\)  Regarding the result of \(\S 3.4.2\), the new space-efficient dynamic semantics are now more closely related to HCC\(^+\). In particular rule \((b:\text{leq})\) of Figure 3.13 is not needed anymore as evidences and coercions are reduced in lock-step. The only difference between both semantics is that identity coercions are eliminated during
reduction, whereas redundant evidences are not (and this is why we have to keep the (b::id) rule).

**Eager space-efficient dynamic semantics** Alternatively we could have defined the space-efficient dynamic semantics without rules (r8) and (r9), and where (r7) is defined as follows:

\[
\langle G_2 \rangle (\langle G_1 \rangle \uparrow \vdash G) \rightarrow_c \begin{cases} 
\langle G_1 \cap G_2 \rangle t 
& \text{if } G_1 \cap G_2 \text{ is not defined} \\
\text{error} 
& \text{otherwise}
\end{cases}
\]

This variant would yield a *more eager* semantics. Going back to the previous example, \(t\) now reduces immediately to an error after trying to combine \(\langle \text{Int} \rangle\) and \(\langle \text{Bool} \rangle\) outer evidences: \(t \vdash \cdot \rightarrow \text{error}\).

The main difference with the previous approach is that a program may diverge using the original dynamic semantics, whereas now it may diverge or fail with an error. To illustrate this, consider the following program

\[
\langle \text{Int} \rangle (\langle ? \rangle (\langle \text{Bool} \rangle (\langle ? \rangle \Omega :: ?) :: \text{Bool}) :: ?) :: \text{Int}
\]

Using the original dynamic semantics this program diverges as \(\Omega\) is evaluated first before combining the outer evidences. Using the first variant of the space-efficient semantics this program also diverges as the outer evidence \(\varepsilon_{\text{err}}\) never triggers an error because \(\Omega\) never reduce to a value. But using the second variant of the space efficient semantics we reduce to an error just after combining the \(\langle \text{Bool} \rangle\) and \(\langle \text{Int} \rangle\) evidences.

### 3.5 Encoding Permissive and Monotonic References in \(\lambda^{\text{REF}}\)

In this section we present \(\lambda^{\text{REF}}_{\text{pm}}\), an extension of \(\lambda^{\text{REF}}\) with support for both permissive and monotonic references. We codify permissive and monotonic references by introducing new term constructors for each form of reference in the source language. Encoding monotonic references is more difficult than encoding permissive references, as it involves extending the dynamic semantics of \(\lambda^{\text{REF}}_{\text{e}}\).

\(\lambda^{\text{REF}}_{\text{pm}}\) supports for programmers two constructors to create references: \texttt{ref} for guarded reference, and \texttt{mref} for monotonic references. For instance, to emulate the behavior of monotonic references in examples 3 and 5, we use \texttt{mref} as illustrated below:

```plaintext
1 let x = mref (4 :: ?) 1 let x = mref (4 :: ?)  
2 let y: Ref Bool = x ← runtime error 2 let y: Ref Int = x  
3 y := true 3 x := true ← runtime error  
4 !y
```

Example 3

Example 5
3.5.1 Static Semantics

We start by extending the syntax of $\lambda^{REF}$ as follows:

$$
\begin{align*}
\tau & ::= \text{g} | \text{p} | \text{m} \quad \text{(reference mode)} \\
v & ::= \ldots | o_{z} \quad \text{(values)} \\
t & ::= \ldots | \text{ref}^{G_{z}} t \quad \text{(terms)}
\end{align*}
$$

A *reference mode* $z$ may be a guarded reference $\text{g}$, a permissive reference $\text{p}$, or a monotonic reference $\text{m}$ (notice that the mref and ref constructors correspond to ref$^{G_{m}}$ and ref$^{G_{g}}$ respectively). Locations are now indexed by a reference mode $o_{z}$, e.g. $o_{m}$ represent a monotonic reference. Reference terms are also indexed with a tag $z$ to know which kind of reference to create during reduction.

Figure 3.15 highlights the changes to the typing rules of Figure 3.3. Rule (Gref) is splitted into (Grefz) and (Grefp), the former to type check guarded and monotonic references, and the latter for permissive references. Note that (Grefp) assigns type $\text{Ref ?}$ to any permissive reference, as it may be used freely with any value of any type.

3.5.2 Dynamic Semantics

Analogous to $\lambda^{REF}$, the dynamic semantics of $\lambda^{REF}_{pm}$ are defined via elaboration to their intrinsic representation called $\lambda^{REF}_{pm}$. The elaboration rules are identical to Figure 3.6, save for terms and types corresponding to references, which are now indexed by a reference mode.

Figure 3.16 presents selected rules of the dynamic semantics of $\lambda^{REF}_{pm}$. We highlight in gray the key changes with respect to Figure 3.5. Rules (r4), (r5) and (r6) are adapted by adding a reference mode $z$ to the corresponding terms and types constructors. Rule (r4), instead of reducing to a location, now reduces to an ascribed location. Although this ascription may seems redundant, it is used later by other rules to push more precise evidence information in the store when working with monotonic locations as shown in rule (r7). Rule (r7) reduces an evidence term and a store, as the store may change during combination of evidence. There is also a new special case when a raw value $u$ is a monotonic location. In that case, the underlying value in the store is updated with information of evidence $\langle G_{3} \rangle$ as it may gain

$^{6}$Notice that in the examples of § 3.3.7 we use mref as syntactic sugar for ref$^{m}$.
Notions of Reduction

\[(r4) \quad \text{ref}^G_{\_} \langle G_1 \rangle u \mid \mu \rightarrow \langle \text{Ref} \ G_2 \rangle o^G_{\_} \:: \text{Ref} \ G_2 \mid \mu[o^G_{\_} \mapsto \langle G_1 \rangle u :: G_2] \]
where \( o_{\_} \not\in \text{dom}(\mu) \)

\[(r5) \quad !^G(\langle \text{Ref} \ G_1 \rangle o^G_{\_}) \mid \mu \rightarrow \langle G_1 \rangle v :: G \mid \mu \quad \text{where } v = \mu(o^G_{\_}) \]

\[(r6) \quad \langle \text{Ref} \ G_1 \rangle o^G_{\_} := G_3 \langle G_2 \rangle u \mid \mu \rightarrow \begin{cases} \text{unit} \mid \mu[o^G_{\_} \mapsto \langle G_3 \cap G_1 \rangle u :: G] & \text{if } G_2 \cap G_1 \text{ is not defined} \\ \text{error} & \text{if } G_2 \cap G_1 \text{ is not defined} \end{cases} \]
where \( \mu(o^G_{\_}) = \langle G' \rangle u :: G \), and if \( z = m \) then \( G_3 = G_2 \cap G' \), otherwise \( G_3 = G_2 \).

\[(r7) \quad \langle G_2 \rangle(\langle G_1 \rangle u :: G) \mid \mu \rightarrow c \begin{cases} \langle G_3 \rangle u \mid \mu & \text{if } u \neq o^G_{m} \\ \langle G_3 \rangle u \mid \mu[u \mapsto \langle G_4 \rangle u' :: G_5] & \text{if } u = o^G_{m} \\ \text{error} & \text{if } G_3 \text{ is not defined, or} \\ \text{error} & \text{if } u = o^G_{m} \text{ and } G_4 \text{ is not defined} \end{cases} \]
where \( \mu(o^G_{m}) = \langle G' \rangle u' :: G_5, G_3 = G_1 \cap G_2, \text{ and } G_4 = \text{ref}(G_3) \cap G' \)

\[\text{Reduction} \quad \ldots \quad (RF) \quad \frac{\text{et} \mid \mu \rightarrow_c \text{et'} \mid \mu'}{F[\text{et}] \mid \mu \rightarrow F[\text{et'}] \mid \mu'} \quad \text{and} \quad \frac{\text{et} \mid \mu \rightarrow_c \text{error}}{F[\text{et}] \mid \mu \rightarrow \text{error}} \quad \ldots\]

Figure 3.16: \( \lambda^{\text{REF}}_{\text{pm}} \cdot \text{Dynamic semantics (selected rules)} \)

precision (or fail!). To illustrate rule \((r7)\), consider the following step of reduction:

\( \langle \text{Ref} \ (? \rightarrow \text{Int}) \rangle (\langle \text{Ref} \ (? \rightarrow ?) \rangle o^7_{m} :: \text{Ref} \ (? \rightarrow ?)) \mid o^7_{m} \rightarrow \langle (? \rightarrow ?)(\lambda x : \_x) :: ? \rightarrow ? \rangle \)

\( \rightarrow_c \langle \text{Ref} \ (? \rightarrow \text{Int}) \rangle o^7_{m} \mid o^7_{m} \rightarrow \langle (? \rightarrow ?)(\lambda x : \_x) :: ? \rightarrow ? \rangle \)

The corresponding value in the store of the monotonic location is updated: its evidence gains precision as \( \text{Int} \rightarrow ? \subseteq ? \rightarrow ? \). Finally, contexts \((RF)\) and \((RFerr)\) are also adapted to include the store when combining evidences.

### 3.5.3 Properties

\( \lambda^{\text{REF}}_{\text{pm}} \) satisfies all the properties described in §3.3.6. As \( \text{ref} \) is not part of the source language \( \lambda^{\text{REF}}_{\text{pm}} \), to make sense of the conservative extension of the static discipline properties (Props [11] and [78]), any \( \text{ref} \) term must be converted to either a \( \text{ref}^G_g \) or a \( \text{ref}^G_m \) term (when considering fully precise programs, a guarded reference and a monotonic reference behave identically).
Notice that converting a ref term to a ref\textsubscript{p} term breaks Props \[11\] e.g. \(\vdash \text{ref 1 : Ref Int}\) but \(\vdash \text{ref\textsubscript{p} 1 : Ref ?}\).

For monotonic references we state a property that best describes their behavior.

**Proposition 19** (Monotonicity of the heap). If \(t^G | \mu \mapsto t'^G | \mu'\), then \(\forall o^G_m \in \text{dom}(\mu), \mu(o^G_m) = \varepsilon u :: G', \) then \(\mu'(o^G_m) = \varepsilon' u' :: G'\) and \(\varepsilon' \subseteq \varepsilon\).

This means that by taking a step, for every monotonic reference the evidence of the corresponding value in the heap may only gain precision upon assignment (this is not true for guarded references as assignments may update the evidence to some less precise type).

### 3.6 Related work

We have already extensively discussed the four main approaches to gradual references found in the literature \[109, 60, 114\]. Vitousek et al. \[125\] present Reticulated Python, a tool for experimenting with gradual typing in Python 3 with support for references. They give two different dynamic semantics for casts: guarded semantics (with support for guarded references), and transient semantics. Instead of performing proxying of function or wrapping of runtime values, the transient semantics translate source programs by inserting type checks at all elimination forms, and at the entry and output of function definitions. These checks only test if values shallowly conform to a given type: only immediately-checkable information is considered. Greenman and Felleisen \[52\] compare guarded and transient semantics, and show that soundness for the transient semantics is a weaker notion that only guarantees preservation of the top-level constructor of the static type of an expression. For instance, consider \(h = (\lambda f : (\? \rightarrow \?) . f :: \text{Int} \rightarrow \text{Int}), g = (\lambda x : \text{Bool}.x), \) and term \(h g\). Using guarded semantics this program reduces to an error after reducing the body of \(h\), because \(\text{Bool} \rightarrow \text{Bool}\) (the type of the returned value, \(g\)) is not consistent with the expected return type, \(\text{Int} \rightarrow \text{Int}\). On the contrary, using the transient semantics, this program reduces successfully to \(g\) as its conforms with the expected top-level type constructor (a function). Of course, if we evaluate the program \((h g) 1\) then we will get an error right after applying \(g\) with \(1\) (thanks to the check at the entry of the body of \(g\)). Similarly, checks involving pair types only test if a given value is a pair. In Reticulated Python, for references (objects) the story is slightly different. Checks involving reference types (object types) recursively inspect a given value. For instance, consider the previous application of \(h g\) where now \(h = (\lambda x : \text{Ref ?}.x :: \text{Ref Int}), \) and \(g = \text{ref true}\). Using transient semantics this program reduces to an error as expected. After reducing the body of \(h\), the resulting location content does not conform with the expected type \(\text{Ref Int}\). But when we combine references and functions, the same issue as before manifests: if \(h = (\lambda x : \text{Ref (? \rightarrow ?)}).x :: \text{Ref Int} \rightarrow \text{Int}, \) and \(g = \text{ref } (\lambda x : \text{Bool}.x), \) then \(h g\) reduces successfully to a location whose content conforms with the expected resulting type: a reference to a function. We believe that the discussion about transient semantics is orthogonal to references, and is extensively analyzed by Greenman and Felleisen \[52\], therefore we did not include it in the main body of this work.

Many prior work on gradual security typing also supports references \[35, 39, 40, 120, \]}.
although imprecision is introduced exclusively via security labels, e.g. types like \( \text{Ref Int} \), are supported but not types like \( \text{Ref } ? \). Toro et al. [120] derived their gradual language using AGT, they also support guarded references, where imprecision is limited to the security levels of references.

Cimini and Siek [26] present the Gradualizer, a methodology and algorithm to systematically derive the static and cast insertion semantics of a gradual language from a static type system. They illustrate the application of this methodology to a language with references. We conjecture that the resulting gradual language treats gradual references as guarded or monotonic references, but it is hard to know precisely as the dynamic semantics were left as future work. They later extend the Gradualizer to also be able derive the dynamic semantics of a gradual language [27]. The handling of auxiliary structures such as the heap is however not supported. The authors mention informally how the algorithm could be adapted to references, and conjecture that the resulting dynamic semantics correspond to the guarded semantics of Herman et al. [60]; however the precise formal treatment of this extension is left as future work.

There are many languages that integrate static and dynamic typing in some way, and support references: TypeScript [28], Flow [71], Hack [36], Dart [30], and Typed Clojure [18]. These languages adopt another approach called \textit{optional} typing [19], which allows programmers to partially introduce type annotations to capture some errors statically, but do not perform any additional runtime checks at runtime beyond those that are performed for fully-untyped programs. This means that the runtime semantics are not sound with respect to its static type system.

Finally, there are many efforts related to gradualizing advanced typing discipline, such as typestates [129, 17], ownership types [107], annotated type systems [117], effects [13, 14, 123], refinement types [76, 72], parametric polymorphism [7, 69, 122], and the security type systems discussed above, among others. Since the formulation of the refined criteria for gradually-typed languages [113], however, only refinement types [76] and data type refinements [72] have been shown to fully respect such guarantees. Toro et al. [120, 122] reveal deep tensions between semantic type-based properties (noninterference, parametricity) and the dynamic gradual guarantee. This work contributes to the gradualization of advanced typing disciplines by deriving a gradual language that satisfies the refined criteria. Also this work presents the first formal statement and proof of the conservative extension of the dynamic semantics of the static language for a gradual language derived using AGT.

### 3.7 Conclusion

In this chapter we present \( \lambda^{\text{REF}} \), a gradual language with support for mutable references. This language is derived step-by-step using the AGT methodology. We compare the resulting language with other gradual languages with mutable references: monotonic references, permissive references, and guarded references. We find that \( \lambda^{\text{REF}} \) treats references as guarded references, similar to how references are treated in the coercion calculus of Herman et al. [60] (HCC).
We formalize this relation by introducing HCC+, an adapted version of HCC with conditionals and binary operations. We prove contextual equivalence between $\lambda_{\text{REF}}$ (the intrinsic semantics of $\lambda_{\text{REF}}$) and HCC+ for elaborated and translated $\lambda_{\text{REF}}$ terms. The main difference between both semantics has nothing to do with references, but with the order in which evidences/casts are combined. Under certain conditions (and contrary to HCC), a gradual language derived with AGT may accumulate an unbounded number of runtime checks. We describe the changes needed in the dynamic semantics of $\lambda_{\text{REF}}$ to recover space efficiency. Finally, we present $\lambda_{\text{pm}}$, an extension of $\lambda_{\text{REF}}$ that supports both permissive and monotonic references.

So far, we have explained how to apply AGT to a static language, and we have validated that AGT can scale to imperative languages. An interesting perspective for future work is to extend $\lambda_{\text{REF}}$ with nominal subtyping. We anticipate that a refined interpretation of evidences (such as pair of type intervals) might be needed to precisely capture type bounds at runtime, similarly to the security label intervals used in the next chapter. It would also be interesting to port the space-efficiency technique developed here to other AGT-derived gradual languages. In the next chapter we explore AGT and subtyping by applying AGT to a complex type discipline: a simply-typed lambda calculus with security types and references.
Chapter 4

Type-driven Gradual Security Typing

In security-typed programming languages, types statically enforce noninterference between potentially conspiring values, such as the arguments and results of functions. But to adopt static security types, like other advanced type disciplines, programmers face a steep wholesale transition, often forcing them to refactor working code just to satisfy their type checker. To provide a gentler path to security typing that supports safe and stylish but hard-to-verify programming idioms, researchers have designed languages that blend static and dynamic checking of security types. Unfortunately most of the resulting languages only support static, type-based reasoning about noninterference if a program is entirely statically secured. This limitation substantially weakens the benefits that dynamic enforcement brings to static security typing. Additionally, current proposals are focused on languages with explicit casts, and therefore do not fulfill the vision of gradual typing, according to which the boundaries between static and dynamic checking only arise from the (im)precision of type annotations, and are transparently mediated by implicit checks.

In this chapter we stress the AGT methodology applying it to a first complex type discipline: a simply-typed lambda calculus with security types and references. We present GSLRef, a gradual security-typed higher-order language with references (§4.4). As a gradual language, GSLRef supports the range of static-to-dynamic security checking exclusively driven by type annotations, without resorting to explicit casts. Additionally, GSLRef lets programmers use types to reason statically about termination-insensitive noninterference in all programs, even those that enforce security dynamically. We prove that GSLRef satisfies all but one of Siek et al.’s criteria for gradually-typed languages, which ensure that programs can seamlessly transition between simple typing and security typing (§4.4.4). A notable exception regards the dynamic gradual guarantee, which some specific programs must violate if they are to satisfy noninterference; it remains an open question whether such a language could fully satisfy the dynamic gradual guarantee. To realize this design, we were led to draw a sharp distinction between syntactic type safety and semantic type soundness, each of which constrains the design of the gradual language (§4.5).

\footnote{This chapter is based on the work of Toro et al. \[120\].}
4.1 Introduction

This work revisits gradual information-flow security typing, with a particular focus on the strong information-flow guarantees that security types have historically implied. We describe a new language, GSL_{Ref}, that introduces a *type-driven* conception of gradual security. Unlike most prior work, GSL_{Ref} supports the same static, type-based reasoning about information-flow for gradually-typed programs as SSL_{Ref}, its purely static counterpart.

**Check-driven approaches break free theorems.** Dynamic security casts give flexibility to programmers, but fundamentally cripple the ability to reason statically using security types. In particular, if security downcasts are added to the language, although noninterference is still preserved, static type judgments no longer imply free theorems about security of programs, as was discussed above. As a result, programmers must reason about the *dynamic semantics*—dynamic labels, dynamic upgrades, and dynamic checks—to uncover which values do not interfere with one another. In particular, a function’s type no longer denotes noninterference properties about its arguments and results. For example, consider the function:

\[
\begin{align*}
\text{let } \text{mix} : \text{Int} \rightarrow \text{Int} & \rightarrow \text{Int} = \\
\text{fun } \text{pub} \text{ priv } \Rightarrow & \text{ if } \text{pub} < (\text{Int}\text{priv}) \text{ then } 1 \text{ else } 2
\end{align*}
\]

This program is statically accepted by languages that only check for compatibility of base types [35, 39]. The type of \text{mix}, while fully static, does not guarantee that \text{mix} never reveals information about its second argument. Rather, the type merely guarantees that the second argument’s security level is \textit{at most} \text{H} and the result is \textit{at most} \text{L}. But upper-bounds on security labels do not suffice to make definitive assertions about the noninterference behavior of this function. Indeed, the program \text{mix} 1L 5L successfully reduces to 1L. In order to avoid such behavior, the programmer must \textit{explicitly} upgrade the dynamic security level of the value passed as second argument at each call site. Alternatively, one can upgrade \text{mix} to its \textit{own} type, thereby forcing the second argument to be upgraded before executing the function body (and hence preventing any information leak about that argument). This highlights the fact that \textit{types} alone do not denote noninterference properties: the two versions of the \text{mix} function behave differently although they have the same type.

This phenomenon, that adding dynamic checking to a static system may weaken type-based reasoning principles, is not unique to security typing. Prior work on cast calculi with parametric polymorphism observes that adding runtime type tests to System F preserves \textit{type safety}—i.e. that programs do not crash—but sacrifices \textit{type soundness}—i.e. that polymorphic types denote strong data abstraction guarantees via parametricity [6 § 5.1].

**Contributions.** Modular, compositional, and type-based reasoning are hallmark benefits of type systems. Thus, to facilitate the seamless transition toward static security typing, the typing judgment of a gradual type system should imply the same semantic invariants that its

---

2Recent work by Fennell and Thiemann [40] on LGJS addresses this particular problem, as described in Sec. 4.7.
fully-static counterpart does. To that end, this work presents GSLRef, a *type-driven* gradual security language that extends a static security type discipline with gradual security labels and corresponding notions of gradual type precision and consistent subtyping. To secure GSLRef programs, one just adds static security labels: dynamic checks arise automatically and implicitly, as needed to enforce the noninterference guarantees denoted by static types.

Unlike most prior work, GSLRef’s static security types denote the same noninterference guarantees as its fully static counterpart language SSLRef. As such, GSLRef’s security types enable modular and compositional type-based reasoning about noninterference, just like the fully static SSLRef, whereas security types in most prior gradual languages do not. GSLRef’s type system supports reasoning about termination-insensitive noninterference because it is sound with respect to a security logical relation defined directly in terms of type structure. This result is standard for a purely-static security language [58], but novel for a gradual security language with imprecise types supported by dynamic checks. In fact the dynamics are guided by the needs of the noninterference proof.

To summarize, this work makes the following contributions:

- This work represents a particularly challenging application of AGT to derive the design of a gradual security language with references. We report on our experience with a number of important considerations that complement the original presentation of AGT. In addition, we highlight the limitation of AGT when applied to semantically-rich type disciplines. (Sec. 4.6)
- We present GSLRef, a gradual security language that supports seamless transition between simply-typed and security-typed programming. Security typing annotations alone drive the balance between static and dynamic information flow checking. (Sec. 4.4)
- We prove that GSLRef’s type discipline enforces termination-insensitive noninterference: GSLRef’s types reflect strong information-flow invariants that hold even in code that contains gradually-typed subexpressions. (Sec. 4.5)
- We prove the static gradual guarantee. Interestingly, in order to ensure noninterference in the presence of references (and hence implicit flows through the heap), GSLRef sacrifices the dynamic gradual guarantee.
- Finally, we contribute more generally to the foundations of gradual typing for advanced type disciplines. We find that GSLRef’s security invariants require separate consideration of syntactic type *safety* and semantic type *soundness*, each of which constrains the design of the gradual language.

Before diving into the development of GSLRef, Sec. 4.2 informally introduces the type-driven approach to gradual security typing through examples. Then, Sec. 4.3 presents SSLRef, the fully-static security type language from which GSLRef is derived. Complete definitions, as well as the proofs of all the results stated in this chapter, can be found in §B. An interactive executable model of GSLRef is available online at [https://pleiad.cl/gradual-security/](https://pleiad.cl/gradual-security/)
4.2 Type-Driven Gradual Security Typing in Action

Static security type systems impose a burdensome all-or-nothing adoption model: all security types must be determined before the type system can check security. Even then, some secure programs have no statically-checkable type assignment, or may require substantial refactoring to satisfy the type checker. **Gradual security typing** addresses these shortcomings by enabling a programmer to incrementally add security information to the program, progressively introducing dynamic and static checks and guarantees.

Let us consider how gradual security typing can progressively introduce security guarantees and help detect and fix bugs in our first example from Sec. 4.1. Recall the problem with the program: `salary` is a high-security value, but `print` is a low-security channel. We can statically reflect these intentions:

```plaintext
1 let age = 31,
2 let salary = 58000_H,
3 let intToString : Int_L \rightarrow String_? = ...
4 let print : String_L \rightarrow Unit_? = ...
5 print(intToString(salary))
```

In practice the programmer just marks the value of `salary` and the input type of `print`: all omitted security annotations desugar to the `unknown` security label `?`. Under our gradual security semantics, this program type checks, but triggers a runtime check failure at line 5. If the highlighted annotations were omitted or `?`, then the program would check and run exactly as a simply-typed one, because it would not impose, and thus not enforce, any security invariants.

How do we repair this program? Simply adding more annotations cannot fix it. Case in point, adding a reasonable security annotation to line 3 escalates the runtime failure to a static type error.

```plaintext
3 let intToString: Int_L \rightarrow String = ...
```

If the security annotations are as intended, however, then the runtime error must be due to some behavioral bug in the program (*e.g.* the programmer might have intended to print the employee’s `age` instead).

**Reasoning with imprecision.** Our type-driven approach adapts prior concepts on gradual typing [109] to gradual security and its natural notion of subtyping.

In this model, the `unknown label` `?` represents imprecise security information. Precision `\sqsubseteq` is a partial order from more-precise labels to less-precise labels: static security labels are perfectly precise, *e.g.* `H \sqsubseteq H`, while `?` denotes utter imprecision, *e.g.* `H \sqsubseteq ?`. Precision extends *covariantly* to security types, *e.g.* `Int_H \rightarrow Int_L \sqsubseteq Int_L \rightarrow Int_?`, in contrast to subtyping.

The ordering on security labels `\preceq` consequently extends to *consistent ordering* `\preceq` on gradual labels. Consistent ordering preserves every order relation among precise labels (*e.g.* `⊥ \preceq T` and `T \not\preceq ⊥`), but mathematically, it is not an ordering relation (*e.g.* both `? \preceq T` and
Rather, it reflects consistent reasoning in the face of imprecise information: since we do not know what label \( ? \) represents, either static order is plausible. Consistent ordering induces an analogous notion of consistent subtyping, e.g. \( \text{int}_T \preceq \text{int}_? \) and \( \text{int}_? \preceq \text{int}_L \), which is not transitive, e.g. \( \text{int}_T \npreceq \text{int}_L \), so it is not a subtyping relation, but embodies imprecise reasoning about static subtyping [110]. An attacker or observer at level \( ol \) can now also observe values that have unknown security levels, as long as the dynamic security information about the value is observable at \( ol \). This is formally explained in § 4.3.

Flexibility As we have seen, GSL\textsubscript{Ref} lets programmers write statically secure programs by first writing the simply-typed version and progressively adding labels. But gradual typing also provides flexibility, so that safe programs that veer from the static type discipline can strategically revert to dynamic checking. GSL\textsubscript{Ref}’s type-driven approach provides this flexibility. Consider an example adapted from Fennell and Thiemann [39].

```plaintext
let infoH : Ref\textsubscript{L}Report\textsubscript{H} = ...  
let sendToFacebook : Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L} = ...  
let sendToManager : Ref\textsubscript{H}Report\textsubscript{H} \rightarrow_{\L} Unit\textsubscript{L} = ...  
let addPrivileged : Bool\textsubscript{H} \rightarrow (Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L}) \rightarrow (Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L}) =  
  fun isPrivileged worker report =>  
    if isPrivileged then report := !report + !infoH else ();  
    worker report  
let sendHi : Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L} = addPrivileged true sendToManager  
let sendLow : Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L} = addPrivileged false sendToFacebook
```

The program starts with the creation of a public reference to a private report, \( \text{infoH} \). It then defines two routines for submitting reports: \( \text{sendToFacebook} \) publishes data publicly, and \( \text{sendToManager} \) publishes data privately. The \( \text{addPrivileged} \) function decides dynamically whether to add high-security information to the sent report, and is used to implement the \( \text{sendHi} \) and \( \text{sendLow} \) functions. This code is secure, but SSL\textsubscript{Ref}, our static security system, cannot type check \( \text{addPrivileged} \) because of its dynamic choice.

Interestingly, GSL\textsubscript{Ref} can type check this program, thanks to a few well-placed \( ? \) labels (line 4), and it dynamically ensures that the program does not leak data. Case in point, the following gradually-typeable function is poised to leak private data:

```plaintext
let sendFail : Ref\textsubscript{L}Report\textsubscript{L} \rightarrow_{\L} Unit\textsubscript{L} = addPrivileged true sendToFacebook
```

but if called, GSL\textsubscript{Ref}’s dynamic security monitor signals an error when \( \text{sendToFacebook} \) dereferences the report, thereby preventing the leak.

Type-based reasoning in GSL\textsubscript{Ref}. Like prior work, GSL\textsubscript{Ref} supports smooth migration to static security and flexible programming idioms. Its most significant innovation is that GSL\textsubscript{Ref} retains the type-based reasoning power of static security typing.

\[^{3}\text{Security labels above function arrows track mutation effects (Sec. 4.3).}\]
Consider again the example mix function of Sec. 4.1. In GSLRef, the function body cannot violate the noninterference property implied by its type, just as in its fully static counterpart language SSLRef. In particular, the following definition is rejected statically as expected:

```ml
let mix : Int L → L Int H → L Int L = fun pub priv => if pub < priv then 1L else 2L
```

In fact, no function body can satisfy this type signature and use its second argument to determine the result. To do so, we must change the type signature, and with it the implied security invariants:

```ml
let mix : Int L → L Int H → L Int L? = fun pub priv => if pub < priv then 1L else 2L
```

The second argument now has statically unknown security. This definition is accepted statically because the function might respect the static security invariants of its clients. Consider two such clients, which only differ in the security level of the second argument:

<table>
<thead>
<tr>
<th>mix 1L 5H</th>
<th>mix 1L 5L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client 1</td>
<td>Client 2</td>
</tr>
</tbody>
</table>

Both type check because the security level of the second argument is consistent with the expected, unknown level. Client 2 returns 1L without incident, because its second argument is public, so applying mix does not leak private information. Client 1, however, signals a runtime security error: the function’s intended result would implicitly leak information from a private input, but the impending leak is trapped and reported. Treating static security levels as precise requirements rather than upper-bounds, and supporting imprecision, provides the same flexibility as the check-driven approach, as demonstrated in the reporting example above. The key difference is that dynamicity manifests as imprecision in a function’s static type, so precise types can preserve their static security interpretation. The interaction between types of different precision is transparently guarded by implicit runtime checks.

If we changed the type signature of mix to Int L → L Int H → L Int L?, making the return type imprecise, then the definition would type check as well. Nonetheless, GSLRef’s dynamic enforcement ensures that the returned value could never leak to a public channel, be it a variable or a heap location, because the result is dynamically secured.

The type-driven model lets programmers use type ascriptions to impose static security guarantees on code that is built from imprecisely typed components. Gradual typing automatically introduces dynamic checks to soundly enforce these invariants. Consider a function called smix that has a fully static signature but is implemented using the imprecisely-typed mix function:

```ml
@let mix : Int L → L Int H → L Int L? = fun pub priv => if pub < priv then 1L else 2L
@let smix : Int L → L Int H → L Int L? = fun pub priv => mix pub priv
```

Type-based reasoning about noninterference dictates that smix cannot reveal any information about its second argument (regardless of the actual security label of the second argument). For instance, consider the clients:
In GSLRef, both clients type check, but both fail at runtime! Client 2 fails because smix’s type dictates a strong noninterference property, independent of the client’s dynamic security levels. To see why, observe that smix accepts as second argument any integer value that has a security level no higher than H. When 5L is substituted in the body of smix, its runtime security information is upgraded to H. This new security level in turn strengthens the confidentiality of the value returned by mix, which contradicts the static return type of mix (L), hence resulting in a runtime error. This behavior preserves local type-based reasoning about the behavior of components, regardless of how they are composed.

To summarize, in GSLRef different gradual security types denote different security guarantees. Most importantly, the flexibility introduced by imprecise security types cannot be abused to violate the type-based noninterference guarantees imposed by static security types.

References and implicit flows. In the presence of mutable references, information-flow security faces the classic problem of implicit flows through the heap [32]. Consider the following program, adapted from Austin and Flanagan [8]:

```
fun x: Bool \Rightarrow
  let y: Ref L Bool = ref true
  let z: Ref L Bool = ref true
  if x then y := false
  else unit
  if !y then z := false
  else unit
  !z
```

This program attempts to downgrade the security of it’s input. A static security type system easily rejects it because the first branch of the first conditional (line 4) assigns a low-security reference under a high-security boolean condition. Indeed, in GSLRef this program is statically rejected as well.

This program is tricky for dynamic information flow monitors, however, and has inspired many approaches, e.g. [8][9][10][55]. Since gradual security typing includes both static and dynamic security checking, GSLRef must also address the challenge of dynamically detecting implicit flows. Consider the same program as above but with some imprecise annotations:

```
fun x: Bool \Rightarrow
  let y: Ref ? Bool = ref true
  let z: Ref L Bool = ref true
  if x then y := false
  else unit
  if !y then z := false
  else unit
  !z
```

This gradually-typed variant type checks because the reference bound to y now has an unknown security level. But if x is bound to true at runtime, then the program fails with an error at the assignment on line 4, because it cannot replace the contents of a reference in a manner that violates the security context H imposed by the conditional expression x. This
restriction, and its motivation, is analogous to the “no-sensitive-upgrade” approach of Austin and Flanagan [8].

Now suppose we make y’s type have unknown static security but force its initial contents to have high security, i.e.:

\[
\text{let } y : \text{Ref}_? \text{ Bool}_? = \text{ref true}_H
\]

Then at runtime the assignment on line 4 succeeds because the assignment on line 2 already refined y’s dynamic security to H, which satisfies the security context. Now if x is false then this program fails at the assignment on line 5, because z’s security level violates the dynamic security context introduced by branching on the contents of y.

To sum up, GSLRef ensures termination-insensitive noninterference, gradually, even in the presence of references.

### 4.3 Static Security Typing with References

This section introduces SSLRef, a higher-order static security-typed language with references, which serves as the static extreme of our gradual language. The language is a straightforward adaptation of prior information-flow security typing disciplines [58, 131, 39]. The most significant novelties include a syntax-directed type system and a dynamic semantics that tracks security levels but performs no security checks: the type system alone guarantees noninterference.

**Syntax.** Fig. 4.1 presents the syntax of SSLRef, at heart a simply-typed higher-order language with references: it includes booleans, functions, unit, mutable references, and type ascription. Each value and type constructor is annotated with a security label \(\ell \in \text{Label} \) with partial order \(\preceq\), where \(\top\) and \(\bot\) denote the greatest and least labels respectively. Function abstractions, and their corresponding types, are annotated with an additional security label called the latent security effect: we explain its static semantics below. Two forms arise only at runtime (highlighted in gray): mutable locations \(o\) and a protection term \(\text{prot}_\ell(t)\), which restricts the security effects of its subterm \(t\).

**Statics.** Fig. 4.1 also presents the type system of SSLRef, which is technically a type-and-effect system [48]. The judgment \(\Gamma; \Sigma; \ell_c \vdash t : S\) says that the term \(t\) has type \(S\) under type environment \(\Gamma\), store type \(\Sigma\), and security effect \(\ell_c \in \text{Label}\). A type environment \(\Gamma\) is a finite map from variables to types. A store type \(\Sigma\) is a finite map from locations to types. The security effect, sometimes called the program counter label [32], is a security label that denotes the least security level of those references that a given term may allocate or mutate [58]. The security effect prevents high-security computations—e.g. the branch of an if expression that is chosen based on a high-security Boolean—from leaking information by assigning to low-security references. An SSLRef source program \(t\) is well-typed if \(\vdash ; ; \bot \vdash t : S\).
• Rule (Sx) and rule (So) type variable and location references as usual. Simple values are also typed as usual, but their types inherit their labels from the values themselves (Sb/Su).

• Rule (Sλ) annotates the type of a function with the latent security effect of its body, as is standard for type-and-effect systems. The greatest (i.e. best) security effect can be inferred from the function body, but for simplicity this type system consults an explicit annotation $\ell'$. Finally label $\ell$ represents the security level of the function.

• Rule (Sprot) imposes a lower bound $\ell$ on the security effect of the subterm $t$. This restriction is captured by *stamping* the label $\ell$ onto the type [8]--e.g. $\text{Bool}_\ell \uplus \ell' = \text{Bool}_{(\ell \uplus \ell')}$, where $\ell \uplus \ell'$ represents the least upper-bound, or *join*, of security levels $\ell$ and $\ell'$.

• Rule (S⊕) types Boolean operations, yielding a result with the join of the operand security levels.
• Rule (Sapp) is mostly standard, but also enforces security restrictions. First, to prevent mutation-based security leaks, the operator’s latent effect \( \ell' \) must upper-bound its security level as well as the latent security effect of the entire expression. Both restrictions are captured with a single label comparison in the premise. Second, to prevent value-based security leaks, the security level of the entire expression must upper-bound the level \( \ell \) of the operator—this is done by stamping label \( \ell \) onto the type. Rule (Sapp) also appeals to the subtyping relation induced by ordering the security labels. Subtyping is driven by security labels: it is invariant on reference types, covariant on security labels, and contravariant on latent effects [98]:

\[
\begin{align*}
\ell \rightarrow \ell' & \quad \text{Bool}_{\ell} \ll: \text{Bool}_{\ell'} \\
\ell \rightarrow \ell' & \quad \text{Unit}_{\ell} \ll: \text{Unit}_{\ell'} \\
\ell \rightarrow \ell' & \quad \text{Ref}_{\ell} \ S \ll: \text{Ref}_{\ell'} \ S
\end{align*}
\]

\[
\begin{align*}
S_1 \ll: S_1 \quad S_2 \ll: S_2' \quad \ell_1 \rightarrow \ell_1' \quad \ell_2 \rightarrow \ell_2' \\
S_1 \ll: S_1 \quad S_2 \ll: S_2'
\end{align*}
\]

• Rule (Sif) incorporates the standard structure for a subtype discipline: the type of the expression involves the subtyping join \( \ll \) of its branches. To protect against explicit information flows, the expression type is stamped to incorporate the security level \( \ell \) of the predicate. Additionally, to prevent effect-based leaks, each branch is type checked with a security effect that incorporates the security level of the predicate.\(^4\)

• Rules (Sref) and (Sasgn), which perform write effects, are constrained by the security effect of the typing judgment to prevent leaks through the store.

Rule (Sref) honors the effect discipline by requiring the current security effect to lower-bound the security level of the stored value. The resulting reference has least security \( \bot \) because it is newly minted and cannot leak information: the type of the stored content is known and its security level prevents further prying.

Rule (Sasgn) ensures that the security level of the location and current security effect lower-bound the assigned value. The result of assignment has \( \bot \) security because \text{unit} cannot leak information.

Rule (Sderef) stamps the security level of the reference onto the resulting type combining the security level of the reference and the security level of the stored content.

• Finally, Rule (S::) is typical for ascription, requiring the ascribed type to be a supertype of the subterm’s type.

**Dynamics.** With fully static security typing, programs execute on a standard runtime with no additional security-enforcing machinery. Type safety—well-typed terms do not get stuck—is guaranteed by the underlying run-of-the-mill simple type discipline. However, to

\(^4\)Note that SSLRef does not have an explicit effect ascription form \( t :: \ell_c \) [13], but this can be encoded using the expression \((\lambda^{\ell_1} \cdot t) \bot \) unit \( \bot \).
establish the *soundness* of security typing—high-security computations have no effect on low-security observations—one must characterize computations and their resulting values with respect to their security levels. To this end, the SSL_{Ref} dynamic semantics explicitly *tracks* security labels as programs evaluate, but never *checks* them. The noninterference proof demonstrates that no such checks are required: static typing suffices. Tracking labels provides weak security guarantees that are exploited in the proof of the stronger noninterference result.

Fig. 4.2 presents the rules of the label-tracking dynamic semantics. The judgment \( t_1 \mid \mu_1 \stackrel{\ell_c}{\rightarrow} t_2 \mid \mu_2 \) says that a term \( t_1 \) and store \( \mu_1 \) step to \( t_2 \) and \( \mu_2 \) respectively, in security effect \( \ell_c \). Reduction of terms is specified using term frames \( f \):

\[
\begin{align*}
& t \mid \mu \quad \frac{t_i \mid \mu_i \stackrel{\ell_c}{\rightarrow} t \mid \mu}{t \mid \mu} \\
& \phantom{t \mid \mu} \quad \frac{(R \rightarrow)}{t_1 \mid \mu_1 \stackrel{\ell_c}{\rightarrow} t_2 \mid \mu_2} \\
& \phantom{t \mid \mu} \quad \frac{(Rf)}{t_1 \mid \mu_1 \stackrel{\ell_c}{\rightarrow} f[t_2] \mid \mu_2} \\
& \phantom{t \mid \mu} \quad \frac{(Rprot)}{t_1 \mid \mu_1 \stackrel{\ell_c}{\rightarrow} \text{prot}_{\ell_c}(t_2) \mid \mu_2}
\end{align*}
\]

Protection terms \( \text{prot}_{\ell_c}(t) \) control the current program counter label. Apart from \( \text{prot} \), all expressions propagate the current program counter to subterms. Rule (Rprot) upgrades \( \ell_c \) for the dynamic extent of \( t \). The resulting value is stamped with the protected label \( \ell \), in case the contents leak information to a context that lacks the confidentiality of \( \ell \). Values are stamped much like types: \( r_{\ell \gamma \ell'} = r_{(\ell \gamma \ell')} \). Protection terms do not exist in source programs: they are introduced by control operations, *i.e.* function calls and conditionals. The intuition is that
calling a function or destructing a Boolean of security level $\ell$ may leak information about the identity of the function or Boolean respectively. As such, the context of the resulting computation should communicate (via mutation) only with reference cells that have high-enough security, and the result of the computation is classified as well.\footnote{Zdancewic \cite{ref2} observes that \textit{e.g.} if $x$ \textbf{then} $\ell_L$ \textbf{else} $\ell_L$ leaks no information about Boolean $x : \text{Bool}_H$ so could be deemed low-security, but security type systems must be conservative for the sake of tractability.} Function calls ignore the operator’s latent effect $\ell'$, which promises the type system that the ensuing computation will not violate the stated confidentiality. However the operator’s security label determines the confidentiality of the ensuing computation.

When stored, a value inherits confidentiality from both the current security effect and the location itself. This behavior tracks both the confidentiality of the location and the induced security effect.

**Properties.** SSL$_{\text{Ref}}$ is type safe: we establish this result via a standard progress and preservation argument (§\textsection\textsection B.2). Since the runtime semantics includes no security checks, progress mirrors the corresponding argument for the underlying simple type discipline. To prove preservation, we must show that after each reduction step the resulting term still has the same security according to the typing rules of Fig. 4.1 modulo subtyping.

**Proposition 20** (Type Safety). If $\emptyset; \Sigma; \ell_c \vdash t : S$ then either

- $t$ is a value $v$, or

- for any store $\mu$ such that $\Sigma \vdash \mu$ and any $\ell'_c \not\ll \ell_c$, we have $t \mid \mu \xrightarrow{\ell'_c} t' \mid \mu'$ and $\emptyset; \Sigma'; \ell_c \vdash t' : S'$ for some $S' <: S$, and some $\Sigma' \supseteq \Sigma$ such that $\Sigma' \vdash \mu'$.

The store typing judgment $\Sigma \vdash \mu$ holds if and only if $\text{dom}(\mu) = \text{dom}(\Sigma)$ and $\emptyset; \Sigma; \ell_c \vdash \mu(o) : \Sigma(o)$ for all $o \in \text{dom}(\mu)$, $\ell_c \in \text{LABEL}$.

The most important property of a security-typed language like SSL$_{\text{Ref}}$ is the soundness of security typing, \textit{i.e.} that well-typed programs have no forbidden information flows. We formally state and prove noninterference using step-indexed logical relations (see §\textsection B.2.2). We do not include the definitions of the logical relations and noninterference statement here because proving that SSL$_{\text{Ref}}$ is secure is not the main focus of this work, and the full treatment of noninterference for the gradual language (§\textsection 4.5) subsumes them.

### 4.4 GSL$_{\text{Ref}}$: Type-Driven Gradual Security Typing

This section presents the static and dynamic semantics of GSL$_{\text{Ref}}$, and addresses its type safety and gradual guarantees. We show that GSL$_{\text{Ref}}$ enforces noninterference in Sec. 4.5.

The reader might (understandably!) wonder how some of the definitions presented in this section were conceived. This section largely appeals to intuition to justify these definitions,
but in practice they were obtained by following the Abstracting Gradual Typing methodology [44], which exploits principles of abstract interpretation [29] to systematically derive a gradual language from a static one. In fact, this work can be seen as a particularly challenging case study for AGT—which has led us to identify the limits of the AGT approach when applied to disciplines where type safety (i.e. “well-typed terms do not get stuck”) does not imply type soundness (i.e. “well-typed terms do not leak”). The gradual language obtained by a straightforward application of AGT is type safe, but does not ensure noninterference because of subtle interactions between security typing imprecision and heap-based flows. We discuss the key elements, pitfalls, and discoveries of this systematic derivation process in Sec. 4.6.

To aid the reader, Fig. 4.3 indicates where important terms, operations and relations are presented, along with their notation.
\textbf{U} ::= \text{Bool}_g \mid U \rightarrow^g U \mid \text{Ref}_g U \mid \text{Unit}_g \quad \text{(gradual types)}\\
\text{g} ::= \ell \mid ? \quad \text{(gradual labels)}\\
\text{b} ::= \text{true} \mid \text{false} \quad \text{(Booleans)}\\
\text{r} ::= b \mid (\lambda^x : U.t) \mid \text{unit} \mid o \quad \text{(base values)}\\
\text{v} ::= r_g \mid x \quad \text{(values)}\\
\text{t} ::= v \mid t \mid t \oplus t \mid \text{if} t \text{ then } t \text{ else } t \mid \text{ref}^U t \mid \text{!t} \mid t := t \mid \text{prot}_g(t) \mid t :: U \quad \text{(terms)}\\
\oplus ::= \land \mid \lor \quad \text{(operations)}\\

\textbf{(Ux)} \quad x : U \in \Gamma \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash x : U} \\

\textbf{(Ub)} \quad o : U \in \Sigma \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash o : \text{Ref}_g U} \\

\textbf{(Ua)} \quad \ell : U \in \Sigma \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash \ell : \text{Ref}_g U} \\

\textbf{(Uc)} \quad \lambda^x : U_1 \Rightarrow U_2 \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash \lambda x : U_1.t : U_2} \\

\textbf{(Uprot)} \quad g \in \Sigma \quad \Gamma = \Sigma \vdash t : U \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash \text{prot}_g(t) : U \gamma g} \\

\textbf{(Uapp)} \quad U_2 \subseteq U_1 \\
\frac{U_2 \subseteq U_1 \quad g \gamma g_c \leq g'}{\Gamma ; \Sigma ; g_c \vdash t_1 : U_1 \quad \Gamma ; \Sigma ; g_c \vdash t_2 : U_2} \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash t_1 : U_1 \quad \Gamma ; \Sigma ; g_c \vdash t_2 : U_2} \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash \text{if} t_1 \text{ then } t_2 : (U_1 \lor U_2) \gamma g} \\

\textbf{(Uasgn)} \quad U_2 \leq U_1 \\
\frac{U_2 \leq U_1 \quad g \gamma g_c \leq \text{label}(U_1)}{\Gamma ; \Sigma ; g_c \vdash t_1 : t_2 : \text{Unit}_g} \\

\textbf{(Uasgn)} \quad U_2 \leq U_1 \\
\frac{U_2 \leq U_1 \quad g \gamma g_c \leq \text{label}(U)}{\Gamma ; \Sigma ; g_c \vdash \text{ref}^U t : \text{Ref}_g U} \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash t : U} \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash U_1 \leq U_2} \\
\frac{}{\Gamma ; \Sigma ; g_c \vdash U_2 \leq U} \\

\textbf{Figure 4.4: GSL}_{\text{Ref}}: \text{Static Semantics}
4.4.1 Static Semantics

Fig. 4.4 presents the syntax and static semantics of GSLRef\textsuperscript{T}. A gradual security label \( g \in \text{GLabel} \) is either a static label \( \ell \) or the unknown label \( ? \), which represents any label whatsoever. Each value and gradual type constructor is now annotated with a gradual security label.

The typing judgment \( \Gamma; \Sigma; g_c \vdash t : U \) says that the term \( t \) has gradual type \( U \) under type environment \( \Gamma \), store environment \( \Sigma \), and gradual security effect \( g_c \). The typing rules are analogous to the static typing rules presented in Fig. 4.1 except that security labels, types, type functions and predicates are all replaced by their gradual counterparts. For instance, static label ordering \( \preceq \) is replaced with consistent label ordering \( \preceq \): 

\[
\begin{align*}
? & \preceq g \\
\ell_1 & \preceq \ell_2 \\
g & \preceq ? \\
\ell_1 & \preceq \ell_2
\end{align*}
\]

Intuitively, if consistent label ordering between two gradual labels holds, then it means that the static relation holds for some static labels represented by the gradual labels. It is always plausible in the presence of \(?\), since the unknown label represents any label. Similarly, subtyping is lifted to consistent subtyping \( \preceq \), whose definition is analogous to static subtyping, but using consistent label ordering:

\[
\begin{align*}
g \preceq g' & \quad \text{Bool}_g \preceq \text{Bool}_{g'} \\
g \preceq g' & \quad \text{Unit}_g \preceq \text{Unit}_{g'} \\
g \preceq g' & \quad U_1 \preceq U_2 \quad U_2 \preceq U_1 \\
g \preceq g' & \quad U_1' \preceq U_1 \quad U_2' \preceq U_2 \\
g \preceq g' & \quad g_1 \preceq g_2 \\
g \preceq g' & \quad g_1' \preceq g_2' \\
g \preceq g' & \quad U_1 \rightarrow g_1 \preceq g_2 \quad U_1 \rightarrow U_2 \\
g \preceq g' & \quad \text{Ref}_g \preceq \text{Ref}_{g'} \quad U_1 \preceq \text{Ref}_g \quad U_2 \preceq \text{Ref}_{g'} \\
g \preceq g' & \quad U_1' \preceq U_1 \quad U_2' \preceq U_2 \\
g \preceq g' & \quad U_1 \rightarrow g_1' \preceq U_2 \rightarrow g_2'
\end{align*}
\]

The label join and meet operators are replaced with consistent join and consistent meet respectively:

\[
\begin{align*}
\top \bar{\top} & = ? \bar{\top} \top = \top \\
\bot \bar{\bot} & = ? \bar{\bot} \bot = \bot \\
g \bar{\top} & = ? \bar{\top} g = ? \quad \text{if } g \neq \top \quad \ell_1 \bar{\top} \ell_2 = \ell_1 \bar{\top} \ell_2 \\
g \bar{\bot} & = ? \bar{\bot} g = ? \quad \text{if } g \neq \bot \quad \ell_1 \bar{\bot} \ell_2 = \ell_1 \bar{\bot} \ell_2
\end{align*}
\]

These operators recover precise label information when the unknown label interacts with the relevant boundary element (\( \top \) for \( \bar{\top} \), and \( \bot \) for \( \bar{\bot} \)), otherwise the result is always unknown. Intuitively, this is because any label \( \ell \) joined (resp. met) with \( \top \) (resp. \( \bot \)), yields \( \top \) (resp. \( \bot \)), so imprecise arguments do not perturb the results. But when the relevant boundary is not involved, then varying \( \ell \) can vary the results, a possibility that is captured by using the unknown label as result.

The join operators for subtyping and label ordering are replaced with consistent join \( \bar{\top} \): \(^6\)

\( ^6 \)In GSLRef\textsuperscript{T}, the \( o \) and \( \text{prot}_g(t) \) forms and typing rules merely serve to induce corresponding GSLRef\textsuperscript{ε} forms (Sec 4.4.2).
and consistent label join \( \tilde{\sqsubseteq} \) respectively:

\[
\begin{align*}
\text{Bool}_g \tilde{\sqsubseteq} \text{Bool}_{g'} &= \text{Bool}_{(g \tilde{\sqsubseteq} g')} \\
\text{Unit}_g \tilde{\sqsubseteq} \text{Unit}_{g'} &= \text{Unit}_{(g \tilde{\sqsubseteq} g')} \\
\text{Ref}_g U \tilde{\sqsubseteq} \text{Ref}_{g'} U' &= \text{Ref}_{(g \tilde{\sqsubseteq} g')} U \sqcap U'
\end{align*}
\]

\[
(U_{11} \xrightarrow{g_1'} U_{12}) \tilde{\sqsubseteq} (U_{21} \xrightarrow{g_2'} U_{22}) = (U_{11} \sqcap U_{21}) \xrightarrow{g_1 \tilde{\sqsubseteq} g_2} (U_{12} \tilde{\sqsubseteq} U_{22})
\]

\[
U \tilde{\sqsubseteq} U \text{ undefined otherwise}
\]

The consistent subtyping meet operator is defined dually (definition in §B.1.3).

Consistent subtyping join appeals to a gradual meet operator \( \sqcap \) on the referent types. This gradual meet arises because static subtyping is invariant for the contents of references, so static subtype join is only defined for references with equal referent types. The gradual meet operator can be understood as the gradual counterpart of a static type equality partial function \( \text{equate} \) (i.e. \( \text{equate}(S, S) = S \), undefined otherwise) \[44\]. Intuitively, if the \( \sqcap \) of two gradual entities is defined, then it means that they are possibly equal. For instance, \( H \sqcap ? = H \). Formally:

\[
\begin{align*}
\quad g \sqcap g &= g \\
\quad g \sqcap ? &= ? \sqcap g &= g \\
\quad \text{Bool}_g \sqcap \text{Bool}_{g'} &= \text{Bool}_{g \sqsubseteq g'} \\
\quad \text{Unit}_g \sqcap \text{Unit}_{g'} &= \text{Unit}_{g \sqsubseteq g'} \\
\quad \text{Ref}_g U \sqcap \text{Ref}_{g'} U' &= \text{Ref}_{g \sqsubseteq g'} U \sqcap U'
\end{align*}
\]

\[
U_1 \xrightarrow{g_1'} U_2 \sqcap U_1' \xrightarrow{g_1'} U_2' = (U_1 \sqcap U_1') \xrightarrow{g_1 \sqsubseteq g_1'} (U_2 \sqcap U_2')
\]

Finally, The SSLRef rules (Sapp) and (asgn) from Fig. 4.1 have compound premises that combine both label join and label ordering, e.g. \( \ell_c \gamma \ell \preceq \ell' \). One subtlety we discovered while applying the AGT methodology is that these premises lose precision when lifted compositionally: simply replacing join with consistent join and label ordering with consistent label ordering yields different results than when lifted in aggregate; we discuss this further in Sec. 4.6. Therefore rules (Uapp) and (Uasgn) use the consistent bounding predicate, which is defined algorithmically as: \( g_1 \gamma g_2 \preceq g_3 \iff g_1 \preceq g_3 \land g_2 \tilde{\preceq} g_3 \). Technically, we could have used this definition to split each premise, but treating the predicate atomically matters when we consider the dynamic semantics.

### 4.4.2 Dynamic Semantics

To present the dynamic semantics of GSLRef, we first define a reduction relation for an internal language GSLRef that directly mirrors GSLRef, except that all terms are augmented
with some evidence information that justifies why the term is well-typed according to the gradual type system. Just like AGT, during reduction steps, units of evidence are combined to form new evidence that supports type preservation between a term and its contractum. If the combination succeeds, reduction goes on; if the combination fails, a runtime error is raised. We first explain what evidence is, then how GSL₇Ref programs are elaborated with evidence information into GSL₇Ref, and finally how evidence is combined, yielding the GSL₇Ref reduction rules.

**Evidence for consistent judgments.** Evidence captures *why* a consistent judgment holds. To explain this concept, we begin with consistent judgments about security labels, then consider the more complex consistent judgments about types.

We use the metavariable \( \varepsilon \) to range over evidence, and write \( \varepsilon \vdash g_1 \lhd g_2 \) to say that evidence \( \varepsilon \) supports the plausibility that \( g_1 \lhd g_2 \) holds.

For instance, consider the consistent ordering judgment \( ? \lhd L \). Even though the unknown label generally denotes any security label, consistent ordering insists that this \( ? \) can only denote labels that are bounded from above by \( L \). Furthermore, this consistent ordering judgment yields no additional information about the right-hand side, which is already precise. We capture this learned information by representing evidence as a *pair of static label intervals*, noted \( \langle \ell_1, \ell_2 \rangle \). If \( \langle \ell_1, \ell_2 \rangle \vdash g_1 \lhd g_2 \) then \( \ell_1 \) and \( \ell_2 \) represent inferred range restrictions for \( g_1 \) and \( g_2 \) respectively. Therefore,

\[
\langle [\bot, L], [L, L] \rangle \vdash ? \lhd L
\]

By analogous reasoning, the consistent judgment \( H \lhd ? \) is initially justified by the evidence \( \langle [H, H], [H, \top] \rangle \), gaining precision about the right-hand side. Interval precision is defined as containment over intervals, i.e. \( [\ell_1, \ell_2] \subseteq [\ell'_1, \ell'_2] \) if and only if \( \ell'_1 \lhd \ell_1 \) and \( \ell_2 \lhd \ell'_2 \). Precision between interval pairs \( \langle \ell_1, \ell_2 \rangle \subseteq \langle \ell'_1, \ell'_2 \rangle \) is defined pointwise.

We represent evidence as pairs of intervals, rather than pairs of labels, essentially because pairs of labels are not precise enough to support gradual security. The formal rationale is involved, so we defer it to Sec. 4.6. For some intuition, though, consider the program

\[
\text{true}_H :: \text{Bool}_H :: \text{Bool}_? :: \text{Bool}_L.
\]

Evaluating it ultimately involves combining evidence for three consecutive judgments\(^7\), \( \varepsilon_1 \vdash ? \lhd H \), \( \varepsilon_2 \vdash H \lhd ? \), and \( \varepsilon_3 \vdash ? \lhd L \). The program should fail at runtime because an \( H \) security value should not be coercable to \( L \), so these three evidences should not compose. Unfortunately, pairs of labels are not precise enough to ensure this: they forget the intermediate step through \( H \). In contrast, pairs of label intervals retain enough precision to warrant the expected runtime failure.

To justify consistent judgments about types like consistent subtyping, we lift label evidence to *type evidence* \( \varepsilon \) by naturally lifting intervals to types: type constructors are now marked with label intervals instead of labels. For instance:

\[
\langle \text{Bool}_{[\bot, L]}, \text{Bool}_{[L, L]} \rangle \vdash \text{Bool}_? \lhd \text{Bool}_L
\]

\(^7\text{in a way that we make precise below.}\)
The syntax of evidence is as follows:

\[
E \in \text{GEType}, \quad i \in \text{Interval}, \quad \varepsilon \in \text{Evidence}
\]

\[
i ::= \langle \ell, \ell \rangle \quad \text{(intervals)}
\]

\[
E ::= \text{Bool} \mid E \rightarrow E \mid \text{Ref} \mid \text{Unit} \quad \text{(type evidences)}
\]

\[
\varepsilon ::= \langle E, E \rangle \mid \langle i, i \rangle \quad \text{(evidences)}
\]

Note that we use the same metavariable \(\varepsilon\) to represent both label evidence and type evidence, since which kind of evidence is meant is always clear from the context.

Terms with evidence. Each well-typed term of GSL\(_{\text{Ref}}\) is recursively elaborated into a GSL\(_{\varepsilon}\) term by decorating it with evidence for the consistent judgments used to establish its well-typedness.

The syntax of GSL\(_{\varepsilon}\) terms follows:

\[
t ::= v \mid \varepsilon t \odot \varepsilon t \mid \varepsilon t \oplus \varepsilon t \mid \text{if } \varepsilon t \text{ then } \varepsilon t \text{ else } \varepsilon t \mid \text{ref}^{\varepsilon} \varepsilon t \mid \varepsilon t \mid \varepsilon t ::= \varepsilon t \mid \text{prot}^{\varepsilon} \varepsilon g(\varepsilon t) \mid \varepsilon t \quad \text{(terms)}
\]

\[
r ::= b \mid (\lambda x : U.t) \mid \text{unit} \mid o \quad \text{(base values)}
\]

\[
u ::= r_g \mid x \quad \text{(raw values)}
\]

\[
v ::= u \mid \varepsilon u \quad \text{(values)}
\]

During reduction, the actual type of a subterm may evolve to a consistent subtype of the statically-determined type. For this reason, each term is augmented with evidence for their immediate sub-redexes (i.e. all subterms that have to be reduced to a value for computation to proceed), justifying why the subterms are consistent subtypes of the types demanded statically by the outer term constructor. For instance, in the term \(\varepsilon_1 t_1 \oplus \varepsilon_2 t_2\), \(\varepsilon_1\) justifies \(t_1\) being a consistent subtype of \(\text{Bool}_{g_1}\), the type deduced during type checking. In particular, \(t_1\) could be such a consistent subtype because it is a value that was ascribed type \(\text{Bool}_{g_1}\) using an explicit ascription. In fact, GSL\(_{\text{Ref}}\) ascriptions are represented simply as evidence-augmented terms \(\varepsilon t\) in GSL\(_{\varepsilon}\); the evidence \(\varepsilon\) holds all the computationally-relevant information about consistent subtyping. For instance, the GSL\(_{\text{Ref}}\) term \((10_L :: \text{Int}_? \downarrow) :: \text{Int}_H\) is translated to \(\varepsilon_2(\varepsilon_1 10_L)\), where \(\varepsilon_1 \vdash \text{Int}_L \GED \text{Int}_T\) and \(\varepsilon_2 \vdash \text{Int}_T \GED \text{Int}_H\).

Note that in addition, some terms carry extra evidences that are needed during reduction to justify type preservation. A conditional \(\varepsilon_1 t_1 \text{ then } \varepsilon_2 t_2 \text{ else } \varepsilon_3 t_3\) carries evidences \(\varepsilon_2\) and \(\varepsilon_3\) that justify that the type of each branch \(t_2\) and \(t_3\) is a consistent subtype of the type of the conditional expression. For instance, if \(U_2\) and \(U_3\) are the types of \(t_2\) and \(t_3\), respectively, then \(\varepsilon_2 \vdash U_2 \GED U_2 \lor U_3\), where \(U_1 \GED U_2 \lor U_3\) is the consistent lifting of the ternary static judgment \(T_1 \GED T_2 \lor T_3\). Similarly, a protection term \(\text{prot}_{\varepsilon_1 g_1} \varepsilon_2 g_2(\varepsilon_3 t)\) carries a security effect \(g_2\) (and its evidence \(\varepsilon_2\)), which represents the security effect of the subterm \(t\); specifically, \(g_2\) is the join of \(g_1\) and the current security effect.

Values are either raw values \(u\) or evidence-augmented raw values \(\varepsilon u\). The latter correspond to ascribed values \(v :: U\) in GSL\(_{\text{Ref}}\); the evidence \(\varepsilon\) confirms that the \(u\)'s type is a consistent subtype of the ascribed type \(U\).
Several terms—applications, references, assignment, and protection—have evidence in addition to that of their subterms. This extra evidence supports the consistent label ordering judgments of their corresponding typing rule, which relate to the current latent effect label. For instance, in the term \( \text{ref}^U \varepsilon t \), the evidence \( \epsilon' \) supports the consistent label ordering judgment \( g_c \preceq \text{label}(U) \). For uniformity, we overload the metavariable \( \varepsilon \) to denote both label and type evidence, since the difference is always clear from the context. Evidence attached to subterms is type evidence, and evidence attached to the security effect or to an expression symbol (\( @, \text{ref}, :=, \) or \( \text{prot} \)) is label evidence.

**Introducing evidence.** Fig. 4.5 presents rules for elaborating \( \text{GSL}_{\text{Ref}} \) source terms to evidence-augmented \( \text{GSL}_{\text{Ref}} \) terms. This elaboration is akin to a cast insertion translation \([109]\), but simpler because it inserts evidence uniformly \([44]\). Basically, each consistent label and type judgment in Fig. 4.4 is replaced by an evidence-computing partial function called an initial evidence operator (\( \mathcal{G} \)). An initial evidence operator computes the most precise evidence that can be deduced from a given judgment. For instance, given a consistent label ordering judgment \( g_1 \preceq g_2 \), the initial evidence for it is computed as follows:

\[
\mathcal{G}[g_1 \preceq g_2] = \text{intr}(\text{bounds}(g_1), \text{bounds}(g_2))
\]

The \textit{bounds} function produces the label interval that corresponds to a given gradual label, \( i.e.\)

\( \text{bounds}(?) = [\bot, \top] \) and \( \text{bounds}(\ell) = [\ell, \ell] \). The interior operator \( \text{intr} \) computes the smallest sub-intervals of its arguments that include all plausible orderings\(^8\). Given two intervals \( \ell_1 \) and \( \ell_2 \), \( \text{intr}(\ell_1, \ell_2) \) yields the greatest pair of sub-intervals \( [\ell_1', \ell_2'] \subseteq [\ell_1, \ell_2] \) such that each label \( \ell_1 \) in the interval \( \ell_1' \) is less than some label \( \ell_1 \) in \( \ell_2' \), and each label in \( \ell_2' \) is greater than some label in \( \ell_1' \). Formally:

\[
\text{intr}([\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}]) = ([\ell_{11}, \ell_{12} \land \ell_{22}], [\ell_{11} \lor \ell_{21}, \ell_{22}])
\]

This operation only changes the upper-bound of the lower interval and the lower-bound of the upper interval. The resulting intervals are well-defined because we only use this operator in \( \mathcal{G} \) after consistent label ordering is already known to hold.

Similarly, the initial evidence of a consistent judgment \( g_1 \gamma g_2 \preceq g_3 \) is computed as

\[
\mathcal{G}[g_1 \gamma g_2 \preceq g_3] = \text{intr}(\text{bounds}(g_1) \gamma \text{bounds}(g_2), \text{bounds}(g_3))
\]

This definition uses join of intervals, defined as \( [\ell_1, \ell_2] \gamma [\ell_1', \ell_2'] = [\ell_1 \gamma \ell_1', \ell_2 \gamma \ell_2'] \). For instance, the initial evidence for consistent judgment \( ? \gamma H \preceq ? \) is:

\[
\mathcal{G}[? \gamma H \preceq ?] = \text{intr}(\text{bounds}(?) \gamma \text{bounds}(H), \text{bounds}(?))
= \text{intr}([H, \top], [\bot, \top])
= ([H, \top], [H, \top])
\]
A generalized definition of $\cal G$, considering any consistent bounding judgment can be found in Fig. [B.3]. The definition of $\cal G$ extends naturally to compute the initial evidence for consistent subtyping judgments (the complete definition can be found in Fig. [B.9]). For instance, in the (Tif) rule, $\cal G[U_2 <: U_2 \lor U_3]$ computes the initial evidence for the consistent lifting of the fact that the type of the first branch is a subtype of the type of the entire conditional expression.

Rule $(T ::)$ recursively translates the subterm $t$, and the consistent subtyping judgment $U_1 <: U_2$ from $(S ::)$ is replaced with $\cal G[U_1 \leq U_2]$, which computes evidence $\varepsilon$ for consistent subtyping. This evidence is eventually placed next to the translated term $t'$. The ascription itself is erased because it does not affect the results of the computation.

Rule $(\text{Tapp})$ works similarly. Since $t_1$ is not constrained by a consistent subtyping judgment, the rule generates evidence for reflexive consistent subtyping: that the type is a consistent subtype of itself, $\cal G[U_{11} \rightarrow U_{12}]$. This seemingly vacuous evidence evolves nontrivially as a program reduces. Evidence for the judgment $\cal G[\gamma \; g \; \leq g']$ is computed as $\cal G[\; g \; \gamma \; \leq g']$, and placed next to the $@$ symbol, since it does not logically belong to any subterm.

The rest of the translation rules are analogous: each term is translated recursively, judgments are replaced by functions that determine the corresponding initial evidence, and the evidence for reflexive consistent subtyping $\cal G[\gamma]$ is associated to otherwise unconstrained types.

As an example, consider the GSLRef program $x := \text{true}_\gamma$, with current security effect $L$ and environment $\Gamma \triangleq x : \text{Ref}_\gamma \; \text{Bool}_H$. It elaborates to GSLRef as follows:

$$
\begin{align*}
\Gamma;.;L \vdash x \rightsquigarrow x : \text{Ref}_\gamma \; \text{Bool}_H & \quad \Gamma;.;L \vdash \text{true}_\gamma \rightsquigarrow \text{true}_\gamma : \text{Bool}_\gamma \\
\varepsilon_1 = \cal G[\text{Ref}_\gamma \; \text{Bool}_H] & \quad = \langle \text{Ref}_\gamma[\; L, \top] \; \text{Bool}_H, \text{Ref}_\gamma[\; L, \top] \; \text{Bool}_H \rangle \\
\varepsilon_2 = \cal G[\text{Bool}_\gamma \leq \text{Bool}_H] & \quad = \langle \text{Bool}_H[\; L, \bot], \text{Bool}_H[\; L, \bot] \rangle \\
\varepsilon_3 = \cal G[L \; ? \; \leq H] & \quad = \langle [L, H], [H, H] \rangle \\
\Gamma;.;L \vdash x := \text{true}_\gamma \rightsquigarrow \varepsilon_1 x := \varepsilon_2 \; \text{true}_\gamma : \text{Unit}_\bot
\end{align*}
$$

**Evolving evidence.** During reduction, evidence for consistent judgments must be combined to justify each reduction step. This combination is realized by two operators: consistent transitivity for label ordering and consistent join monotonicity.

The consistent transitivity operator $\circ \triangleq$ attempts to combine evidence for $g_1 \triangleq g_2$ and $g_2 \triangleq g_3$ to produce evidence for $g_1 \triangleq g_3$. Since $\triangleright$ is not in general transitive, $\circ \triangleq$ is partial, giving rise to runtime errors. For instance, both $H \triangleq ?$ and $? \triangleq L$ hold, but can they be combined to deduce that $H \triangleq L$? Of course not, otherwise high-confidence data could flow to low-confidence positions. To understand this failure of consistent transitivity, consider the initial evidence for these judgments, $\langle [H, H], [H, \top] \rangle$ and $\langle [\bot, L], [L, L] \rangle$. They cannot be combined because “they do not meet in the middle”, i.e. the middle intervals $[H, \top]$ and $[\bot, L]$ share no labels in common, which would justify transitivity. This intuition is formalized as
These type operators inherit properties from the label operators, produce evidence for \( \triangle \) joined labels. Consistent join monotonicity combines evidence for \( g \) of subtyping \( \circ \). In contrast to consistent transitivity, this operator is total.

\[
\langle \ell_1, \ell_2 \rangle \circ \leq \langle \ell_2, \ell_3 \rangle = \triangle \leq (\ell_1, \ell_2) \cap (\ell_2, \ell_3)
\]

where \([\ell_1, \ell_2] \cap [\ell'_1, \ell'_2] = [\ell_1 \vee \ell'_1, \ell_2 \wedge \ell'_2]\) if \( \ell_1 \vee \ell'_1 \leq \ell_2 \wedge \ell'_2 \)

and \( \triangle \leq ([\ell_1, \ell_2], [\ell'_1, \ell'_2], [\ell''_1, \ell''_2]) = \)

\[
\langle [\ell_1, \ell_2 \wedge \ell'_2 \wedge \ell''_2], [\ell_1 \vee \ell'_1 \vee \ell''_1] \rangle \quad \text{if} \quad \ell_1 \leq \ell'_2, \ell'_1 \leq \ell''_2, \ell_1 \leq \ell''_1
\]

The meet operator \( \cap \) denotes the intersection of two intervals. Given three intervals \( \nu_1, \nu_2, \nu_3 \), the \( \triangle \leq \) operator calculates, if possible, a pair of intervals \( \langle \nu'_1, \nu'_2 \rangle \subseteq \langle \nu_1, \nu_3 \rangle \) such that transitivity of label ordering through elements of \( \nu_2 \) is always plausible. Both operators are undefined if their side conditions do not hold.

The consistent join monotonicity operator \( \tilde{\gamma} \) reflects another facet of reasoning about consistent ordering relationships. Recall from Fig. 4.2 that during reduction, labels are sometimes joined, either for stamping values or for augmenting the security effect. Similarly, in \( \text{GSL}^\varepsilon \) evidence must be combined to support new consistent judgments that involve these joined labels. Consistent join monotonicity combines evidence for \( g_1 \leq g_2 \) and \( g_3 \leq g_4 \) to produce evidence for \( g_1 \vee g_3 \leq g_2 \vee g_4 \), the consistent lifting of the static judgment \( \ell_1 \vee \ell_3 \leq \ell_2 \vee \ell_4 \).

\[
\langle \nu_1, \nu_2 \rangle \tilde{\gamma} \langle \nu'_1, \nu'_2 \rangle = \langle \nu_1 \vee \nu'_1, \nu_2 \vee \nu'_2 \rangle
\]

In contrast to consistent transitivity, this operator is total.

Lifting these label operators to types is direct, albeit verbose, and can be found in § B.1.5. These type operators inherit properties from the label operators, e.g. consistent transitivity of subtyping \( \circ \leq \) is partial just like consistent transitivity of label ordering.

**Reduction rules.** Fig. 4.6 presents reduction semantics for \( \text{GSL}^\varepsilon_{\text{Ref}} \). Reduction operates on configurations \( C \), which consist of a term and a store, and a security effect. Specifically, \( t_1 | \mu_1 \xrightarrow{g} t_2 | \mu_2 \) denotes the reduction of term \( t_1 \) in store \( \mu_1 \) to term \( t_2 \) in store \( \mu_2 \) under security effect \( g \); the label evidence \( \varepsilon \) confirms that the runtime security effect is a sublabel of the label that was used statically to type check the original term (and is preserved by reduction).

The semantics is defined using two notions of reduction, \( \rightarrow \) and \( \rightarrow \leq \). The rules directly mirror the rules of \( \text{SSL}_{\text{Ref}} \) (Fig. 4.2), except that they also manage evidence at subexpression borders and combine evidence as needed to justify the preserved typing of the contractum. If evidence fails to combine, the program ends with an error.

A word about notation: to select evidences for sub-components of types, we use evidence inversion functions \[44\]. For instance, given a function type evidence \( \varepsilon \), \( \text{idom}(\varepsilon) \) (resp. \( \text{icod}(\varepsilon) \)) retrieves the type evidence of the domain (resp. co-domain). Similarly, \( \text{ilat} \) retrieves latent effect evidence from the evidence for a function type, and \( \text{iref} \) performs likewise for reference types. Finally, given type evidence \( \varepsilon \), \( \text{ilbl}(\varepsilon) \) yields the corresponding label evidence.

We now describe each reduction rule in turn.
\[\Gamma; \Sigma; g_c \vdash t \leadsto t': U\]

\[(Tx) \quad \frac{\Gamma(x) = U}{\Gamma; \Sigma; g_c \vdash x \leadsto x : U}\]

\[(Tb) \quad \frac{\Gamma; \Sigma; g_c \vdash b_g \leadsto b_g : \text{Bool}_g}{\Gamma; \Sigma; g_c \vdash b_g \leadsto b_g : \text{Bool}_g}\]

\[(Tu) \quad \frac{\Gamma; \Sigma; g_c \vdash \text{unit}_g \leadsto \text{unit}_g : \text{Unit}_g}{\Gamma; \Sigma; g_c \vdash \text{unit}_g \leadsto \text{unit}_g : \text{Unit}_g}\]

\[(T\lambda) \quad \frac{\Gamma; \Sigma; g_c \vdash (\lambda \alpha' x : U_1. t)_g \leadsto (\lambda \alpha' x : U_1. t')_g : U_1 \xrightarrow{g} U_2}{\Gamma; \Sigma; g_c \vdash t \leadsto t' : U_2}\]

\[(T\oplus) \quad \frac{\Gamma; \Sigma; g_c \vdash t_1 \leadsto t'_1 : \text{Bool}_g \quad \Gamma; \Sigma; g_c \vdash t_2 \leadsto t'_2 : \text{Bool}_g}{\Gamma; \Sigma; g_c \vdash t_1 \oplus t_2 \leadsto \varepsilon_1 t'_1 \oplus \varepsilon_2 t'_2 : \text{Bool}_g \equiv g_1 \oplus g_2}\]

\[(T\text{app}) \quad \frac{\Gamma; \Sigma; g_c \vdash t_1 \leadsto t'_1 : \text{Bool}_g \quad \varepsilon_1 = g^\gamma[\text{Bool}_g] \quad \varepsilon_2 = g^\gamma[\text{Bool}_g]}{\Gamma; \Sigma; g_c \vdash t_1 \text{app} \leadsto \varepsilon_1 t'_1 \rightarrow \varepsilon_2 t'_2 : \text{Bool}_g \equiv g_1 \oplus g_2}\]

\[(T\text{if}) \quad \frac{\Gamma; \Sigma; g_c \vdash t \leadsto t'_1 : \text{Bool}_g \quad g_c = g \equiv \gamma g \quad \Gamma; \Sigma; g_c \vdash t_2 \leadsto t'_2 : U_2 \quad \Gamma; \Sigma; g_c \vdash t_3 \leadsto t'_3 : U_3}{\Gamma; \Sigma; g_c \vdash \text{if } t \text{ then } t_1 \text{ else } t_3 \leadsto \varepsilon_1 t_1 \text{ else } \varepsilon_2 t_2 \text{ else } \varepsilon_3 t_3 : (U_2 \equiv U_3) \equiv \gamma g}\]

\[(T\text{assign}) \quad \frac{\Gamma; \Sigma; g_c \vdash t \leadsto t'_1 : \text{Ref}_g U \quad \varepsilon_1 = g^\gamma[\text{Ref}_g U] \quad \varepsilon_2 = g[U_2 \equiv U_1]}{\Gamma; \Sigma; g_c \vdash t_1 := t_2 \leadsto \varepsilon_1 t'_1 := \varepsilon_2 t'_2 : \text{Unit}_\perp}\]

\[(T\text{ref}) \quad \frac{\Gamma; \Sigma; g_c \vdash t \leadsto t' : U' \quad \varepsilon_1 = g[U' \equiv U] \quad \varepsilon_2 = g[g_c \equiv \text{label}(U)]}{\Gamma; \Sigma; g_c \vdash \text{ref} t \leadsto \text{ref} U \equiv \varepsilon_1 t'_1 : \text{Ref}_U}\]

\[(T\text{deref}) \quad \frac{\Gamma; \Sigma; g_c \vdash t \leadsto ! t' : \text{Ref}_g U \quad \varepsilon = g^\gamma[\text{Ref}_g U]}{\Gamma; \Sigma; g_c \vdash ! t \leadsto ! \varepsilon t' : U \equiv \gamma g}\]

where \(g^\gamma = g^{\equiv \gamma g}\) and \(g[U] = g[U \equiv U]\)

Figure 4.5: GSL$_{\text{Ref}}$: elaboration to GSL$_{\text{Ref}}$ terms
\[(r1) \quad \varepsilon_1(b_1)_{g_1} \oplus \varepsilon_2(b_2)_{g_2} \mid \mu \xrightarrow{\varepsilon g_c} \varepsilon_1 \wedge \varepsilon_2 \mid \mu \\xrightarrow{\varepsilon g_c} (\varepsilon_1 \vee \varepsilon_2)(b_1 [\oplus] b_2)_{(g_1 \top g_2)} \mid \mu \quad \xrightarrow{\varepsilon g_c} C \times (C \cup \{\text{error}\})\]

\[(r2) \quad \text{prot}_{\varepsilon_1} \varepsilon_2 g_2(\varepsilon_3 u) \mid \mu \xrightarrow{\varepsilon g_c} \varepsilon_3 \vee \varepsilon_1(\mu \vee g_1) \mid \mu \]

\[(r3) \quad \varepsilon_1(\lambda^x : U.t)_{g} \at \varepsilon_2 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \text{prot}_{\text{ibl}(\varepsilon_1)} g' \mid \mu & \text{if } \varepsilon' \text{ or } \varepsilon_2 \text{ are not defined} \\ \text{error} & \text{if } \varepsilon' \text{ is not defined} \end{cases} \]

where:

\[\varepsilon'_1 = \varepsilon_1 \vee \text{ibl}(\varepsilon_1) \circ \varepsilon_2 \circ \text{ilat}(\varepsilon_1)\]

\[\varepsilon'_2 = \varepsilon_2 \circ \text{idom}(\varepsilon_1)\]

\[g'_1 = (g_c \vee g)\]

\[(r4) \quad \text{if } \varepsilon_1 b_{g_1} \text{ then } t_2 \text{ else } t_3 \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \text{prot}_{\text{ibl}(\varepsilon_1)} b_{g_2}(\varepsilon_2 t_2) \mid \mu & \text{if } b = \text{true} \\ \text{prot}_{\text{ibl}(\varepsilon_1)} b_{g_3}(\varepsilon_3 t_3) \mid \mu & \text{if } b = \text{false} \end{cases} \]

where:

\[\varepsilon' = \varepsilon_1 \vee \text{ibl}(\varepsilon_1)\]

\[g' = g_c \vee g_1\]

\[(r5) \quad \text{ref}_{\varepsilon_2}^{U} u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} o_{=1} | \mu[o \mapsto \varepsilon'(u \vee g_c)] & \text{if } (\varepsilon \circ \varepsilon_2) \text{ is not defined} \\ \text{error} & \text{if } (\varepsilon \circ \varepsilon_2) \text{ is not defined} \end{cases} \]

where:

\[o \notin \text{dom}(\mu)\]

\[\varepsilon' = \varepsilon_1 \vee (\varepsilon \circ \varepsilon_2)\]

\[(r6) \quad !\varepsilon_1 o_{g} \mid \mu \xrightarrow{\varepsilon g_c} \text{prot}_{\text{ibl}(\varepsilon_1)} g' \mid \mu \xrightarrow{\text{ibl}(\varepsilon_1) v} \]

where:

\[\mu(o) = v\]

\[\varepsilon' = \varepsilon_1 \vee \text{ibl}(\varepsilon_1)\]

\[g' = (g_c \vee g)\]

\[(r7) \quad \varepsilon_1 o_{g} := \varepsilon_2 u \mid \mu \xrightarrow{\varepsilon g_c} \begin{cases} \text{unit}_{\bot} | \mu[o \mapsto \varepsilon'(u \vee (g_c \vee g))] & \text{if } \varepsilon' \text{ is not defined, or } \varepsilon \not\subseteq \text{ilbl}(\varepsilon) \text{does not hold} \\ \text{error} & \text{if } \varepsilon' \text{ is not defined, or } \varepsilon \not\subseteq \text{ilbl}(\varepsilon) \text{does not hold} \end{cases} \]

where:

\[\mu(o) = \varepsilon'' u'\]

\[\varepsilon' = (\varepsilon_2 \circ \text{ilat}(\varepsilon_1)) \vee ((\varepsilon_1 \vee \text{ibl}(\varepsilon_1)) \circ \varepsilon_3 \circ \text{ilat}(\varepsilon_1))\]

\[\varepsilon_1(\varepsilon_2 u) \rightarrow_{\text{error}} \begin{cases} \varepsilon_2 \circ \varepsilon_1 u & \text{if not defined} \\ \text{error} & \text{if not defined} \end{cases} \quad \xrightarrow{\text{EvTerm} \times (\text{EvTerm} \cup \{\text{error}\})}

Figure 4.6: GSL_{Ref}^\varepsilon: Dynamic semantics
• Rule \((r1)\) reduces a binary operation by joining the evidence of both operands to confirm that type preservation holds.

• Rule \((r2)\) reduces a protected value by stamping the security effect of the \texttt{prot} on the value and joining both evidences accordingly. We stamp \(g_1\) on the value to prevent it from leaking information to the current context when \(g_1\) is more confidential than the current security effect \(g_c\). Note that \(g_2\)—which represents the join between \(g_1\) and the current security effect \(g_c\)—is not used in this rule; it is used during reduction of the protected subterm.

• Rule \((r3)\) reduces a function application either to a protected body or to an error. The term reduces to an error if consistent transitivity fails to justify that the type of the actual argument is a consistent subtype of the formal argument type. This prevents an evident invalid information flow from the actual argument to the formal argument. Also, to prevent implicit flows via the store, an error is signaled if consistent transitivity fails to confirm that the latent effect of the function is greater than both the current security effect and that of the function. If the function application is valid, then the body is protected at the security level of the function. Label \(g'_1\) represents the security effect that is used to reduce the body, where \(\varepsilon'_1\) confirms that \(g'_1\) is no more confidential than the latent effect \(g'\).

• Similarly, rule \((r4)\) reduces a conditional expression by protecting the chosen branch. The resulting \texttt{prot} term is constructed using the dynamic information of the conditional.

• Rule \((r5)\) reduces a reference term to a fresh location. To prevent invalid implicit flows, the current security effect is stamped on the stored value. The term reduces to an error if consistent transitivity fails to confirm that the current security effect is lower than the statically-determined security level of the reference content \(U\).

• Rule \((r6)\) reduces a dereference term. In the dynamic semantics of SSL_{Ref}, dereferencing a store location causes the actual security of the location to be stamped on the resulting value. Here, the term reduces instead to a protected expression, which is equivalent but simplifies the proofs.

• Rule \((r7)\) is critical to ensuring noninterference. It can reduce to an error, and thereby preventing either implicit or explicit invalid flows, for three reasons:

1. the security level of the stored value should be no more confidential than the statically-determined security level of the reference content (explicit flow).

2. both the current security effect and the actual security level of the reference should be no more confidential than the static security level of the reference content (implicit flow).

3. the evidence of the current security effect must denote possible labels that are \textit{necessarily lower} than those denoted by the evidence of the stored value (implicit flow).

The third condition above, highlighted in gray in Fig. 4.6, is expressed with the lower-
The reduction rules for frames are presented in Fig. 4.7. Rule (Rf) reduces under term frames. Rule (Rprot) reduces a term to either a term or error, using → from Fig. 4.6. Similarly, Rules (Rh) and (Rproth) reduce the subterm using the evidence-combining reduction →<:, using evidence frames. Rule (Rprot) allows the protected subterm to step under a higher security level, which may be a sublabel of the one determined statically. Finally, rules (Rferr) and (Rproterr) propagate errors when the subterm reduces to an error, and rules (Rherr) and (Rprotherr) propagate errors when evidence fails to combine.

Figure 4.7: GSL Ref: Evaluation frames and reduction

bound comparison operator \([\leq]\) between evidences:

\[
\langle [\ell_1, \ell_2], \ell_3, \ell_4] \leq \langle [\ell'_1, \ell'_2], \ell'_3, \ell'_4] \iff \ell_3 \leq \ell'_3
\]

This check is necessary to ensure noninterference, and as explained in Sec. 4.6.3 it arises not from the type preservation argument, but from the noninterference argument. In Sec. 4.4.3 we illustrate each of these three scenarios.

The \(\rightarrow<:\) reduction rule uses consistent transitivity to combine, if possible, strings of evidence that accumulate on a raw value. It fails with a runtime error if the evidence cannot be combined. Sec. 4.4.3 presents an example of such a reduction.

Finally, contextual term reduction is specified using term frames \(f\) and evidence frames \(h\):

\[
\begin{align*}
\text{term} &::= h[\varepsilon] \\
\text{evidence} &::= \square \vartriangleleft \varepsilon t | \varepsilon u \vartriangleleft \square | \square \vartriangleleft \varepsilon t | \varepsilon u \vartriangleleft \square | \varepsilon \square | \varepsilon \varepsilon \square \\
\text{if} &\quad \text{then} \varepsilon t \text{ else } \varepsilon t | \text{!} \square | \square := \varepsilon \varepsilon t | \varepsilon u := \varepsilon \square | \text{ref}^t_1 \square
\end{align*}
\]

The reduction rules for frames are presented in Fig. 4.7. Rule (Rf) reduces under term frames. Rule (Rprot) reduces a term to either a term or error, using \(\rightarrow\) from Fig. 4.6. Similarly, Rules (Rh) and (Rproth) reduce the subterm using the evidence-combining reduction \(\rightarrow<:\). Rule (Rprot) allows the protected subterm to step under a higher security level, which may be a sublabel of the one determined statically. Finally, rules (Rferr) and (Rproterr) propagate errors when the subterm reduces to an error, and rules (Rherr) and (Rprotherr) propagate errors when evidence fails to combine.

82
4.4.3 Examples of Reduction

To illustrate the runtime semantics of GSLRef we first illustrate the three scenarios for which an assignment can fail, as per Rule (r7).

1. Consider the following program, which attempts to assign a high-confidentiality value into a low-confidentiality reference, and its translation (under security effect ⊥):

\[
\bot \vdash \text{ref}^\text{IntL} 20_L := (10_H :: \text{Int}_? \rightsquigarrow t : \text{Unit}_\bot
\]

Abbreviating \([\bot, \top]\) as \(?, \{\ell, \ell\} \text{ as } \ell, \langle i, i \rangle \text{ as } \langle i \rangle\), and \(\_\) for irrelevant evidence, we have:

\[
t \xrightarrow{\varepsilon_3 \circ \varepsilon_1} \varepsilon_2 10_H
\]

where \(\varepsilon_1 = \langle \text{Ref}_\bot \text{Int}_L \rangle \vdash \text{Ref}_\bot \text{Int}_L \preceq \text{Ref}_\bot \text{Int}_L, \varepsilon_2 = \langle \text{Int}_H, \text{Int}[H, \top] \rangle \vdash \text{Int}_H \preceq \text{Int}_?\). Then as \(\varepsilon_2 \circ \varepsilon_1 = \langle \text{Int}_H, \text{Int}[H, \top] \rangle \circ \langle \text{Int}_L \rangle\) is not defined, the term reduces to an error, as expected.

2. The following program attempts to update a low-confidentiality reference under a high-confidentiality security effect. Considering a security effect \(\bot\), a location \(\vdash o_\bot : \text{Ref}_\bot \text{Int}_L\), the program and its translation are:

\[
\bot \vdash \text{true}_H :: \text{Bool}_? \text{ then } o_\bot := 10_L \text{ else unit } \rightsquigarrow t : \text{Unit}_?,
\]

The conditional reduces to the first branch under a security effect \(H\).

\[
t \xrightarrow{\varepsilon_1} \text{prot}_H \varepsilon_1 H(\langle \varepsilon_2 o_\bot := \varepsilon_3 10_L \rangle)
\]

where \(\varepsilon_1 = \langle H, [H, \top] \rangle \vdash \bot \gamma H \preceq \bot \gamma ?\) and \(\varepsilon_2 = \langle \text{Ref}_\bot \text{Int}_L \rangle \vdash \text{Ref}_\bot \text{Int}_L \preceq \text{Ref}_\bot \text{Int}_L\). Also, because the static security effect of the assignment is ?, we have \(\varepsilon_3 = \langle [\bot, L], L \rangle \vdash ? \gamma \bot \preceq L\). Then as \(\langle \varepsilon_1 \circ \varepsilon_2 \rangle \circ \varepsilon_3 \circ \varepsilon_1 \circ \varepsilon_2 \rangle \neq \langle H, [H, \top] \rangle \circ \langle [\bot, L], L \rangle \circ \langle \bot, L \rangle\) is not defined, the term reduces to an error, successfully preventing an invalid implicit flow.

3. Consider a program fragment similar to the previous one, with security effect \(\bot\), a variable \(x : \text{Bool}_H\), and a location \(\vdash o_\bot : \text{Ref}_\bot \text{Int}_x\):

\[
\bot \vdash \text{if} x :: \text{Bool}_? \text{ then } o_\bot := 10_H \text{ else unit } \rightsquigarrow t : \text{Unit}_?,
\]

Suppose as well that \(\mu(o) = \varepsilon_2 0_?\), where \(\varepsilon_2 = \langle [\bot, T], [\bot, T] \rangle \vdash ? \gamma ?\) \(i.e.\) the stored number and heap cell have not acquired any security commitments yet). If \(x\) is \(\text{true}_H\), then the first branch is taken:

\[
t \xrightarrow{\varepsilon_1} \text{prot}_H \varepsilon_1 H(\langle o_\bot := \bot 10_H \rangle)
\]

where \(\varepsilon_1 = \langle H, [H, \top] \rangle \vdash \bot \gamma H \preceq \bot \gamma ?\). Since \(\varepsilon_1 \preceq \bot \gamma H \neq \bot\), the program reduces to an error. The problem is that if \(x\) were changed to \(\text{false}_H\), then the unchanged imprecisely labeled contents of \(o\) could be treated as low-security and thereby used to leak information about \(x\), using for instance a test of \(lo\) that conditionally assigns to some other low-security reference (for more see the example of Sec. 4.2 and Sec. 4.6.3).

83
Type-based reasoning. Finally, we revisit the mix and smix functions from Sec. 4.2, which illustrate how GSLRef preserves type-based reasoning principles in the gradual setting. The desugared GSLRef program follows:

\[
mix = (\lambda pub : L. (\lambda priv : ?. (\text{if } pub < priv \text{ then } 1_L \text{ else } 2_L) : L)_L)
\]

\[
smix = mix : L \rightarrow H \rightarrow L
\]

\[
smix:: L \rightarrow H \rightarrow L
\]

This program elaborates to the following GSLRef program:

\[
mix = (\lambda pub : L. (\lambda priv : ?. (\langle L \rangle pub < ? priv) \text{ then } 1_L \text{ else } 2_L)_L)_L
\]

\[
smix = \langle L \rightarrow L \rightarrow H \rightarrow L \rangle \text{ mix}
\]

\[
\langle H \rightarrow L \rangle \langle (L \rightarrow H \rightarrow L) smix \circ \langle[\langle\langle L,T\rangle\rangle)\rangle \rangle \rangle \rangle \langle L \rangle 1_L \circ \langle[\langle\langle L,T\rangle\rangle)\rangle \rangle \rangle \langle L, H \rangle 5_L
\]

A trace of the program is given in Fig. 4.8. As before, we abbreviate \([\perp, T]\) as \(?\), \([\ell, \ell]\) as \(\ell\), and \(\langle \ell, \ell \rangle\) as \(\langle \ell \rangle\). We omit the security effect of the reduction, which is always \(\langle \perp \rangle \perp\), as well as the heap, since the program is pure. The program fails as expected because low-security evidence is attached to the conditional term by a static ascription, which fails to combine with the high-security evidence of the value produced by the conditional. In other words, reduction fails to prove that \(H \ll L\).

4.4.4 GSLRef: Safety and Graduality

GSLRef satisfies a standard type safety property, whose proofs are in §C.1.5. More precisely, type safety is formulated for the evidence-augmented language GSLRef, and hence appeals to a corresponding typing judgment. As expected, this typing judgment, denoted \(\Gamma; \Sigma; \varepsilon g_c \vdash t : U\), is based on the GSLRef typing judgment. The only difference is that the security effect \(g_c\) is enriched with evidence \(\varepsilon\). This evidence accounts for how the runtime security effect can evolve to (consistently) lower levels than the security effect originally determined by the type system.

Proposition 21 (Type Safety). If \(\varnothing; \Sigma; \varepsilon g_c \vdash t : U\), and consider \(\mu\), such that \(\Sigma \vdash \mu\), then either:

- \(t\) is a value \(v\)
- \(t \mid \mu \xrightarrow{\varepsilon g_c} \text{error}\)
- \(t \mid \mu \xrightarrow{\varepsilon g_c} t' \mid \mu'\) and \(\varnothing; \Sigma' ; \varepsilon g_c \vdash t' : U\) for some \(\Sigma' \supseteq \Sigma\) such that \(\Sigma' \vdash \mu'\)

\footnote{For brevity, we only show the labels of base types, and omit latent effect annotations on pure functions.}

\footnote{The full definition of the GSLRef type system can be found in §B.1.3, the (straightforward) theorem that elaboration preserves typing is in §B.4.2.}
Definition 11 (Type and label precision).

\[
\begin{align*}
g & \subseteq ? \\
g & \subseteq g \\
g_1 & \subseteq g_2 \\
\text{Bool}_{g_1} & \subseteq \text{Bool}_{g_2} \\
g_1 & \subseteq g_2 \\
\text{Unit}_{g_1} & \subseteq \text{Unit}_{g_2} \\
U_{11} & \subseteq U_{21} \\
U_{12} & \subseteq U_{22} \\
g_{11} & \subseteq g_{21} \\
g_{12} & \subseteq g_{22} \\
g_{11} & \subseteq g_{21} \\
U_{11} & \rightarrow_{g_{11}} U_{12} \subseteq U_{21} \rightarrow_{g_{21}} U_{22} \\
g_{12} & \subseteq g_{22} \\
g_{12} & \subseteq g_{22} \\
\text{Ref}_{g_1} & U_1 \subseteq \text{Ref}_{g_2} U_2
\end{align*}
\]

Type and label precision are naturally lifted to term precision.

Proposition 23 (Static gradual guarantee). Suppose \( g_{c1} \subseteq g_{c2} \) and \( t_1 \subseteq t_2 \).
If $\emptyset; \emptyset; g_{c1} \vdash t_1 : U_1$ then $\emptyset; \emptyset; g_{c2} \vdash t_2 : U_2$ where $U_1 \subseteq U_2$.

This guarantee is best understood in reverse: if a \textit{simply-typed} program (where all security labels are ?) has a security-typed counterpart (where all security labels are precise), then $\text{GSL}_{\text{Ref}}$ statically accepts every intermediate security typing of that program: type checking is continuous with respect to security precision, so security information can be added in any order and at any rate \cite{113}.

Siek et al. \cite{113} also present a dynamic gradual guarantee, unfortunately, we have uncovered a tension between the dynamic gradual guarantee and noninterference. To ensure noninterference, the dynamic semantics of $\text{GSL}_{\text{Ref}}$ includes a specific runtime check (highlighted in gray in Fig. 4.6) which breaks the dynamic gradual guarantee. Dually, without this check, $\text{GSL}_{\text{Ref}}$ satisfies the dynamic gradual guarantee, but does not enforce noninterference for all programs. We discuss this subtlety in more detail in Sec. 4.6.3.

Nevertheless, an interesting conservative extension result holds for the dynamic semantics. Specifically, static $\text{GSL}_{\text{Ref}}$ terms never produce errors at runtime.

\textbf{Proposition 24} (Static terms do not fail). Let $\text{STATICTERM}$ be the static subset of $\text{GSL}_{\text{Ref}}$ terms, \emph{i.e.} with fully-static annotations, and $\text{STATICSTORE}$ the set of stores whose codomains are subsets of $\text{STATICTERM}$. Then consider $t \in \text{STATICTERM}$, $\mu \in \text{STATICSTORE}$, and $\varepsilon \ell_c$ such that $\varepsilon = g[\ell_c \sim \ell_c']$. If $\emptyset; \Sigma; \varepsilon \ell_c \vdash t : U$, then either $t$ is a value, or $t \mid \mu_s \xrightarrow{\varepsilon \ell_c} t'_s \mid \mu'_s$, with $t' \in \text{STATICTERM}$ and $\mu' \in \text{STATICSTORE}$.

\subsection{Prototype Implementation}

We have implemented $\text{GSL}_{\text{Ref}}$ in an interactive prototype available online at: https://pleiad.cl/gradual-security/.

The implementation, realized in Scala, supports all of $\text{GSL}_{\text{Ref}}$ plus let-bindings. Given a source program, it either shows the result of the elaboration to $\text{GSL}_{\text{Ref}}$, or reports a static type error. If the source program is well-typed, the evidence-augmented term can be explored interactively, either collapsing or expanding premises of its well-typedness, including evidences. The user can then reduce the term step by step, similarly to PLT Redex’s trace facility. At each step, the full typing derivation of the term can again be explored. The reduction shows how evidences are combined by consistent subtyping transitivity, eventually ending up in a value or a runtime security error.

All examples presented in this chapter are available as pre-loaded source examples.

\section{GSL_{\text{Ref}}: Noninterference}

This section establishes the type \textit{soundness} of $\text{GSL}_{\text{Ref}}$, \emph{i.e.} that gradual security types ensure noninterference. Noninterference formalizes the intuition that low-security observers of a computation cannot detect changes in high-security inputs. Therefore noninterference inher-
ently reflects a relationship between different runs of the same program with different inputs. We establish noninterference for GSL_{Ref} using logical relations \[58, 131\]. More precisely, because general references introduce nontermination, we apply step-indexed relations \[4\]. As standard, we focus on termination-insensitive noninterference: interference between two executions is only acknowledged when both terminate in values that are observably different. In line with prior work on gradual security \[35, 39\], we consider runtime check errors to be akin to non-termination, because in principle the semantics could deal with errors by diverging and directly reporting the error through a secure channel.

Observing values. The security type of a value dictates both an observation protocol and the clearance required to observe it. Consider a value \( \vdash v : U_1 \rightarrow g U_2 \), and an observer with security level \( ol \): Can \( ol \) observe the value? If so, what observations can it make? First, \( ol \) cannot make any observations if its security level does not subsume that of the function \( g \). If clearance is granted \( (g \lessapprox ol) \), then \( ol \) may make observations in accordance with the structure of \( v \)'s type: it may construct another value \( v' : U_1 \) and apply it to the function; the observations that \( ol \) can make of the result are then dictated by the type \( U_2 \vdash g \).

The predicate \( \text{obsVal}_{ol}(v) : U \), defined formally below, intuitively captures what it means for a value \( v \) of type \( U \) to be observable at \( ol \): \( ol \) must be consistently greater than the security label of \( U \). To account for the gradual security setting, we need to extend this intuitive notion in two ways. First, observation must deal with the potential for values to carry type ascriptions, such as \( v = \text{true}_H : \text{Bool} \). An observer at security level \( L \) must not observe the underlying high-security value. The key intuition is that the observation should ultimately be equivalent to applying the source language context \( \Box : \text{Bool}_L \text{then } \text{true}_L \text{else } \text{false}_L \) to the value, thereby asserting credentials and then using them. Doing so would trigger a runtime check error, which amounts to a non-observation. In GSL_{Ref}, \( v \) would be represented as an evidence value \( \varepsilon \text{true}_H \), where \( \varepsilon \) confirms that \( \text{Bool}_H \lessapprox \text{Bool}_L \). We capture the observability of the underlying value by defining the notion of observable evidence at a given observation level. Then, an evidence value \( v = \varepsilon u \) is observable if its label evidence \( \text{ilbl}(\varepsilon) \) is observable.

**Definition 12** (Observable evidence). Suppose observation level \( ol \) and an evidence judgment \( \varepsilon \vdash g \lessapprox g' \) for some \( g \) and \( g' \). For the evidence \( \varepsilon \) to be observable at \( ol \), it must be possible to confirm \( g \lessapprox ol \) using consistent transitivity of label ordering through \( g' \). Formally:

\[
\text{obsEv}_{ol}(\varepsilon) : g' \iff \varepsilon \circ \varepsilon \vdash g' \lessapprox ol
\]

Second, observation must account for dynamic security effect clearance: observation leaks a value from its context, so the observer must have the proper credentials. Recall that execution happens under a dynamic security effect \( g \) that, at runtime, can be consistently lower than the security effect originally determined by the type system. Therefore the dynamic security effect is accompanied by evidence \( \varepsilon \) that confirms that \( g \lessapprox g' \), where \( g' \) is the static security effect. Observation is allowed if such evidence is observable, i.e. \( g \lessapprox ol \).

Adding these two refinements of observability to the original notion of observable value yields the following definition.

**Definition 13** (Observable value). Given an observation level \( ol \), we define that a value \( v \),
typed as $U$, is observable as:

$$\text{obsVal}_{ol}(v) : U \iff g \preceq ol \land ((v = \varepsilon_1 u) \implies \text{obsEv}_{ol}(\text{ilbl}(\varepsilon_1)) : g) \quad \text{where} \quad g = \text{label}(U)$$

**Security logical relations.** We define logical relations between both computations and values in Figs. 4.9 and 4.10. The notions of related values and related computations are mutually recursive, as explained below. Note that the logical relations are only defined for pairs of GSLRef terms that have the same type $U$, so simple type safety ensures that the behaviors dictated by $U$ will produce defined behavior (including runtime error). To make the relations well-defined in the presence of nontermination, we index them on the number of steps $k$ that the observer $ol$ may take. If no inequivalent observations are made after $k$ steps, the terms are deemed equivalent. Ultimately we require that $ol$ observes equivalence for any arbitrary number of steps, which implies that nonterminating computations also respect the noninterference guarantees. This is the essence of step-indexing [4].

The definition of related values is presented in Fig. 4.9. We use notation $\hat{g}_i$ to denote the evidence-augmented security context $\varepsilon_i g_i$. The notation $\Sigma; g_c \vdash \langle \hat{g}_1, v_1, \mu_1 \rangle \approx^k_{ol} \langle \hat{g}_2, v_2, \mu_2 \rangle : U$ indicates that the triple of security context $\hat{g}_1$, value $v_1$ and store $\mu_1$, is related to the triple of dynamic security context $\hat{g}_2$, value $v_2$ and store $\mu_2$ at type $U$ for $k$ steps under store typing $\Sigma$ and static security context $g_c$ when observed at the security level $ol$. For two such triples to be related, four conditions must be satisfied:

1. The security effects must be related under security effect $g_c$, meaning they denote execution contexts that are either both above $ol$ (high-security), or both below (low-security). Formally, two security effects are related if their underlying evidences are either both observable or both not observable:

$$g_c \vdash \varepsilon_1 g_1 \approx_{ol} \varepsilon_2 g_2 \iff (\text{obsEv}_{ol}(\varepsilon_1) : g_c \land \text{obsEv}_{ol}(\varepsilon_2) : g_c) \lor (\neg \text{obsEv}_{ol}(\varepsilon_1) : g_c \land \neg \text{obsEv}_{ol}(\varepsilon_2) : g_c)$$

where $\varepsilon_1 \vdash g_1 \preceq g_c$.

2. The stores must be related for $k$ steps under store typing $\Sigma$, notation $\Sigma \vdash \mu_1 \approx^k_{ol} \mu_2$. This means that, for locations that are common to both stores $\Sigma$ the stored values are related at $j < k$ steps. Formally:

$$\Sigma \vdash \mu_1 \approx^k_{ol} \mu_2 \iff \forall g_c, \hat{g}_1, \varepsilon_1 \vdash g_1 \preceq g_c, g_c \vdash \hat{g}_1 \approx_{ol} \hat{g}_2, j < k, \Sigma \vdash \mu_1,$$

$$\forall o \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2), \Sigma; g_c \vdash \langle \hat{g}_1, \mu_1(o), \mu_1 \rangle \approx^j_{ol} \langle \hat{g}_2, \mu_2(o), \mu_2 \rangle : \Sigma(U)$$

In particular, stored values must be related at all related security effects $\hat{g}_1, \hat{g}_2$. This generality is necessary because all reference operations involve stamping the current security effect (and its evidence) onto the stored value, and doing so must preserve relatedness. For instance, two runs of a program can update a store location with different values under a high-security effect because both will be stamped high-security, and thus indistinguishable by a low-security observer $ol$.

---

11 For simplicity and without loss of generality, like Austin and Flanagan [8], we assume that a new reference in two related executions is allocated at the same address.
\[\Sigma; g_c \vdash \langle \hat{g}_1, \nu_1, \mu_1 \rangle \approx_k^{ol} \langle \hat{g}_2, \nu_2, \mu_2 \rangle : U \iff g_c \vdash \hat{g}_1 \approx_{ol} \hat{g}_2 \land \Sigma \vdash \mu_1 \approx_k^{ol} \mu_2 \land \forall; \Sigma; \hat{g}_1 \vdash \nu_1 : U \land (\text{obsVal}_{ol}(\nu_1) : U \lor \neg \text{obsVal}_{ol}(\nu_1) : U) \land \\
(\text{obsVal}_{ol}(\nu_1) : U \land \text{obsEv}_{ol}(\nu_1) : \mu_1) \implies \Sigma; g_c \vdash \text{obsRel}_{ol}^k(\hat{g}_1, \nu_1, \mu_1, \hat{g}_2, \nu_2, \mu_2) : U)\]

\[\Sigma; g_c \vdash \text{obsRel}_{ol}^k(\hat{g}_1, \nu_1, \mu_1, \hat{g}_2, \nu_2, \mu_2) : U \iff \text{rval}(\nu_1) = \text{rval}(\nu_2)\]

if \(U \in \{\text{Bool}_g, \text{Unit}_g, \text{Ref}_g, U'\}\)

\[\Sigma; g_c \vdash \text{obsRel}_{ol}^k(\hat{g}_1, \nu_1, \mu_1, \hat{g}_2, \nu_2, \mu_2) : U \iff \forall j \leq k, \forall U' = U'_1 \overset{g_{32}^{ol}}{\rightarrow} U'_2, \forall U''_1,
\forall g'_1, \forall g'_2 = \varepsilon_1 g'_1, \text{ where }\varepsilon_1 \vdash g'_1 \approx g'_2, \text{ s.t. } \hat{g}_1 \leq_{ol} \hat{g}_2,
\varepsilon_{11} \vdash U_1 \overset{g_{32}}{\rightarrow} U_2 \preceq U', \varepsilon_{12} \vdash U''_1 \preceq U'_1, \text{ and } \varepsilon_{31} \vdash g'_1 \gamma g_{32} \preceq g_{32}, \text{ we have:}
\forall \nu'_1, \mu'_1, \Sigma' \subseteq \Sigma', \Sigma'; g_c \vdash \langle \hat{g}_1, \nu_1, \mu_1 \rangle \approx_j^{ol} \langle \hat{g}_2, \nu_2, \mu_2 \rangle : U''_1, \text{ dom}(\mu_1) \subseteq \text{dom}(\mu'_1),
\Sigma'; g_c \vdash \langle \hat{g}_1, (\varepsilon_{11} \nu_1 \circ_{z_{31}} \varepsilon_{12} \nu'_1), \mu'_1 \rangle \approx_j^{ol} \langle \hat{g}_2, (\varepsilon_{11} \nu_2 \circ_{z_{32}} \varepsilon_{12} \nu'_2), \mu_2 \rangle : C(U'_2 \gamma g_{31})\]

**Figure 4.9:** Related values

\[\Sigma; g_c \vdash \langle \hat{g}_1, t_1, \mu_1 \rangle \approx_k^{ol} \langle \hat{g}_2, t_2, \mu_2 \rangle : C(U) \iff g_c \vdash \hat{g}_1 \approx_{ol} \hat{g}_2 \land \Sigma \vdash \mu_1 \approx_k^{ol} \mu_2 \land \forall \hat{g}_1, \text{ s.t. } \hat{g}_1 \leq_{ol} \hat{g}_2 \text{ and } \forall; \Sigma; \hat{g}_1 \vdash t_1 : U, \forall j < k, (t_1 | \mu_1 \vdash_{ol} \hat{g}_1 \rightarrow t'_1 | \mu'_1) \implies \exists \Sigma', \Sigma \subseteq \Sigma',
\Sigma' \vdash \mu'_1 \approx_k^{ol} \mu'_2 \land ((\text{irred}(t'_1) \land \text{irred}(t'_2)) \implies \Sigma'; g_c \vdash \langle \hat{g}_1, t'_1, \mu'_1 \rangle \approx_k^{ol} \langle \hat{g}_2, t'_2, \mu'_2 \rangle : U)\]

**Figure 4.10:** Related computations

3. The values must both have the same type \(U\) under an empty type environment and valid store type.

4. The values must be either both observable or both not observable. If the values are not observable, they are deemed equivalent. If they are observable, then they must be related at their specific type, as specified by the auxiliary relation \(\Sigma; g_c \vdash \text{obsRel}_{ol}^k(g_1, \nu_1, \mu_1, g_2, \nu_2, \mu_2) : U\), defined by case analysis on \(U\). If \(U\) is either \(\text{Bool}_g, \text{Unit}_g\) or \(\text{Ref}_g\), two values are related simply if their raw values are equal (raw strips away checking-related information such as labels and evidences). Two functions are related if their application to two related argument values, in related stores, for \(j \leq k\) steps, are related computations, as explained below.

The definition of related computations is presented in Fig. 4.10. First, two triples of security effect, term, and store are related computations for \(k\) steps at type \(U\) if the security effects and the stores are related, as defined previously. Second, the terms must have type \(U\) under
any observationally higher security effect $\hat{g}'$. We say $\hat{g}' = \varepsilon g'$ is observationally higher than $\hat{g} = \varepsilon g$, notation $\hat{g} \leq_{\mathit{ol}} \hat{g}'$ if $-\mathit{obsEv}_{\mathit{ol}}(\varepsilon) : g_{c} \Rightarrow -\mathit{obsEv}_{\mathit{ol}}(\varepsilon') : g'_{c}$, where $\varepsilon \vdash g \approx g_{c}$ and $\varepsilon' \vdash g' \approx g'_{c}$. For instance, in the static language it is the case that for any $\ell$, $H \leq_{\mathit{ol}} H \gamma \ell$, because by monotonicity of the join $H \not\ll_{\mathit{ol}} H \forall \ell \not\ll_{\mathit{ol}}$. Additionally, for any $j < k$, if both terms can be reduced for at least $j$ steps under security effect $\hat{g}'_{c}$, then the resulting stores should be related for the remaining $k - j$ steps. Finally, if the resulting terms are irreducible, they must be related values for the remaining $k - j$ steps at type $U$, as defined previously.

The logical relation relates computations that do not terminate as long as the stores are also related after $k$ steps.

**Noninterference.** Armed with these logical relations, we can state a semantics-driven notion of noninterference, and prove that well-typed terms of the internal language are sound with respect to it. The judgment $\Gamma; \Sigma; \hat{g} \vdash t : U$ says that term $t$ is *semantically well-typed*, meaning that it respects the security protocol $U$ for all observers, substitutions, stores, and steps $[4]$.

**Definition 14** (Semantic Security Typing).

\[
\Gamma; \Sigma; \hat{g} \vdash t : U \iff \forall \text{ol} \in \mathit{Label}, k \geq 0, \sigma_1, \sigma_2 \in \mathit{Subst} \text{ and } \mu_1, \mu_2 \in \mathit{Store}, \forall g_{c}, \hat{g} = \varepsilon g, \\
\varepsilon \vdash g \approx g_{c}, \text{ such that } \Sigma \vdash \mu_i \text{ and } \Gamma; \Sigma; g_{c} \vdash \langle \hat{g}, \sigma_1, \mu_1 \rangle \approx_{\mathit{ol}}^{k} \langle g, \sigma_2, \mu_2 \rangle, \text{ we have } \Sigma; g_{c} \vdash \langle \hat{g}, \sigma_1(t), \mu_1 \rangle \approx_{\mathit{ol}}^{k} \langle g, \sigma_2(t), \mu_2 \rangle : C(U)
\]

The definition above appeals to a notion of related substitutions. Indeed, the term $t$ may have free variables, indicating “input parameters”. The term is semantically well-typed if applying related substitutions (and stores) yields related computations at type $U$, for any number of steps $k$, and for any observer $\text{ol}$. Two substitutions are related if they map each variable in the term to related closed values:

**Definition 15** (Related Closed Values).

\[
\forall x \in \text{dom}(\Gamma). \Sigma; g_{c} \vdash \langle \hat{g}_1, \sigma_1(x), \mu_1 \rangle \approx_{\mathit{ol}}^{k} \langle g_2, \sigma_2(x), \mu_2 \rangle : \Gamma(x)
\]

Note that because a low-security observer equates all high-security values, the actual substitutions and stores can be wildly different, up to the strictures that the logical relation imposes on their types.

Finally, Security Type Soundness says that the syntactic type system enforces noninterference.

**Proposition 25** (Security Type Soundness).

\[
\Gamma; \Sigma; \hat{g} \vdash t : U \implies \Gamma; \Sigma; \hat{g} \vdash t : U
\]

\[\text{12This requirement is motivated by the proof, in order to obtain a stronger induction hypothesis (see \S B.5.6).}\]
4.6 Deriving GSL\textsubscript{Ref} with AGT (almost)

So far the presentation of GSL\textsubscript{Ref} has focused on describing the language as it is and its properties, without explaining how it came to be designed that way. Several definitions in both the static and dynamic semantics may seem to come out of nowhere, and hard to accept without further justification.

This work originated from our desire to apply the Abstracting Gradual Typing (AGT) methodology \cite{garcia2021abstracting} in a challenging setting: AGT has never been applied to a type discipline that denotes a relational property over multiple executions. Therefore, we have systematically derived GSL\textsubscript{Ref} from SSL\textsubscript{Ref} using AGT. This methodology, which starts from considering gradual types as abstractions of static types, drove the entire design of GSL\textsubscript{Ref}. Each algorithmic characterization from Sec. 4.4 is equivalent to its semantic definition, obtained using AGT and presented hereafter. These equivalences are proven in §B.4.5.

**Applying AGT to security typing.** As mentioned before, applying AGT ensures by construction that the derived gradual language is type safe and satisfies the gradual guarantees. In prior work, Garcia \textit{et al.} applied AGT to a pure language with security typing, and found the resulting language to satisfy noninterference \cite{garcia2020abstracting}. However, in this work, where the languages support mutable references, applying AGT to SSL\textsubscript{Ref} yielded a gradual language that violates noninterference! By applying AGT, we surely obtained a gradual language that was type safe and satisfied the gradual guarantees, but unfortunately, the crucial semantic property of security types was broken. In brief, we had to apply two refinements. The first was proposed in the AGT methodology, though not needed in prior work. The second is novel, but conflicts with the dynamic gradual guarantee.

This section reports on these wrinkles and refinements so that future efforts to apply AGT to rich type disciplines can build on our experience. In particular:

- Sec. 4.6.1 sets up the basics to derive the static semantics of GSL\textsubscript{Ref} with AGT, which was a successful endeavor. In the process, we identified one subtlety (about compositional lifting) that is worth highlighting.

- Sec. 4.6.2 explains the AGT approach to deriving the dynamic semantics of the gradual language. Here, we discover that evidence must use a more precise abstraction than the one used in the static semantics. While this possibility is briefly mentioned in \cite{garcia2021abstracting}, it was not necessary in other applications of AGT.

- Sec. 4.6.3 discusses a crucial point related to enforcing noninterference in the presence of references, and hence potential implicit flows. This observation led us to add an extra check to GSL\textsubscript{Ref}’s dynamic semantics. The check ensures noninterference, but breaks the dynamic gradual guarantee.
4.6.1 Deriving the Statics

Following the AGT approach, we give meaning to gradual security labels directly in terms of the original static security labels. The driving intuition is that the unknown label \(?\) represents any label whatsoever, while a gradual label \(\ell\) represents a single static security label. We formalize this with a concretization function.

**Definition 16 (Label Concretization).** \(\gamma : \text{GLABEL} \rightarrow \mathcal{P}(\text{LABEL})\)

\[
\begin{align*}
\gamma(\ell) &= \{ \ell \} \\
\gamma(?) &= \text{LABEL}
\end{align*}
\]

Concretization immediately induces the notion of precision, which orders the static information content of gradual labels from most to least:

**Definition 17 (Label Precision).** \(g_1 \sqsubseteq g_2\) if and only if \(\gamma(g_1) \subseteq \gamma(g_2)\).

In order to exploit AGT to gradualize SSL\textsubscript{Ref}, we also require an abstraction function to precisely summarize a set of static labels as a single gradual label (round hats \(\hat{x}\) denote sets of \(x\)):

**Definition 18 (Label Abstraction).** \(\alpha : \mathcal{P}(\text{LABEL}) \rightarrow \text{GLABEL} :\)

\[
\begin{align*}
\alpha(\{ \ell \}) &= \ell \\
\alpha(\emptyset) &\text{ is undefined} \\
\alpha(\hat{\ell}) &= ? \text{ otherwise}
\end{align*}
\]

The \(\gamma\) and \(\alpha\) functions are tightly connected by two properties that together form a Galois connection [29].

**Proposition 26 (\(\alpha\) is Sound and Optimal).** If \(\hat{\ell} \neq \emptyset\) then,

(i) \(\hat{\ell} \subseteq \gamma(\alpha(\hat{\ell}))\).

(ii) If \(\hat{\ell} \subseteq \gamma(g)\) then \(\alpha(\hat{\ell}) \sqsubseteq g\).

The meaning of gradual security types is derived from the meaning of gradual security labels. Therefore, we naturally define a Galois connection for gradual security types (see §B.1.3).

**Lifting predicates and functions.** Following AGT, we exploit the Galois connections to lift all predicates and functions over labels and types from SSL\textsubscript{Ref} to obtain the definition of their counterparts in GSL\textsubscript{Ref}. In essence, each gradual entity (label, type) represents some set of static entities, so a consistent predicate holds among gradual entities so long as the underlying static predicate could plausibly hold. For instance, consistent ordering on gradual labels is defined as follows:
Definition 19 (Consistent label ordering). \( g_1 \sim g_2 \iff \ell_1 \triangleleft \ell_2 \text{ for some } (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \).

Consistent ordering conservatively extends static label ordering because each static label, when treated as a gradual label, concretizes to a singleton set that contains only itself; conservative extension is central to the concept of graduality [113]. On the other hand, consistent ordering holds universally for the unknown label \(?\), since it concretizes to all possible static labels.

Similarly, the join of two gradual labels is defined by lifting static label join:

Definition 20 (Gradual label join). \( g_1 \uplus g_2 = \alpha(\{ \ell_1 \gamma \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \}) \)

The gradual join of two gradual labels is the best abstraction of the set of all plausible static joins. For more insight, recall its equational characterization in Sec. 4.4: the unknown label disappears when joined with \(\top\), while it otherwise survives all joins. This is an emergent property of lifting: we did not anticipate it.

Compositional vs. aggregate lifting. One unanticipated subtlety observed in Sec. 4.4 involves the compound premises of the (Sapp) and (Sref) rules, such as \( \ell_c \gamma \ell \triangleleft \ell' \). One might be tempted to lift this premise compositionally as \( g_c \uplus g \triangleleft g' \). But Garcia et al. [14] explicitly warn against blindly lifting static predicates compositionally: compositional lifting must be proven (for instance, they show that lifting their subtyping premises compositionally yields the same result as lifting them aggregately). Here it matters! Consider the definition induced by AGT:

Definition 21 (Consistent bounding).

\[
\overline{g_1 \gamma g_2} \trianglelefteq g_3 \iff \ell_1 \gamma \ell_2 \trianglelefteq \ell_3 \text{ for some } (\ell_1, \ell_2, \ell_3) \in \gamma(g_1) \times \gamma(g_2) \times \gamma(g_3)
\]

This definition is not equivalent to compositional lifting. For instance, the relation \( \overline{H \gamma ?} \trianglelefteq L \) holds, but we know that no static label \( \ell \) satisfies \( H \gamma \ell \triangleleft L \) (because \( H \gamma \ell \) must be at least as high as \( H \)) [12]. In fact, aggregate lifting becomes critical when we reason about combining such lattice relations in the dynamic semantics. To the best of our knowledge, this is the first instance of aggregate lifting affecting the application of AGT.

4.6.2 Deriving the Dynamics

Intrinsic terms are heavy notationally because they carry all type annotations, yielding to reduction rules that are hard to read. To alleviate this burden, we have chosen to present the dynamic semantics by reducing evidence-augmented terms, which are more lightweight

\footnote{To be honest, despite the warning of Garcia et al., we first overlooked the issue and applied compositional lifting, assuming it would hold. We then observed that the resulting design loses enough precision to miss some evident inconsistencies, with dramatic consequences for security.}
notationally, and establish a more direct connection with the traditional translational approach. The counterpart of this choice is that we had to present a translation from source GSLRef terms to evidence-augmented GSLRefε terms. Apart from this cosmetic difference, the central approach to reduction is the same: evidence is combined during reduction, producing either new evidence to support the plausibility of the contractum, or a runtime error if no evidence remains, thereby refuting type safety.

In essence, GSLRefε terms are intrinsic terms from which computationally irrelevant static annotations have been erased. Proofs of theorems about GSLRef’s dynamic semantics need these annotations, so they use intrinsic terms. § 3.5.3 formalizes the relationship between intrinsic terms and evidence-augmented terms by giving a translation from intrinsic terms to evidence-augmented terms. We show that, intrinsic terms can always be erased to GSLRef terms, and that the process can be reversed for well-typed GSLRefε terms. Furthermore, related intrinsic and GSLRef terms either reduce to related terms or yield errors. Therefore the theorems about intrinsic terms transfer to GSLRef terms.

**Reduction and consistent deductions.** All instances of combining evidence in the reduction rules are dictated by SSLRef’s type safety proof. This “reduction” applies reasoning steps with a computational flavor: it composes ≼ relations to deduce new ones, using both join monotonicity and order transitivity. Let us remember that evidence is represented by an abstraction of the possible static candidates. Which abstraction to use turns out to be a crucial decision in order to preserve noninterference, as discussed next.

**Problems with evidence as gradual labels.** The “natural” abstraction of sets of labels are gradual labels, as used in the static semantics. In fact, Garcia et al. [44] use the same abstraction to represent both runtime evidence and static gradual types; we initially followed suit. However, the first major subtlety we uncovered while deriving GSLRef’s dynamic semantics is that using gradual labels (and consequently, gradual types) for evidence yields a design that achieves both type safety and the gradual criteria, but violates noninterference!

This problem manifested in two parts of the noninterference proof. First, the noninterference proof relies on the associativity of consistent transitivity. However, consistent transitivity of label ordering is not associative if gradual labels are used to represent evidence. Recall the program true :: Bool4 :: Bool7 :: BoolL, introduced in Sec. 4.4.2, which we expect to fail at runtime, and which ultimately involves combining three consistent label ordering judgments: ε1 ⊢ ? ≼ H, ε2 ⊢ H ≼ ?, and ε3 ⊢ ? ≼ L. If we use a pair of gradual labels to represent evidence, eventually we have to calculate (ε1 o≺ ε2) o≺ ε3. But ε1 = ⟨?, H⟩, ε2 = ⟨H, ?⟩, and ε3 = ⟨?, L⟩, then ε1 o≺ ε2 = ⟨?, ?⟩ and ⟨?, ?⟩ o≺ ε3 = ⟨?, L⟩, so no runtime error is produced. Note that ε1 o≺ (ε2 o≺ ε3) fails as expected, because ε2 o≺ ε3 is not defined, but this is not the composition order that arises at runtime.

Second, the proof of noninterference relies on the observational completeness of the con-

---

14Note that associativity of cast composition is also critical for space-efficient semantics of gradual typing, e.g. Siek and Wadler [112]. We conjecture that associativity may be a fundamentally desirable property, and intend to pursue this question.
Lifting consistent lattice relations. We now explain how the definitions of consistent transitivity and join monotonicity are semantically justified. As discussed in Sec. 4.6.1, premises such as \( \ell \, \gamma \, \ell \leq \ell' \) must be lifted as aggregates. In fact, such a judgment is likely the consequence of similar deductions from earlier reduction steps. For instance \( \ell \) must

\footnote{One could design a gradual security language that uses label intervals instead of gradual labels right from the start, including in the static semantics. While this would unify the abstractions used in the statics and dynamics, it would yield a gradual type system that rejects more secure programs than GSLRef does. For instance, the program \((\text{if false} \; \ell \; \text{then} \; 1_H \; \text{else} \; 2_L) :: \ell \) is accepted and runs without errors in GSLRef. But if we use intervals in the static semantics, then the security level of the conditional expression which boils down to the join between \( \?, \? \) and \( L \) would be \([L, H] \), therefore the program would be rejected statically. Applying a \( ? \) ascription to \( 1_H \) would fix this program.}

\[ \text{Lemma 27. Suppose } \varepsilon_1 \vdash g_1' \preceq g_1 \text{ and } \varepsilon_2 \vdash g_2' \preceq g_2 \text{ such that } \varepsilon_1 \, \gamma \, \varepsilon_2 \vdash g_1' \, \gamma \, g_2' \preceq g_1 \, \gamma \, g_2. \]

Then \( \neg \text{obsEv}_\text{al}(\varepsilon_1) : g_1 \lor \neg \text{obsEv}_\text{al}(\varepsilon_2) : g_2 \iff \neg \text{obsEv}_\text{al}(\varepsilon_1 \, \gamma \, \varepsilon_2) : g_1 \, \gamma \, g_2 \).

The analogous static lemma, i.e. \( \neg \text{obsEv}_\text{al}(\varepsilon_1) : \ell_1 \lor \neg \text{obsEv}_\text{al}(\varepsilon_2) : \ell_2 \iff \neg \text{obsEv}_\text{al}(\varepsilon_1 \, \gamma \, \varepsilon_2) : \ell_1 \, \gamma \, \ell_2 \), holds trivially by the very definition of the join, but this property fails to hold in the presence of the unknown label. Suppose \( \varepsilon_1' \vdash H \preceq \? \) and \( \varepsilon_2' \vdash ? \preceq \? \). If we use a pair of gradual labels to represent evidence, then \( \varepsilon_1' = (H, \?), \varepsilon_2' = (?), \text{ and } \varepsilon_1' \, \gamma \, \varepsilon_2' = (? , ?) \) losing information about \( H \). But \( \neg \text{obsEv}_L((H, \?)) : ? \) and \( \text{obsEv}_L((?, ?)) : ? \), therefore invalidating the lemma.

Representing evidence as intervals. These observations forced us to seek a more precise abstraction whose composition (through consistent transitivity) is associative and preserves the observational completeness of consistent join. Since it suffices to know whether the upper- and lower-bounds of the plausible static labels overlap to deduce the plausibility of consistent ordering, intervals seem to be a fitting abstraction. Indeed, this abstraction is sufficiently precise to guarantee the desired properties.

Definition 22 (Interval Concretization). \( \gamma_i : \text{INTERVAL} \to \mathcal{P}(\text{LABEL}) \), where \( \text{INTERVAL} = \{[\ell_1, \ell_2] \in \text{LABEL}^2 \mid \ell_1 \ll \ell_2\} \)

\[ \gamma_i([\ell_1, \ell_2]) = \{\ell \mid \ell \in \text{LABEL}, \ell_1 \ll \ell \ll \ell_2\}. \]

Definition 23 (Interval Abstraction). \( \alpha_i : \mathcal{P}(\text{LABEL}) \to \text{INTERVAL} \)

\( \alpha_i(\emptyset) \) is undefined \( \alpha_i(\{\ell_1\}) = [\ell_1, \gamma \ell_1] \) otherwise

With evidence based on intervals, \( \varepsilon_1 \circ^\varepsilon \varepsilon_2 \circ^\varepsilon \varepsilon_3 \) and \( \varepsilon_1 \circ^\varepsilon (\varepsilon_2 \circ^\varepsilon \varepsilon_3) \) are equivalent. Back to the example, now \( \varepsilon_1 = ([\bot, H], [H, H]), \varepsilon_2 = ([H, H], [H, T]) \) and \( \varepsilon_3 = ([\bot, L], [L, L]) \), then \( \varepsilon_1 \circ^\varepsilon \varepsilon_2 = ([\bot, H], [H, T]) \). Because \( ([\bot, H], [H, T]) \circ^\varepsilon \varepsilon_3 \) is undefined, a runtime error is raised, avoiding the breach of noninterference. Also, the observational-monotonicity of the join is preserved. Now \( \varepsilon_1' = ([H, H], [H, T]) \) and \( \varepsilon_2' = ([\bot, T], [\bot, T]) \), then \( \varepsilon_1' \, \gamma \, \varepsilon_2' = ([H, T], [H, T]) \) and now \( \neg \text{obsEv}_L((H, T), [H, T])) : ? \) as expected.
be some lattice expression $F(\vec{l}_i)$ comprising joins (and meets) of source program labels $\vec{l}_i$. Therefore, to mirror static type safety reasoning steps at runtime, and catch inconsistencies if they arise, we must generalize each ordering premise in a derivation and consider it as some lattice relation $F_1(\vec{l}_i) \ll F_2(\vec{l}_j)$. The notion of evidence must consequently account for the plausibility of consistent lattice relations:

$$\langle \ell_1, \ell_2 \rangle \vdash F_1(\vec{g}_i) \ll F_2(\vec{g}_j)$$

The definitions of consistent join monotonicity and consistent transitivity then follow directly from AGT by consistent lifting.

**Definition 24** (Consistent transitivity for label ordering).

$$\circ_{\ll} : \text{INTERVAL}^2 \times \text{INTERVAL}^2 \to \text{INTERVAL}^2$$

$$\langle \ell_{11}, \ell_{21} \rangle \circ_{\ll} \langle \ell_{22}, \ell_{32} \rangle = \alpha_{\ll}^2(\{\langle \ell_1, \ell_3 \rangle \in \gamma_{\ll}(\langle \ell_{11}, \ell_{32} \rangle) \mid \exists \ell \in \gamma_{\ll}(\langle \ell_{11}, \ell_{22} \rangle) \prec \ell \land \ell \ll \ell_3\})$$

Consistent transitivity produces evidence for all plausible instances of consistent ordering that can be deduced using transitivity from the plausible instances of ordering represented by the two inputs. By design, $\alpha_{\ll}^2(\emptyset)$ is undefined, so consistent transitivity is also undefined if no plausible pairings remain to support a deduction.

**Definition 25** (Consistent join monotonicity).

$$\varepsilon_1 \varepsilon_2 = \alpha_{\ll}^2(\{\langle \ell_1, \ell_2 \rangle \mid \exists \langle \ell_{11}, \ell_{12} \rangle \in \gamma_{\ll}(\varepsilon_1), \langle \ell_{21}, \ell_{22} \rangle \in \gamma_{\ll}(\varepsilon_2), \ell_1 = \ell_{11} \prec \ell_{21}, \ell_2 = \ell_{12} \prec \ell_{22}, \ell_1 \ll \ell_2\})$$

Consistent join monotonicity is analogous, but note that due to lattice and interval properties, consistent join monotonicity is really a total function. Also, the $\ell_1 \ll \ell_2$ condition is superfluous; we present the definition in this form to preserve the general structure of consistent deduction definitions.

The algorithmic characterizations from Sec. 4.4.2 are equivalent to the above definitions. More importantly, we can prove that these operators indeed yield valid evidence for the combined consistent judgments.

**Proposition 28.** Suppose $\varepsilon_1 \vdash F_{11}(\vec{g}_i) \ll F_{12}(\vec{g}_j)$ and $\varepsilon_2 \vdash F_{21}(\vec{g}_i) \ll F_{22}(\vec{g}_j)$

Then $\varepsilon_1 \varepsilon_2 \vdash F_{11}(\vec{g}_i) \gamma F_{21}(\vec{g}_i) \ll F_{12}(\vec{g}_j) \gamma F_{22}(\vec{g}_j)$

**Proposition 29.** Suppose $\varepsilon_1 \vdash F_{1}(\vec{g}_i) \ll F_{2}(\vec{g}_j)$ and $\varepsilon_2 \vdash F_{2}(\vec{g}_j) \ll F_{3}(\vec{g}_k)$.

If $\varepsilon_1 \circ_{\ll} \varepsilon_2$ is defined, then $\varepsilon_1 \circ_{\ll} \varepsilon_2 \vdash F_{1}(\vec{g}_i) \ll F_{3}(\vec{g}_k)$

**From labels to types.** Finally, in addition to reasoning about consistent label ordering, the dynamic semantics must track and check the plausibility of consistent subtyping. Since (consistent) subtyping is induced by (consistent) ordering, the reasoning in question arises by lifting the same constructions to gradual security types, consistent subtyping, and consistent subtyping join and meet.
Just as we extend gradual labels $g$ to gradual security types $U$ (e.g. $\text{Int}_g$) in the source language, so do we extend label intervals $ı$ point-wise to type intervals $E$ (e.g. $\langle\text{Int}_1, \text{Int}_2\rangle$) and corresponding notions of evidence for consistent subtyping $\varepsilon$ (e.g. $\langle\text{Int}_1, \text{Int}_2\rangle$), which represent sets of pairs of candidates for plausible subtyping. We introduce evidence judgments $\varepsilon \vdash U_1 \leq U_2$ to associate runtime evidence with particular consistent subtyping judgments. The entire development mirrors the one for labels, and does not convey any new insights (see §B.4.1).

### 4.6.3 Policing Dynamic Heap Updates

Although adopting label intervals for evidence of consistent label judgments addressed some aspects of the noninterference proof, this refinement alone is not sufficient.

To illustrate the remaining problem, recall the example of implicit flows from Sec. 4.2, in particular the second version of the example, which has some missing static annotations.

```plaintext
1 fun x: Bool_H =>
2   let y: Ref Bool_H = ref true_H
3   let z: Ref Bool_L = ref true_L
4   if x then y := false_H else unit
5   if !y then z := false_L else unit
6   !z
```

This program is accepted statically and also runs without errors: if $x$ is $\text{true}_H$ then the program reduces to $\text{true}_L$, and if $x$ is $\text{false}_H$ it reduces to $\text{false}_L$: a clear breach of noninterference!

To understand the problem, consider what happens for the different values of $x$. When $x$ is $\text{true}_H$ the assignment in line 4 under security effect $H$ is valid, because $H \preceq ?$. In that moment we know that the security level of the content of $y$, must be higher than $H$. But when $x$ is $\text{false}_H$, in line 5 we assume that the security level of the content of $y$ is lower than $L$. In other words, under supposedly-related executions we get contradictory evidence for $y$. Notice that in the assignment at line 4, the judgment $H \preceq ?$ holds, but so does its negation $H \npreceq ?$. To preserve noninterference, we must ensure that its negation never holds.

To recover noninterference, we add an extra check to the assignment reduction rule $(r7)$ from Fig. 4.6:

\[
\begin{align*}
\varepsilon_{1g} &:= \varepsilon_3 \parallel \mu \xrightarrow{\varepsilon_{g_3}} \\
&\begin{cases}
\text{unit}_L & | \mu[o \mapsto \varepsilon'(u \mapsto \gamma(g, c \mapsto \gamma g))] \\
\text{error} & \text{if } \varepsilon' \text{ is not defined, or } \varepsilon \not\leq \text{dbl}(\varepsilon'') \text{ does not hold}
\end{cases}
\end{align*}
\]

where $\mu(o) = \varepsilon''u'$. The highlighted check ensures that if the security effect is not observable, then the content of the heap to be replaced must also be not observable\(^{16}\). This concept is formalized in the following lemma, which is used in the noninterference proof:

\(^{16}\)This check is analogous to the no-sensitive-upgrade check introduced by Austin and Flanagan [8], taken to the gradual context, and hence involving unknown labels, evidences and consistent judgments.
Lemma 30. Consider $\varepsilon_1 \vdash g'_1 \approx g_1$ and $\varepsilon_2 \vdash g'_2 \approx g_2$. Then $\neg \text{obsEv}_{ol}(\varepsilon_1) : g_1 \land \varepsilon_1 \preceq \varepsilon_2 \Rightarrow \neg \text{obsEv}_{ol}(\varepsilon_2) : g_2$.

With the additional check, if $x$ is $\text{true}_H$, the program fails at runtime, preserving noninterference.

The necessity of the check shows up in the noninterference proof for the if case. When two computations have related non-observable conditionals, the booleans can be different. This may lead to two related computations that reduce different branches under a high-security context. At that point, we must enforce that those different executions only write high-security values to the heap. In other words, as long as both executions reduce under high-security contexts, their executions can desynchronize only on private information. Formally, the following lemma should hold:

Lemma 31. Consider $\emptyset; \Sigma; \varepsilon g_c \vdash t : U$, $g'_c$ and $\mu$ such that, $\varepsilon \vdash g_c \approx g'_c$, $\neg \text{obsEv}_{ol}(\varepsilon) : g'_c$ and $\Sigma \vdash \mu$, and $\forall k > 0$, such that $t \parallel \mu \xrightarrow{\varepsilon g_c \vdash k t'} \parallel \mu'$,

1. $\forall o \in \text{dom}(\mu') \setminus \text{dom}(\mu), \neg \text{obsVal}_{ol}(\mu'(o)) : U$.
2. $\forall o \in \text{dom}(\mu') \cap \text{dom}(\mu)$ where $\mu'(o) \neq \mu(o)$,
   (a) $\neg \text{obsVal}_{ol}(\mu(o)) : U$, and
   (b) $\neg \text{obsVal}_{ol}(\mu'(o)) : U$.

Without the additional check in rule ($r7$), we cannot prove (2.a): before updating a reference, the current content should be non observable. And as we can see in the example above, without the check, the reference before the assignment would be observable, hence breaking the Lemma.

In its current formulation [44], AGT derives the dynamic semantics of the gradual language from the type safety argument of the static language. Here, we are facing a typing discipline in which type safety does not imply type soundness (i.e. noninterference), and hence, the methodology falls short of naturally preserving that property. This suggests that extending AGT to ensure type soundness of the derived gradual language might require adapting the conceptual framework to take the purely static type soundness proof as a source of design insight.

Noninterference vs. Dynamic gradual guarantee. Although the extra check above allows GSL\text{Ref} to ensure noninterference, it sacrifices the dynamic gradual guarantee. Recall that this guarantee says that removing a static security annotation cannot introduce new runtime errors.

Consider the following example:

```
1 fun x: Bool_H =>
2   let y: Ref Bool_H = ref true_H
3   if x then y := false_H else unit
```
The program is accepted statically and runs without error as it does not break noninterference. If we remove the type annotations on line 2:

```plaintext
1 fun x: Bool_H =>
2   let y: Ref Bool_H = ref true
3   if x then y := false_H else unit
```

then the program is conservatively rejected at runtime, because of the additional check for assignments. This behavior violates the dynamic gradual guarantee.

To sum up, if decreasing the precision of a type annotation results in performing an assignment to a reference whose content now has an unknown security label, and that assignment occurs under a non-public security effect, a runtime error can be raised, whereas the more precise program did not fail. More precisely, even in such situations, a runtime error will only be raised if the dynamic security information about the stored value up to the point of the actual assignment is lower than the current security effect. For instance, in our example above, if we modify the security level of the boolean in line 2 to \( H \) (leaving the type of \( y \) as it is), then the program performs a valid assignment on a reference whose content has a statically-unknown security level, but dynamically \( H \); therefore no runtime error is raised. Unfortunately, beyond pure and read-only programs, it seems impossible to provide any useful syntactic characterization of the programs for which the dynamic gradual guarantee holds, because both the current security effect and the accumulated evidence about a given value are essentially dynamic information.

## 4.7 Related Work

Static and dynamic information-flow control techniques have been extensively studied in the literature. The area is too vast to exhaustively review here: we refer to [105, 104, 56] for broad overviews of the area. This section first focuses on security type systems, as well as some specific approaches to dynamic information flow control, given the static-to-dynamic spectrum that gradual security typing covers. We also discuss existing proposals that combine static and dynamic checking. Finally we relate our work to other efforts to gradualize advanced type disciplines.

**Static information flow control.** Volpano et al. [126] present one of the first type systems for information flow analysis, developed for a first-order imperative language with conditionals and loops. They present and formalize the first soundness result for a security-typed language, namely that altering the initial values of locations cannot affect resulting values of locations with a lesser security level.

Subsequently, Heintze and Riecke [58] present a security-typed higher-order language called the Secure Lambda Calculus (SLam). SLam is a functional language extended with

---

17Removing the additional check on assignments recovers the dynamic gradual guarantee, but it breaks noninterference: there is no free lunch in the presence of mutable references.
splits, products, and recursion, that supports both confidentiality and its dual notion, integrity [17]. They introduce the prot expression, which we also use, to increase the ambient security level for the dynamic extent of evaluating a term. The noninterference proof for SLam is also based on logical relations. The authors extend SLam with concurrency and references. They prove that the resulting language is type safe, but they do not prove noninterference, deemed too problematic in a concurrent setting. SSLRef is also a higher-order language with references, but it does not support sums, products, recursion and concurrency. We prove noninterference for both GSLRef and SSLRef. Extending GSLRef to richer types and concurrency is a challenge worth addressing in future work.

To consolidate different related efforts, Abadi et al. [1] develop the Dependency Core Calculus (DCC), an extension of the lambda calculus that tracks dependencies such as security, partial evaluation, program slicing and call-tracking. In particular, they show that different languages such as SLam can be translated to DCC. They present a semantic model of DCC that helps to provide a simple proof of noninterference. It would be interesting to study the application of AGT to DCC, to provide a general account of gradual dependency tracking.

JFlow [85, 84], which later evolved into Jif [86], is a practical extension of the Java language that protects both confidentiality and integrity of sensitive data. Jif supports statically-checked information flow annotations, a decentralized label model with principals, automatic label inference, and security label polymorphism, all integrated with object-oriented features like class inheritance, as well as exceptions, among other features. Jif supports runtime label tests that can be used to encode explicit security casts, although such casts break type-based reasoning about noninterference. Scaling up GSLRef to cover the feature set of Jif would open the door to a practical implementation of gradual security typing.

Zdancewic [131] proposes $\lambda^{SEC}$, a simple security language similar to SLam, and proves noninterference using logical relations. He then extends the language with references, yielding $\lambda^{SEC}_{REF}$, which was the starting point for our design of SSLRef. Unlike SSLRef, the operational semantics of $\lambda^{SEC}_{REF}$ includes additional checks to control whether it is safe to assign to references; the type system then makes these checks redundant. In SSLRef, we omit these checks, and the runtime only tracks security levels. The runtime checks needed in the gradual setting arise as evidence combination. Also, Zdancewic does not prove noninterference for $\lambda^{SEC}_{REF}$ directly, but instead by a CPS translation to a lower-level imperative language with explicit continuations, for which noninterference is established [132]. This setting permits studying information flow with concurrency and as such could be a judicious starting point to study the interaction of gradual security typing and concurrency.

Much work on static information flow analysis focuses on declassification, which is the limited, intentional, and controlled release of confidential information. Declassification is outside the scope of this work, though a very interesting perspective for future work; we refer to [106] for an introductory survey.

An important distinction in information flow analysis is whether an analysis is flow-sensitive, i.e. whether memory cells are allowed to store values of different security levels at different times. Hunt and Sands [64] explore families of sound flow-sensitive type systems, indexed by the choice of the security lattice. In particular, they show that every program typeable in a flow-sensitive static type system can be translated to an equivalent program
typeable in a flow-insensitive type system. SSLRef is a flow-insensitive purely static analysis; GSLRef inherits flow-insensitivity for its static semantics. However, at runtime the security level of references is allowed to vary (through evidence composition) within the bounds imposed by the static type of the reference. This means that a reference that is created with an unknown security label can store values of any security level at different times. This leads us to sharing challenges faced by dynamic information-flow control techniques, discussed hereafter.

**Dynamic information flow control.** Russo and Sabelfeld [104] show that static mechanisms can be more precise than dynamic ones about certain kinds of information flows. Indeed, noninterference can be characterized as a 2-safety property, meaning that it can only be refuted by observing two different executions of the same program with different inputs. This makes it particularly challenging for dynamic information flow control, which traditionally makes decisions based on a single execution. Most work on dynamic information flow analysis therefore monitors a 1-safety property that conservatively approximates noninterference, but has the advantage of being observable in a single execution. Such approximations necessarily introduce false alarms, especially when mutable references are involved.

To avoid implicit leaks through the heap in a purely dynamic information-flow analysis, Austin and Flanagan [8] introduce a no-sensitive-upgrade check to prevent implicit security leaks through partially-leaked data, i.e. data produced from updates to public heap data that depend on private information. We adapt this approach to GSLRef, imposing an extra check when assigning to references. Subsequently, Austin and Flanagan [9] propose a more permissive analysis, where partially-leaked data is allowed, but carefully tracked to ensure that it is upgraded before being used in conditional tests. This allows programmers to iteratively add security upgrades to partially-leak data only when needed, through multiple executions of a program.

Later, Austin and Flanagan [10] introduce a completely different approach: faceted execution, which simulates multiple executions of a program for different security levels in a single run. A faceted execution yields a faceted value, which in a traditional two-point lattice is a pair of a public and a private value. This novel approach enables a characterization of noninterference as a 1-safety property, without introducing false alarms. It does however raise questions regarding how to efficiently implement such faceted executions, especially in the presence of complex security lattices. Faceted execution was recently extended to support dynamic information flow with exceptions, declassification and clearance [11]. It would be interesting to explore whether basing GSLRef on faceted execution might yield a gradual security language that fully respects the dynamic gradual guarantee, by avoiding the extra runtime check in assignments.

Stefan et al. [116] present a dynamic information-flow control system called LIO. Contrary to most approaches to dynamic information flow, LIO does not modify the underlying language runtime semantics, being implemented as a Haskell library. LIO supports both mutable references and exceptions. Exceptions are used to recover from security monitor failures, preserving both confidentiality and integrity. The possibility of securely recovering from runtime security exceptions is an interesting perspective to study in the context of
gradual security typing. More generally, recovering from runtime type errors raises a number of questions about the metatheory of gradual typing, because doing so can directly affect the dynamic gradual guarantee as well as type-based reasoning (e.g. it becomes possible to encode explicit type tests).

**Hybrid information flow control.** To resolve the tension between flexibility and soundness of flow-sensitive analyses, Russo and Sabelfeld [104] propose a general hybrid approach, in which a static effect analysis is used to dynamically upgrade the security level of variables of untaken branches of conditionals, thereby preventing implicit leaks through the heap. This hybrid approach is developed on top of a (first-order) imperative language. Moore and Chong [81] later show how to implement this hybrid approach more efficiently using additional static analyses.

A variety of hybrid information-flow control systems have been investigated, whose designs combine static and dynamic techniques that buttress one another to balance permissiveness and efficiency. Note that although gradual typing also combines static and dynamic techniques, hybrid approaches differ essentially from gradual ones. The key specificity of gradual typing is to smoothly support the continuum between static and dynamic checking based on the (programmer-controlled) precision of type annotations [109, 113]. This central notion of type precision is absent from hybrid approaches, in which the balance between static and dynamic checking is often driven by other concerns—such as the (un)decidability of a static predicate [75], or the need to pre-compute information for enhancing runtime checking.

Chandra and Franz [24] implement hybrid security information flow control for the Java Virtual Machine. The operational semantics permits policies to change during execution. To prevent invalid implicit flows through the heap, they perform a static analysis of effects similar to Russo and Sabelfeld [104]. Information about conditionals is gathered ahead of execution, then used to update labels at runtime, as if all branching alternatives had been taken. They also statically determine when the current security effect can be lowered again after a conditional. Performing an effect analysis statically to drive runtime monitoring is appealing as it could obviate the extra assignment check in GSLRef that compromised the dynamic gradual guarantee. However, in the setting of a higher-order imperative language, the effect analysis could easily become too conservative or too demanding for programmers. Combining gradual security and gradual effects [14] may temper this issue, but represents a considerable challenge in itself.

Shroff et al. [108] present a dynamic information flow system based on runtime tracking of indirect dependencies between program points, allowing a lazier, hence more flexible, detection of implicit flows. In particular, they track indirect dependency between dereference points and branching points. They present two languages, one that captures dependencies statically, and one that uses multiple executions of a program to record dependencies. This is yet another approach to runtime tracking that is worth considering in order to achieve a more flexible gradual security language that fully respects the dynamic gradual guarantee.

Hybrid approaches can also support programmer-controlled flexibility. Buiras et al. [20] propose Hybrid LIO (HLIO), a flexible monadic information-flow control library for Haskell. HLIO is not gradual in the sense that it does not include an unknown security label; instead,
HLIO provides a primitive to explicitly and selectively defer label-ordering checks to runtime. Their approach to defer static typing constraints to runtime can even be exploited to postpone type checks beyond security label constraints, opening the door to hybrid type checking in Haskell. In contrast, as a gradual security language, GSLRef supports a notion of unknown security information and implicitly mediates the interactions between static and dynamic security checking.

Gradual security typing. Most directly related to our proposal is prior work on gradual security typing, which combines static and dynamic checking with the express intent of supporting a smooth migration between both checking disciplines by introducing a dynamic (i.e. statically unknown) security label. Disney and Flanagan [35] and Fennell and Thiemann [39] pioneered what we describe in Sec. 4.1 as a check-driven approach to gradual security typing, starting from dynamic checking. Both develop notions of blame tracking and prove blame theorems for their semantics. It is important to recall that these approaches, while dubbed “gradual”, are based on explicit security casts, and are therefore more akin to cast calculi than to gradual languages. In particular, this means that these languages do not respect the gradual guarantees by design, including the static one, because changing the precision of type annotations requires adding/removing explicit casts. Additionally, as discussed in the introduction, both proposals break type-based reasoning about noninterference.

Recently, Fennell and Thiemann [40] extend their prior work on gradual security typing with references to the object-oriented setting, in a language called LJGS. Like Jif, LJGS performs local inference of security labels, and supports polymorphic security signatures. Local variables in LJGS are typed in a flow-sensitive manner, whereas both SSLRef and GSLRef are flow insensitive regarding security levels. Although LJGS is based on explicit casts like prior work, its semantics differ in important ways. For instance, recall the example given in Sec. 4.1:

```plaintext
let mix : L → L Int H → L Int H L
  = fun pub priv => if pub < (Int L ⇐ Int H) priv then 1 L else 2 L
  mix 1 5 0
```

This example does not type check in LJGS because the target type of a security cast cannot be less secure than the source type. The only way to write this example is to go through the dynamic security level explicitly:

```plaintext
let mix : L → L Int H → L Int H L
  = fun pub priv => if pub < (Int L ⇐ Int H) (Int L ⇐ Int H) priv then 1 L else 2 L
  mix 1 5 0
```

This well-typed program fails at runtime because (Int L ⇐ Int H) upgrades 5 L to 5 H, but (Int L ⇐ Int H) 5 H is not defined. This approach to upgrade the security level of values that are cast to the dynamic label using the statically-determined source label seems to restore type-based reasoning about noninterference in LJGS. Interestingly, the change in semantics in LGJS is solely motivated by the design goal to avoid having to dynamically track security labels of statically-typed program fragments, so the relation with type-based reasoning appears to be accidental.
Similar to the approach of Russo and Sabelfeld [104] and Shroff et al. [108] discussed above, LJGS relies on a side-effect analysis to tracks the updated variables in method bodies. More precisely, when typing a method, LJGS generates a set of constraints that represent the information flow dependencies between parameters and return values, as well as two sets of effects: a local effect that lists the variables modified in branches of a conditional, used to update local variables of untaken branches; and a global effect that records the security types whose fields may be updated with sensitive information. This type analysis and constraint/effect inference is facilitated by the fact that classes in LJGS are not first-class entities, i.e. all class definitions are top-level and known ahead-of-time. This means in particular that at every call site, one statically knows the precise inferred constraints and effects of methods (modulo a standard subsumption criteria to account for subtyping). In a setting with higher-order types, this information would be more complex to track. Additionally, the inferred global effect of a method is insufficient information per se for the dynamic information flow control part of LJGS. Therefore, LJGS also appeals to an external effect analysis (left opaque) to obtain precise information about heap write effects.

Gradualizing expressive typing disciplines. Since the initial formulation of gradual typing [109], there has been many efforts to gradualize advanced typing disciplines, like typestates [129, 47], ownership types [107], annotated type systems [117], effects [13, 14, 123], refinement types [76, 72], parametric polymorphism [7, 69], and the security type systems discussed above, among others.

Since the formulation of the refined criteria for gradually-typed languages [113], however, only refinement types [76, 72] have been shown to fully respect such guarantees. This work contributes to the general research agenda of gradual typing disciplines by explicitly attempting to achieve both the gradual guarantees and a rich semantic property, like noninterference. Indeed, noninterference is not implied by type safety; in contrast, soundness of refinement types directly follows from type safety. We have shown that GSL adopts respect the static gradual guarantee (as opposed to other gradual security type systems); but GSL sacrifices the dynamic gradual guarantee due to a modification of the runtime semantics to enforce noninterference in the presence of mutable references.

Initial work on gradual parametricity [69] also suggests that parametricity may be incompatible with the dynamic gradual guarantee, unless one is willing to tweak the type precision relation; even then, the dynamic gradual guarantee is left as a conjecture. Ahmed et al. [7] prove parametricity for a polymorphic cast calculus—not a source language—and also leave the gradual guarantees as an open question. Actually in §6 we apply AGT to System F and show that the classical notion of parametricity is actually incompatible with the dynamic gradual guarantee. Therefore, further work is needed to fully understand if and how the gradual guarantees can be reconciled with rich semantic typing disciplines, and if additional design criteria for such gradual languages should be devised.
4.8 Conclusion

We develop a novel, \textit{type-driven} approach to gradual security typing, in which gradual security types provide strong security invariants, while admitting flexible programming idioms. This is the first work to address the gradualization of a rich typing discipline in which type safety does not imply type soundness, while pursuing the most elaborate formulation of criteria for gradually-typed languages \cite{113}, and preserving type-based reasoning principles. This means that the amount of static checking is entirely driven by the precision of static security annotations, and that programmers can reason modularly about the noninterference guarantees of program fragments by just looking at types.

Using the AGT methodology \cite{44} to derive the gradual security language GSL$_{Ref}$, this work sheds light on key semantic issues in the design of gradual languages. AGT was central in our endeavor to separate the elements of the design that follow by systematically following the methodology from those that require careful consideration. In particular, we identify a tension between the smooth continuum on the static-to-dynamic spectrum that the gradual guarantees mandate, and the semantic property of noninterference, which manifests in GSL$_{Ref}$ because of mutable references. This tension also raises interesting questions for the principled design of gradually-typed languages, whenever the semantics of types has a relational flavor. In particular, while we have addressed noninterference, relational parametricity remains to be addressed. Overall, this work suggests that it might be necessary to extend AGT to integrate the purely static type soundness proof—as opposed to only the type safety proof—as a source for the design of the dynamic semantics of a gradual language.

Within the context of gradual security typing, our work leaves open the question of whether it is possible to reconcile both noninterference and the dynamic gradual guarantee. Specifically, it would be informative to study whether other approaches to sound dynamic information flow control could help us recover the dynamic gradual guarantee. We believe that there might be an inherent incompatibility between the strictness required to enforce a hyper-property like noninterference, and the optimistic flexibility dictated by the dynamic gradual guarantee.

Another interesting track for future work is to explore a “pay-as-you-go” \cite{109} semantics, which only introduces runtime checks for imprecisely-typed expressions, as well as scaling the security discipline to other language-based security features such as integrity, flow sensitivity and declassification. Additionally, we want to explore the applicability of Garcia and Cimini \cite{43}’s approach to type inference in gradual languages to address security label inference \cite{98} in GSL$_{Ref}$.
Chapter 5

A Gradual Interpretation of Union Types

Union types enable programmers to capture the possibility of a term to be of several possibly unrelated types. Traditional static approaches to union types are untagged and tagged unions, which present dual advantages in their use. Inspired by recent work on using abstract interpretation to understand gradual typing, in this chapter we present a novel design for union types, called gradual union types\(^1\). Gradual union types combine the advantages of tagged and untagged union types, backed by dynamic checks. Seen as a gradual typing discipline, gradual union types are restricted imprecise types that denote a finite number of static types. We apply the AGT methodology to derive the static (§5.3) and dynamic semantics (§5.4) of a language that supports both gradual unions and the traditional, totally-unknown type. We uncover that gradual unions interact with the unknown type in a way that mandates a stratified approach to AGT, relying on a composition of two distinct abstract interpretations in order to retain optimality (§5.3.1). Thanks to the abstract interpretation framework, the resulting language is type safe and satisfies the refined criteria for gradual languages (§5.4). We also show how to compile such a language to a threesome cast calculus, and prove that the compilation preserves the semantics and properties of the language (§5.5).

5.1 Introduction

Let us remember that AGT reinforces a broader interpretation of gradual typing: that of soundly dealing with imprecision at the type level. Indeed, one can see dynamically-typed languages as languages with highly-imprecise static type information, and the original gradually-typed languages as enabling us to reason about partial type information. For instance, consider a function \(f\) of gradual type \(\text{Int} \rightarrow ?\); this type is imprecise in that it does not provide any information about the values returned by \(f\), but it does specify precisely that \(f\) is a function, which furthermore expects an integer argument. Therefore, the gradual

\(^1\)This chapter is based on the work of Toro and Tanter [124].
language can statically reject \( f + 1 \) or \( f(\text{true}) \), accept \( f(1) \), and optimistically accept \( f(1) + 2 \) subject to a dynamic check that the value of \( f(1) \) is indeed an integer. Similarly, integrating a simply-typed language with gradual support for effects \[13\] can be viewed as dealing with imprecision of effect information.

Inspired by this focus on imprecision, we observe that standard static type systems have long been proposed to deal with a basic form of imprecision: the possibility for a value to be of several, possibly-unrelated types. In the literature, two approaches have been developed to safely, and fully statically, deal with the possibility of an expression to have possibly different types: disjoint (or tagged) union types, such as sum types \( T_1 + T_2 \) and variant types, and untagged union types, usually noted \( T_1 \vee T_2 \) \[92\]. Both forms of union types have complementary pros and cons when viewed from a pragmatic angle.

The understanding of both gradual types and union types as different ways to deal with imprecision at the type level suggests a novel, gradual interpretation of union types. Following the abstract interpretation of gradual types put forth in AGT, a gradual union \( T_1 \oplus T_2 \) is a gradual type that abstracts both \( T_1 \) and \( T_2 \). Seen in this light, a gradual union is a gradual type that is more precise than the prototypical, fully-unknown, gradual type \(?\). Starting from this insight, systematically applying the AGT methodology yields a novel point in the design space of both union types and gradual types.

Adding gradual unions to a simply-typed language relaxes the typing discipline, but does not support for full dynamic type checking. To achieve this, one needs to include both the unknown type \(?\) and gradual unions. A second contribution of this chapter is to uncover that combining these two gradual type constructors in the same language demands a stratified approach to AGT, in which the semantics of gradual types comes from the composition of two distinct abstract interpretations.

**Contributions.** This chapter makes the following specific contributions:

- A novel design of union types that combines benefits of both tagged and untagged unions, with added static flexibility backed by runtime checks. Compared to a standard gradually-typed language with only the totally-unknown type \(?\), the resulting design is stricter, allowing more blatantly wrong programs to be statically rejected.

- A first example of a stratified approach to AGT. To derive the static semantics of a gradual language, AGT requires a Galois connection between gradual types and sets of static types, which then guides the lifting of functions and predicates on static types to their gradual counterparts \[44\]. We observe that applying AGT directly to introduce both the unknown type and gradual unions breaks optimality of the abstraction, thereby weakening the meaning of type information, both statically and dynamically. To address this, we develop a stratified approach to AGT that allows us to recover optimality. More specifically, we first apply AGT to support only the unknown type, and lift this Galois connection and derived liftings to their powerset counterpart. We then apply AGT once more with another Galois connection to introduce support for gradual unions, which allows us to define liftings based on the previously-defined powerset liftings. We prove that the composed abstraction is optimal. We conjecture that
this technique might prove helpful in integrating other gradualization efforts.

- The formalization and meta-theory of the proposed language, including type safety and the gradual guarantees of Siek et al. [113]; these results follow directly, by construction, from relying on the AGT methodology.
- Although we derive the dynamic semantics using AGT, we also present a compilation scheme to an internal language with threesome, a space-efficient representation for casts [112]. We prove the correctness of the compilation with respect to the reference semantics derived by AGT using logical relations.

**Structure.** § 5.2 briefly reviews tagged and untagged unions, highlighting their pros and cons, and then informally introduces gradual unions, comparing them with standard gradual types and with the other kinds of unions, including those supported by several recent languages such as Flow and TypeScript, among others. § 5.3 describes the static semantics of GTFL⊕, a language with both gradual unions and the unknown type, using AGT. § 5.4 describes the runtime semantics of the language by translation to a threesome cast calculus, and gives the formal properties of the language. § 5.6 discusses related work and § 5.7 concludes.

Complete definitions, as well as the proofs of all the results stated in this chapter, can be found in §[C]. A prototype implementation is available online, showing interactive typing and reduction derivations for arbitrary source programs: http://pleiad.cl/gradual-unions/

## 5.2 Background and Motivation

We first briefly review standard tagged and untagged union types [92], highlighting the tradeoffs associated with each approach, and then introduce gradual union types as a novel point in the design space. We compare gradual unions to other approaches to union types, including practical languages with unions supported by runtime type tests. Finally, we compare gradual unions with the standard gradual types introduced by Siek and Taha [109].

### 5.2.1 Tagged Unions

Tagged unions, also called *disjoint* union types, denote values of possibly different types. The “disjointness” of the union comes from the fact that elements must be explicitly *tagged* so that it is clear to which type an element belongs. Tagging allows type-safe disambiguation through a case analysis construct.

The simplest form of tagged unions are binary *sum types*, noted $T_1 + T_2$, with injection forms *inl* and *inr*, and a disambiguation *case* expression. For instance, *inl* 10 :: $\text{Int} + \text{Bool}$ injects the integer 10 into the sum type $\text{Int} + \text{Bool}$. The tag *inl* denotes the *left* part of the sum. Similarly, *inr* true :: $\text{Int} + \text{Bool}$ injects *true* to the right of the sum. Note that
the ascription :: is necessary to maintain a simple syntax-directed type system; different techniques can be used to alleviate notation for programmers [92].

Given a value of type \texttt{Int+Bool}, one cannot use it directly. For instance, \( \lambda x : \texttt{Int+Bool}.x+1 \) is not well typed. To use a tagged value, one must first disambiguate through an explicit case analysis, considering each tag explicitly, \textit{e.g.} \( \lambda x : \texttt{Int+Bool}.\text{case } x \texttt{of inl } x \Rightarrow x + 1 \mid \text{inr } x \Rightarrow \text{if } x \texttt{then } 1 \texttt{else } 0. \)

Note that \texttt{Int+Bool} is different from \texttt{Bool+Int} because the injection tag is relative to the \textit{position} in the sum type. Sums can be generalized to \textit{variants}, which are n-ary sums with custom labels instead of the positional \texttt{inl} and \texttt{inr} tags. In the case of variants, a type-case construct similar to \texttt{case} forces programmers to consider all possible alternatives, thereby statically ensuring the absence of runtime type errors.

To deal with with values of statically-unknown types, several proposals add a type \texttt{Dynamic} whose values are pairs of a plain value and a type tag [2, 59]. The type \texttt{Dynamic} is therefore akin to an infinite tagged union, where tags are types. Disambiguation through case analysis therefore requires a default branch to handle unconsidered alternatives generally.

This general approach also explains how several languages support union types without needing any explicit tagging operation. For instance, in safe dynamic languages, all values are readily tagged with their \textit{class} (either in the sense of Harper [54], \textit{e.g.} \texttt{Int, Bool, Function}, or, for class-based object-oriented languages, their actual class). This allows disambiguation of unions through runtime type testing (either via a type-case analysis or casts that can fail). This approach is exploited in several retrofitted type systems such as TypeScript [28], Flow [71] and Typed Racket [118, 119]. Explicit disambiguation of unions can also be supported through pattern matching, as in CDuce [16] and Dotty [102].

5.2.2 Untagged Unions

An untagged union, noted \( T_1 \lor T_2 \), denotes the union of the values of type \( T_1 \) and of type \( T_2 \), without any tagging mechanism to support disambiguation. In this set-theoretic interpretation [15, 42, 91], \( \texttt{Int \lor Int} \) is the same type as \( \texttt{Int} \); and a value of type \( T_1 \) is a value of type \( T_1 \lor T_2 \), without any injection construct.

Untagged unions can be used to allow the branches of conditionals to have unrelated types: for instance, the function \( \lambda x : \texttt{Bool}.\text{if } x \texttt{then } 1 \texttt{else } \text{false} \) can be considered well-typed at \( \texttt{Bool} \rightarrow \texttt{Int\lorBool} \). This is the approach followed by the CDuce programming language [16], for instance.

Automatically introducing imprecision through untagged unions can however lead to unwanted programs being accepted. An alternative approach is for the typing rule for conditional expressions to require both branches to be of the same type, and to expect the programmer to use an explicit type \textit{ascription} to specify that imprecision is desired; \textit{e.g.} \( \lambda x : \texttt{Bool}.\text{if } x \texttt{then } (1 :: \texttt{Int\lorBool}) \texttt{else } \text{false} \). Note that the ascription does not imply any runtime tagging; it is a purely static artifact. Also, because of the set-theoretic interpretation
of types, it is sufficient to ascribe imprecision in one of the two branches.

Untagged unions have no projection construct either; the only safe operations on a value of type $T_1 \lor T_2$ are those that are supported by both $T_1$ and $T_2$. Note that this makes untagged union restrictive to use; for instance, nothing useful can be done with a value of type $\text{Int} \lor \text{Bool}$. For instance, $\lambda x : \text{Int} \lor \text{Bool}.x + 1$ is not well-typed, because $x$ could be a boolean value; and there is no disambiguation expression like case to handle each alternative separately.

This does not mean that untagged unions are useless; for instance, if the language has records, then it is safe to access fields that are common to both types. As noted by Pierce [92], untagged unions have traditionally been much more frequent in program analysis than in programming languages, where they were mostly used in type systems for semi-structured data [21, 61], before being generalized in CDuce. Finally, note that the C language supports unsafe untagged unions, allowing programmers to use operations that are supported by either $T_1$ or $T_2$, at their own risk!

### 5.2.3 Gradual Unions

Tagged and untagged unions are the only safe approaches to statically deal with imprecision: either explicitly tag the imprecision so as to be able to safely discriminate later on, or assume the loss of precision and restrict what can be done with imprecisely-typed values. Tagged unions have the benefit of allowing programmers to fully use values, but only after explicit case-based disambiguations. Untagged unions have the benefit of requiring neither explicit injection nor projection, but only allow restricted usage of values.

If we are willing to accept some form of dynamic check errors, however, we can combine the benefits of both tagged and untagged unions by viewing a union type as a kind of gradual type: $T_1 \oplus T_2$ is a gradual type that represents both $T_1$ and $T_2$. A gradual union supports the same kind of optimistic static checking that standard gradual typing provides.

For instance, $f \triangleq \lambda x : \text{Int} \oplus \text{Bool}.x + 1$ is (optimistically) wellyped, because $x$ might possibly be an Int, without any explicit projection or case analysis. The expressions $f \text{ true}$ and $f \text{ 1}$ are also well-typed because injection to a gradual union is implicit. The expression $f \text{ true}$ evaluates to 2, as expected. But because $x$ might in fact be a Bool, a runtime check is implicitly introduced before applying the + operator: hence the expression $f \text{ true}$ produces a runtime cast error.

As a gradual type, a gradual union allows clearly incorrect programs to be rejected statically. For instance, changing the body of $f$ to $x \text{ 1}$ is statically rejected, because $x$ cannot possibly be a function. Similarly, $f \text{ "hola"}$ is statically rejected, because $f$ only tolerates integer or boolean arguments.

Note that compared to untagged unions, the use of a value with a gradual union type $T_1 \oplus T_2$ is accepted if the operations make sense for either $T_1$ or $T_2$ (and not both). This is just like untagged unions in C, but backed by runtime checks to ensure type safety. Injecting values
into a gradual union type can be done implicitly as when applying \( g \) in the example above, or using an ascription, e.g. \( g \triangleq \lambda x : \text{Bool}. \text{if } x \text{ then } (1 :: \text{Int} \oplus \text{Bool}) \text{ else } (\text{false} :: \text{Int} \oplus \text{Bool}) \) has type \( \text{Bool} \rightarrow \text{Int} \oplus \text{Bool} \).2

### 5.2.4 Comparing Unions

We summarize the characteristics of each form of union types as follows:

<table>
<thead>
<tr>
<th></th>
<th>injection</th>
<th>projection</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tagged unions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sums</td>
<td>explicit</td>
<td>explicit</td>
<td>full</td>
</tr>
<tr>
<td>type tests/casts</td>
<td>implicit</td>
<td>explicit</td>
<td>full</td>
</tr>
<tr>
<td>Untagged unions</td>
<td>none</td>
<td>none</td>
<td>restricted</td>
</tr>
<tr>
<td>Gradual unions</td>
<td>implicit</td>
<td>implicit</td>
<td>full</td>
</tr>
</tbody>
</table>

To illustrate the convenience of gradual unions compared to alternative approaches, consider the following simple program:

```plaintext
let x: Bool \oplus \text{Int} \oplus \text{String} = 10
(\lambda x: \text{Int} \oplus \text{Bool}. x+1) x
```

The program introduces a variable \( x \) that can be one of three types, and initializes it to the number 10. It then passes it as argument to a lambda that expects either an \text{Int} or a \text{Bool}, and adds 1 to it. This program is well-typed, and returns 11. If \( x \) is initialized with a string, the program fails at runtime before the application of the function; if it is initialized with a boolean, the runtime error occurs before the addition.

This example would not be well-typed with untagged unions. This is because the intersection of \text{Int} and \text{Bool} and \text{String} is empty. If all three types have a common method, say \text{toString}, then the body of the lambda can only safely invoke \( x.\text{toString()} \).

Turning to tagged unions, using standard sum types, the equivalent program would be fairly cumbersome to write because all injections and projections have to be manually introduced by the programmer, and deal with exact positions:

```plaintext
let x: \text{Bool} + (\text{Int} + \text{String}) = \text{inr} \text{ inl} 10
let x2: \text{Int} + \text{Bool} = \text{case } x \text{ of}
  | \text{inl} y => \text{inr} y
  | \text{inr} y => \text{case } y \text{ of}
  | \text{inl} z => \text{inl} z
  | \text{inr} z => \text{throw new Error("not an Int or Bool")}
(\lambda x: \text{Int} \oplus \text{Bool}. \text{case } x \text{ of}
  | \text{inl} y => y + 1
  | \text{inr} y => \text{throw new Error("not an Int")}) x2
```

2Similarly to untagged unions, one could design a language whose conditional expression implicitly introduces imprecision, without the need for any ascription (Sect. 5.2.2); we do not further consider this possible design and use ascriptions explicitly.
Note the need for an explicit intermediate step ($x_2$) to safely go from the ternary union to the binary union.

The same program can also be written using implicitly-tagged unions with type-test disambiguation. As expected, the code is more lightweight than with sums thanks to implicit injection. For instance, in Flow:

```javascript
const x: boolean | number | string = 10
const foo = (x: number | boolean): number => {
  if(typeof x == "number") return x + 1
  else throw new Error("not a number")
}
if(typeof x == "boolean" || typeof x == "number") foo(x)
else throw new Error("not a boolean or number")
```

Note that projections must be realized manually via `typeof` (lines 3, 4, 6 and 7).

**Evolving precision.** The advantage of gradual unions does not only lie in the simplicity and compactness of the program definition. It also lies in its robustness in the face of precision-related changes. For instance, suppose that as the software matures, the programmer is now convinced that $x$ will always be initialized with a number and that the function can simply only accept numbers. With sums, the program is so fragile that it would need to be modified at every injection and projection point to account for this change in precision. The Flow version would still run as is, but would feature a lot of dead code. Further decreasing precision would require adding checks at various projection points. With gradual unions, it is enough to adjust the type annotations—the rest of the program is unchanged!

The fact that the static-to-dynamic spectrum is navigated solely through the precision of type annotations, without requiring further modification of the program, is a key asset of gradual typing in general. The gradual guarantee of Siek et al. [113] further characterizes the relation between the static and dynamic semantics of programs that only differ in the precision of their type annotations, and will be discussed further when addressing the meta-theory of GTFL.$^\oplus$

**Higher-order types.** Finally, a major limitation of projections from unions using explicit type tests is that they do not support higher-order types. For instance, because one cannot decide whether an arbitrary function (e.g. of tag/class `Function`) always behave as a function of a particular type, programmers have to manually wrap functions with pre-post type checks.

Consider, in Flow or TypeScript, two functions of the following types:

```
f: (number | boolean) => (number | boolean)
g: ((number | string) => (boolean | string)) => string
```

To safely support the application $g(f)$, one needs to explicitly wrap $f$ as follows:

```javascript
const wrapper = (x: number | string): string | boolean => {
  if (typeof x === "number") {
```
const result = f(x)
if (typeof result === "boolean") return result
else throw new Error("not a boolean")
} else throw new Error("not a number")

and then pass the wrapped function as argument: `g(wrapper)`.

Conversely, with gradual unions, one can simply write:

```javascript
let f: (Int ⊕ Bool) -> (Int ⊕ Bool) = ...
let g: ((Int ⊕ String) -> (Bool ⊕ String)) -> String = ...
g(f)
```

for the exact same behavior; all the necessary checks and wrappers are handled under the hood.

### 5.2.5 Gradual Unions vs. Standard Gradual Types

Gradual typing has always been formulated in terms of an unknown type, frequently written `?`, which denotes any possible type \[109\]. Furthermore, when structural types are supported, gradual types can be more precise than the fully-unknown type: for instance `Int -> ?` denotes all function types from `Int` to possibly any type.

To illustrate the key difference between gradual unions and standard gradual types, consider a function `h` that always returns either an `Int` or a `Bool`. Starting from a simple typing discipline, with standard gradual types, the most precise type one can give to `h` is `Bool -> ?`. However, this type allows for too much flexibility that was not intended: because `h true` has type `?`, it can subsequently be used in any context, even `(h true) 1`, which is clearly always going to fail since `h` never returns a function. The problem comes form the fact that the gradual type used for the codomain of `h`, `?`, is too imprecise—yet it is the only available type to denote both `Int` and `Bool`. Hence, the programmer cannot express a more restricted form of flexibility. Gradual unions address this need. For instance, recalling function `g` from §5.2.3 above, `(g true) 1` is statically rejected.

Gradual union types are a novel way to relax a static typing discipline in a restricted manner. While the discussion above insists on the advantages of this restricted flexibility, it necessarily presents drawbacks as well. In particular, a language with only gradual unions cannot fully embed the untyped lambda calculus. In order to get the best of both worlds, one needs a language that supports both the fully-unknown gradual type `?` in addition to gradual unions. This way, programmers can navigate the full static-to-dynamic spectrum, with more interesting intermediate points offered by gradual unions. In the rest of this chapter, we design and formalize such a language.
5.3 GTFL⊕: Static Semantics

We now formalize GTFL⊕, a gradual language with both gradual unions and the unknown type. As hinted previously, we follow AGT to derive the static semantics of GTFL⊕:

1. We start from a language with a fully static typing discipline, STFL.

2. We define the syntax of gradual types, and give them meaning via a concretization function and its corresponding most precise abstraction, forming a Galois connection. Crucially, in this step we realize that the two forms of gradual types must be handled in a stratified manner in order to ensure optimality.

3. We derive the static semantics of the gradual language by lifting type predicates and type functions used in the static type system through the Galois connection.

The most novel part of our development are steps 2 and 3, which showcase how to compose Galois connections related to different gradual type constructors. We address the dynamic semantics of GTFL⊕ to §5.4.

5.3.1 The Static Language: STFL

Our starting point is the simply-typed functional language STFL. A term can be a lambda abstraction, a boolean, a number, a variable, an application, an addition, a conditional, or an ascription. The typing rules, dynamic semantics and type safety of STFL are completely standard, and can be found in §C.1.1.

5.3.2 Defining Gradual Types Separately

GTFL⊕ supports both the unknown type ? and gradual unions with ⊕. In this section, we look at both gradual type constructors separately in order to precisely define their meaning.

Let us first recall from [44] the Galois connection for gradual types made up with the (nullary constructor) ?, here denoted GTYPE.

\[
G \in \text{GTYPE} \\
G ::= ? \mid \text{Bool} \mid \text{Int} \mid G \rightarrow G
\]

The meaning of these gradual types is standard, and defined through concretization by Garcia et al. [44] as follows:
**Definition 26** (GType Concretization). \( \gamma_? : \text{GType} \rightarrow \mathcal{P}(\text{TYPE}) \)

\[
\begin{align*}
\gamma_?(\text{Int}) &= \{ \text{Int} \} \\
\gamma_?(\text{Bool}) &= \{ \text{Bool} \} \\
\gamma_?(?) &= \text{TYPE} \\
\gamma_?(G_1 \rightarrow G_2) &= \{ T_1 \rightarrow T_2 \mid T_1 \in \gamma_?(G_1) \land T_2 \in \gamma_?(G_2) \} 
\end{align*}
\]

Concretization naturally induces the notion of precision among gradual types:

**Definition 27** (GType Precision). \( G_1 \) is less imprecise than \( G_2 \), notation \( G_1 \sqsubseteq G_2 \), if and only if \( \gamma_?(G_1) \subseteq \gamma_?(G_2) \).

The following abstraction \( \alpha_? \) naturally forms a Galois connection with \( \gamma_? \):

**Definition 28** (GType Abstraction). \( \alpha_? : \mathcal{P}(\text{TYPE}) \rightarrow \text{GType} \)

\[
\begin{align*}
\alpha_?\{ T \} &= T \\
\alpha_?(T_1 \rightarrow T_2) &= \alpha_?(T_1) \rightarrow \alpha_?(T_2) \\
\alpha_?(\emptyset) &= \text{undefined} \\
\alpha_?(\top) &= ? \text{ otherwise}
\end{align*}
\]

Importantly, \( \gamma_? \) and \( \alpha_? \) form a Galois connection:

**Proposition 32** (\( \alpha_? \) is Sound and Optimal). If \( \top \) is not empty, then

\[
\begin{align*}
(a) \quad \top \subseteq \gamma_?(\alpha_?(\top)). \\
(b) \quad \top \subseteq \gamma_?(G) \Rightarrow \alpha_?(\top) \sqsubseteq G.
\end{align*}
\]

Let us now consider a Galois connection for the novel gradual type constructor introduced in this work, gradual unions. We use \( \text{STYPE} \) to denote gradual types made up only of gradual unions, \( i.e. \) without \( ? \).

\[
S \in \text{STYPE} \\
S ::= S \oplus S \mid \text{Bool} \mid \text{Int} \mid S \rightarrow S
\]

Note that this syntax admits n-ary unions recursively through \( S \oplus S \). We consider gradual unions to be syntactically equivalent up to associativity of \( \oplus \), \( i.e. \) \( S_1 \oplus (S_2 \oplus S_3) \equiv (S_1 \oplus S_2) \oplus S_3 \). Gradual unions represent the finite set of types represented (recursively) by each constituent:

**Definition 29** (STYPE Concretization). \( \gamma_\oplus : \text{STYPE} \rightarrow \mathcal{P}_{\text{fin}}(\text{TYPE}) \)

\[
\begin{align*}
\gamma_\oplus(\text{Int}) &= \{ \text{Int} \} \\
\gamma_\oplus(\text{Bool}) &= \{ \text{Bool} \} \\
\gamma_\oplus(S_1 \oplus S_2) &= \gamma_\oplus(S_1) \cup \gamma_\oplus(S_2) \\
\gamma_\oplus(S_1 \rightarrow S_2) &= \{ T_1 \rightarrow T_2 \mid T_1 \in \gamma_\oplus(S_1) \land T_2 \in \gamma_\oplus(S_2) \}
\end{align*}
\]

For instance \( \gamma_\oplus(\text{Int} \oplus \text{Bool} \oplus (\text{Int} \rightarrow \text{Bool})) = \{ \text{Int}, \text{Bool}, \text{Int} \rightarrow \text{Bool} \} \). Because gradual unions only produce finite sets of static types, the corresponding abstraction also only needs to be defined on finite sets, and therefore can produce the gradual union with all the elements, noted \( \oplus \top \):

---

3We use the hat notation \( \hat{X} \) to refer to a set of elements \( X \) (\( e.g. \) \( \hat{T} \) is a set of static types, and \( \hat{\top} \) is a set of sets of static types).
Definition 30 (STYPE Abstraction). $\alpha_\oplus : \mathcal{P}_{\text{fin}}(\text{Type}) \rightarrow \text{STYPE}$

$$\alpha_\oplus(T) = \oplus\hat{T} \quad \text{if } \hat{T} \neq \emptyset$$

Here again, $\langle \gamma_\oplus, \alpha_\oplus \rangle$ forms a Galois connection.

5.3.3 Combining Gradual Types: Take 1

Now that we have defined the meaning of gradual types formed with the unknown type $\_\_\_$, as well as the meaning of gradual types formed with gradual unions $\oplus$, we turn to defining the meaning of gradual types in $\text{GTFL}^{\oplus}$, which combine both constructors, denoted $\text{UType}$:

$$U \in \text{UType}$$

$$U ::= \_\_\_ | U \oplus U | \text{Bool} | \text{Int} | U \rightarrow U \quad (\text{gradual types})$$

A first seemingly natural approach is to define the concretization function for $\text{UType}$ by combining both concretization functions for $\text{GType}$ and $\text{STYPE}$:

Definition 31 (UType Concretization, Take 1). $\gamma : \text{UType} \rightarrow \mathcal{P}(\text{Type})$

$$\gamma(\text{Int}) = \{ \text{Int} \}$$

$$\gamma(\text{Bool}) = \{ \text{Bool} \}$$

$$\gamma(U_1 \oplus U_2) = \gamma(U_1) \cup \gamma(U_2)$$

$$\gamma(\_\_\_) = \text{TYPE}$$

$$\gamma(U_1 \rightarrow U_2) = \{ T_1 \rightarrow T_2 \mid T_1 \in \gamma(U_1) \land T_2 \in \gamma(U_2) \}$$

While this definition seems sensible, it does not accommodate a corresponding optimal abstraction. Indeed, the abstraction functions for $\text{GType}$ and $\text{STYPE}$ conflict with each other: how should we abstract a set of different types?

For a set of base types, say $\{ \text{Int}, \text{Bool} \}$, we can either abstract to $\_\_\_$ or to $\text{Int} \oplus \text{Bool}$; the latter being optimal, while the former is not. In fact, to preserve optimality, we ought to defer to the unknown type only for heterogeneous infinite sets. Even if we would adjust the definition of the combined abstraction to make such a distinction, it would not be optimal. To see why, consider the type $(\_\_\_ \rightarrow \text{Int}) \oplus (\text{Int} \rightarrow \_\_\_)$, whose concretization is:

$$\gamma((\_\_\_ \rightarrow \text{Int}) \oplus (\text{Int} \rightarrow \_\_\_)) = \gamma(\_\_\_ \rightarrow \text{Int}) \cup \gamma(\text{Int} \rightarrow \_\_\_)$$

$$= \{ T \rightarrow \text{Int} \mid T \in \text{TYPE} \} \cup \{ \text{Int} \rightarrow T \mid T \in \text{TYPE} \}$$

$$= \{ \text{Int} \rightarrow \text{Int}, \text{Bool} \rightarrow \text{Int}, \text{Int} \rightarrow \text{Bool}, \ldots \}$$

$$\triangleq \hat{T}$$

By taking the union of both sets, we “forget” a specificity of the original gradual type—namely that it only represents functions that necessarily have $\text{Int}$ either as domain or as codomain. For instance, $\text{Bool} \rightarrow \text{Bool}$ is not present in the resulting set $\hat{T}$. However, the abstraction function that we obtain by directly combining the two abstractions we have seen
above is unable to recover an optimal gradual type: because $\hat{T}$ is infinite and only contains arrow types, the best the abstraction can do is to keep the arrow constructor, and then separately abstracts the domain and codomain types (just like $\alpha_?$). As a result:

$$\alpha(\hat{T}) = ? \to ?$$

While this abstraction is sound, it is not optimal: there exists a more precise gradual type that represents $\hat{T}$, the type $(? \to \text{Int}) \oplus (\text{Int} \to ?)$ we started with.

Losing optimality directly affects the programmer’s experience. For instance, in the type system, this means that the gain of precision that gradual unions are supposed to provide (recall §5.2.5) is lost; similarly, type annotations would not be strictly enforced at runtime.

### 5.3.4 Combining Gradual Types: Take 2

In order to define a proper Galois connection to give meaning to the gradual types of GTFL$^\oplus$, we introduce a stratified approach, sketched in Figure 5.1:

- **Step 1.** We start from the Galois connection between GTYPE and $\mathcal{P}(\text{Type})$, named classic interpretation hereafter, which interprets the unknown type. We already described this Galois connection in §5.3.2.

- **Step 2.** We lift this connection to operate on finite sets of gradual types, with the standard collecting semantics, forming a new Galois connection between $\mathcal{P}_{\text{fin}}(\text{GTYPE})$ and $\mathcal{P}_{\text{fin}}(\mathcal{P}(\text{Type}))$, named the classic set interpretation.

- **Step 3.** We introduce a Galois connection between UTYPE and $\mathcal{P}_{\text{fin}}(\text{GTYPE})$, named union interpretation, which adds support for gradual unions among gradual types that include the unknown type.

- **Step 4.** We combine the classic set interpretation and the union interpretation. This combination gives a stratified interpretation of GTFL$^\oplus$ gradual types, UTYPE, in terms of finite sets of (possibly-infinite) sets of static types.
As we show, the stratified interpretation is itself a proper Galois connection, and we can subsequently use it to lift the static (and dynamic) semantics of STFL in order to define the semantics of GTFL\(^\oplus\).

**Step 2. Lifting the classic interpretation.**

Recall that \(\langle \gamma?, \alpha? \rangle\) from Defs 26 and 28 form a Galois connection between G\(\text{Type}\) and \(\mathcal{P}(\text{TYPE})\). Our first step is to lift this connection to operate on sets of gradual types with the unknown type, i.e. to relate \(\mathcal{P}_{\text{fin}}(\text{GType})\) and \(\mathcal{P}_{\text{fin}}(\mathcal{P}(\text{TYPE}))\). The powerset lifting of \(\gamma?\), denoted \(\hat{\gamma}\), is simply the piecewise application of \(\gamma?\):

**Definition 32** \((\mathcal{P}_{\text{fin}}(\text{GType}) \text{ Concretization})\). \(\hat{\gamma}(G) = \{ \gamma?(G) \mid G \in \hat{G} \}\)

Similarly, the powerset lifting of the abstraction function \(\alpha?\), denoted \(\hat{\alpha}\), is the union of the piecewise application of \(\alpha?\):

**Definition 33** \((\mathcal{P}_{\text{fin}}(\text{GType}) \text{ Abstraction})\). \(\hat{\alpha}(\emptyset) = \text{undefined} \quad \hat{\alpha}(\hat{T}) = \bigcup_{T \in \hat{T}} \alpha?(T)\)

As expected, \(\langle \hat{\gamma}, \hat{\alpha} \rangle\) is a proper Galois connection.

**Proposition 33** \((\hat{\alpha} \text{ is Sound and Optimal})\). If \(\hat{T}\) is not empty, then

\[ a) \hat{T} \subseteq \hat{\gamma}(\hat{T}) \quad b) \hat{T} \subseteq \hat{\gamma}(\hat{G}) \Rightarrow \hat{\alpha}(\hat{T}) \subseteq \hat{G} \]

**Step 3. Introducing the union interpretation.**

We define a Galois connection between U\(\text{TYPE}\) and \(\mathcal{P}_{\text{fin}}(\text{GType})\) by naturally extending the definition of the connection between ST\(\text{TYPE}\) and \(\mathcal{P}_{\text{fin}}(\text{TYPE})\) from Defs 29 and 30, so that it now operates over types in U\(\text{TYPE}\) instead of only types in ST\(\text{TYPE}\).

**Definition 34** \((\text{U}\text{TYPE} \text{ Concretization})\). \(\gamma_\oplus : \text{U}\text{TYPE} \to \mathcal{P}_{\text{fin}}(\text{GType})\)

\[
\gamma_\oplus(\text{Int}) = \{ \text{Int} \} \quad \gamma_\oplus(\text{Bool}) = \{ \text{Bool} \} \quad \gamma_\oplus(?) = \{ ? \} \\
\gamma_\oplus(U_1 \to U_2) = \{ T_1 \to T_2 \mid T_1 \in \gamma_\oplus(U_1) \land T_2 \in \gamma_\oplus(U_2) \} \quad \gamma_\oplus(U_1 \oplus U_2) = \gamma_\oplus(U_1) \cup \gamma_\oplus(U_2) 
\]

Compared to Definition 29, the only additional case to consider is that the unknown type \(?\) can now occur: it is handled like other nullary type constructors, by concretizing to a singleton.

The abstraction is direct from Definition 30.
Definition 35 (UType Abstraction). \( \alpha_\oplus : \mathcal{P}_{\text{fin}}(\mathcal{P}(\text{Type})) \to \text{UType} \)

\[
\alpha_\oplus(\hat{G}) = \oplus\hat{G} \quad \text{if } \hat{G} \neq \emptyset
\]

where \( \oplus\hat{G} \) denotes the gradual union of all the types in the set \( \hat{T} \).

Again, \( \langle \gamma_\oplus, \alpha_\oplus \rangle \) is a Galois connection.

Proposition 34 (\( \alpha_\oplus \) is Sound and Optimal). If \( \hat{G} \) is not empty, then

\[
a) \hat{G} \subseteq \gamma_\oplus(\alpha_\oplus(\hat{G})). \quad b) \hat{G} \subseteq \gamma_\oplus(U) \Rightarrow \alpha_\oplus(\hat{G}) \sqsubseteq U.
\]

Step 4. Composing the connections.

We can now compose the two Galois connections in order to define a stratified interpretation for \( \text{UType} \) in terms of sets of sets of static types.

Definition 36 (Concretization). \( \gamma : \text{UType} \to \mathcal{P}_{\text{fin}}(\mathcal{P}(\text{Type})), \gamma = \overline{\gamma} \circ \gamma_\oplus \)

Definition 37 (Abstraction). \( \alpha : \mathcal{P}_{\text{fin}}(\mathcal{P}(\text{Type})) \to \text{UType}, \alpha = \alpha_\oplus \circ \overline{\alpha} \)

Because the composition of two Galois connection is a Galois connection, the stratified interpretation \( \langle \gamma, \alpha \rangle \) is a Galois connection.

Proposition 35 (\( \alpha \) is Sound and Optimal). If \( \hat{T} \) is not empty, then

\[
a) \hat{T} \subseteq \gamma(\alpha(\hat{T})). \quad b) \hat{T} \subseteq \gamma(U) \Rightarrow \alpha(\hat{T}) \sqsubseteq U.
\]

The notion of precision for gradual types used above is similarly induced by concretization, i.e. \( U_1 \sqsubseteq U_2 \iff \gamma(U_1) \subseteq \gamma(U_2) \). Note that these definitions use containment over sets of sets, defined as \( \hat{T}_1 \subseteq \hat{T}_2 \iff \forall \hat{T}_1 \in \hat{T}_1, \exists \hat{T}_2 \in \hat{T}_2, \hat{T}_1 \subseteq \hat{T}_2 \). Precision can equivalently be defined in terms of the lifted classic abstraction, i.e. \( U_1 \sqsubseteq U_2 \iff \gamma_\oplus(U_1) \sqsubseteq \gamma_\oplus(U_2) \), where \( \hat{G}_1 \sqsubseteq \hat{G}_2 \iff \overline{\gamma}(G_1) \subseteq \overline{\gamma}(G_2) \).

Illustration.

Let us come back to the example of §5.3.3 that motivated the need for a stratified interpretation of \( \text{UType} \).

\[
\gamma((? \to \text{Int}) \oplus (\text{Int} \to ?)) = \{ \overline{\gamma}(? \to \text{Int}), \overline{\gamma}(\text{Int} \to ?) \}
\]
\[
= \{ \{ T \to \text{Int} \mid T \in \text{Type} \}, \{ \text{Int} \to T \mid T \in \text{Type} \} \}
\]
The application of the static function to all the possible static types denoted by the involved predicate lifting of the predicate through the Galois connection. In other words, for a given binary predicates and functions, such as \( \text{et al.} [44] \). This is because the novelty of gradual unions is encapsulated in gradual type existential functions. For instance, the consistent lifting of a predicate over static types is the \( \gamma \) rules present no surprise with respect to the gradual language with \( \sim \). Consequently, terms \( t \) are lifted to gradual terms \( \bar{t} \in \text{UTERM} \), i.e. terms with gradual type annotations. The typing rules present no surprise with respect to the gradual language with \( ? \) presented by Garcia et al. [44]. This is because the novelty of gradual unions is encapsulated in gradual type predicates and functions, such as \( \sim \), \( \cap \), \( \widetilde{\text{dom}} \), etc.

Let us recall from AGT that the Galois connection specifies how to lift predicates and functions. For instance, the consistent lifting of a predicate over static types is the existential lifting of the predicate through the Galois connection. In other words, for a given binary predicate \( P \in \text{TYPE}^2 \), its consistent lifting \( \bar{P} \in \text{GTYPE}^2 \) is defined as: \( \bar{P}(U_1, U_2) \iff \exists T_1 \in \gamma(U_1), \exists T_2 \in \gamma(U_2), P(T_1, T_2) \). Similarly for functions: a lifted function is the abstraction of the application of the static function to all the possible static types denoted by the involved

\[
U \in \text{UType}, \quad x \in \text{Var}, \quad \bar{t} \in \text{UTerm}, \Gamma \in \text{Var}^{\text{fin}} \text{UType}
\]

\[
\begin{align*}
U & ::= U \oplus U \mid \text{Int} \mid \text{Bool} \mid U \to U \quad \text{(types)} \\
v & ::= n \mid \text{true} \mid \text{false} \mid (\lambda x : U \bar{t}) \quad \text{(values)} \\
\bar{t} & ::= v \mid x \mid \bar{t} \bar{t} \mid \bar{t} + \bar{t} \mid \text{if } \bar{t} \text{ then } \bar{t} \text{ else } \bar{t} : U \quad \text{(terms)}
\end{align*}
\]

\[
\frac{}{\Gamma \vdash x : U} \quad (Ux) \quad \frac{}{\Gamma \mid b : \text{Bool}} \quad (Ub) \quad \frac{}{\Gamma \mid n : \text{Int}} \quad (Un)
\]

\[
\frac{\Gamma, x : U_1 \vdash \bar{t} : U_2}{\Gamma \vdash (\lambda x : U_1 \bar{t}) : U_1 \to U_2} \quad (U\lambda) \quad \frac{\Gamma \vdash \bar{t}_1 : U_1 \quad \Gamma \vdash \bar{t}_2 : U_2}{\Gamma \vdash \bar{t}_1 + \bar{t}_2 : \text{Int}} \quad (U+) \quad \frac{\Gamma \vdash \bar{t}_1 : U_1 \quad \Gamma \vdash \bar{t}_2 : U_2}{\Gamma \vdash \text{if } \bar{t}_1 \text{ then } \bar{t}_2 \text{ else } \bar{t}_3 : U_1 \cap U_3} \quad (U\text{if})
\]

\[
\frac{U_1 \sim \text{Int} \quad U_2 \sim \text{Int}}{\Gamma \vdash \bar{t}_1 + \bar{t}_2 : \text{Int}} \quad (U:\sim) \quad \frac{\text{dom : UType } \to \text{UType} \quad \text{cod : UType } \to \text{UType}}{\text{dom}(U) = \alpha(\text{dom}(\gamma(U))) \quad \text{cod}(U) = \alpha(\text{cod}(\gamma(U)))}
\]

Figure 5.2: GTFL\textsuperscript{⊕}: Syntax and typing rules

we can now recover exactly the same gradual type

\[
\alpha(\{\{T \to \text{Int} \mid T \in \text{TYPE}\}, \{\text{Int} \to T \mid T \in \text{TYPE}\}\}) = \alpha\oplus\alpha(\{\{T \to \text{Int} \mid T \in \text{TYPE}\}\}) \cup \alpha(\{\text{Int} \to T \mid T \in \text{TYPE}\})) = \alpha\oplus(\text{Int} \to ?) = (\text{Int} \to ?) + (\text{Int} \to ?)
\]

5.3.5 Static Semantics of GTFL\textsuperscript{⊕}

The syntax and type system of GTFL\textsuperscript{⊕} is presented in Figure 5.2. The syntax is the same as that of STFL, save for the introduction of gradual types \( U \). Consequently, terms \( t \) are lifted to gradual terms \( \bar{t} \in \text{UTERM} \), i.e. terms with gradual type annotations. The typing rules present no surprise with respect to the gradual language with \( ? \) presented by Garcia et al. [44]. This is because the novelty of gradual unions is encapsulated in gradual type predicates and functions, such as \( \sim \), \( \cap \), \( \widetilde{\text{dom}} \), etc.
gradual types. Formally, $\tilde{f} = \alpha \circ \tilde{f} \circ \gamma$, where $\tilde{f}$ is the pointwise application of $f$ to all elements.

**Lifting for stratified interpretation.**

We need to adapt these definitions from AGT to our stratified setting; indeed, our Galois connection relates gradual types with sets of sets of static types, rather than just sets of static types.

We can base our liftings of predicates and types on inclusion and pointwise application that are extended to sets of sets.

**Definition 38 (Predicate Lifting).**

$\tilde{P}(U_1, U_2) \iff \exists T_1 \in \gamma(U_1), T_2 \in \gamma(U_2), P(T_1, T_2)$ where $\tilde{\in}$ is the existential lifting of $\in$ to powersets: $T \in \tilde{T} \iff \exists \tilde{T} \in \tilde{T}, T \in \tilde{T}$

Equivalently: $\tilde{P}(U_1, U_2) \iff \exists \tilde{T}_1 \in \gamma(U_1), \exists \tilde{T}_2 \in \gamma(U_2), \exists T_1 \in \tilde{T}_1, \exists T_2 \in \tilde{T}_2, P(T_1, T_2)$

The lifting of a predicate can also be defined in terms of each of the composed interpretations:

**Proposition 36.** $\tilde{P}(U_1, U_2) \iff \exists G_1 \in \gamma_{\oplus}(U_1), \exists G_2 \in \gamma_{\oplus}(U_2), \tilde{P}_{\gamma}(G_1, G_2)$ where $\tilde{P}_{\gamma}$ is the predicate $P$ lifted with $\gamma_{\oplus}$.

The lifting of a type function $f$ uses the pointwise application of $f$ to all elements of each subset of a powerset, which we note $\tilde{f}$.

**Definition 39 (Function Lifting).**

$\tilde{f} = \alpha \circ \tilde{f} \circ \gamma$

Again, we can define the lifting using the separate abstractions: $\tilde{f} = \alpha_{\oplus} \circ \tilde{f}_\gamma \circ \gamma_{\oplus}$

**Example liftings.**

Let us look at the lifting of a type predicate and a type function. We start with consistency, $\sim$, which corresponds to the lifting of type equality: two gradual types are consistent if some static types in their concretization are equal.

**Definition 40 (Consistency).** $U_1 \sim U_2$ if and only if $\exists \tilde{T}_1 \in \gamma(U_1), \exists T_1 \in \tilde{T}_1, \exists \tilde{T}_2 \in \gamma(U_2), \exists T_2 \in \tilde{T}_2, T_1 = T_2$.

This definition is equivalent to the following inductive definition:
Proposition 37.

\[
\begin{align*}
U \sim U_1 & \quad U \sim U_2 \quad U_1 \sim U \quad U_2 \sim U \\
U \sim U_1 \oplus U_2 & \\
? \sim U & \\
U \sim ? & \\
U_2 \sim U_1 & \\
U_{11} \sim U_{12} & \\
U_{21} \sim U_{22} & \\
U_{21} \sim U_{11} & \\
U_{12} \sim U_{22} & \\
U_{21} \sim U_{21} & \\
U \sim U & \\
? \sim U & \\
U \sim U & \\
U_{11} \rightarrow U_{12} \sim U_{21} & \\
U_{12} \sim U_{22} & \\
\end{align*}
\]

Let us now consider the (precision) meet of gradual types, which corresponds to the lifting of the *equate* function used in the typing rule for conditionals. Its algorithmic definition is:

**Definition 41** (Gradual Meet). Let $\sqcap : \text{UType} \rightarrow \text{UType}$ be defined as:

1. $U \sqcap U = U$
2. $? \sqcap U = U \sqcap ? = U$
3. $U \sqcap (U_1 \oplus U_2) = (U_1 \oplus U_2) \sqcap U = \begin{cases} U \sqcap U_1 & \text{if } U \sqcap U_2 \text{ is undefined} \\ U \sqcap U_2 & \text{if } U \sqcap U_1 \text{ is undefined} \\ (U \sqcap U_1) \oplus (U \sqcap U_2) & \text{otherwise} \end{cases}$
4. $(U_{11} \rightarrow U_{12}) \sqcap (U_{21} \rightarrow U_{22}) = (U_{11} \sqcap U_{21}) \rightarrow (U_{12} \sqcap U_{22})$
5. $U_1 \sqcap U_2$ is undefined otherwise.

This algorithmic definition coincides with the lifting of equate:

**Proposition 38.** $\sqcap = \alpha \circ \text{equate} \circ \gamma$

### 5.4 GTFL$\oplus$: Dynamic Semantics and Properties

Instead of presenting directly the reference dynamic semantics derived with AGT we follow an alternative approach. Following the tradition [109], we first give the dynamic semantics of GTFL$\oplus$ programs by a cast insertion translation to an internal language with explicit casts. We describe the internal language GTFL$\oplus_\leadsto$, which is adapted from the (blameless\footnote{We do not include blame as it is out of scope of this thesis.}) threesome calculus of Siek et al. [112], and then present a cast insertion translation from GTFL$\oplus$ to GTFL$\oplus_\leadsto$. The properties of GTFL$\oplus$ are presented using the translation to GTFL$\oplus_\leadsto$, but the proofs are done by proving equivalence between translated GTFL$\oplus_\leadsto$ terms and the reference semantic of AGT (§5.5).

**Intermediate language.**

GTFL$\oplus_\leadsto$ is an adaptation of the original threesome calculus without blame [112]. A threesome $\langle T_2 \xrightarrow{\cdot} T_1 \rangle$ is a cast composed of three types: the source type $T_1$, the target type $T_2$, and...
The middle type $T_3$. Initially, the middle type of a threesome is the greatest lower bound (in terms of precision), or meet, of the source and target types. The key benefit of threesomes is that two threesomes can be merged into a single threesome by taking the meet of their middle types, hence avoiding space issues [40].

The syntax and typing rules are presented in Figure 5.3. The syntax of GTFL⊕ is a simple extension of that of GTFL□, with cast expressions $\langle U_2 \downarrow U_1 \rangle t$ and casted values $\langle u \downarrow U_1 \rangle$, where $u$ denotes the simple values of GTFL□. In the typing rule for cast expressions (IT),$ the consistency premises are required for a threesome to be well-formed (this is always the case by construction). The other typing rules are basically those of GTFL□, except that type consistency is replaced with type equality; this is because uses of consistency will be guarded by the insertion of casts. For instance, in rule (ITapp), while the GTFL□ typing rule (Uapp, Figure 5.2) uses the premise $U_2 \sim \text{dom}(U_1)$, the new rule requires the type of $t_2$ to exactly be $\text{dom}(U_1)$.

Figure 5.4 presents the dynamic semantics of GTFL⊕, which are similar to [112]. Two threesomes that coincide on their source/target types are combined by meeting their middle types. If the meet is undefined then the term steps to error. Otherwise both casts are merged to a new cast where the middle type is now the meet between the middle types. Note that casts are introduced using the following metafunction, which avoids producing useless threesomes:

$$\langle U_2 \downarrow U_1 \rangle t = t \text{ if } U_1 = U_2 = U_3; \text{ and } \langle U_2 \downarrow U_1 \rangle t \text{ otherwise}$$
\[
\begin{align*}
  u & ::= \text{true} \mid \text{false} \mid n \mid \lambda x.t \quad v ::= u \mid \langle U \Downarrow U \rangle u \\
  f & ::= \Box + t \mid v + \Box \mid \Box t \mid v \Box \mid \langle U \Downarrow U \rangle \Box \mid \text{if } \Box \text{ then } t \text{ else } t
\end{align*}
\]

(values)

(frames)

\[
\begin{align*}
  \Gamma \vdash t & \rightarrow t \\
  n_3 &= \text{rval}(v_1) \; \langle + \rangle \; \text{rval}(v_2) \\
  \Gamma \vdash \lambda x.t \; v & \rightarrow [v/x]t
\end{align*}
\]

(\[\lambda x.t\] v \rightarrow [v/x]t)

\[
\begin{align*}
  \text{if } v \text{ then } t_1 \text{ else } t_2 & \rightarrow \\
  & \begin{cases} 
    t_2 & \text{if } \text{rval}(v) = \text{true} \\
    t_3 & \text{if } \text{rval}(v) = \text{false}
  \end{cases}
\end{align*}
\]

\[
\begin{align*}
  \langle U_3 \Downarrow U_2 \rangle \langle U_2 \Downarrow U_1 \rangle v & \rightarrow \\
  & \begin{cases} 
    \langle U_3 \Downarrow U_2 \rangle \langle U_1 \rangle v & \text{error if } U_{32} \cap U_{21} \text{ is undefined}
  \end{cases}
\end{align*}
\]

\[
\begin{align*}
  \Gamma \vdash t & \rightarrow \Gamma \\
  t_1 & \rightarrow t_2 \quad t_1 \rightarrow t_2 \quad f[t_1] & \rightarrow f[t_2] \\
  f[\text{error}] & \rightarrow \text{error}
\end{align*}
\]

Figure 5.4: GTFL\{\oplus\}\Rightarrow Dynamic Semantics of GTFL\{\oplus\}

Cast insertion.

A GTFL\{\oplus\} program is elaborated through a type-driven cast insertion translation. The key idea of the transformation is to insert casts in places where consistency is used to justify the typing derivation. For instance, if \( \bar{t} : \text{Int} \oplus \text{Bool} \) is used where \( \text{Int} \) is required, the translation inserts a cast \( \langle \text{Int} \leftarrow \text{Int} \oplus \text{Bool} \rangle t \), where \( t \) is the recursive translation of \( \bar{t} \). This cast plays the role of the implicit projection from the gradual union type. Dually, when a term of type \( \text{Int} \) is used where a gradual union is expected, the translation adds a cast that performs the implicit injection to the gradual union, \( e.g. \langle \text{Int} \oplus \text{Bool} \leftarrow \text{Int} \rangle \). Note that a value with a cast that loses precision is like a tagged value in tagged union type systems; the difference again is that the “tag” is inserted implicitly.

The translation judgment has the form \( \Gamma \vdash \bar{t} \Rightarrow t : U \): under type environment \( \Gamma \), GTFL\{\oplus\} term \( \bar{t} \) of type \( U \), is translated to GTFL\{\oplus\} term \( t \). The translation rules given in Figure 5.5 are standard. Cast insertion rules use twosomes to ease readability; a twosome \( \langle U_2 \leftarrow U_1 \rangle t \) is equal to \( \langle U_2 \Downarrow U_1 \rangle t \): the initial middle type is the meet of both ends [112].

Properties of GTFL\{\oplus\}.

GTFL\{\oplus\} satisfies a number of properties. First GTFL\{\oplus\} satisfies a standard type safety property:

**Proposition 39** (Type safety). Suppose that \( \cdot \vdash \bar{t} \Rightarrow t : U \), then either: \( t \) is a value \( v \); \( t \rightarrow \text{error} \); or \( t \rightarrow t' \) for some \( t' \) such that \( \cdot \vdash t' : U \)
guarantees must then be established separately. Usually, one establishes type safety of the internal language and then proving following a route that differs from prior work. A technical novelty of our work is that we establish all the properties presented in §5.5 Correctness of the Translational Semantics.

Proposition 40 (Equivalence for fully-annotated terms). For any $t \in \text{TERM}$, $\vdash t : T$ if and only if $\vdash t : T$

Precision on terms, noted $\bar{t}_1 \subseteq \bar{t}_2$, is the natural lifting of type precision to terms. The gradual type system satisfies the static and dynamic gradual guarantees:

Proposition 41 (Static gradual guarantee). If $\vdash \bar{t}_1 : U_1$ and $\bar{t}_1 \subseteq \bar{t}_2$, then $\vdash \bar{t}_2 : U_2$, for some $U_2$ such that $U_1 \subseteq U_2$.

Proposition 42 (Dynamic gradual guarantee). Suppose $\cdot \vdash \bar{t}_1 \Rightarrow t_1 : U_1$, $\vdash \bar{t}_2 \Rightarrow t_2 : U_2$, and $t_1 \subseteq t'_1$. If $t_1 \mapsto t_2$ then $t'_1 \mapsto t'_2$ where $t_2 \subseteq t'_2$.

5.5 Correctness of the Translational Semantics

A technical novelty of our work is that we establish all the properties presented in §5.4 following a route that differs from prior work. Usually, one establishes type safety of the gradual language by first proving type safety of the internal language and then proving that the cast insertion translation preserves typing \[109\]. With this approach, the gradual guarantees must then be established separately \[113\].

Our approach, presented in Figure 5.6, exploits the AGT methodology: we first systemat-
Figure 5.6: Proof of dynamic properties approach

Figure 5.7: Gradual Intrinsic Terms for GTFL⊕

GTLF⊕

\[\begin{array}{ll}
\text{Static semantics} & \text{Dynamic semantics} \\
\text{Translation} & \text{Type Safety} \\
\text{Gradual Guarantee} & \text{Equivalence}
\end{array}\]

\[\begin{align*}
(IUx) & \quad U x \in \text{TERM}_U \\
(IU\lambda) & \quad x U_2 \in \text{TERM}_{U_2} \\
(IU\) & \quad (\lambda x U_1 . t U_2) \in \text{TERM}_{U_1 \rightarrow U_2} \\
(IUb) & \quad b \in \text{TERM}_{\text{Bool}} \\
(IUn) & \quad n \in \text{TERM}_{\text{Int}} \\
(IU+) & \quad x U_1 \in \text{TERM}_{U_1} \\
(IU\text{if}) & \quad \text{if } x U_1 \text{ then } x U_2 \text{ else } x U_3 \in \text{TERM}_U \\
\end{align*}\]

Figure 5.8: Static and Dynamic Semantics of Intrinsic Terms

Figure 5.9: Gradual Intrinsic Terms for GTFL⊕

Directly derive the direct runtime semantics of GTFL⊕ (i.e. which do not rely on a cast insertion translation). We then prove safety and the gradual guarantees, which in fact directly follow from AGT. Then, we prove that the compilation to threesome combined with the semantics of the internal language together are equivalent to the dynamic semantics derived with AGT. This correctness argument proceeds using logical relations.

**Direct dynamic semantics of GTFL⊕** Following AGT, we first derive the dynamic semantics from the type safety proof argument, using intrinsic terms instead of type derivation trees. Figures C.5, 5.8, and 5.9 presents the static and dynamic semantics of the intrinsic terms, and are completely analogous to the system presented by Garcia et al. [44].

**Equivalence of the translational semantics and the direct semantics** We establish the equivalence between the semantics using step-indexed logical relations. We use step-indexed logical relations so the relation is well-founded: the definition without indexes may contain some vicious cycles in presence of gradual unions. Equivalence between two terms is acknowledged when either both evaluate to the same value, or both lead to an error.
\[\varepsilon \in \text{Evidence}, \quad et \in \text{EvTerm}, \quad ev \in \text{EvValue}, \quad t \in \text{Term},
\]
\[v \in \text{Value}, \quad u \in \text{SimpleValue}, \quad g \in \text{EvFrame}, \quad f \in \text{TmFrame}
\]
\[et ::= \varepsilon t\]
\[ev ::= \varepsilon u\]
\[u ::= x | n | b | \lambda x.t\]
\[v ::= u | \varepsilon u:::U\]
\[g ::= \Box + et | ev + \Box | \Box @U | \Box :::U | \text{if } \Box \text{ then } et \text{ else } et\]
\[f ::= g[\varepsilon \Box]\]

### Notions of Reduction

\[
\begin{align*}
-\rightarrow: & \; \text{TERM}_U \times (\text{TERM}_U \cup \{\text{error}\}) \\
-\rightarrow_c: & \; \text{EvTerm} \times (\text{EvTerm} \cup \{\text{error}\})
\end{align*}
\]

\[
\varepsilon n_1 + \varepsilon n_2 \longrightarrow n_3 \text{ where } n_3 = n_1 [+ ] n_2
\]
\[
\varepsilon (\lambda x^{U_{11}}.t) @U_1 \longrightarrow U_2 \; \varepsilon u \longrightarrow \begin{cases} 
\text{icod}(\varepsilon_1)([(\varepsilon_2 \circ \text{idom}(\varepsilon_1))u :: U_{11}] / x^{U_{11}})]t :: U_2 \\
\text{error} \quad \text{if not defined}
\end{cases}
\]
\[
\text{if } \varepsilon b \text{ then } \varepsilon_2 U_2 \text{ else } \varepsilon_3 U_3 \longrightarrow \begin{cases} 
\varepsilon_2 U_2 :: U_2 \ominus U_3 & b = \text{true} \\
\varepsilon_3 U_3 :: U_2 \ominus U_3 & b = \text{false}
\end{cases}
\]
\[
\varepsilon (\varepsilon_2 U :: U) \longrightarrow_c \begin{cases} 
(\varepsilon_2 \circ \varepsilon_1)u \\
\text{error} \quad \text{if not defined}
\end{cases}
\]

Figure 5.8: Syntax and notions of Reduction

Figure 5.10 presents the logical relations between simple values, values and computations, which are defined mutually recursively. The logical relations are defined for pairs composed of an intrinsic term \( \hat{t} \), which denotes the typing derivation for a GTFL \( \oplus \) term \( t \), and a GTFL \( \Rightarrow \) term \( t \). For simplicity, we write \( t : U \) for \( \cdot \vdash i \; t : U \).

A pair of simple values \((\hat{u}_1, u_2)\) are related for \( k \) steps at type \( U \), notation \((\hat{u}_1, u_2) \in U_k[\overline{U}]\), if they both have the same type \( U \) and, if \( U \) is either \textbf{Bool} or \textbf{Int}, then the values are also equal. If the simple values are functions, then they are related if their application to related arguments , for \( j \leq k \) steps, yields related computations, as explained below. Note that the relation between simple values need not consider the case of gradual types, as no literal values have gradual types.

A pair of values \((\hat{v}_1, v_2)\) are related for \( k \) steps at type \( U \), notation \((\hat{v}_1, v_2) \in V_k[\overline{U}]\), if both have the same type and their underlying simple values are related. One important point to notice is that we may only relate an ascribed value \( \varepsilon \hat{u}_1 :: U \) to a simple value \( u_2 \) if we do not learn anything new from the ascription, \( i.e. \) both the type of \( \hat{u}_1 \) and the evidence \( \varepsilon \) are \( U \). This corresponds to the case where the reference semantics carries useless evidence—recall that the cast insertion translation does not insert useless casts. Additionally, an ascribed value \( \varepsilon \hat{u}_1 :: U \) is related to a casted value if the evidence and ascription correspond to the threesome. More precisely, the evidence \( \varepsilon \) must be exactly the middle type of the threesome,
and the source and target types of the threesome must correspond to the type of $\bar{u}_1$ and the ascribed type $U$, respectively. Finally, in order to reason about the underlying simple values, the ascription and cast must be eliminated by combining them with an evidence and a cast respectively. Because of this extra step, the underlying simple values must be related for $k - 1$ steps instead.

A pair of terms $(\bar{t}_1, t_2)$ are related computations for $k$ steps at type $U$, notation $(\bar{t}_1, t_2) \in T_k[U]$, if both terms have the same type $U$, then either both terms reduce to related values at type $U$, or both terms reduce to an error. Formally, for any $j < k$, if the evaluation of the intrinsic term $\bar{t}_1$ terminates in a value $v_1$ at least in $j$ steps, then the compiled term $t_2$ also reduces to a value $v_2$, and the resulting values are related values for $k - j$ steps at type $U$. Analogously, if the evaluation of the compiled term $t_2$ reduces to a value $v_2$ at least in $j$ steps, then the intrinsic term $\bar{t}_1$ also reduces to a related value $v_1$. Finally, if either term reduces to an error in at least $j$ steps, then the other also reduces to an error. Note that this last condition is only required because we do not assume type safety of GTFL$^\oplus$. 

![Figure 5.9: Intrinsic Reduction](image)

![Figure 5.10: Logical relations between intrinsic terms and cast calculus terms](image)
Armed with these logical relations we can state the notion of semantic equivalence between a GTFL$^\oplus$ intrinsic term and a GTFL$^\oplus$ term.

**Definition 42** (Semantic equivalence). Let $\tilde{t} \in \text{TERM}_U$, $\Gamma = \text{FV}(\tilde{t})$ and a GTFL$^\oplus$ term $t$ such that $\Gamma \vdash t : U$. We say that $\tilde{t}$ and $t$ are semantically equivalent, notation $\tilde{t} \approx t : U$, if and only if for any $k \geq 0$, $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\Gamma]$, we have $(\sigma_1(\tilde{t}), \sigma_2(t)) \in \mathcal{T}_k[U]$.

The definition of semantic equivalence appeals to the notion of related substitutions. Two substitutions $\sigma_1$ and $\sigma_2$ are related for $k$ steps at type environment $\Gamma$, notation $(\sigma_1, \sigma_2) \in \mathcal{G}_k[\Gamma]$, if they map each variable in $\Gamma$ to related values.

Note that we write $\tilde{t}$ instead of $t^U$ when it is clear from the context that it is an intrinsic term.

**Definition 43.** Let $\sigma$ be a substitution and $\Gamma$ a type substitution. We say that substitution $\sigma$ satisfy environment $\Gamma$, written $\sigma \models \Gamma$, if and only if $\text{dom}(\sigma) = \text{dom}(\Gamma)$.

**Definition 44** (Related substitutions). Let $\sigma_1$ be a substitution function from intrinsic variables to intrinsic values, and let $\sigma_2$ be a substitution function from variables to values from the intermediate language. Then we define related substitution as follows:

$$(\sigma_1, \sigma_2) \in \mathcal{G}_k[\Gamma] \iff \sigma_1 \models \Gamma \land \forall x \in \Gamma.(\sigma_1(x^{\Gamma(x)}), \sigma_2(x)) \in \mathcal{V}_k[\Gamma(x)]$$

Finally, semantic equivalence between the reference and the translational semantics says that given a well-typed term $\tilde{t}$ from the gradual source language, its corresponding intrinsic term $\tilde{t}$ is semantically equivalent to the cast calculus term $t$ obtained after the cast insertion translation.

**Proposition 43** (Equivalence of reference and translational semantics). If $\Gamma \vdash \tilde{t} : U$, represented as the intrinsic term $\tilde{t} \in \text{TERM}_U$, and $\Gamma \vdash \tilde{t} \Rightarrow t : U$, then $\tilde{t} \approx t : U$.

### 5.6 Related Work

In § 5.2, we have compared gradual unions to both tagged and untagged unions from the standard type system literature [92], highlighting their key characteristics and differences. Gradual unions are unique in admitting runtime errors, with the benefits of more flexible programming patterns. We have also compared gradual unions to retrofitted type systems for dynamic languages with support for unions. Note that in Flow, TypeScript, CDuce and Typed Racket, a function that expects an argument of type $A + B$ can accept arguments of type $A$, $B$, or $A + B$, but neither of type $A + B + C$ nor $A + D$. In contrast, in GTFL$^\oplus$, as long as two gradual union types have at least one compatible type in their denotation, then they are compatible. So, a function that expects an argument of type $A \oplus B$ accepts arguments of types such as $A \oplus B \oplus C$ and $A \oplus D$.

Flow-sensitive typing approaches such as occurrence typing [74] support more precise type assignments based on the result of some (type) predicate check. Such techniques can avoid
the insertion of unnecessary casts. However, in general, the combination of gradual types with type tests raises questions regarding the dynamic gradual guarantee, which have not yet been answered.

Interestingly, languages with set-theoretic (untagged) unions usually also consider intersection types, with distributivity relations such as \((T_1 \lor T_2) \rightarrow T_3 \equiv (T_1 \rightarrow T_3) \land (T_2 \rightarrow T_3)\). This law states that if a function accepts a value that is either of type \(T_1\) or of type \(T_2\), then it behaves as both a function of type \(T_1 \rightarrow T_3\) and a function of type \(T_2 \rightarrow T_3\). Gradual unions encompass both interpretations, without having to resort to a notion of intersection types: \((T_1 \oplus T_2) \rightarrow T_3 \equiv (T_1 \rightarrow T_3) \oplus (T_2 \rightarrow T_3)\), because both types have the same interpretation, i.e. they represent the same concrete set of static types. This simplicity is a consequence of the optimistic interpretation with dynamic checks that is characteristic of gradual typing. (Note that it resonates with the fact that type precision is covariant in both positive and negative positions.)

Siek and Tobin-Hochstadt studied the interaction between gradual typing and union types. While seemingly related, the focus of their work is very different: the addition of the unknown type to a language with static union types. Additionally, they only support the union of types with different type constructors, so for instance the union of two function types is not supported.

Similarly, in parallel with this work, Castagna and Lanvin developed a theory for gradual set-theoretic types, supporting union, intersection and the unknown type. Their system can express constructs similar to gradual unions by using a combination of unions and intersections with the unknown type, e.g. \(\text{Int} \oplus \text{Bool}\) is equivalent to \((\text{Int} \mid \text{Bool})\&?\). They also exploit AGT to derive the static semantics, although the more expressive setting with static unions and intersections makes the design of the Galois connection much more challenging. Our design is minimalist, providing a novel form of union types to languages that do not initially support such set-theoretic types. They mention compilation to threesomes and proving the gradual guarantees as future work.

Jafery and Dunfield present a gradual language that features two types of (datasort) refinement sums, for either exhaustive or non-exhaustive matches. Non-exhaustive matches are backed by dynamic checks in case of an unsuccessful match. Elements with a sum type must be explicitly injected; the sum constructors are neither commutative nor associative. Also, they do not discuss the interaction with the fully-unknown type.

5.7 Conclusion

Inspired by the interpretation of gradual types as a general approach to deal with imprecision at the type level, and recognizing that unions types are a form of imprecision, we proposed the novel notion of gradual union types. Gradual unions are a new design for dealing with the possibility for expressions to have different, unrelated types. Accepting the possibility of runtime cast errors, gradual unions combine and extend the convenience of both tagged and untagged union types. We have presented the meta-theory of gradual union types and
their interaction with the traditional unknown type, using the AGT methodology. We have described a compilation semantics to a threesome calculus, and established its desired properties through logical relations. The combination of both gradual type constructors forced us to explore a stratified approach to AGT, whereby each gradual type constructor is interpreted separately and then carefully composed in order to ensure optimality of the resulting abstraction. This compositional approach to designing a gradual language is novel. We hope that it helps understanding how to combine different gradualization efforts that have been developed independently, and may not be fully orthogonal.
Chapter 6

Gradual Parametricity, Revisited

Bringing the benefits of gradual typing to a language with parametric polymorphism like System F, while preserving relational parametricity, has proven extremely challenging: first attempts were formulated a decade ago, and several designs were recently proposed. Among other issues, these proposals can however signal parametricity errors in unexpected situations, and improperly handle type instantiations when imprecise types are involved. These observations further suggest that existing polymorphic cast calculi are not well suited for supporting a gradual counterpart of System F.

In this chapter we further stress the AGT methodology applying it to derive a gradual language with support to explicit parametric polymorphism. We introduce GSF, a gradual polymorphic language which addresses the design issues identified in prior work and satisfies parametricity.§6.8. We explicitly lay out the design principles, goals and non-goals of GSF and introduce the language briefly through examples.§6.3. We then explain how we derive GSF from a variant of System F called SF, by following the AGT methodology. While the statics of GSF follow naturally from SF using AGT, the dynamic semantics are more challenging. §6.6/§6.7. GSF satisfies the expected properties of gradual languages, §6.5/§6.7, except the dynamic gradual guarantee. This property was left open as a conjecture in prior work; here we prove that it is in fact incompatible with parametricity. §6.9. We uncover a novel, weaker property that GSF satisfies, which allows us to disprove several claims related to gradual free theorems for imprecise type signatures. §6.10.

6.1 Introduction

Gradual typing has been explored numerous times in a setting with parametric polymorphism [6, 7, 70, 69, 130]. A long-standing challenge has been to prove that the gradual language preserves a rich semantic property known as relational parametricity [101], which dictates that a polymorphic value must behave uniformly for all possible instantiations. The first gradual language to come with a proof of parametricity is the cast calculus λB [7], re-

\footnote{1This chapter is based on the work of Toro et al. [122].}
cently used as a target language for CSA, a language with implicit polymorphism developed by Xie et al. [130]. Another recent effort is System F\(_G\), an actual gradual source language (i.e. without explicit casts), which is compiled to a cast calculus akin to \(\lambda B\), called System F\(_C\) [69].

**Contributions.** This work starts from the novel identification of design issues in existing proposals, especially in their dynamic semantics. In short, both lexical scoping of type variables and type instantiations can be violated when imprecise types are involved. Consequently, we argue that neither \(\lambda B\) nor System F\(_C\) are adequate targets for an explicitly-parametric gradual language (§6.2).

- We introduce GSF, a gradual counterpart of System F that addresses the design issues identified in prior work and satisfies parametricity (§6.8).
- We apply the AGT methodology to derive GSF (§6.4), and we uncovered that a direct application breaks parametricity (§6.7.1). To recover parametricity we have to take into account multiple considerations regarding the dynamic semantics (§6.7.2).
- We prove properties of the static semantics such as the static gradual guarantee, and the static equivalence for static terms (§6.5.3).
- We prove properties of the dynamic semantics such as type safety, a weak version of the conservative extension of the dynamic semantics of Siek et al. (§6.7.3), and last but not least, parametricity which was the big challenge of this work (§6.8).
- We show that parametricity is incompatible with the dynamic gradual guarantee (§6.9).
- We establish a novel property of GSF regarding the wrapping of System F terms into less precise types, which allows us to disprove some claims from the literature about gradual free theorems (§6.10).

Complete definitions and proofs of the main results can be found in §D. Additionally, GSF is implemented as an interactive prototype that exhibits both typing derivations and reduction traces. All the examples mentioned in this work, as well as others, are readily available in the online demo: http://pleiad.cl/gsf/.

### 6.2 The Need to Revisit Gradual Parametricity

We start by exposing different issues in both the static and dynamic semantics of existing gradual polymorphic languages.
6.2.1 Static Semantics Issues

While the static semantics of simple gradual languages are uncontroversial, devising the static semantics of gradual polymorphic languages has proven to be fairly challenging, yielding systems that are arguably hard to grasp. We highlight the most salient issues with $\lambda B$ and System $F_G$ below, and then relate to CSA, which addresses them to some extent.

**Mixing Explicit and Implicit Polymorphism.** Both $\lambda B$ and System $F_G$ are languages with *explicit* polymorphism, i.e. with explicit type abstraction and type application terms. However, instead of focusing on explicit polymorphism only, both languages accommodate some form of implicitness, but with different flavors. Consider the type of a polymorphic identity function, $\forall X.X \rightarrow X$. In $\lambda B$ this type is compatible with $\text{Int} \rightarrow \text{Int}$, which is a defining feature of *implicit* polymorphism. More surprisingly, this type is also compatible with $\text{Int} \rightarrow \text{Bool}$. (Runtime errors will account for the obvious mistake.) This means in particular that $\lambda B$ is not a proper conservative extension of System $F$, as both type systems disagree on some fully static terms. Technically, instead of the traditional consistency relation, $\lambda B$ introduces two close but distinct relations on types, called convertibility and compatibility, in order to orchestrate these non-trivial semantics. Conversely, System $F_G$ relies on a notion of consistency, and is a proper conservative extension of System $F$. As an explicitly polymorphic language, System $F_G$ does not relate $\forall X.X \rightarrow X$ with any of its static instantiations. However, it *does* relate that type with $? \rightarrow ?$, on the basis that using the unknown type should bring some of the flexibility of implicit polymorphism.

**Ad-hoc Precision.** Conversely to System $F_G$, $\lambda B$ has no notion of type precision, and does not discuss any of the gradual guarantees. The precision relation of System $F_G$ features some constraints that might be surprising to programmers. Specifically, System $F_G$ allows loss of precision only in non-parametric positions of a polymorphic type. For instance, $\forall X.X \rightarrow \text{Int}$ is considered more precise than $\forall X.X \rightarrow ???$, but unrelated to $\forall X.? \rightarrow \text{Int}$. Because precision induces consistency, it means that $\forall X.X \rightarrow \text{Int}$ and $\forall X.? \rightarrow \text{Int}$ are inconsistent with each other. This choice is motivated by the desire to avoid a counterexample of the gradual guarantee: they claim that a function of type $\forall X.? \rightarrow X$ must fail on all inputs in order to respect parametricity (we disprove this claim in §6.10), so accepting that this type is less precise than $\forall X.X \rightarrow X$ breaks the dynamic gradual guarantee.

But tailoring the precision relation to avoid a class of counterexamples is not benign. First, changing the definition of precision to accommodate a theorem does not necessarily result in a programmer’s expectations being adjusted. Let us recall that the gradual guarantees were introduced to formally capture the expectations of programmers using gradual languages. The restriction on precision imposed by System $F_G$ breaks the intuition of programmers that, starting program from a well-typed program, removing static type information yields a program that is *by definition* less precise—and should also be well-typed.

Second, the restricted rule excludes instances of precision that are harmless for the dynamic gradual guarantee. For instance, in System $F_G$, $\forall X.X \rightarrow X$ is not more precise than $\forall X.X \rightarrow ???$, despite the fact that a function of type $\forall X.X \rightarrow ?$ can be a proper identity function (§6.10).
Third, Igarashi et al. [69] only prove the static guarantee based on this ad hoc precision, and leave the dynamic guarantee as a conjecture, so it is unclear whether the restriction on precision imposed by System F\(_G\) is indeed sufficient.

Separating Concerns. Recently, Xie et al. [130] raise similar concerns about the static semantics of \(\lambda B\) and System F\(_G\), in particular regarding the mixing of explicit and implicit polymorphism. In response, they clearly separate the subtyping relation induced by implicit polymorphism from the consistency relation induced by gradual types. Their notion of consistent subtyping extends the notion of Siek and Taha [110]. As a result, CSA features intuitive and straightforward definitions of precision and consistency, while accommodating the flexibility of implicit polymorphism in full.

We fully concur with the necessity to untangle implicitness from consistency in order to achieve a principled design. Xie et al. leave open the question of designing an explicitly-polymorphic gradual language. Additionally, Xie et al. do not deal with the dynamic semantics of their language beyond a translation to \(\lambda B\). Therefore CSA inherits both the virtues of \(\lambda B\), such as parametricity, and its issues, uncovered next.\(^2\)

6.2.2 Dynamic Semantics Issues

In the design of gradually-typed languages, cast calculi are typically used as target languages to give runtime semantics to gradual programs. However, as observed by Garcia et al. [44], there is little justification or guidance available to design or choose a cast calculus for interpreting a given gradual source language. To this date, only the AGT and The Gradualizer [27] methodologies provide a systematic approach to derive the dynamic semantics of gradual languages.

Since the early work on the polymorphic blame calculus [5, 6], all existing work has built upon variants of that cast calculus. While a cast language like \(\lambda B\) can be used as a source language [7], \(\lambda B\) has been used in recent work as the target language of choice for gradual source languages [69, 130]. In this section, we identify two questionable design decisions in both \(\lambda B\) and System F\(_C\) that arguably make them inadequate as internal languages of a gradual version of System F.

Excess of Failure. Consider the following example, written in System F\(_G\) (the \(\lambda B\) and System F\(_C\) versions are more verbose because of explicit casts):

\[
\text{let } f \;::\; \forall X.X \rightarrow ? = \Lambda X.\lambda x:X.x \text{ in } (f \;[\text{Int}] \;1) + 1
\]

What would the programmer expect out of this program? While the annotated return type of \(f\) is left unknown, the function itself is the System F identity function. Therefore, one might expect that instantiating the function to \text{Int}, passing 1 and adding 1, should yield 2 as a result.

\(^2\)The implicit polymorphism of Xie et al. [130] faces other challenges, most notably the lack of coherence of the runtime semantics. This issue is entirely related to implicit polymorphism and is therefore not addressed here.
However, in both $\lambda B$ and System $F_C$, the above program fails with a runtime error. The reason is that the result of $f$ [Int] 1 is sealed, and therefore unusable. Ahmed et al. [6] justify this behavior (already present in early work [5]), or the alternative of always failing before returning, based on a claim about gradual free theorems. Intuitively, this can be surprising because the underlying value is the System F identity function, which does behave parametrically; it is therefore unclear what parametricity violation is being reported. As we will see later, this failing behavior is in fact not formally demanded by parametricity ($\S$ 6.10).

**Lack of Failure.** A major interest of gradual types is that they soundly augment the expressiveness of the original static type system. Let us illustrate first in a simply-typed setting (STLC refers to the simply-typed lambda calculus with base types):

Consider the STLC term $t = \lambda x : \_x$, which behaves as the identity function. $t$ is incomplete because the type annotation on $x$ is missing so far. $t$ is operationally valid at different types, but it cannot be given a general type in STLC. Its type has to be fixed at either $\text{Int} \to \text{Int}$, $\text{Bool} \to \text{Bool}$, etc. Intuitively, a proper characterization of $t$ requires going from simple types to parametric polymorphism, such as System F. In System F, we could use the type $\forall X.X \to X$ to precisely specify that $t$ can be applied with any argument type and return the same type. With a gradual variant of STLC, we can give term $t$ the imprecise type $\_ \to \_$. This lack of precision is soundly backed by runtime enforcement, such that the term $(t \ 3) \ 1$ evaluates to a runtime type error.

The exact same line of reasoning should apply when starting from System F, as follows:

Consider the System F term $t = \lambda x : \_ (x [\text{Int}])$, which behaves as an instantiation function. $t$ is incomplete because the type annotation on $x$ is missing so far. $t$ is operationally valid at different types, but it cannot be given a general type in System F. It has to be fixed at either $(\forall X.X \to X) \to (\text{Int} \to \text{Int})$, $(\forall XY.X \to Y \to X) \to (\forall Y.\text{Int} \to Y \to \text{Int})$, etc. Intuitively, a proper characterization of $t$ requires going from System F to higher-order polymorphism, such as System $F_\omega$. In System $F_\omega$, we could use the type $\forall P. (\forall X. P X) \to (P \ \text{Int})$ to precisely specify that $t$ instantiates any polymorphic argument to $\text{Int}$. With a gradual variant of System F, we ought to be able to give term $t$ the imprecise type $(\forall X.\_ \to \_)$ to statically capture the fact that $t$ is definitely a function, without committing to specific domain and codomain types. This lack of precision ought to be soundly backed by runtime enforcement, such that, given $id : \forall X.X \to X$, the term $(t \ id) \ true$ should evaluate to a runtime type error.

However, the runtime semantics of $\lambda B$ and System $F_C$ suffer from a fundamental issue that breaks the argument above: they do not respect type instantiations that involve the unknown type, and consequently do not fail as expected.

Below is another simple example in System $F_G$ in which the polymorphic identity function is instantiated to $\text{Int}$ and passed a $\text{Bool}$ value:

```
let g : ? = \A.X.\x:X.x in g [Int] true
```

3In System $F_C$, $(t \ id) \ true$ fails because $\forall X.?$ is not deemed consistent with $\forall X.X \to X$. Consequently, $t$ must be declared to take an argument of type $\_ \to \_$ instead of $\forall X.?$. The result is the same as in $\lambda B$ however: no runtime error is raised.
This System Fₖ program (and its translation to λB) returns true, despite the explicit instantiation to Int. Internally, this happens because g is first consistently considered to be of type ∀X.? in order to accommodate the type instantiation, but then the instantiation yields a substitution of Int for X in ?, which in both languages is just ?. There is no tracking of the decision to instantiate the underlying value to Int. Consequently, current polymorphic cast calculi such as λB and System Fₖ are inadequate to serve as the runtime support of a gradual variant of System F.

6.3 GSF, Informally

This chapter presents the design, semantics and metatheory of GSF, a gradual counterpart of System F that addresses the issues raised above. This section focuses on the informal aspects of GSF: design principles and methodology, as well as some illustrative examples of GSF in action.

6.3.1 Design Principles, Goals and Non-Goals

Considering the many concerns involved in developing a gradual language with parametric polymorphism, we should be very clear about the principles, goals and non-goals of a specific design. In designing GSF, we respect the following design principles:

Explicit polymorphism: GSF is a gradual counterpart to System F, and as such, is a fully explicitly polymorphic language: type abstraction and type application are part of the term language, reflected in types. GSF gradualizes type information, not term structure.

Simple statics: GSF must embody the complexity of dynamically enforcing parametricity solely in its dynamic semantics; its static semantics should be as straightforward as possible.

Natural precision: Precision is intended to capture the level of static typing information of a gradual type, with ? as the least precise and static types as the most precise [113]. GSF should preserve this simple intuition.

The mandatory goals for GSF, i.e. the properties that it should definitely satisfy, are:

Type safety: GSF should be type safe, meaning all programs should either evaluate to a value, halt with a runtime error, or diverge. Well-typed GSF terms should not get stuck.

Conservative extension: GSF should be a conservative extension of System F: both languages should coincide in their static and dynamic semantics for fully static programs.

Faithful instantiations: GSF should respect type instantiations. In particular, explicit instantiations of imprecise types should be enforced (§6.2.2).

Parametricity: GSF should enforce the notion of parametricity understood for gradual programs [7]. In particular, a polymorphic function should behave uniformly across all its
instantiations—i.e. always take related inputs to related outputs, or always fail or diverge.

**Static gradual guarantee:** By virtue of the simple statics principle stated above, GSF should satisfy the static gradual guarantee, i.e. typeability should be monotonic with respect to the natural notion of precision.

Similarly important are the explicit *non-goals* that we adopt when designing GSF:

**Dynamic gradual guarantee:** While GSF should strive to satisfy the dynamic gradual guarantee, this should *not* be at the expense of any of the above-stated principles and goals. In other words, the dynamic gradual guarantee is the first candidate property to abandon (or weaken) if need be.

**Implicit polymorphism:** GSF is a gradual counterpart to System F, and as such, is a fully explicitly polymorphic language. While implicit polymorphism is certainly a desirable feature for usability, the integration of implicit polymorphism in GSF is future work.

**Blame tracking:** Tracking blame in order to report more informative error messages is very important, but that is beyond the scope of this thesis.

**Performance:** We focus on the semantics and meta-theoretical properties of GSF, without explicitly taking into account efficiency considerations such as pay-as-you-go [109, 69], space efficiency [60, 112], cast elimination [99], etc. Optimizing the dynamic semantics of GSF is left for future work.

### 6.3.2 GSF in Action

Recall the example from §2.3.3 in which a function \( f \) defined as:

\[
\text{let } f = \lambda g : (\forall X.X \rightarrow X).g \ [\text{Int}] \ 10
\]

is applied to a function \( h \) of unknown type. GSF behaves exactly as described with each of the three variant implementations of \( h \), namely:

\[
\begin{align*}
\text{let } h : ? &= \Lambda X.\lambda x:X.x \text{ in } f \ h & \text{----> 10} \\
\text{let } h : ? &= \Lambda X.\lambda x:? .x \text{ in } f \ h & \text{----> 10} \\
\text{let } h : ? &= \Lambda X.\lambda x:? .x+1 \text{ in } f \ h & \text{----> error}
\end{align*}
\]

In the last case, the runtime error is raised when the body of the function attempts to perform an addition, since this type-specific operation is a violation of parametricity.

The fact that GSF adopts explicit polymorphism *à la* System F means that a polymorphic type is not consistent with any of its instantiations. In practice, this means that:

\[
\text{let } h : ? = \lambda x:? .x \text{ in } f \ h \quad \text{----> error}
\]

The runtime error occurs when the body of \( f \) performs the type application, because the value bound to \( g \) is not of the appropriate constructor (\( \Lambda \)). If changing the definition of \( h \) to
include the \( \Lambda \) constructor is not an option, one can perform this adaptation explicitly upon application of \( f \):

\[
\text{let } h : ? = \lambda x:?.x \text{ in } f (\Lambda X.h) \quad \longrightarrow \quad 10
\]

Finally, GSF does not report spurious parametricity violations, and enforces type instantiations even when applied to an imprecisely-typed value:

\[
\text{let } f : \forall X.X \rightarrow ? = \Lambda X.\lambda x:X.x \text{ in } (f \left[\text{Int}\right] 1) + 1 \quad \longrightarrow \quad 2
\]
\[
\text{let } g : ? = \Lambda X.\lambda x:X \text{ in } g \left[\text{Int}\right] \text{true} \quad \longrightarrow \quad \text{error}
\]

Hence GSF addresses the issues in the dynamic semantics of \( \lambda B \) and System F\(_C\), and soundly augments the expressiveness of System F (§6.2.2). Other illustrative examples are available online at http://pleiad.cl/gsf/.

## 6.4 Preliminary: The Static Language SF

We systematically derive GSF by applying AGT to a largely standard polymorphic language similar to System F, called SF (Figure 6.1). In addition to the standard System F types and terms, SF includes base types \( B \) inhabited by constants \( b \), typed using the auxiliary function \( ty \), and primitive n-ary operations \( op \) that operate on base types and are given meaning by the function \( \delta \). SF also includes pairs \( \langle t_1, t_2 \rangle \), and the associated projection operations \( \pi_i(t) \)\(^4\) as well as type ascriptions \( t :: T \).

The statics are standard. The typing judgment is defined over three contexts: a type name store \( \Sigma \) (explained below), a type variable set \( \Delta \) that keeps track of type variables in scope, and a standard type environment \( \Gamma \) that associates term variables to types. We adopt the convention of using partial type functions to denote computed types in the rules: \( \text{dom} \) and \( \text{cod} \) for domain and codomain types, \( \text{inst} \) for the resulting type of an instantiation, and \( \text{proj}_i \) for projected types. These partial functions are undefined if the argument is not of the appropriate shape. We also make the use of type equality explicit as a premise whenever necessary. These conventions are helpful for lifting the static semantics to the gradual setting \[14\]. For closed terms, we write \( \cdot;\cdot;\cdot; \vdash t : T \), or simply \( \vdash t : T \).

The dynamics are standard call-by-value semantics, specified using reduction frames. The only peculiarity is that they rely on runtime type generation: upon type application, a fresh type name \( \alpha \) is generated and bound to the instantiation type \( T \) in a global type name store \( \Sigma \). The notion of reduction and reduction rules all carry along the type name store. While type names only occur at runtime, and not in source programs, reasoning about SF terms as they reduce requires accounting for programs with type names in them. This is why the typing rules are defined relative to a type name store as well. Similarly, type equality is relative to a type name store: a type name \( \alpha \) is considered equal to its associated type in the store. The recursive definition of equality modulo type names is necessary to derive equalities \[69\]. For instance, in the reduction of the well-typed program \((id \left[\text{Int} \rightarrow \text{Int}\right]) \ (id \left[\text{Int}\right])\), where \( id \)

\[4\]We omit the constraint \( i \in \{1, 2\} \) when operating on pairs throughout this chapter.
\[
x \in \text{VAR}, X \in \text{TYPEVAR}, \alpha \in \text{TYPENAME} \\
T \ ::= \ B \mid T \to T \mid \forall X . T \mid T \times T \mid X \mid \alpha \quad \text{(types)} \\
t \ ::= \ b \mid \lambda x : T . t \mid \Delta X . t \mid (t, t) \mid x \mid t :: T \mid \text{op}(\bar{t}) \mid t \ f t \mid t [T] \mid \pi_i(t) \quad \text{(terms)} \\
v \ ::= \ b \mid \lambda x : T . t \mid \Delta X . t \mid \langle v, v \rangle \\
\]

\[
\Sigma; \Delta; \Gamma \vdash t : T \\
\]

\[
\begin{array}{c}
\text{Well-typed terms} \\
(\text{Tb}) \quad \frac{ty(b) = B}{\Sigma; \Delta; \Gamma \vdash b : B} \\
(\text{TA}) \quad \frac{\Sigma; \Delta; \Gamma \vdash t : T \quad \Sigma; \Delta \vdash \sigma}{\Sigma; \Delta; \Gamma \vdash \sigma(t) : T} \\
(\text{Tx}) \quad \frac{x : T \in \Gamma}{\Sigma; \Delta; \Gamma \vdash x : T} \\
(\text{Top}) \quad \frac{\Sigma; \Delta; \Gamma \vdash t \quad \Sigma; \Delta; \Gamma \vdash \Delta X . t \vdash \forall X . T}{\Sigma; \Delta; \Gamma \vdash \Delta \vdash \forall X . T} \\
(\text{Tapp}) \quad \frac{\Sigma; \Delta; \Gamma \vdash t_1 : T_1 \quad \Sigma; \Delta; \Gamma \vdash t_2 : T_2 \quad \Sigma; \Delta; \Gamma \vdash \sigma}{\Sigma; \Delta; \Gamma \vdash \sigma(t_1, t_2) : T_1 \times T_2} \\
(\text{TappT}) \quad \frac{\Sigma; \Delta; \Gamma \vdash t \quad \Sigma; \Delta; \Gamma \vdash \Gamma \vdash \langle t \rangle \vdash \text{inst}(T, T')}{\Sigma; \Delta; \Gamma \vdash \Gamma \vdash \text{inst}(T, T')} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Type equality} \\
\Sigma; \Delta \vdash B = B \quad \Sigma; \Delta \vdash X = X \\
\Sigma; \Delta \vdash \forall X . T_1 = \forall X . T_2 \\
\Sigma; \Delta \vdash T_1 = T_2 \quad \Sigma; \Delta \vdash T_1 \to T_2 = T_2 \to T_1 \\
\Sigma; \Delta \vdash \alpha \in \text{dom}(\Sigma) \quad \Sigma; \Delta \vdash \alpha = \alpha \\
\Sigma; \Delta \vdash \alpha = T \quad \Sigma; \Delta \vdash T = \Sigma(\alpha) \\
\end{array}
\]

\[
\Sigma \triangleright t \to \Sigma' \triangleright t' \\
\]

\[
\begin{array}{c}
\Sigma \triangleright \sigma \triangleright \Delta \triangleright t \to \Sigma' \triangleright \sigma' \triangleright t' \\
\Sigma \triangleright \text{term frames and reduction} \\
\end{array}
\]

\[
\begin{array}{c}
f \ ::= \ \Box :: T \mid \text{op}(\bar{t}, \Box, \bar{t}) \mid \Box \mid v \mid \Box \mid \langle v, \Box \rangle \mid \pi_i(\Box) \quad \text{(term frames)} \\
\Sigma \triangleright f[t] \to \Sigma' \triangleright f'[t'] \\
\end{array}
\]

Figure 6.1: SF: Simple Static Polymorphic Language with Runtime Type Generation
is the polymorphic identity function, the equality $\alpha := \text{Int} \rightarrow \text{Int}, \beta := \text{Int}; \Delta \vdash \alpha = \beta \rightarrow \beta$ should be derivable.

Rules in Figure 6.1 appeal to auxiliary well-formedness judgments, which can be found in §[D.1]. A type $T$ is well-formed ($\Sigma; \Delta \vdash T$) if it only contains type variables in the type variable environment $\Delta$, and type names bound in a well-formed type name store. A type name store is well-formed ($\vdash \Sigma$) if all type names are distinct, and associated to well-formed types. A type environment $\Gamma$ binds term variables to types, and is well-formed ($\Sigma; \Delta \vdash \Gamma$) if all types are well-formed.

The decision of using type names instead of the traditional substitution semantics is in anticipation of gradualization: indeed, prior work has shown that runtime type generation is crucial in order to be able to distinguish between different type variables instantiated with the same type [78, 6, 7]. We follow the approach already in SF because we want the dynamics and type soundness argument of the static language to help us with GSF.

Unsurprisingly, SF is type safe, and all well-typed terms are parametric. These results also follow from the properties of GSF, and the strong relation between both languages.

6.5 GSF: Statics

The first step of the AGT methodology is to define the syntax of gradual types and give them meaning through a concretization function to the set of static types they denote. Then, by finding the corresponding abstraction function to establish a Galois connection, the static semantics of the static language can be lifted to the gradual setting.

6.5.1 Syntax and Syntactic Meaning of Gradual Types

We introduce the syntactic category of gradual types $G \in \text{GType}$, by admitting the unknown type in any position, namely:

$$G ::= B \mid G \rightarrow G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ?$$

Observe that static types $T$ are syntactically included in gradual types $G$.

The syntactic meaning of gradual types is straightforward: the unknown type represents any type. Perhaps surprisingly, we can simply extend this syntactic approach to deal with universal types, type variables, and type names; the concretization function $C$ is defined in Figure 6.2. Note that the definition is purely syntactic and does not even consider well-formedness (?) stands for any static type); notions built above concretization, such as consistency, will naturally embed the necessary restrictions (§6.5.2).

---

5To be more in connection to the literature, in this chapter we use different letters to denote the concretization and abstraction function, because letters $\gamma$ and $\alpha$ are used for substitution and type names respectively.
\[ C(B) = \{ B \} \]
\[ C(G_1 \to G_2) = \{ T_1 \to T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2) \} \]
\[ C(G_1 \times G_2) = \{ T_1 \times T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2) \} \]
\[ C(X) = \{ X \} \]
\[ C(\alpha) = \{ \alpha \} \]
\[ C(\forall X.G) = \{ \forall X.T \mid T \in C(G) \} \]
\[ C(\square) = \text{TYPE} \]

\[ A(\{ B \}) = B \]
\[ A(\{ T_{i1} \to T_{i2} \}) = A(\{ T_{i1} \}) \to A(\{ T_{i2} \}) \]
\[ A(\{ T_{i1} \times T_{i2} \}) = A(\{ T_{i1} \}) \times A(\{ T_{i2} \}) \]
\[ A(\{ T \}) = X \]
\[ A(\{ \alpha \}) = \alpha \]
\[ A(\{ \forall X.T \}) = \forall X.A(\{ T \}) \]
\[ A(\{ \square \}) = ? \text{ otherwise, } \{ \square \} \neq \emptyset \]
\[ A(\emptyset) \text{ is undefined} \]

Figure 6.2: Type concretization \((C)\) and abstraction \((A)\)

\[ G \sqsubseteq G \]
\[ B \sqsubseteq B \]
\[ X \sqsubseteq X \]
\[ G_1 \sqsubseteq G_1' \quad G_2 \sqsubseteq G_2' \quad G_1 \to G_2 \sqsubseteq G_1' \to G_2' \]
\[ \forall X.G_1 \sqsubseteq \forall X.G_2 \]
\[ G_1 \times G_2 \sqsubseteq G_1' \times G_2' \]
\[ \alpha \sqsubseteq \alpha \]

Figure 6.3: GSF: Inductive definition of type precision

Following the abstract interpretation framework, the notion of precision is not subject to tailoring: precision coincides with set inclusion of the denoted static types [44].

**Definition 45** (Type Precision). \( G_1 \sqsubseteq G_2 \) if and only if \( C(G_1) \subseteq C(G_2) \).

**Proposition 44** (Precision, inductively). The inductive definition of type precision given in Figure 6.3 is equivalent to Definition 45.

Observe that both \( \forall X.X \to ? \) and \( \forall X.? \to X \) are more precise than \( \forall X.? \to ? \), and less precise than \( \forall X.X \to X \), thereby reflecting the original intuition about precision [113]. Also \( \forall X.? \to ? \) and \( ? \to ? \) are unrelated by precision, since they correspond to different constructors (and GSF is a language with explicit polymorphism); they are both more precise than ?, of course.

Dual to concretization is abstraction, which produces a gradual type from a set of static types. The abstraction function \( A \) is direct (Figure 6.2) and is both sound and optimal.

**Proposition 45** (Galois connection). \( \langle C, A \rangle \) is a Galois connection, i.e.:  

a) (Soundness) for any non-empty set of static types \( S = \{ T \} \), we have \( S \subseteq C(A(S)) \)

b) (Optimality) for any gradual type \( G \), we have \( A(C(G)) \subseteq G \).
Type consistency

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊢ Ξ</td>
<td>Ξ; Δ ⊢ B ∼ B</td>
</tr>
<tr>
<td>⊢ Ξ</td>
<td>Ξ; Δ ⊢ X ∈ Δ</td>
</tr>
<tr>
<td>⊢ Ξ</td>
<td>Ξ; Δ ⊢ X ∼ X</td>
</tr>
<tr>
<td>⊢ Ξ</td>
<td>Ξ; Δ ⊢ G_1 ∼ G_1'</td>
</tr>
<tr>
<td>⊢ Ξ</td>
<td>Ξ; Δ ⊢ G_2 ∼ G_2'</td>
</tr>
<tr>
<td>⊢ Ξ; Δ, X ⊢ G_1 ∼ G_2</td>
<td></td>
</tr>
<tr>
<td>⊢ Ξ; Δ ⊢ ∀ X.G</td>
<td></td>
</tr>
<tr>
<td>⊢ Ξ; Δ ⊢ G_1 × G_2 ∼ G_1' × G_2'</td>
<td></td>
</tr>
<tr>
<td>⊢ Ξ; Δ ⊢ α ∈ dom(Ξ)</td>
<td></td>
</tr>
<tr>
<td>⊢ Ξ; Δ ⊢ α ∼ α</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: GSF: Inductive definition of type consistency

```
dom^♯ : GTYPE → GTYPE
cod^♯ : GTYPE → GTYPE

dom^♯(G_1 → G_2) = G_1

cod^♯(G_1 → G_2) = G_2

dom^♯(?) = ?

cod^♯(?) = ?

dom^♯(G) undefined o/w

cod^♯(G) undefined o/w

inst^♯ : GTYPE^2 → GTYPE

inst^♯(∀ X.G, G') = G'[G'/X]

proj^♯ : GTYPE → GTYPE

proj^♯(G_1 × G_2) = G_i

proj^♯(?) = ?

inst^♯(G, G') undefined o/w

proj^♯(G) undefined o/w
```

Figure 6.5: GSF: Inductive definitions of type functions

6.5.2 Lifting the Static Semantics

Following AGT we lift predicates and functions using existential lifting. Our only predicate in SF is type equality, whose existential lifting is type consistency:

**Definition 46** (Consistency). Ξ; Δ ⊢ G_1 ∼ G_2 if and only if Σ; Δ ⊢ T_1 = T_2 for some Σ ∈ C(Ξ), T_1 ∈ C(G_i).

For closed types we write G_1 ∼ G_2. This definition uses a *gradual* type name store Ξ, which binds type names to gradual types. Its concretization is the obvious pointwise concretization:

C(·) = ∅

C(Ξ, α := G) = { Σ, α := T | Σ ∈ C(Ξ), T ∈ C(G) }

Note that because consistency is the consistent lifting of static type equality, which does impose well-formedness, consistency is only defined on well-formed types (i.e. Ξ; Δ ⊢ X ∼ X does not hold).

**Proposition 46** (Consistency, inductively). The inductive definition of type consistency given in Figure 6.4 is equivalent to Definition 46.

Again, observe that the resulting definition of consistency relates any two types that only differ in unknown type components, without any restriction. Also, because of explicit polymorphism, top-level constructors must match, so ? → ? is not consistent with ∀ X.? → ?.
\( x \in \text{Var}, X \in \text{TypeVar}, \alpha \in \text{TypeName} \quad \Xi \in \text{TypeName} \xrightarrow{\text{fin}} \text{GType}, \)

\( \Delta \subset \text{TypeVar}, \Gamma \in \text{Var} \xrightarrow{\text{fin}} \text{GType} \)

\[
\begin{align*}
G & ::= B \mid G \rightarrow G \mid \forall X.G \mid G \times G \mid X \mid \alpha \mid ? \\
(\text{gradual types})
\end{align*}
\]

\[
\begin{align*}
t & ::= b \mid \lambda x : G.t \mid \Lambda X.t \mid \langle t, t \rangle \mid x \mid t :: G \mid \text{op}(\overline{t}) \mid t \mid t [G] \mid \pi_i(t) \\
(\text{gradual terms})
\end{align*}
\]

\[
\begin{align*}
\Xi; \Delta; \Gamma \vdash t : G & \qquad \text{Well-typed terms} \\
(Gb) & \frac{t(b) = B}{\Xi; \Delta; \Gamma \vdash b : B} \\
(GA) & \frac{\Xi; \Delta, X; \Gamma \vdash t : G}{\Xi; \Delta; \Gamma \vdash \Lambda X.t : \forall X.G} \\
(Gx) & \frac{x : G \in \Gamma}{\Xi; \Delta; \Gamma \vdash x : G} \\
(Gop) & \frac{\Xi; \Delta; \Gamma \vdash t : \overline{G_1}}{\Xi; \Delta; \Gamma \vdash \text{op}(\overline{t}) : G} \\
(GpairG) & \frac{\Xi; \Delta; \Gamma \vdash t : G}{\Xi; \Delta; \Gamma \vdash [G'] : \text{inst}^\sharp(G, G')} \\
(GappG) & \frac{\Xi; \Delta; \Gamma \vdash t : G'}{\Xi; \Delta; \Gamma \vdash \pi_i(t) : \text{proj}^\sharp(G)}
\end{align*}
\]

Figure 6.6: GSF: Syntax and Static Semantics

However, in line with gradual typing, both are consistent with ?, as expected. Therefore GSF does not treat ? → ? as a special “quasi-polymorphic” type, unlike System F. Rather, consistency in GSF coincides with that of CSA. Consistent lifting of functions requires abstraction: a lifted function is the abstraction of the results of applying the static function to all the denoted static types:

**Definition 47** (Consistent lifting of functions). Let \( F_n \) be a function of type \( \text{Type}^n \rightarrow \text{Type} \). Its consistent lifting \( F^\sharp_n \), of type \( \text{GType}^n \rightarrow \text{GType} \), is defined as:

\[
F^\sharp_n(\overline{G}) = \text{A} \{ F_n(T) \mid T \in \overline{C(G)} \}
\]

The abstract interpretation framework allows us to prove the following definitions:

**Proposition 47** (Consistent type functions). The definitions of \( \text{dom}^\sharp, \text{cod}^\sharp, \text{inst}^\sharp, \text{proj}^\sharp \) given in Fig. 6.5 are consistent liftings, as per Def. 47, of the corresponding functions from Fig. 6.1.

Finally, the gradual typing rules of GSF are given in Figure 6.6. Of interest here is the fact that a term of unknown type can be optimistically treated as a polymorphic term and hence be instantiated, yielding ? as the result type of the type application.
6.5.3 Static Properties of GSF

As established by Siek and Taha \cite{109} in the context of simple types, we can prove that the GSF type system is equivalent to the SF type system on fully-static terms. We say that a gradual type is static if the unknown type does not occur in it, and a term is static if it is fully annotated with static types. Let $\vdash_S$ denote the typing judgment of SF.$^4$

**Proposition 48** (Static equivalence for static terms). Let $t$ be a static term and $G$ a static type ($G = T$). We have $\vdash_S t : T$ if and only if $\vdash t : T$.

The second important property of the static semantics of a gradual language is the static gradual guarantee (§2.1), where term precision is the natural extension of precision to terms.

**Proposition 49** (Static gradual guarantee). Let $t$ and $t'$ be closed GSF terms such that $t \sqsubseteq t'$ and $\vdash t : G$. Then $\vdash t' : G'$ and $G \sqsubseteq G'$.

6.6 GSF: Evidence-Based Dynamics

We now turn to the dynamic semantics of GSF. As anticipated, this is where the complexity of gradual parametricity manifests. Still, in addition to streamlining the design of the static semantics, AGT provides effective (though incomplete) guidance for the dynamics.

Just as in §4, we simplify the exposition of the dynamic semantics by avoiding the use of intrinsic terms.$^7$ Instead, we rely on a type-directed, straightforward translation that inserts explicit ascriptions everywhere consistency is used—very much in the spirit of the coercion-based semantics of subtyping \cite{92}. For instance, this small program $(\lambda x : ?.x + 1)$ false, is translated to:

$$(\varepsilon_{?\rightarrow \text{Int}}(\lambda x : ?.x :: \text{Int}) + (\varepsilon_{\text{Int}1 :: \text{Int}})) :: ? \rightarrow \text{Int} ~ (\varepsilon_{2}(\varepsilon_{\text{Bool}false :: \text{Bool}}) :: ?)$$

where $\varepsilon_G$ denotes evidence of the reflexive judgment $G \sim G$ (e.g. $\varepsilon_{\text{Int}}$ supports $\text{Int} \sim \text{Int}$), $\varepsilon_1$ supports $? \sim \text{Int}$, and $\varepsilon_2$ supports $\text{Bool} \sim ?$.$^8$

Despite this translation, we do preserve the essence of the AGT dynamics approach in which evidence and consistent transitivity drive the runtime monitoring aspect of gradual typing. Furthermore, by making the translation explicitly ascribe all base values to their base type, we can present a uniform syntax and greatly simplify reduction rules compared to the original AGT exposition. This presentation also streamlines the proofs by reducing the number of cases to consider.

---

$^4$As usual, the propositions here are stated over closed terms, but are proven as corollaries of statements over open terms.

$^7$We use blue color for evidence $\varepsilon$ to enhance readability of the structure of terms in the next section and beyond.

$^8$Such initial evidences are computed by means of an *interior* function, given by the abstract interpretation framework \cite{44}. The definition of interior (§D.3.2) and the type-preserving translation (§D.3.5) are direct.
### Well-typed terms (for conciseness, $s$ ranges over both $t$ and $u$)

- $(Eb)\quad \frac{\vdash t b = B}{\vdash X; \Delta; \Gamma \vdash b : B}$
- $(E\lambda)\quad \frac{\vdash \Delta; \Gamma \vdash X : \forall X . G \quad \vdash \Delta; \Gamma \vdash \lambda x : G . t : G \to G'}{\vdash \Delta; \Gamma \vdash \lambda x : G . t : G \to G'}$
- $(Epair)\quad \frac{\vdash \Delta; \Gamma \vdash s_1 : G_1 \quad \vdash \Delta; \Gamma \vdash s_2 : G_2}{\vdash \Delta; \Gamma \vdash \langle s_1, s_2 \rangle : G_1 \times G_2}$
- $(Ex)\quad \frac{\vdash \Delta; \Gamma \vdash x \in G}{\vdash \Delta; \Gamma \vdash x : G}$
- $(Easc)\quad \frac{\vdash \Delta; \Gamma \vdash s \in G \quad \vdash \Delta; \Gamma \vdash s : G'}{\vdash \Delta; \Gamma \vdash s : G'}$
- $(Eop)\quad \frac{\vdash \Delta; \Gamma \vdash \bar{t} : B \quad \vdash \Delta; \Gamma \vdash op(\bar{t}) : B'}{\vdash \Delta; \Gamma \vdash \bar{t} : B}$
- $(Eapp)\quad \frac{\vdash \Delta; \Gamma \vdash t_1 : G \to G' \quad \vdash \Delta; \Gamma \vdash t_2 : G}{\vdash \Delta; \Gamma \vdash t_1 t_2 : G'}$
- $(EappG)\quad \frac{\vdash \Delta; \Gamma \vdash \forall X . G \quad \vdash \Delta; \Gamma \vdash \pi_i (t) : G'}{\vdash \Delta; \Gamma \vdash \pi_i (t) : G'}$
- $(Epair)\quad \frac{\vdash \Delta; \Gamma \vdash t_1 : G_1 \times G_2}{\vdash \Delta; \Gamma \vdash \pi_i (t) : G_i}$

#### Figures 6.6, 6.7, and 6.8

Figures 6.7 and 6.8 presents the syntax and semantics of GSF$\varepsilon$, a simple variant of GSF in which all values are ascribed, and ascriptions carry evidence. Key changes with respect to Figure 6.6 are highlighted in gray. Here, we treat evidence as a pair of elements of an abstract datatype; we define its actual value (and operations) in the next section.

Because the translation from GSF to GSF$\varepsilon$ introduces explicit ascriptions everywhere consistency is used, the only remaining use of consistency in the typing rules of GSF$\varepsilon$ is in the rule ($Easc$). The evidence of the term itself supports the consistency judgment in the premise. All other rules require types to match exactly; the translation inserts ascriptions to ensure that top-level constructors match in every elimination form.

The notion of reduction for GSF$\varepsilon$ terms deals with evidence propagation and composition with consistent transitivity. Rule ($Rasc$) specifies how an ascription around an ascribed value reduces to a single value if consistent transitivity holds: $\varepsilon_1$ justifies that $G_u \sim G_1$, where $G_u$ is the type of the underlying simple value $u$, and $\varepsilon_2$ is evidence that $G_1 \sim G_2$. The composition via consistent transitivity, if defined, justifies that $G_u \sim G_2$; if undefined, reduction steps to $error$. Rule ($Rop$) simply strips the underlying simple values, applies the primitive operation, and then wraps the result in an ascription, using a canonical base evidence $\varepsilon_B$ (which trivially justifies that $B \sim B$). Rule ($Rapp$) combines the evidence from the argument value $\varepsilon_2$ with the domain evidence of the function value $dom(\varepsilon_1)$ in an attempt to transitively justify that $G_u \sim G_{11}$. Failure to justify that judgment produces $error$. The return value is ascribed to the expected return type, using the codomain evidence of the
\[ \Xi \vdash t \rightarrow \Xi \vdash t \text{ or error} \]

Notion of reduction

\[ (Rasc) \]
\[ \Xi \vdash e_2(e_1 u :: G_1) :: G_2 \rightarrow \begin{cases} \Xi \vdash (e_1 \circ e_2)u :: G_2 \\ \text{error} & \text{if not defined} \end{cases} \]

(Rop)
\[ \Xi \vdash op(e u :: G) \rightarrow \Xi \vdash e_B \delta(op, \overline{a}) :: B \text{ where } B \triangleq \text{cod}(ty(op)) \]

(Rapp)
\[ \Xi \vdash (e_1(\lambda x : G_{11}.t) :: G_1 \rightarrow G_2)(e_2 u :: G_1) \rightarrow \begin{cases} \Xi \vdash \text{cod}(e_1)(t[(e_2 \circ \text{dom}(e_1))u :: G_{11})/x]) :: G_2 \\ \text{error} & \text{if not defined} \end{cases} \]

(Rpair)
\[ \Xi \vdash \langle e_1 u_1 :: G_1, e_2 u_2 :: G_2 \rangle \rightarrow \Xi \vdash (e_1 \times e_2)(u_1, u_2) :: G_1 \times G_2 \]

(Rproji)
\[ \Xi \vdash \pi_1(e(u_1, u_2) :: G_1 \times G_2) \rightarrow \Xi \vdash p_1(e)u_1 :: G_1 \]

(RappG)
\[ \Xi \vdash (\varepsilon \Lambda X.t :: \forall X.G)[G'] \rightarrow \Xi' \vdash e_{\text{out}}(\varepsilon(\bar{\alpha})t[\bar{\hat{\alpha}}/X] :: G[\alpha/X]) :: G[G'/X] \]

where \( \Xi' \triangleq \Xi, \alpha := G', \alpha \notin \text{dom}(\Xi), \bar{\hat{\alpha}} = \text{lift}_{\Xi'}(\alpha) \)

\[ \Xi \vdash t \mapsto \Xi' \vdash t \text{ or error} \]

Evaluation frames and reduction

\[ f ::= \begin{array}{c} \varepsilon \Box :: G \mid \text{op}(\overline{a}, \Box, t) \mid \Box t \mid v \Box \mid \Box [G] \mid (\Box, t) \mid \langle v, \Box \rangle \end{array} \]

\[ (R) \]
\[ \frac{\Xi \vdash t \rightarrow \Xi' \vdash t'}{\Xi \vdash f \mapsto \Xi' \vdash f[t']} \]

\[ (Rerr) \]
\[ \frac{\Xi \vdash t \rightarrow \text{error}}{\Xi \vdash f \mapsto \text{error}} \]

\[ (Rf) \]
\[ \frac{\Xi \vdash f \mapsto \Xi' \vdash f[t']} {\Xi \vdash f \mapsto \Xi' \vdash f[t']} \]

\[ (Rferr) \]
\[ \frac{\Xi \vdash f \mapsto \text{error}} {\Xi \vdash f \mapsto \text{error}} \]

Figure 6.8: GSF\(\varepsilon\): Dynamic Semantics

function \(\text{cod}(e_1)\). Rule (Rpair) produces a pair value when the subterms of a pair have been reduced to values themselves, using the product operator on evidences \(e_1 \times e_2\). This rule is necessary to enforce a uniform presentation of all values as ascribed values, which simplifies technicalities. Dually, Rule (Rproji) extracts a component of a pair and ascribes it to the projected type, using the corresponding evidence obtained with \(p_1(\varepsilon)\).

Apart from the presentational details, the above rules are standard for an evidence-based reduction semantics. Rule (RappG) is the rule that specifically deals with parametric polymorphism, reducing a type application. This is where most of the complexity of gradual parametricity concentrates.

The use of \(\bar{\hat{\alpha}}\) is a technicality: because up to now we treat evidence as an abstract datatype from an as-yet-unspecified domain, say pairs of ETYP\(E\), we cannot directly use gradual types (GTYPE) inside evidences. The connection between GTYPE and ETYP\(E\) is specified by lifting operations, \(\text{lift}_\Xi : \text{GTYPE} \rightarrow \text{ETYP}\) and \(\text{unlift} : \text{ETYP} \rightarrow \text{GTYPE}\). Because type names have meaning related to a store, the lifting is parametrized by the store \(\Xi\). Term

\[ ^9 \text{We use } p_1(\varepsilon) \text{ to avoid confusion with } \pi_1(\varepsilon), \text{ which refers to extracting the first component of evidence (itself a metalanguage pair).} \]

\[ ^{10} \text{In standard AGT the lifting is simply the identity, } i.e. \text{ ETYP} = \text{GTYPE}. \]
substitution is mostly standard: it uses $unlift$ to recover $\alpha$, and is extended to substitute recursively in evidences. Substitution in evidence, also triggered by evidence instantiation, is simply component-wise substitution on evidence types.

Turning to the reduction rule, observe that there are two ascriptions in the produced term:

- The *inner* ascription (to $G[\alpha/X]$) is for the body of the polymorphic term, asserting that substituting a fresh type name $\alpha$ for the type variable $X$ preserves typing. The associated evidence $\varepsilon[\hat{\alpha}]$ is the result of instantiating $\varepsilon$ (which justifies that the actual type of $\Lambda X.t$ is consistent with $\forall X.G$) with the fresh type name, hence justifying that the body after substitution is consistent with $G[\alpha/X]$.

- The *outer* ascription asserts that $G[\alpha/X]$ is consistent with $G[G'/X]$, witnessed by evidence $\varepsilon_{out}$. This evidence plays a key role in avoiding unjustified failures as described in §6.2.2. We define $\varepsilon_{out}$ in §6.7.2 below, once the representation of evidence is introduced.

Finally, the evaluation frames and associated reduction rules in Figure 6.7 are straightforward; in particular ($R$err) and ($Rf$err) propagate error to the top-level.

6.7 Evidence for Gradual Parametricity

We now turn to the actual representation of evidence. We first explain in §6.7.1 why the standard representation of evidence as pair of gradual types is insufficient for gradual parametricity. We then introduce the refined representation of evidence to enforce parametricity (§6.7.2), and basic properties of the language. Richer properties of GSF are discussed in §6.8, §6.9 and §6.10.

6.7.1 Simple Evidence, and Why It Fails

In standard AGT [44], evidence is simply represented as a pair of gradual types, i.e. $\text{ETYPE} = \text{GTYPE}$. Let us recall the definition of consistent transitivity from §3.3.4.

**Definition 48** (Consistent transitivity). Suppose $\varepsilon_{ab} \vdash G_a \sim G_b$ and $\varepsilon_{bc} \vdash G_b \sim G_c$. Evidence for consistent transitivity is deduced as $(\varepsilon_{ab} \circ \varepsilon_{bc}) \vdash G_a \sim G_b$, where:

$$\langle G_1, G_{21} \rangle \circ \langle G_{22}, G_3 \rangle = A^2(\{(T_1, T_3) \in C(G_1) \times C(G_3) \mid \exists T_2 \in C(G_{21}) \cap C(G_{22}), T_1 = T_2 \land T_2 = T_3\})$$

Consistent transitivity satisfies some important properties. First, it is associative. Second, the resulting evidence is more precise than the outer evidence types, reflecting that during evaluation, typing justification only gets more precise (or fails). Violating this property breaks type safety. The third property is key for establishing the dynamic gradual guarantee [44].

**Lemma 50.** (Properties of consistent transitivity).

(a) Associativity. $(\varepsilon_1 \circ \varepsilon_2) \circ \varepsilon_3 = \varepsilon_1 \circ (\varepsilon_2 \circ \varepsilon_3)$, or both are undefined.
(b) Optimality. If $\varepsilon = \varepsilon_1 \circ \varepsilon_2$ is defined, then $\pi_1(\varepsilon) \subseteq \pi_1(\varepsilon_1)$ and $\pi_2(\varepsilon) \subseteq \pi_2(\varepsilon_2)$.

(c) Monotonicity. If $\varepsilon_1 \subseteq \varepsilon_1'$ and $\varepsilon_2 \subseteq \varepsilon_2'$ and $\varepsilon_1 \circ \varepsilon_2$ is defined, then $\varepsilon_1 \circ \varepsilon_2 \subseteq \varepsilon_1' \circ \varepsilon_2'$.

Unfortunately, adopting gradual types for evidence types and simply extending the consistent transitivity definition to deal with GSF types and consistency judgments yields a gradual language that breaks parametricity. To illustrate, consider this simple program:

\[
\lambda X. (\lambda x : X. \text{let } y : ? = x \text{ in let } z : ? = y \text{ in } z + 1)) \ [\text{Int}] \ 1
\]

The function is not parametric because it ends up adding 1 to its argument, although it does so after two intermediate bindings, typed as $?$. Without further precaution, the parametricity violation of this program would not be detected at runtime. Assume that the type application generates the fresh name $\alpha$, bound to Int in the store. For justifying that $x$ can flow to $y$ (the let-binding is equivalent to a function application), we need evidence for $\text{Int} \sim ?$ by consistent transitivity between the evidences $\langle X, X \rangle$, which justifies $\text{Int} \sim \alpha^{11}$, and $\langle \alpha, \alpha \rangle$, which justifies $\alpha \sim ?^{12}$. This evidence is defined, namely $\langle \text{Int}, \alpha \rangle$. Using the definition of consistent transitivity (Def. 48), $\langle \text{Int}, \alpha \rangle \circ \langle \alpha, \alpha \rangle = \langle \text{Int}, \alpha \rangle$. Similarly, for justifying the flow of $y$ to $z$, the previous evidence must be combined with $\langle ?, ? \rangle$, which justifies $? \sim ?^{13}$. Using the definition of consistent transitivity (Def. 48), $\langle \text{Int}, \alpha \rangle \circ \langle ?, ? \rangle = A^2(\{ \langle \text{Int}, \text{Int} \rangle, \langle \text{Int}, \alpha \rangle \}) = \langle \text{Int}, ? \rangle$. This evidence can subsequently be used to produce evidence to justify that the addition is well-typed, since $\langle \text{Int}, ? \rangle \circ \langle \text{Int}, \text{Int} \rangle = \langle \text{Int}, \text{Int} \rangle$. Therefore the program produces 2, without errors; parametricity is violated.

### 6.7.2 Refining Evidence

For gradual parametricity, evidence must do more than just ensure type safety. It needs to safeguard the sealing that type variables are meant to represent, also taking care of unsealing as necessary.

**Evidence types.** We define evidence types, $E \in \text{ETYPE}$, to be an enriched version of gradual types:

\[
E ::= B \mid E \rightarrow E \mid \forall X. E \mid E \times E \mid \alpha^E \mid X \mid ?
\]

SF equality judgments, and hence GSF consistency judgments, are relative to a store. It is therefore not enough to use type names in evidence: we need to keep track of their associated types in the store. An evidence type name $\alpha^E$ therefore captures the type associated to the type name $\alpha$ through the store. For instance, evidence that a variable has a polymorphic type $X$ is initially $\langle X, X \rangle$. When $X$ is instantiated, say to Int, and a fresh type name $\alpha$ is introduced, the evidence becomes $\langle \alpha^{\text{Int}}, \alpha^{\text{Int}} \rangle$. An evidence type name keeps track of its most precise associated type in the store. This is necessary because as a program evolves, evidence

---

11 The obtained language is type safe, and satisfies the dynamic gradual guarantee. This novel design could make sense to gradualize impure polymorphic languages, which do not enforce parametricity. Exploring this perspective is future work.

12 Note that conversely to the simply-typed setting, both components of evidence are not necessarily equal, as in this case.

13 This evidence is obtained by substituting $\alpha$ for $X$ in the initial evidence $\langle X, X \rangle$ for $X \sim ?$. 

---

can not only become more precise than statically-used types, but also than the global store. For instance, it can be the case that \( \alpha := ? \) in the global store \( \Xi \), but that locally, the evidence for \( \alpha \) has gotten more precise, e.g. \( \alpha^{\text{Int}} \). Formally, a type name is enriched with its transitive bindings in the store, lift\(_\Xi\)(\(\alpha\)) = \(\alpha^{\text{lift}}\Xi(\Xi(\alpha))\). Unlifting simply forgets the additional information: unlift\(_\Xi\)(\(\alpha^{E}\)) = \(\alpha\). In all other cases, both operations recur structurally.

Importantly, an evidence type name does not only record the end type to which it is bound, but the whole path. For instance, \(\alpha^{\beta^{\text{Int}}}\) is a valid evidence type name that embeds the fact that \(\alpha\) is bound to \(\beta\), which is itself bound to \(\text{Int}\). It is also crucial to understand the intuition behind the position of type names in a given evidence. The position of \(\alpha^{E}\) in an evidence can correspond to a sealing, an unsealing, or neither. If \(\alpha^{E}\) is only on the right-side, e.g. \(\langle \text{Int}, \alpha^{\text{Int}} \rangle\), then the evidence is a sealing (here, of \(\text{Int}\) with \(\alpha\)). Dually, if \(\alpha^{E}\) is only on the left-side, e.g. \(\langle ?, \alpha^{E} \rangle\), the evidence is an unsealing (here, of \(?\) from \(\alpha\)). Otherwise, the evidence denotes neither, e.g. \(\langle \alpha^{\text{Int}}, \alpha^{\text{Int}} \rangle\).

**Consistent transitivity.** With this syntactic enrichment, consistent transitivity can be strengthened to account for sealing and unsealing, ensuring parametricity. Consistent transitivity is defined inductively; selected rules are presented in Figure 6.9.

Rule (unsl) specifies that when a sealing and an unsealing of the same type name meet in the middle positions of a consistent transitivity step, the type name can be eliminated in order to calculate the resulting evidence. For instance, \(\langle \text{Int}, \alpha^{\text{Int}} \rangle \circ \langle ?, \alpha^{E} \rangle = \langle \text{Int}, \alpha^{\text{Int}} \rangle \circ \langle ?, ?, \beta^{\text{Int}} \rangle = \langle \text{Int}, \text{Int} \rangle\).

As shown in § 6.7.1 it is important for consistent transitivity to not lose precision when combining an evidence with an unknown evidence. To this end, rule (identL) in Fig. 6.9 preserves the left evidence. Going back to the example of § 6.7.1, we now have \(\langle \text{Int}, \alpha^{\text{Int}} \rangle \circ \langle ?, ?, \beta^{\text{Int}} \rangle = \langle \text{Int}, \alpha^{\text{Int}} \rangle\), instead of \(\langle \text{Int}, \beta^{\text{Int}} \rangle\). Because \(\langle \text{Int}, \alpha^{\text{Int}} \rangle \circ \langle \text{Int}, \text{Int} \rangle\) is undefined (\(\alpha \neq \text{Int}\)), reduction steps to error as desired.

Rule (sealL) shows that when an evidence is combined with a sealing, the resulting evidence is also a sealing. This sealing can be more precise, e.g. \(\langle \text{Int}, \text{Int} \rangle \circ \langle ?, \alpha^{E} \rangle = \langle \text{Int}, \alpha^{\text{Int}} \rangle\).
Figure [6.9] only shows one structurally-recursive rule, corresponding to the function case (func); consistent transitivity is computed recursively with the domain and codomain evidence. To combine a function evidence with unknown evidence, the unknown evidence is first “expanded” to match the type constructor (func?L). There are similar rules for the other type constructors. Also, there are symmetric variants of the above rules—such as (identR) and (sealR)—in which every evidence and every evidence type is swapped. The complete definition is provided in §D.3.3.

Importantly, this refined definition of consistent transitivity preserves associativity and optimality, based on a natural notion of precision for evidence types (§D.3.1). It does however break monotonicity\[14] and hence the dynamic gradual guarantee—in §6.9 we give a semantic argument establishing that the dynamic gradual guarantee is fundamentally incompatible with parametricity anyway, independently of this refinement.

**Outer evidence.** The reduction rule of a type application (RappG) produces an outer evidence εout that justifies that G[α/X] is consistent with G[G'/X]. The precise definition of this evidence is delicate, addressing a subtle tension between the precision required for justifying unsealing when possible, and the imprecision required for parametricity.

εout ≜ ⟨E_s[α^E], E_s[α^E']⟩ where E_s = lift_{R}(unlift(π_2(ε))), α^E = lift_{R}(α), E' = lift_{R}(G')

In this definition, ε, α, G, Ξ, and Ξ' come from rule (RappG). Determining E_s is the key challenge. The second evidence type of ε refines ∀X.G by exploiting the fact that the underlying polymorphic value ΛX.t is consistent with it; this extra precision is crucial for unsealing. The roundtrip unlift/lift “resets” the sealing information of evidence type names to that contained in the store (e.g. lift_{α→1}(unlift(α^int → α^int)) = α^? → α^?)—this relaxation is crucial for parametricity (to prove the compositionality lemma—§6.8).

To illustrate how to compute εout, consider the identity function with evidence ε_{∀X.X→X} = (∀X.X→X, ∀X.X→X), instantiated to Int. As we are instantiating to Int, then α^E = lift_{α→Int}(α) = α^int and E' = lift.(Int) = Int. Then E_s = lift.(unlift(π_2(∀X.X→X, ∀X.X→X))) = lift.(unlift(∀X.X→X)) = ∀X.X→X. Finally εout = (∀X.X→X)[α^int], (∀X.X→X)[Int]) = (α^int → α^int, Int → Int).

Note that εout will never cause a runtime error when combined with the resulting evidence of the parametric term result, because both are necessarily related by precision.

**Illustration.** The following reduction trace illustrates all the important aspects of reduction:

(ε_{∀X.X→X}(ΛX.λx : X.x) :: ∀X.X→X) | (Int) (ε_{Int 1 :: Int})

(RappG) → ((α^int → α^int, Int → Int) (ε_{α→X}) :: α→?) | (Int → ?)

(ε_{Int 1 :: Int})

(Rasc) → ((ε_{Int} :: Int) (λx : α.x) :: α→?) (ε_{Int 1 :: Int})

(Rapp) → (Int, α^Int) (Int, α^Int) 1 :: α :: ?

(Rasc) → (Int, Int) 1 :: ?

\[14\] For instance, consider (Int, α^Int) ⊆ (Int, α^Int) and (α^int, Int) ⊆ (? , ?). By consistent transitivity, (Int, α^Int) o (α^int, Int) = (Int, Int) (rule unsl), and (Int, α^Int) o (? , ?) = (Int, α^Int) (rule idL), but (Int, Int) ⊆ (Int, α^Int).
Crucially, the initial evidence of the identity function is fully precise, even though it is ascribed an imprecise type. Consequently, in the first reduction step above, $\varepsilon_{\text{out}}$ is calculated as:

$$
\varepsilon_{\text{out}} \triangleq \langle E_s[\alpha^E], E_s[E'] \rangle = \langle (\forall X.X \rightarrow X)[\alpha^{\text{Int}}], (\forall X.X \rightarrow X)[\text{Int}] \rangle = \langle \alpha^{\text{Int}} \rightarrow \alpha^{\text{Int}}, \text{Int} \rightarrow \text{Int} \rangle
$$

The application step ($R_{\text{app}}$) then gives rise to sealing and unsealing evidences after deconstructing $\varepsilon_{\text{out}}$: the inner evidence $\langle \text{Int}, \alpha^{\text{Int}} \rangle$ seals the number 1 at type $\alpha$, while the outer evidence $\langle \alpha^{\text{Int}}, \text{Int} \rangle$ allows the subsequent unsealing in the ascription step ($R_{\text{asc}}$). As a result, the ascribed identity function yields usable values, because the outer evidence subsequently takes care of unsealing. This addresses the excess of failure reported with $\lambda B$ and System $F_C$ in §6.2.2. Note that if the function were not intrinsically precise on its return type, e.g. $\Lambda X.\lambda x : X.(x :: ?)$, then initial evidence would likewise be imprecise, and deconstructing $\varepsilon_{\text{out}}$ would not justify unsealing the result anymore.

### 6.7.3 Basic Properties of GSF Evaluation

The runtime semantics of a GSF term are given by first translating the term to GSF $\varepsilon$ (noted $\vdash t \rightsquigarrow t : G$) and then reducing the GSF $\varepsilon$ term. We write $t \downarrow \Xi \triangleright v$ (resp. $t \downarrow \text{error}$) if $\vdash t \rightsquigarrow t_\varepsilon : G$ and $\cdot \triangleright t_\varepsilon \rightsquigarrow^* \Xi \triangleright v$ (resp. $\cdot \triangleright t_\varepsilon \rightsquigarrow^* \Xi \triangleright \text{error}$) for some resulting store $\Xi$. We write $\Xi \triangleright v : G$ for $\Xi ; \cdot ; \vdash v : G$. We write $t \uparrow$ if the translation of $t$ diverges, and $t \downarrow v$ when the store is irrelevant.

The properties of GSF follow from the same properties of GSF $\varepsilon$, expressed using the small-step reduction relation, due to the fact that the translation $\rightsquigarrow$ preserves typing (§D.3.5). In particular, GSF terms do not get stuck, although they might produce error or diverge:

**Proposition 51 (Type Safety).** If $\vdash t : G$ then either $t \downarrow \Xi \triangleright v$ with $\Xi \triangleright v : G$, $t \downarrow \text{error}$, or $t \uparrow$.

Proposition 48 established that GSF typing coincides with SF typing on static terms. A similar result holds considering the dynamic semantics. In particular, static GSF terms never produce error:

**Proposition 52 (Static terms do not fail).** Let $t$ be a static term, and $G$ a static type ($G = T$). If $\vdash t : T$ then $\neg(t \downarrow \text{error})$.

This result follows from the fact that all evidences in a static program are static, hence never gain precision; the initial type checking ensures that combination through transitivity never fails. As we will see in §6.10 a static term is also guaranteed to terminate.
\[ \nu_\rho [B] = \{ (W,v,v) \in \text{Atom}^\rho [B] \} \]

\[ \nu_\rho [G_1 \rightarrow G_2] = \{ (W,v_1,v_2) \in \text{Atom}^\rho [G_1 \rightarrow G_2] \mid \forall W' \supseteq W . v_1' , v_2' . (W',v_1',v_2') \in \nu_\rho [G_1] \Rightarrow (W',v_1',v_2') \in \nu_\rho [G_2] \} \]

\[ \nu_\rho [\forall X . G] = \{ (W,v_1,v_2) \in \text{Atom}^\rho [\forall X . G] \mid \forall W' \supseteq W . v_1 , t_1 , G_1 , G_2 , G_\alpha , \varepsilon_1 , \varepsilon_2 . \forall R \in \text{Rel}_{W_\gamma} [\forall X . G] . (W',\Xi_1 \vdash G_1 \land W',\Xi_2 \vdash G_2 \land W'.\Xi_1 \triangleright v_1 [G_1] \Rightarrow W'.\Xi_1 , \alpha := G_1 \triangleright \varepsilon_1 t_1 :: \rho (\Gamma) [G_1 / X] \land W'.\Xi_2 \triangleright v_2 [G_2] \Rightarrow W'.\Xi_2 , \alpha := G_2 \triangleright \varepsilon_2 t_2 :: \rho (\Gamma) [G_2 / X] \Rightarrow (\downarrow (W' \varnothing (\alpha, G_1 , G_2 , R) , t_1 , t_2 )) \in \nu_\rho [\forall X \rightarrow \alpha [\Gamma]] \} \]

\[ \nu_\rho [G_1 \times G_2] = \{ (W,v_1,v_2) \in \text{Atom}^\rho [G_1 \times G_2] \mid (W,\pi_1 (v_1) , \pi_1 (v_2)) \in \nu_\rho [G_1] \land (W,\pi_2 (v_1) , \pi_2 (v_2)) \in \nu_\rho [G_2] \} \]

\[ \nu_\rho [X] = \nu_\rho [\rho (X)] \]

\[ \nu_\rho [\alpha] = \{ (W, (\alpha_{E_1} , \alpha_{E_2}) u_1 :: \alpha , (\alpha_{E_2} , \alpha_{E_2}) u_2 :: \alpha ) \in \text{Atom}^\rho [\alpha] \mid (W , \alpha_{E_1} , \alpha_{E_2} u_1 :: W . \Xi_1 (\alpha) , \alpha_{E_2} , u_2 :: W . \Xi_2 (\alpha) ) \in W . \kappa (\alpha) \} \]

\[ \nu_\rho [\varepsilon] = \{ (W, \varepsilon_1 u_1 :: \varepsilon_2 u_2 :: \varepsilon) \in \text{Atom}^\rho [\varepsilon] \mid \text{const} (\tau (\varepsilon)) = G \land (W, \varepsilon_1 u_1 :: G , \varepsilon_2 u_2 :: G ) \in \nu_\rho [G] \} \]

\[ \mathcal{T}_\rho [G] = \{ (W,t_1,t_2) \in \text{Atom}^\rho [G] \mid \forall i . W . t_1 , W . \Xi_1 \triangleright i . \Xi_1 \triangleright v_1 \Rightarrow \exists W', \Xi_1 , W . \Xi_2 \triangleright t_2 \rightarrow^* W'. \Xi_2 \triangleright v_2 \land W'. \Xi_1 = \Xi_1 \land (W',v_1,v_2) \in \nu_\rho [G] \land \forall v_1 . W . \Xi_1 \triangleright v_1 \rightarrow^* \Xi_1 \triangleright \text{error} \Rightarrow \exists W . W . \Xi_2 \triangleright t_2 \rightarrow^* \Xi_2 \triangleright \text{error} \} \]

\[ \delta [\cdot] \quad \text{World} \]

\[ \delta [\Xi, \alpha := G] = \delta [\Xi] \cap \{ W \in \text{World} \mid W . \Xi_1 (\alpha) = G \land W . \Xi_2 (\alpha) = G \land W . \Xi_1 \vdash W . \Xi_2 \land W . \kappa (\alpha) = [\nu_\rho [G]] W \} \]

\[ \mathcal{D} [\cdot] = \{ (W, \emptyset) \mid W \in \text{World} \} \]

\[ \mathcal{D} [\Delta, X] = \{ (W, \rho [X \leftarrow \alpha]) \mid (W, \rho) \in \mathcal{D} [\Delta] \land \alpha \in \text{dom}(W . \kappa) \} \]

\[ G_\rho [\cdot] = \{ (W, \emptyset) \mid W \in \text{World} \} \]

\[ G_\rho [\Gamma, x : G] = \{ (W, \gamma [x : G \leftarrow (v_1 , v_2)]) \mid (W, \gamma) \in G_\rho [\Gamma] \land (W, v_1 , v_2) \in \nu_\rho [G] \} \]

\[ \Xi ; \Delta ; \Gamma \vdash t_1 \leq t_2 : G \triangleq \Xi ; \Delta ; \Gamma \vdash t_1 : G \land \Xi ; \Delta ; \Gamma \vdash t_2 : G \land \forall W \in \delta [\Xi] , \rho , \gamma . ((W , \rho ) \in \mathcal{D} [\Delta] \land (W , \gamma ) \in G_\rho [\Gamma] ) \Rightarrow (W , \rho (\gamma (t_1)) , \rho (\gamma (t_2))) \in \mathcal{T}_\rho [G] \]

\[ \Xi ; \Delta ; \Gamma \vdash t_1 = t_2 : G \triangleq \Xi ; \Delta ; \Gamma \vdash t_1 : G \land \Xi ; \Delta ; \Gamma \vdash t_2 : G \land \forall W \in \delta [\Xi] , \rho , \gamma . ((W , \rho ) \in \mathcal{D} [\Delta] \land (W , \gamma ) \in G_\rho [\Gamma] ) \Rightarrow (W , \rho (\gamma (t_1)) , \rho (\gamma (t_2))) \in \mathcal{T}_\rho [G] \]

\[ \Xi ; \Delta ; \Gamma \vdash t_1 = t_2 : G \Rightarrow \Xi ; \Delta ; \Gamma \vdash t_1 \leq t_2 : G \land \Xi ; \Delta ; \Gamma \vdash t_2 \leq t_1 : G \]

Figure 6.10: Gradual logical relation and auxiliary definitions

### 6.8 GSF: Parametricity

We establish parametricity for GSF by proving parametricity for GSF\( \varepsilon \) terms, closely following the relation for \( \lambda B \) [7]. In the following, we only go briefly over the definition of the relation (Figure 6.10), and focus on the few differences with the \( \lambda B \) relation, essentially dealing with evidences.

The relation is defined on tuples \((W, t_1, t_2)\) that denote two related terms \(t_1, t_2\) in a world \(W\). A world is composed of a step index \(j\), two stores \(\Xi_1\) and \(\Xi_2\) used to typecheck and evaluate the related terms, and a mapping \(\kappa\), which maps type names to relations \(R\), used to relate sealed values. The components of a world are accessed through a dot notation, e.g. \(W.j\) for the step index.
\[(\text{Atom}_n[G_1, G_2] = \{(W, t_1, t_2) \mid W. j < n \land W \in \text{WORLD} \land W. \Xi_1, \cdots \vdash t_1 : G_1 \land W. \Xi_2, \cdots \vdash t_2 : G_2\}) \]

\[(\text{Atom}_{\text{val}}_n[G_1, G_2] = \{(W, v_1, v_2) \in \text{Atom}_n[G_1, G_2]\}) \]

\[\text{Atom}_p[G] = \bigcup_{n \geq 0} \{(W, t_1, t_2) \in \text{Atom}_n[\rho(G), \rho(G)]\} \]

\[\text{Atom}_p^\sim[G] = \{(W, v_1, v_2) \in \text{Atom}_p[G] \mid \text{unlift}(\pi_2(\text{ev}(v_1))) = \text{unlift}(\pi_2(\text{ev}(v_2)))\} \]

\[\text{WORLD} = \bigcup_{n \geq 0} \text{WORLD}_n \]

\[\text{WORLD}_n = \{(j, \Xi_1, \Xi_2, \kappa) \in \text{Nat} \times \text{Store} \times \text{Store} \times (\text{Name} \rightarrow \text{Rel}_j) \mid j < n \land \Xi_1 \land \Xi_2 \land \forall \alpha \in \text{dom}(\kappa).\kappa(\alpha) = \text{Rel}_j(\Xi_1(\alpha), \Xi_2(\alpha))\} \]

\[\text{Rel}_n[G_1, G_2] = \{R \in \text{Atom}_{\text{val}}_n[G_1, G_2] \mid \forall (W, v_1, v_2) \in R. \forall W' \succeq W. (W', v_1, v_2) \in R\} \]

\[
[R]_n = \{(W, e_1, e_2) \in R \mid W. j \leq n\}
\]

\[
[\kappa]_n = \{\alpha \mapsto [R]_n \mid \kappa(\alpha) = R\}
\]

\[
\kappa' \succeq \kappa \triangleq \forall \alpha \in \text{dom}(\kappa).\kappa'(\alpha) = \kappa(\alpha)
\]

\[
W' \succeq W \triangleq W. j \land W'. \Xi_1 \succeq W. \Xi_1 \land W'. \Xi_2 \succeq W. \Xi_2 \land W'. \kappa \succeq [W. \kappa]_{W. j} \land W', \text{W} \in \text{WORLD}
\]

\[
\downarrow^i W = (j, W. \Xi_1, W. \Xi_2, [W. \kappa]_j) \quad \text{where} \quad j = W. j - i
\]

Figure 6.11: Gradual logical relation and auxiliary definitions

The interpretations of values, terms, stores, name environments, and type environments are mutually defined, using the auxiliary definitions of Figure 6.11. As usual, the value and term interpretations are indexed by a type and a type substitution \(\rho\). We use \(\text{Atom}_n[G_1, G_2]\) to denote a set of pair of terms of type \(G_1\) and \(G_2\), and worlds with a step index less than \(n\). We write \(\text{Atom}_{\text{val}}_n[G_1, G_2]\) to restrict that set to values, and \(\text{Atom}_p[G]\) to denote a set of terms of the same type after substitution. The \(\text{Atom}_p^\sim[G]\) variant is similar to \(\text{Atom}_{\text{val}}^n_1[G_1, G_2]\) but restricts the set to values that have, after substitution, equally precise evidences (the equality is after unlifting because two sealed values may be related under different instantiations). \(\text{Rel}_n[G_1, G_2]\) defines the set of relations of values of type \(G_1\) and \(G_2\). We use \([R]_n\) and \([\kappa]_n\) to restrict the step index of the worlds to less than \(n\). Finally, \(\kappa' \succeq \kappa\) specifies that \(\kappa'\) is a future relation mapping of \(\kappa\) (and extension), and similarly \(W' \succeq W\) expresses that \(W'\) is a future world of \(W\). The \(\downarrow^i\) lowers the step index of a world by \(i\) (1 if unspecified).

The logical interpretation of a given type enforces a “termination-sensitive” view of parametricity: if the first term yields a value, the second must produce a related value at that type; if the first term fails, so must the second. Note that \(\text{Atom}_p^\sim[G]\) requires the second component of the evidence of each value to have the same precision in order to enforce such sensitivity. Indeed, if one is allowed to be more precise than the other, then when later combined in the same context, the more precise value may induce failure while the other does not.

The logical interpretation of values is also very similar to that of \(\lambda B\). Two base values are related if they are equal. Two functions are related if their application to related values yields related results. Two type abstractions are related if given any two types and any relation between them, the instantiated terms (without their unsealing evidence) are also related in a world extended (\(\Xi\)) with \(\alpha\), the two instantiation types \(G_1\) and \(G_2\) and the chosen relation \(R\).
between sealed values. Note that the step index of this extended world is decreased by one, because we take a reduction step. Two pairs are related if their components are pointwise related. Two sealed values are related at a type name $\alpha$ if, after unsealing, the resulting values are in the relation corresponding to $\alpha$ in the current world, $W.\kappa(\alpha)$.

Finally, two values are related at type $\_\_\_$ if they are related at the least-precise type with the same top-level constructor as the second component of the evidence, $\text{const}(\pi_2(\varepsilon))^{15}$. The intuition is that to be able to relate these unknown values we must take a step towards relating their actual content; evidence necessarily captures at least the top-level constructor (e.g. if a value is a function, the second evidence type is no less precise than $\_\_\_ \to \_\_\_$, i.e. $\text{const}(G_1 \to G_2)$).

The logical relation is well-founded for two reasons: (i) in the $\_\_\_$ case, $\text{const}(\pi_2(\varepsilon))$ cannot itself be $\_\_\_$, as just explained; (ii) in each other recursive cases, the step index is lowered: for functions and pairs, the relation is between reducible expressions (applications, projections) that either take a step or fail; for type abstractions, the relation is with respect to a world whose indexed is lowered.

The interpretations of stores, type name environments and type environments are straightforward (Figure 6.10). The logical relation allows us to define logical approximation, whose symmetric extension is logical equivalence. Any well-typed GSF$\varepsilon$ term is related to itself at its type:

**Theorem 53** (Fundamental Property). If $\Xi; \Delta; \Gamma \vdash t : G$ then $\Xi; \Delta; \Gamma \vdash t \leq t : G$.

As standard, the proof of the fundamental property uses compatibility lemmas for each term constructor and the compositionality lemma. Almost every compatibility lemma relies on the fact that the ascription of two related values yield related terms.

**Lemma 54** (Ascriptions Preserve Relations). If $(W, v_1, v_2) \in V(\rho)[G]$, $\varepsilon \vdash \Xi; \Delta \vdash G \sim G'$, $W \in S[\Xi]$, and $(W, \rho) \in D[\Delta]$, then $(W, \rho_1(\varepsilon)v_1 :: \rho(G'), \rho_2(\varepsilon)v_2 :: \rho(G')) \in T[\rho[G']]$.  

Note that type substitution on evidences takes as parameter the corresponding store: $\rho_i(\varepsilon)$ is syntactic sugar for $\rho(W.\Xi_i, \varepsilon)$, lifting each substituted type name in the process, e.g. if $\rho(X) = \alpha$, $W.\Xi_1(\alpha) = \text{Int}$, and $W.\Xi_2(\alpha) = \text{Bool}$, then $\rho_1((X, X)) = (\alpha^\text{Int}, \alpha^\text{Int})$, and $\rho_2((X, X)) = (\alpha^\text{Bool}, \alpha^\text{Bool})$.

### 6.9 Parametricity vs. Dynamic Gradual Guarantee

We now turn to the dynamic gradual guarantee \[113\] as described in § 2.1. We show that parametricity as defined in § 6.8 is however incompatible with this guarantee. First, we can prove the following lemma:

**Lemma 55.** For any $\vdash v : \_\_\_$ and $\vdash G$, we have $(\Lambda X. \lambda x : \_\_\_ . x :: X) [G] v \Downarrow \text{error}$.

\[15\] $\text{const}$ extracts the top-level constructor of an evidence type, e.g. $\text{const}(E_1 \to E_2) = \_\_\_ \to \_\_\_$ and $\text{const}(\forall X.E) = \forall X.\_\_\_$. 

155
Proof. Let \( v = (AX.\lambda x : ?.x :: X), \vdash v \rightsquigarrow v_\gamma : \forall X.? \to X, \) and \( v' \) s.t. \( \vdash v \rightsquigarrow v_\gamma : ? \).

By the fundamental property (Th. 53), \( \vdash v_\gamma \preceq v_\gamma : ? \); we can instantiate this definition with some \( W_0 \in \Delta \). By reduction, \( v_\gamma \) is a value, \( (W_0, v_\gamma, v_\gamma) \in \mathcal{I}_0[\forall X.? \to X] \). Because \( v_\gamma \) is a value, \( (W_0, v_\gamma, v_\gamma) \in \mathcal{I}_0[\forall X.? \to X] \). By reduction, \( \vdash v_\gamma \cdot G_1 \mapsto \varepsilon_1 v_\gamma :: ? \to G_1 \) for some \( \varepsilon_1, \varepsilon_1 $ and \( \varepsilon_1 $, where \( \varepsilon_1 = \{ \alpha = G_1 \} \) and \( v_1 = \varepsilon_1 (\lambda x : ?.(\varepsilon_1 x :: \alpha)) :: ? \to \alpha \). We can instantiate the definition of \( v_0[\forall X.? \to X] \) with \( W_0, G_1 = G \) and \( G_2 \) structurally different (and different from \( G \)), some \( R \in \text{REL}_{W_0,j}[G_1, G_2], v_1, v_2, \varepsilon_1 $ and \( \varepsilon_2 $, then we have that \( (W_1, v_1, v_2) \in \mathcal{I}_{X \to \alpha}[? \to X], \) where \( W_1 = (\downarrow (W_0 \times (\alpha, G_1, G_2, R)) \). As \( v_1 \) and \( v_2 \) are values, \( (W_1, v_1, v_2) \in \mathcal{I}_{X \to \alpha}[? \to X] \). Also, by associativity of consistent transitivity, the reduction of \( \varepsilon_1 \cdot G_1 \mapsto \varepsilon_2 v_1 :: ? \to G_1 \) \( v_\gamma \) is equivalent to that of \( \varepsilon_1 \cdot \varepsilon_2 v_1 :: ? \to G_1 \).

By the fundamental property (Th. 53), we know that \( \vdash v_\gamma \preceq v_\gamma : ? \); we can instantiate this definition with some \( W_2 \in W_1 \), and we have that \( (W_2, v_\gamma, v_\gamma) \in \mathcal{I}_0[?]. \) Since \( v_\gamma \) is a value, \( (W_2, v_\gamma, v_\gamma) \in \mathcal{I}_0[?]. \) By the ascription lemma (54), \( (W_2, \text{dom}(\varepsilon_1)v_\gamma :: ?, \text{dom}(\varepsilon_2)v_\gamma :: ?) \in \mathcal{I}_0[?]. \) If \( \text{dom}(\varepsilon_1) \cdot \text{dom}(\varepsilon_2)v_\gamma :: ? \rightarrow \text{error} \) then the result follows immediately. Otherwise, \( \varepsilon_1 \cdot \text{dom}(\varepsilon_1)v_\gamma :: ? \rightarrow^* \varepsilon_1 \cdot v_1 \), and \( (W_3, v_1', v_2', \varepsilon_1) \in \mathcal{I}_0[?], \) where \( W_1 = \downarrow W_2, \) and some \( v_1' \) and \( v_2' \). We can instantiate the definition of \( \mathcal{I}_{X \to \alpha}[? \to X] \) with \( W_3, v_1', v_2', \varepsilon_1 $, obtaining that \( (W_3, v_1', v_2', v_2', v_2', \varepsilon_1) \in \mathcal{I}_{X \to \alpha}[X]. \) We then proceed by contradiction. Suppose that \( \varepsilon_1 \cdot v_1 \rightarrow^* \varepsilon_1 v_2 \cdot v_1' \) (for a big-enough step index). If \( v_1'' = \varepsilon_1 u :: ? \), then by evaluation \( \varepsilon_1 u :: ? \), but by definition of \( \mathcal{I}_{X \to \alpha}[X] \), it must be the case that for some \( W_4 \geq W_3, (W_4, \varepsilon_1 u :: G_1, \varepsilon_2 u :: G_2) \in R, \) which is impossible because \( u \) cannot be ascribed to structurally different types \( G_1 \) and \( G_2 \). Therefore \( v_1 v_1'' \) cannot reduce to a value, and hence the term \( v_\gamma \cdot G_1 \) \( v_\gamma \) cannot reduce to a value either. Because \( v_\gamma \) is non-diverging, its application must produce \text{error}.

Consequently, the dynamic gradual guarantee is violated:

**Corollary 56.** There exist \( \vdash t_1 : G \) and \( t_2 \sqsubseteq t_1 \) such that \( t_1 \downarrow v \) and \( t_2 \downarrow \text{error}. \)

**Proof.** Let \( id_X \triangleq \Lambda X.\lambda x : X.x :: X, \) and \( id_\gamma \triangleq \Lambda X.\lambda x : ?.x :: X. \) By definition of precision, we have \( id_X \sqsubseteq id_\gamma \). Let \( \vdash v : G \) and \( \vdash v' : ?, \) such that \( v \sqsubseteq v'. \) Pose \( t_1 \triangleq id_X \cdot [G] v \) and \( t_2 \triangleq id_\gamma \cdot [G] v'. \) By definition of precision, we have \( t_1 \sqsubseteq t_2 \). By evaluation, \( t_1 \downarrow v. \) But by Lemma 55, \( t_2 \downarrow \text{error}. \)

Interestingly, Lemma 55 holds irrespective of the actual choices for representing evidence in GSF. The key element is the (standard) logical interpretation of \( \forall \cdot X \cdot G. \) Therefore the incompatibility described here does not apply only to GSF: in fact, we have been able to prove that Lemma 55 also holds in \( \lambda B \) [7], whose notion of parametricity is essentially the same as GSF.

By sticking to this standard notion of parametricity, one way to accommodate the dynamic gradual guarantee is to change the definition of precision, as done by Igarashi et al. [69] (denying that \( t_1 \sqsubseteq t_2 \) in the proof of Corollary 56). We believe this is questionable, because precision is a syntactic and intuitive notion describing “how static a type is”, and replacing parts of a type with ? is clearly making it “less static” (recall §6.2.1). Dually, if one sticks to the natural notion of precision, as adopted by both GSF and CSA, and justified by the AGT.
interpretation, reconciliation might come from considering other forms of parametricity, or perhaps less flexible gradual language designs [33].

Currently, it seems that the incompatibility of the dynamic gradual guarantee with parametricity has to be understood, in conjunction with a similar observation regarding non-interference (§4), as hinting towards further refined criteria for semantically-rich gradual typing. In particular, weaker forms of the dynamic gradual guarantee might still be useful, as explored next.

6.10 Gradual Free Theorems in GSF

The parametricity logical relation (§6.8) allows us to define notions of logical approximation (≤) and equivalence (≈) that are sound with respect to contextual approximation (≤ctx) and equivalence (≈ctx), and hence can be used to derive free theorems about well-typed GSF terms [127, 7]. The definitions of contextual approximation and equivalence, and the soundness of the logical relation, are fairly standard.

As shown by Ahmed et al. [7], in a gradual setting, the free theorems that hold for System F are weaker, as they have to be understood “modulo errors and divergence”. Ahmed et al. [7] prove two such free theorems in λB. However, these free theorems only concern fully static type signatures. This leaves unanswered the question of what imprecise free theorems are enabled by gradual parametricity. To the best of our knowledge, this topic has not been formally developed in the literature so far, despite several claims about expected theorems, exposed hereafter.

Igarashi et al. [69] report that the System F polymorphic identity function, if allowed to be cast to ∀X.? → X, would always trigger a runtime error when applied, suggesting that functions of type ∀X.? → X are always failing. Consequently, System F_G rejects such a cast by tweaking the precision relation (§6.2.1). But the corresponding free theorem is not proven. Also, Ahmed et al. [6] declare that parametricity dictates that any value of type ∀X.X → ? is either constant or always failing or diverging (p.7). This gradual free theorem is not proven either. In fact, in both an older system [5] and its newest version [7], as well as in System F_G, casting the identity function to ∀X.X → ? yields a function that returns without errors, though the returned value is still sealed, and as such unusable (§6.2.2). Considering that the underlying function is intrinsically parametric, why shall we expect it to fail or return unusable values? In fact, while the specific choice of runtime semantics may decree failure, such behavior is not imposed by the parametricity relation per se. Parametricity only imposes uniformity of behavior, including failure, of polymorphic terms, which leaves some freedom regarding when to fail.

Disproving gradual free claims. We initially settled to prove the above claims about both ∀X.? → X and ∀X.X → ? as free theorems, but failed: the parametricity relation does not impose the claimed behaviors. We eventually uncovered a novel property of GSF: it preserves the strong normalization property of System F terms even as they are ascribed to

---

16Interestingly, nor does the parametricity relation of Ahmed et al. [7], which essentially coincides with
\[ n^\Sigma_\rho[B \sqsubseteq G] = \{ v = \varepsilon b :: G \mid v \in \text{ImpSV}^\Sigma_\rho[B \sqsubseteq G] \} \]
\[ n^\Sigma_\rho[T_1 \rightarrow T_2 \sqsubseteq G] = \{ v = \varepsilon u :: \rho(G) \mid v \in \text{ImpSV}^\Sigma_\rho[T_1 \rightarrow T_2 \sqsubseteq G] \land \forall v' \in n^\Sigma_\rho[T_1 \subseteq \text{dom}^2(G)] \} \]
\[ (\varepsilon u :: \text{dom}^2(\rho(G)) \rightarrow \text{cod}^2(\rho(G))) v' \in C^\Sigma_\rho[T_2 \subseteq \text{cod}^2(G)] \}
\[ n^\Sigma_\rho[\forall X.T \sqsubseteq G] = \{ v = \varepsilon u :: \rho(G) \mid v \in \text{ImpSV}^\Sigma_\rho[\forall X.T \sqsubseteq G] \land (\forall T'.\Sigma \vdash T'.\Sigma \triangleright (\varepsilon u :: \forall X.\text{schm}^\Sigma(\rho(G))))[T'] \mapsto \Sigma, \alpha := T' \triangleright \varepsilon' t' :: \text{schm}^\Sigma(\rho(G))[T'/X] \land t' \in C^\Sigma_{\rho[\alpha \rightarrow \alpha]}[T \sqsubseteq \text{schm}^\Sigma(\rho(G))] \}
\[ n^\Sigma_\rho[T_1 \times T_2 \sqsubseteq G] = \{ v = \varepsilon u :: \rho(G) \mid v \in \text{ImpSV}^\Sigma_\rho[T_1 \times T_2 \sqsubseteq G] \land (\rho(\varepsilon)\pi_1(u) :: \text{proj}_1^\Sigma(\rho(G))) \in n^\Sigma_\rho[T_1 \sqsubseteq \text{proj}_1^\Sigma(\rho(G))] \}
\] \[ n^\Sigma_\rho[X \sqsubseteq G] = n^\Sigma_\rho[\rho(X) \sqsubseteq \rho(G)] \]
\[ n^\Sigma_\rho[\alpha \sqsubseteq G] = \{ v = \varepsilon u :: \rho(G) \mid v \in \text{ImpSV}^\Sigma_\rho[\alpha \sqsubseteq G] \land \Sigma(\alpha) = T \land \forall \Gamma.\Sigma \sqsubseteq G'.(\langle \pi_1(\varepsilon), \text{lift}_\Sigma(T)\rangle::u :: \rho(G')) \in C^\Sigma_\rho[T \sqsubseteq G'] \}
\[ C^\Sigma_\rho[T \sqsubseteq G] = \{ \Gamma \vdash \Sigma; :: t :: \rho(G) \land \Sigma \triangleright v \triangleright \varepsilon' \Sigma' \triangleright v \land v \in n^\Sigma_\rho[T \sqsubseteq G] \}
\[ \text{ImpSV}^\Sigma_\rho[T \sqsubseteq G] = \{ v = \varepsilon u :: \rho(G) \mid \text{static}(u) \land \pi_2(\varepsilon) = \text{lift}_\Sigma(\rho(T)) \land \Sigma; :: \vdash v :: \rho(G) \}
\]
\[ \Sigma; \Delta; \Gamma \vdash t :: T \sqsubseteq G \triangleq \Sigma; \Delta; \Gamma \vdash t : G \land \forall \rho \in \mathcal{B}[\Delta], \forall \gamma \in \mathcal{C}^{\Sigma}_\rho[\Gamma], \rho(\gamma(t)) \in C^\Sigma_\rho[T \sqsubseteq G] \]

Figure 6.12: Imprecise termination logical predicate.

less precise types, as long as they are used with similarly-terminating terms, and instantiated at static types.

We establish this result using a logical predicate, named *imprecise termination* (Figure 6.12\textsuperscript{17}), whose statement \( \vdash t :: T \sqsubseteq G \) expresses that \( t \) is a static term of type \( T \) that has been ascribed a less precise type \( G \). As usual, the predicates for values and terms carry a type environment and type name store; we do not need step indexing because the logical relation is defined inductively on the structure of \( T \) (not \( G \)). At the function type, the predicate specifies that when applied to an imprecisely-terminating argument, the application terminates and yields an imprecisely-terminating result. For type application, only static type instantiations are considered. The predicate \( \text{ImpSV}^\Sigma_\rho[T \sqsubseteq G] \) characterizes imprecisely-ascribed static values. The rest of the definitions are essentially administrative ascriptions to align types as required by GSF\( \varepsilon \).

Static terms satisfy the imprecise termination predicate, and are hence hereditarily terminating:

**Lemma 57.** Let \( t \) be a static term. If \( \vdash t :: T \sqsubseteq G \), then \( \vdash (t :: G) \triangleright t' : G \) and \( \vdash t' :: T \sqsubseteq G \).

This property is related to the dynamic gradual guarantee—which we have seen is incompatible with parametricity (§ 6.9)—but it is much weaker. Nevertheless, it is powerful

\textsuperscript{17} schm\( ^\varepsilon \) (consistently) extracts the schema of a gradual type, \textit{i.e.} schm\( ^\varepsilon (\forall X.G) = G \), schm\( ^\varepsilon (?) = \? \), undefined o/w.

158
enough to disprove the claims from the literature about $\forall X.? \rightarrow X$ and $\forall X.X \rightarrow ?$: both types admit the ascribed System F identity function, among many others\textsuperscript{18} as a non-constant, non-failing, parametricity-preserving inhabitant. We believe this result constitutes a valuable compositionality guarantee when embedding fully-static (System F) terms in a gradual world. Another corollary is that closed static terms always terminate (by $\models t : T \subseteq T$), hence superseding Proposition \[183].

Cheap theorems. The intuition of $\forall X.? \rightarrow X$ denoting always-failing functions is not entirely misguided: this result does hold for a large subset of the terms of that type. This leads us to observe that we can derive “cheap theorems” with gradual parametricity: obtained not by looking only at the type, but by also considering the head constructors of a term. For instance:

Theorem 58. Let $v \triangleq \Lambda X.\lambda x : ?.t$ for some $t$.

If $\vdash v : \forall X.? \rightarrow X$, then for any $\vdash v' : G$, we either have $v [G] v' \downarrow \text{error}$ or $v [G] v' \uparrow$.

This result holds independently of the body $t$, therefore without having to analyze the whole term. Not as good as a free theorem, but cheap.

6.11 Related Work

We have already discussed at length related work on gradual parametricity, especially the most recent developments \[7, 69, 130\]. In addition to static semantics issues in $\lambda B$ and System $F_G$, all theses languages suffer from dynamic semantics that do not accurately track type instantiations (§6.2.2). Note that, conversely to $\lambda B$, GSF does not impose any syntactic value restriction on polymorphic terms; such a restriction might be necessary when exploring the extension of GSF with implicit polymorphism. Finally, instead of leaving the dynamic gradual guarantee as a conjecture, we show that it is incompatible with parametricity, at least given the standard definitions of both notions. Note that some language features are also known to break the dynamic gradual guarantee, such as structural type tests and object identity \[113\], as well as method overloading and extension methods \[83\].

The relation between parametric polymorphism in general and dynamic typing much predates the work on gradual typing. Abadi et al. \[2\] first note that without further precaution, type abstraction might be violated. Subsequent work explored different approaches to protect parametricity, especially runtime-type generation (RTG) \[77, 3, 103\]. Pierce and Sumii \[90\] and Guha et al. \[53\] use dynamic sealing, originally proposed by Morris \[82\], in order to dynamically enforce type abstraction. Matthews and Ahmed \[78\] also use RTG in order to protect polymorphic functions in an integration of Scheme and ML. This line of work eventually led to the polymorphic blame calculus \[6\] and its most recent version with the proof of parametricity by Ahmed et al. \[7\]. We adapt their proof techniques to the evidence-based semantics of GSF.

Hou et al. \[62\] prove the correctness of compiling polymorphism to dynamic typing with

\textsuperscript{18}e.g. $\Lambda X.\lambda x : X.\lambda f : X \rightarrow f x$ of type $\forall X.X \rightarrow (X \rightarrow X) \rightarrow X$ can also be ascribed to $\forall X.X \rightarrow ?$. 159
embeddings and partial projections; the compilation setting however differs significantly from gradual typing. New and Ahmed [88] use embedding-projection pairs to formulate a semantic account of the dynamic gradual guarantee, coined graduality, in a language with explicit casts. It would be interesting to extend their simply-typed setting to parametric polymorphism, and study the interplay of parametricity and graduality when casts, and possibly seals, are explicit as in the work of Neis et al. [87] on parametricity in a non-parametric language.

Devriese et al. [33] disprove a conjecture by Pierce and Sumii [90] according to which the compilation of System F to an untyped language with dynamic sealing is fully abstract, i.e. preserves contextual equivalences. They show that, for similar reasons, the embedding of System F in current polymorphic blame calculi is not fully abstract; their observation also applies to GSF. Full abstraction might be too strong a criteria for gradual typing: already in the simply-typed setting, embedding typed terms in gradual contexts is not fully abstract, because gradual types admit non-terminating terms. Imprecise termination (§ 6.10) is a weaker, yet useful result that sheds light on gradual free theorems about imprecise type signatures. It should be possible to generalize this result to account for the harmless content of imprecise ascriptions; we leave this perspective for future work.

This work is generally related to gradualization of advanced typing disciplines, in particular to gradual information-flow security typing [35, 39, 40, 45, 120]. In these systems, one aims at preserving noninterference, i.e. that private values do not affect public outputs. Both parametricity and noninterference are 2-safety properties, expressed as a relation of two program executions. While Garcia and Tanter [45] show that one can derive a pure security language with AGT that satisfies both noninterference and the dynamic gradual guarantee, Toro et al. [120] find that in presence of mutable references, one can have either the dynamic gradual guarantee, or noninterference, but not both. Also similarly to this work, AGT for security typing needs a more precise abstraction for evidence types (based on security label intervals) in order to enforce noninterference. Together, these results suggest that new criteria are needed to characterize the spectrum of type-based reasoning that gradual typing supports when applied to semantically-rich disciplines.

6.12 Conclusion

We uncover design flaws in prior work on gradual parametric languages that enforce relational parametricity. We exploit the Abstracting Gradual Typing (AGT) methodology to design a new gradual language with explicit parametric polymorphism, GSF. We find that AGT greatly streamlines the static semantics of GSF, but does not yield a language that respects parametricity by default; non-trivial exploration was necessary to uncover how to strengthen the structure and treatment of runtime evidence in order to recover parametricity. We show that parametricity is, like noninterference, incompatible with the dynamic gradual guarantee laid forth by Siek et al. [113]. We nevertheless establish a novel, weaker property of GSF regarding the embedding of System F terms at less precise types, which allows us to disprove some claims from the literature about gradual free theorems.

Future work also includes extending GSF and its associated reasoning with existential
types, both in terms of their encoding, and as primitives in the language. We shall also study the integration of implicit polymorphism on top of GSF, most likely following the approach of Xie et al. [130]. Finally, it would be interesting to understand whether the evidence-based runtime semantics presented here can be used to derive a cast calculus akin to $\lambda B$, and then address efficiency considerations.
Chapter 7

Conclusions and Future Work

In this thesis we have shown that the Abstracting Gradual Typing (AGT) methodology can be applied to complex type disciplines and language constructs. In particular, we applied AGT to a simply-typed lambda calculus with references ($\lambda^{\text{REF}}$), to a security-typed language with references ($\text{SSL}_{\text{Ref}}$), and to the parametric language System F. The resulting gradual languages are named $\lambda^{\text{REF}}$, $\text{GSL}_{\text{Ref}}$, and GSF respectively.

From our experience applying AGT to $\lambda^{\text{REF}}$ we learned that a straightforward application yields operational semantics that are equivalent to that of Herman et al. [60], but not space efficient. In particular, nested ascriptions may be accumulated during function applications. We have presented a simple fix to the dynamic semantics in order to regain space efficiency.

The straightforward application of AGT only guarantees type safety and the refined criteria for gradual languages [113]. Preserving other properties of the static language requires extra considerations. To preserve hyper-properties such as noninterference [50] and parametricity [100] we had to customize the dynamic semantics in two ways. First, we had to use a more refined abstraction in the representation of evidence. For $\text{GSL}_{\text{Ref}}$ we use label intervals, and for GSF type names enriched with store information. Second, we had to customize the reduction rules. For $\text{GSL}_{\text{Ref}}$ we add an extra check in the assignment reduction rule, and for GSF we customized the consistent transitivity operator to gain more precision. Sadly, these customizations to the reduction rules break the dynamic gradual guarantee: losing precision on a term may introduce new runtime errors. For GSF, we actually proved that there is an incompatibility between the dynamic gradual guarantee and the standard notion of parametricity. Nevertheless, we established a novel, weaker property regarding the embedding of terms of the static language at less precise types.

Regarding composability of AGT, we have shown that applying AGT directly to introduce both the unknown type and gradual unions breaks optimality of the abstraction. To address this, we developed a stratified approach that allows us to recover optimality. In particular, we first apply AGT to only introduce the unknown type, followed by a second application of AGT on top of the gradual language to introduce gradual unions. We prove that the composed abstraction is optimal.
There are many tracks of future work specifically related to some of the chapters of this thesis.

**Type-driven Gradual Security Typing**  One track of future work is to try to reconcile the dynamic gradual guarantee and noninterference. We could try other techniques to deal with implicit flows, such as multiple execution [34], or faceted values [10], but in this setting we first have to give meaning to the dynamic gradual guarantee. We could also try to do some effect analysis to collect information of untaken branches [104], but then programmers would have to deal with extra complexity of working with a static effect system in a higher-order language. This effect analysis would not be transparent for programmers as they would have to insert extra effect annotations. It may be simpler (and transparent for programmers) if we consider a restricted language without first-class functions, but still we do not know if doing an effect analysis in this setting would allow us to recover the dynamic gradual guarantee. Also as future work we can try to integrate gradual labels with the unknown type, support for integrity, and implement “pay-as-you-go” semantics.

**A Gradual Interpretation of Union Types**  We think that the stratified approach to AGT can be generalized to multiple applications of AGT (not just two). We conjecture that this technique might prove helpful in integrating other gradualization efforts, such as combining $\lambda_{\text{REF}}$ and GSL$_{\text{Ref}}$ to support for both the unknown type and the unknown security label.

Unlike gradual unions (without unknown), we know that $(\text{Int} \rightarrow ?) \oplus (? \rightarrow \text{Int})$ and $(\text{Int} \oplus ?) \rightarrow (? \oplus \text{Int})$ represent different functions. In particular the type $\text{Bool} \rightarrow \text{Bool}$ is not in the concretization of the former but it is contained in the concretization of the later. Consider the following program:

```plaintext
let f : (\text{Int} \rightarrow ?) \oplus (? \rightarrow \text{Int}) = \lambda x:?.x in f true
```

This program runs without errors. But by looking at the signature of the function, that programs should fail because we are ascribing the function to a function that either receives an integer and return “anything”, or it receives “anything” and returns an integer. The argument only matches with the latter so it should return an integer, not a boolean.

Technically, what is happening is that when combining evidence, we are considering that $(\text{Int} \oplus ?) \rightarrow (\text{Int} \oplus ?)$ and $(\text{Int} \rightarrow ?) \oplus (? \rightarrow \text{Int})$ are the same. In particular the codomain of both types are the same. We would like to study as future work if we can address this issue by either customizing the codomain operator, refining the interpretation of types or following the approaches of Castagna and Lanvin [22] and Keil and Thiemann [73].

**Gradual Parametricity, Revisited**  As GSF supports only explicit polymorphism, one immediate track of future work is to extend GSF with support for implicit polymorphism. A possible plan to do this is to first support for implicit polymorphism dynamically in two ways. First, we can instantiate type abstractions with the unknown type when used as functions, e.g. $((\Lambda X.\lambda x : X.x) :: ?) 1$ would now reduce to $((\Lambda X.\lambda x : X.x)[?] :: ?) 1$. Second, we could
wrap non-type abstraction terms into new type abstractions when used in a polymorphic way, e.g. \((\lambda x : \text{Int}.x :: ?) [\text{Int}]\) would now reduce now to \(((\Lambda X.\lambda x : \text{Int}.x) :: ?) [\text{Int}]\). Although this technique is used statically by Ahmed et al. [7], we agree with Xie et al. [130] that this technique should not be used statically as it breaks the conservative extension of System F. For instance, program \((\lambda f : \text{Int} \rightarrow \text{Bool}.f)1)((\Lambda X.\lambda x : X.x))\), ill-typed in System F, would be accepted as \(\text{Int} \rightarrow \text{Bool} \sim ? \rightarrow ?\). For the static semantics, following an approach similar to Xie et al. [130] seems promising.

Another track of future work is to extend GSF with more language features such as existentials, recursion, and references. Also we can generalize the imprecise embedding property, and capture what are the key conditions on terms, types, and embeddings, so embedding of System F terms preserves behavior.
Bibliography


[40] Luminous Fennell and Peter Thiemann. 2016. LJGS: Gradual Security Types for Object-Oriented Languages. In Proceedings of the 30th European Conference on Object-


[73] Matthias Keil and Peter Thiemann. 2015. Blame assignment for higher-order contracts with intersection and union, See [66], 375–386.


USA.


Appendices
Appendix A

First step: Gradualizing References

In this appendix we present additional definitions and proofs that were not included in § 3.

A.1 Gradualizing $\lambda^{\text{REF}}$, Elaborating $\lambda^{\text{REF}}_\varepsilon$

In this section we present some proofs used in the gradualization of $\lambda^{\text{REF}}$ and elaboration of $\lambda^{\text{REF}}_\varepsilon$.

**Proposition 44** (Precision, inductively). The inductive definition of type precision given in Figure 6.3 is equivalent to Definition 45.

**Proof.** We have to prove that $\gamma(G_1) \subseteq \gamma(G_2) \iff G_1 \sqsubseteq G_2$, where $G_1 \sqsubseteq G_2$ correspond to the inductive definition of type precision. We prove $\Rightarrow$ (the other direction is analogous). We proceed by induction on $\gamma(G_1) \subseteq \gamma(G_2)$.

Case ($\gamma(B) \subseteq \gamma(G_2)$). If $G_2 = B$ then we have to prove that $\gamma(B) \subseteq \gamma(B) \Rightarrow B \sqsubseteq B$, which is trivial. If $G_2 = ?$ then we have to prove that $\gamma(B) \subseteq \text{TYPE} \Rightarrow B \sqsubseteq ?$, which is also trivial.

Case ($\gamma(G_{11} \rightarrow G_{12}) \subseteq \gamma(G_2)$). If $\gamma(G_{11} \rightarrow G_{12}) = \{T_{11} \rightarrow T_{12} \mid T_{11} \in \gamma(G_{11}) \wedge T_{12} \in \gamma(G_{12})\}$, then $G_2$ is either $? \sqsubseteq ? \rightarrow ?$ or $\gamma(G_2) = \text{TYPE}$, but $G_{11} \sqsubseteq G_2$ and the result holds, or $G_2$ is $G_{21} \rightarrow G_{22}$ such that $\gamma(G_{21} \rightarrow G_{22}) = \{T_{21} \rightarrow T_{22} \mid T_{21} \in \gamma(G_{21}) \wedge T_{22} \in \gamma(G_{22})\}$ and $\{T_{11} \rightarrow T_{12} \mid T_{11} \in \gamma(G_{11}) \wedge T_{12} \in \gamma(G_{12})\} \subseteq \{T_{21} \rightarrow T_{22} \mid T_{21} \in \gamma(G_{21}) \wedge T_{22} \in \gamma(G_{22})\}$. For this to be true then $\gamma(G_{11}) = \{T_{11} \in \gamma(G_{11})\} \subseteq \gamma(G_{21}) = \{T_{21} \in \gamma(G_{21})\}$, and $\gamma(G_{12}) = \{T_{12} \in \gamma(G_{12})\} \subseteq \gamma(G_{22}) = \{T_{22} \in \gamma(G_{22})\}$. By induction hypotheses on $\gamma(G_{11}) \sqsubseteq \gamma(G_{21})$ and $\gamma(G_{12}) \sqsubseteq \gamma(G_{22})$ we know that $G_{11} \sqsubseteq G_{21}$ and $G_{12} \sqsubseteq G_{22}$. Therefore $G_{11} \rightarrow G_{12} \sqsubseteq G_{21} \rightarrow G_{22}$ and the result holds.

Case ($\gamma(\text{Ref } G_{11}) \subseteq \gamma(G_2)$). We proceed similar to case function.

**Proposition 3** (Galois connection). $(\gamma, \alpha)$ is a Galois connection, i.e.:

a) (Soundness) for any non-empty set of static types $S = \{T\}$, we have $S \subseteq \gamma(\alpha(S))$
b) (Optimality) for any gradual type \( G \), we have \( \alpha(\gamma(G)) \subseteq G \).

**Proof.** We first proceed to prove a) by induction on the structure of the non-empty set \( S \).

**Case (\{ \{ \} \}).** Then \( \alpha(\{ \{ \} \}) = B \). But \( \gamma(B) = \{ \} \) and the result holds.

**Case (\{ \{ T_1 \rightarrow T_2 \} \}).** Then \( \alpha(\{ \{ T_1 \rightarrow T_2 \} \}) = \alpha(\{ \{ T_1 \} \}) \rightarrow \alpha(\{ \{ T_2 \} \}) \). But by definition of \( \gamma \), \( \gamma(\alpha(\{ \{ T_1 \} \}) \rightarrow \alpha(\{ \{ T_2 \} \})) = \{ T_1 \rightarrow T_2 \mid T_1 \in \gamma(\alpha(\{ \{ T_1 \} \}), T_2 \in \gamma(\alpha(\{ \{ T_2 \} \})) \}. \) By induction hypotheses, \( \{ \{ T_1 \} \} \subseteq \gamma(\alpha(\{ \{ T_1 \} \})) \) and \( \{ \{ T_2 \} \} \subseteq \gamma(\alpha(\{ \{ T_2 \} \})) \), therefore \( \{ \{ T_1 \rightarrow T_2 \} \} \subseteq \{ T_1 \rightarrow T_2 \mid T_1 \in \{ \{ T_1 \} \}, T_2 \in \{ \{ T_2 \} \} \} \subseteq \{ \{ T_1 \} \rightarrow \{ T_2 \} \mid T_1 \in \gamma(\alpha(\{ \{ T_1 \} \}), T_2 \in \gamma(\alpha(\{ \{ T_2 \} \})) \} \) and the result holds.

**Case (\{ \{ \text{Ref } T \} \}).** Then \( \alpha(\{ \{ \text{Ref } T \} \}) = \text{Ref } \alpha(\{ \{ T \} \}) \). But by definition of \( \gamma \), \( \gamma(\text{Ref } \alpha(\{ \{ T \} \})) = \{ \text{Ref } T \mid T \in \gamma(\alpha(\{ \{ T \} \})) \}. \) By induction hypothesis, \( \{ \{ \text{Ref } T \} \} \subseteq \gamma(\alpha(\{ \{ T \} \})) \), therefore \( \{ \{ \text{Ref } T \} \} \subseteq \{ \text{Ref } T \mid T \in \gamma(\alpha(\{ \{ T \} \})) \} \) and the result holds.

**Case (\{ \{ T \} \} heterogeneous).** Then \( \alpha(\{ \{ T \} \}) \equiv ? \) and therefore \( \gamma(\alpha(\{ \{ T \} \})) = \text{TYPE} \), but \( \{ \{ T \} \} \subseteq \text{TYPE} \) and the result holds.

Now let us proceed to prove b) by induction on gradual type \( G \).

**Case (\{ \}).** Trivial because \( \gamma(B) = \{ \} \), and \( \alpha(\{ \} ) = B \).

**Case (\{ G_1 \rightarrow G_2 \}).** We have to prove that \( \alpha(\gamma(G_1 \rightarrow G_2)) \subseteq G_1 \rightarrow G_2 \), which is equivalent to prove that \( \gamma(\alpha(\{ G \})) \subseteq \{ G \} \), where \( \{ G \} = \gamma(\alpha(G)) = \{ T \mid T \in \gamma(\alpha(G)) \} \). Then \( \{ G \} \) has the form \( \{ \{ T \} \rightarrow \{ T \} \} \), such that \( \forall i, T_i \in \gamma(G) \) and \( \{ T_i \} \subseteq \gamma(G) \). Also note that \( \{ \{ T \} \} \subseteq \gamma(G) \) and \( \{ \{ T \} \} \subseteq \gamma(G) \). But by definition of \( \alpha \), \( \alpha(\{ \{ T \} \} \rightarrow \{ \{ T \} \}) \) is equivalent to prove that \( \gamma(\alpha(\{ \{ T \} \})) \subseteq \{ \{ T \} \} \), where \( \{ \{ T \} \} = \gamma(\alpha(\{ \{ T \} \})) = \{ \text{Ref } T \mid T \in \gamma(\alpha(\{ \{ T \} \})) \} \). Then \( \{ \{ T \} \} \) has the form \( \{ \{ \text{Ref } T \} \} \), such that \( \forall i, T_i \in \gamma(\alpha(\{ \{ T \} \})) \). Also note that \( \{ \{ \text{Ref } T \} \} \subseteq \gamma(\alpha(\{ \{ T \} \})) \). But by definition of \( \alpha \), \( \alpha(\{ \{ \text{Ref } T \} \}) = \text{Ref } \alpha(\{ \{ T \} \}) \) and therefore \( \gamma(\text{Ref } \alpha(\{ \{ T \} \})) = \{ \text{Ref } T \mid T \in \gamma(\alpha(\{ \{ T \} \})) \}. \) But by induction hypothesis most of \( \gamma(\alpha(\{ \{ T \} \})) \subseteq \gamma(G) \) and the result holds.

**Case (?).** Then we have to prove that \( \gamma(\alpha(?)) \subseteq \gamma(?) = \text{TYPE} \), but this is always true and the result holds immediately.

\[ \square \]

**Proposition 4.** \( \text{domate}(G_1, G_2) = G_1 \cap G_2 \).

The meet operator is defined as \( G_1 \cap G_2 = \alpha(\gamma(G_1) \cap \gamma(G_2)) \), and inductively as:

\[
B \cap B = B \\
G_1 \cap G_2 = G_2 \cap G_1 \\
G \cap ? = ? \cap G = G \\
(G_{11} \rightarrow G_{12}) \cap (G_{21} \rightarrow G_{22}) = (G_{11} \cap G_{21}) \rightarrow (G_{12} \cap G_{22}) \\
\text{Ref } G_1 \cap \text{Ref } G_2 = \text{Ref } G_1 \cap G_2 \\
G_1 \cap G_2 \text{ is undefined otherwise}
\]

**Proof.** By induction on \( G_1 \) and \( G_2 \). \[ \square \]

**Proposition 5.** Let \( P_1(T_1, T_2) \triangleq T_1 = \text{dom}(T_2) \). Then \( \text{domate}(G_1, G_2) \iff G_1 \sim \text{dom}(G_2) \).
Proposition 6. Let \( P_2(T_1, T_2) \triangleq T_1 = \text{tref}(T_2) \). Then \( P_2(G_1, G_2) \iff G_1 \sim \text{tref}(G_2) \).

Proof. The \( \Rightarrow \) direction by induction on \( \text{tref}(G_1, G_2) \), and the \( \Leftarrow \) direction by induction on \( G_1 \sim \text{dom}(G_2) \).

Proposition 7. If \( G_1 \sim G_2 \), then \( \subseteq(G_1, G_2) = (G_1 \cap G_2) \).

Proof. Notice that in this setting \( \subseteq(G_1, G_2) = \alpha(\{\langle T \rangle \mid T \in \gamma(G_1) \cap \gamma(G_2)\}) = \alpha(\{\langle T \rangle \mid T \in \gamma(G_1) \cap \gamma(G_2)\}) = (\alpha(\gamma(G_1) \cap \gamma(G_2))) = (G_1 \cap G_2) \).

Lemma 8. \( \langle G_1 \rangle \subseteq \langle G_2 \rangle = (G_1 \cap G_2) \).

Proof. Similar to Prop 7.

Proposition 9 (Elaboration preserves typing). If \( \Gamma; \Sigma \vdash t : G \) and \( \Gamma; \Sigma \vdash t \wedge \sim \varepsilon t^G : G \), then \( t^G \in T[G] \).

Proof. Straightforward induction on \( \Gamma; \Sigma \vdash t : G \). We only present one case as the other are analogous.

Case \((\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit})\). We know by \((\text{Gasgn})\) that

\[
\begin{array}{c}
(\text{Gasgn}) \\
\Gamma; \Sigma \vdash t_1 : G_1 & \quad \Gamma; \Sigma \vdash t_2 : G_2 & \quad G_1 \sim \text{tref}(G_1) \\
\hline
\Gamma; \Sigma \vdash t_1 := t_2 : \text{Unit}
\end{array}
\]

Then by \((\text{TRasgn})\):

\[
\begin{array}{c}
(\text{TRasgn}) \\
\Gamma; \Sigma \vdash t_1 \wedge \sim \varepsilon t_{G_1} : G_1 & \quad \Gamma; \Sigma \vdash t_2 \wedge \sim \varepsilon t_{G_2} : G_2 \\
G_3 = \text{tref}(G_1) & \quad \varepsilon_1 = \subseteq(G_1, \text{Ref} G_3) & \quad \varepsilon_2 = \subseteq(G_2, G_3) \\
\hline
\Gamma; \Sigma \vdash t_1 := t_2 \wedge \sim \varepsilon t_{G_1} := \varepsilon_1 t^G_{G_1} \wedge \varepsilon_2 t^G_{G_2} : \text{Unit}
\end{array}
\]

By induction hypothesis on \( \Gamma; \Sigma \vdash t_1 : G_1 \), if \( \Gamma; \Sigma \vdash t_1 \wedge \sim \varepsilon t^G_{G_1} : G_1 \) then \( t^G_{G_1} \in T[G_1] \).

Similarly by induction hypothesis on \( \Gamma; \Sigma \vdash t_2 : G_2 \), if \( \Gamma; \Sigma \vdash t_2 \wedge \sim \varepsilon t^G_{G_2} : G_2 \) then \( t^G_{G_2} \in T[G_2] \).

Also by definition of the interior function, \( \varepsilon_1 \vdash \text{Ref} G_3 \) and \( \varepsilon_2 \vdash G_2 \sim G_3 \). Then by \((\text{IGasgn})\):

\[
\begin{array}{c}
(\text{IGasgn}) \\
t^G_{G_1} \in T[G_1] & \quad \varepsilon_1 \vdash G_1 \sim \text{Ref} G_3 \\
t^G_{G_2} \in T[G_2] & \quad \varepsilon_2 \vdash G_2 \sim G_3 \\
\hline
\varepsilon_1 t^G_{G_1} := \varepsilon_2 t^G_{G_2} \in T[\text{Unit}]
\end{array}
\]

and the result holds.
A.2 Type Safety

In this section we present the proof of type safety for \( \lambda^\text{REF} \).

**Lemma 59** (Canonical forms). Consider a value \( v \in T[G] \). Then either \( v = u \), or \( v = \varepsilon u :: G \) with \( u \in T[G'] \) and \( \varepsilon \vdash G' \sim G \). Furthermore:

1. If \( G = \text{Bool} \) then either \( v = b \) or \( v = \varepsilon b :: \text{Bool} \) with \( b \in T[\text{Bool}] \).
2. If \( G = \text{Int} \) then either \( v = n \) or \( v = \varepsilon n :: \text{Int} \) with \( n \in T[\text{Int}] \).
3. If \( G = G_1 \rightarrow G_2 \) then either \( v = (\lambda x^{G_1}, t^{G_2}) \) with \( t^{G_2} \in T[G_2] \) or \( v = \varepsilon (\lambda x^{G_1}, t^{G_2}) :: G_1 \rightarrow G_2 \) with \( t^{G_2} \in T[G'_2] \) and \( \varepsilon \vdash G'_1 \rightarrow G'_2 \sim G_1 \rightarrow G_2 \).
4. If \( G = \text{Ref} G' \) then either \( v = o^{G'} \) or \( v = \varepsilon o^{G'} \) with \( o^{G'} \in T[\text{Ref} G'] \) and \( \varepsilon \vdash \text{Ref} G' \rightarrow \text{Ref} G \).

**Proof.** By direct inspection of the formation rules of gradual intrinsic terms (Figure 3.4).

**Lemma 60** (Substitution). If \( t^G \in T[G] \) and \( v \in T[G_1] \), then \( [v/x^{G_1}] t^G \in T[G] \).

**Proof.** By induction on the derivation of \( t^G \).

**Proposition 61** (\( \rightarrow \) is well defined). If \( t^G \vdash \mu' \) and \( t^G \rightarrow r \), then \( r \in \text{CONFIG} \cup \{ \text{error} \} \), and if \( r = t^G \mid \mu' \), then also \( t^G \vdash \mu' \) and \( \text{dom} (\mu) \subseteq \text{dom} (\mu') \).

**Proof.** By induction on the structure of a derivation of \( t^G \rightarrow r \), considering the last rule used in the derivation.

**Case** \((r1)\). Then \( t^G = t^{B_3} = \varepsilon_1 b_1 \oplus \varepsilon_2 b_2 \). Then

\[
(b_1 \in T[B_1]) \quad \varepsilon_1 \vdash B_1 \sim B_1 \quad b_2 \in T[B_2] \quad \varepsilon_2 \vdash B_2 \sim B_2 \quad t y (\oplus) = B_1 x B_2 \rightarrow B_3
\]

Therefore

\[
\varepsilon_1 b_1 \oplus \varepsilon_2 b_2 | \mu \rightarrow b_3 | \mu \text{ where } b_3 = b_1 [\oplus] b_2
\]

But \( b_3 \in T[B_3] \) and the result holds.

**Case** \((r2)\). Then \( t^G = \varepsilon_1 (\lambda x^{G_1} , t_1^{G_12}) @^{G_1 \rightarrow G_2} (\varepsilon_2 u) \) and \( G = G_2 \). Then

\[
(\lambda x^{G_1}, t_1^{G_12}) \in T[G_{11} \rightarrow G_{12}] \quad u \in T[G'_2] \quad \varepsilon_2 \vdash G'_2 \sim G_1
\]

Therefore

\[
\varepsilon_1 (\lambda x^{G_1}, t_1^{G_12}) @^{G_1 \rightarrow G_2} \varepsilon_2 u \in T[G_2]
\]
If $\varepsilon' = (\varepsilon_2 \circ idom(\varepsilon_1))$ is not defined, then $t^G \rightarrow \textbf{error}$, and then the result holds immediately. Suppose that consistent transitivity does hold, then

$$
\varepsilon_1(\lambda x^{G_{11}}, t^{G_{12}}_1) @^{G_1 \rightarrow G_2} \varepsilon_2 \varepsilon u | \mu \rightarrow icod(\varepsilon_1)([(\varepsilon' u :: G_{11})/x^{G_{11}}]t) :: G_2 | \mu
$$

As $\varepsilon_2 \vdash G'_2 \sim G_1$ and by inversion lemma $idom(\varepsilon_1) \vdash G_1 \sim G_{11}$, then $\varepsilon' \vdash G'_2 \sim G_{11}$. Therefore $\varepsilon' u :: G_{11} \in T[G_{11}]$, and by Lemma 275, $t^{G_{12}} = [(\varepsilon' u :: G_{11})/x^{G_{11}}]t^{G_{12}} \in T[G_{12}]$.

Then

$$
(IG::) \dfrac{t^{G_{12}} \in T[G_{12}] \quad icod(\varepsilon_1) \vdash G_{12} \sim G_2}{icod(\varepsilon_1)t^{G_{12}} :: G_2 \in T[G_2]}
$$

and the result holds.

\textit{Case (r3 = true).} Then $t^G = \text{if } \varepsilon_1 \text{b then } \varepsilon_2 t^{G_2} \text{ else } \varepsilon_3 t^{G_3}, G = G_2 \cap G_3$ and

$$
\begin{align*}
b & \in T[G_1] \quad \varepsilon_1 \vdash G_1 \sim \text{Bool} \quad G = (G_2 \cap G_3)\\
t^{G_2} & \in T[G_2] \quad \varepsilon_2 \vdash G_2 \sim G \\
t^{G_3} & \in T[G_3] \quad \varepsilon_3 \vdash G_3 \sim G
\end{align*}
$$

(IGif)

if $\varepsilon_1 \text{b then } \varepsilon_2 t^{G_2} \text{ else } \varepsilon_3 t^{G_3} \in T[G]$ 

Therefore

if $\varepsilon_1 \text{b then } \varepsilon_2 t^{G_2} \text{ else } \varepsilon_3 t^{G_3} | \mu \rightarrow \varepsilon_2 t^{G_2} :: G_2 \cap G_3 | \mu$

But

$$
(IG::) \dfrac{t^{G_2} \in T[G_2] \quad \varepsilon \vdash G_2 \sim G_2 \cap G_3}{\varepsilon t^{G_2} :: G_2 \cap G_3 \in T[G_2 \cap G_3]}
$$

and the result holds.

\textit{Case (r3 = false).} Analogous to case (if-true).

\textit{Case (r4).} Then $t^G = \text{ref}^{G_2} \varepsilon u$. Then

$$
(IGref) \dfrac{u \in T[G_1] \quad \varepsilon \vdash G_1 \sim G_2}{\text{ref}^{G_2} \varepsilon u \in T[\text{Ref} \ G_2]}
$$

Then

$$
\text{ref}^{G_2} \varepsilon u | \mu \rightarrow o^{G_2} | \mu[o^{G_2} \mapsto \varepsilon u :: G_2]
$$

where $o \not\in \text{dom}(\mu)$. But as $\varepsilon u :: G_2 \in T[G_2]$, then $o^{G_2} \vdash \mu[o^{G_2} \mapsto \varepsilon u :: G_2]$. Also $o^{G_2} \in T[\text{Ref} \ G_2]$ and the result holds.

\textit{Case (r5).} Then $t^G = !^{G_2}(\varepsilon o^{G_1})$. Then

$$
(IGasgn) \dfrac{o^{G_1} \in T[\text{Ref} \ G_1] \quad \varepsilon \vdash \text{Ref} \ G_1 \sim \text{Ref} \ G_2}{!^{G_2}(\varepsilon o^{G_1}) \in T[G_2]}
$$

Then

$$
!^{G_2}(\varepsilon o^{G_1}) | \mu \rightarrow i\text{ref}(\varepsilon) v :: G_2 | \mu
$$

where $v = \mu(o^{G_2})$ As $\mu$ is well formed, then $v \in T[G_1]$. Then by inversion lemma $i\text{ref}(\varepsilon) \vdash G_1 \sim G_2$, therefore $i\text{ref}(\varepsilon) v :: G_2 \in T[G_2]$ and the result holds.
We know that $\varepsilon = \varepsilon_1 o^{G_1} := G_3 \varepsilon_2 u$. Then

\[
\begin{align*}
o^{G_1} & \in T[Ref\ G_1] \\
u & \in T[G_2] \\
\varepsilon_1 & \vdash Ref\ G_1 \sim Ref\ G_3 \\
\varepsilon_2 & \vdash G_2 \sim G_3 \\
\varepsilon_1 o^{G_1} := G_3 & \varepsilon_2 u \in T[Unit]
\end{align*}
\]

If $\varepsilon' = (\varepsilon_2 \triangleright iref(\varepsilon_1))$ is not defined, then $t^G \rightarrow error$, and then the result hold immediately. Suppose that consistent transitivity does hold, then

\[
\varepsilon_1 o^{G_1} := G_3 \varepsilon_2 u \mid \mu \rightarrow \text{unit} \mid \mu[o^{G_1} \mapsto \varepsilon' u :: G_1]
\]

As $\varepsilon_2 \vdash G_2 \sim G_3$ and by inversion lemma $iref(\varepsilon_1) \vdash G_1 \sim G_3$, and as evidence is simmetrical $iref(\varepsilon_1) \vdash G_3 \sim G_1$, then $\varepsilon' \vdash G_2 \sim G_1$. Therefore $\varepsilon' u :: G_1 \in T[G_1]$, and therefore $\mu \vdash \mu[o^{G_1} \mapsto \varepsilon' u :: G_1]$. Also

\[
\theta(\text{unit}) = \text{Unit} \implies \text{unit} \in T[Unit]
\]

and the result holds.

\[\square\]

**Proposition 62** ( $\rightarrow$ is well defined). If $t^G \vdash \mu$ and $t^G \mid \mu \rightarrow r$, then $r \in Config \cup \{ error \}$, and if $r = t^G \mid \mu'$, then also $t^G \vdash \mu'$ and $dom(\mu) \subseteq dom(\mu')$.

**Proof.** By induction on the structure of a derivation of $t^G \rightarrow r$.

**Case** (R $\rightarrow$ ). $t^G \mid \mu \rightarrow r$. By well-definedness of $\rightarrow$ (Prop 280), $r \in Config \cup \{ error \}$, and if $r = t^G \mid \mu'$, then also $t^G \vdash \mu'$ and $dom(\mu) \subseteq dom(\mu')$.

**Case** (Ref). $t^G = E[t^G_1]$, $E[t^G_1] \in T[G]$, $t^G_1 \mid \mu \rightarrow t^G_2 \mid \mu'$, $t^G_1 \in T[G']$, and $E : T[G'] \rightarrow T[G]$. By induction hypothesis, $t^G_2 \in T[G']$, so $E[t^G_2] \in T[G]$.

By induction hypothesis we also know that $t^G_2 \vdash \mu'$.

If $freeLocs(t^G_2) \subseteq \mu'$, $freeLocs(f[t^G_1]) \subseteq \mu$, and $dom(\mu) \subseteq dom(\mu')$, then it is easy to see that $freeLocs(f[t^G_1]) \subseteq \mu'$, and therefore conclude that $f[t^G_2] \vdash \mu'$.

**Case** (Ref, RFerr). $r = error$.

**Case** (RF). Let $EvTerm_{G_2}$ be notation for the family of evidence terms $\varepsilon t^{G_1}$ such that $\varepsilon \vdash G_1 \sim G_2$. Then $t^G = F[et], F[et] \in T[G]$, and $F : EvTerm_{G_2} \rightarrow T[G]$, and $et \rightarrow_c et'$. Then there exists $G_e, G_e$ such that $et = \varepsilon_e t_e^{G_2}$ and $\varepsilon_e \vdash G_e \sim G_x$. Also, $t_e = \varepsilon_e u :: G_e$, with $u \in T[G_e]$ and $\varepsilon_e \vdash G_e \sim G_x$.

We know that $\varepsilon_e = \varepsilon_e o^c \varepsilon_e$ is defined, and $et = \varepsilon_e t_e \rightarrow_c \varepsilon_e u = et'$. By definition of $o^c$ we have $\varepsilon_e \vdash G_e \sim G_x$, so $F[et'] \in T[G]$.

As $freeLocs(et) = freeLocs(et')$ and $\mu' = \mu$ then it is easy to conclude that $F[et'] \vdash \mu$.

\[\square\]

Now we can establish type safety: programs do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

**Proposition 63** (Type Safety). If $t^G \in T[G]$ then one of the following is true:
1. \(t^G\) is a value \(v\);

2. if \(t^G \vdash \mu\) then \(t^G \mid \mu \mapsto t'^G \mid \mu'\) for some term \(t'^G \in T[G]\) and some \(\mu'\) such that \(t^G \vdash \mu'\) and \(\text{dom}(\mu) \subseteq \text{dom}(\mu')\);

3. \(t^G \mid \mu \mapsto \text{error}\).

**Proof.** By induction on the structure of \(t^G\). We only present some cases as all proceed the same way.

**Case** \((IGc, IGx, IG\lambda, IGo)\). \(t^G\) is a value.

**Case** \((IG::)\). \(t^G = \varepsilon_1 t^G_1 :: G_2\), and

\[
\begin{array}{c}
\text{(I::)} \quad t^G_1 \in T[G_1] \quad \varepsilon_1 \vdash G_1 \sim G_2 \\
\varepsilon_1 t^G_1 :: G_2 \in T[G_2]
\end{array}
\]

By induction hypothesis on \(t^G_1\), one of the following holds:

1. \(t^G_1\) is a simple value \(u\), in which case \(t^G\) is also a value.

2. \(t^G_1\) is an ascribed value \(v\), then the result holds by Prop 282 and either (RF), or (RFerr).

3. \(t^G_1 \mid \mu \mapsto r_1\) for some \(r_1 \in \text{CONFIG}_G \cup \{\text{error}\}\). Hence \(t^G \mid \mu \mapsto r\) for some \(r \in \text{CONFIG}_G \cup \{\text{error}\}\) by Prop 282 and either (RE), or (REerr).

**Case** \((IGf)\). \(t^G = \begin{cases} \varepsilon_1 t^G_1 \text{ then } \varepsilon_2 t^G_2 \text{ else } \varepsilon_3 t^G_3 \end{cases}\) and

\[
\begin{array}{c}
t^G_1 \in T[G_1] \quad \varepsilon_1 \vdash G_1 \sim \text{Bool} \\
t^G_2 \in T[G_2] \quad \varepsilon_2 \vdash G_2 \sim G \\
t^G_3 \in T[G_3] \quad \varepsilon_3 \vdash G_3 \sim G
\end{array}
\]

By induction hypothesis on \(t^G_1\), one of the following holds:

1. \(t^G_1\) is a simple value \(u\), then by \((R\rightsquigarrow)\), \(t^G \mid \mu \mapsto r\) and \(r \in \text{CONFIG}_G \cup \{\text{error}\}\) by Prop 282.

2. \(t^G_1\) is an ascribed value \(v\), then, \(\varepsilon_1 t^G_1 \rightarrow_c r'\) for some \(r' \in \text{EVTERM}_{\text{Bool}} \cup \{\text{error}\}\). Hence \(t^G \mid \mu \mapsto r\) for some \(r \in \text{CONFIG}_G \cup \{\text{error}\}\) by Prop 282 and either (RF), or (RFerr).

3. \(t^G_1 \mid \mu \mapsto r_1\) for some \(r_1 \in T[G_1] \cup \{\text{error}\}\). Hence \(t^G \mid \mu \mapsto r\) for some \(r \in \text{CONFIG}_G \cup \{\text{error}\}\) by Prop 282 and either (RE), or (REerr).

**Case** \((IGapp)\). \(t^G = (\varepsilon_1 t^G_1) @^{G_1 \rightarrow G_{12}} (\varepsilon_2 t^G_2)\)

\[
\begin{array}{c}
t^G_1 \in T[G_1] \quad \varepsilon_1 \vdash G_1 \rightarrow G_{12} \\
t^G_2 \in T[G_2] \quad \varepsilon_2 \vdash G_2 \rightarrow G_{11} \\
(\varepsilon_1 t^G_1) @^{G_1 \rightarrow G_{12}} (\varepsilon_2 t^G_2) \in T[G_{12}]
\end{array}
\]

By induction hypothesis on \(t^G_1\), one of the following holds:
1. \( t^{G_1} \) is a value \((\lambda x^{G_{11}}, t^{G_{12}})\) (by canonical forms Lemma \(274\)), posing \( G_1 = G'_{11} \rightarrow G'_{12} \). Then by induction hypothesis on \( t^{G_2} \), one of the following holds:

(a) \( t^{G_2} \) is a simple value \( u \), then by \((R \rightarrow)\), \( t^G | \mu \mapsto r \) and \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Prop \(282\).

(b) \( t^{G_2} \) is an ascribed value \( v \), then, \( \varepsilon_2 t^{G_2} \rightarrow_c r' \) for some \( r' \in \text{EvTerm}_{G_{11}} \cup \{\text{error}\} \). Hence \( t^G | \mu \mapsto r \) for some \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Prop \(282\) and either \((R_F)\), or \((R_{F \text{err}})\).

(c) \( t^{G_2} | \mu \mapsto r_2 \) for some \( r_2 \in \text{CONFIG}_{G_{2}} \cup \{\text{error}\} \). Hence \( t^G | \mu \mapsto r \) for some \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Prop \(282\) and either \((R)\), or \((R_{\text{Eerr}})\).

2. \( t^{G_1} \) is an ascribed value \( v \), then, \( \varepsilon_1 t^{G_1} \rightarrow_c r' \) for some \( r' \in \text{EvTerm}_{G_{11} \rightarrow G_{12}} \cup \{\text{error}\} \). Hence \( t^G | \mu \mapsto r \) for some \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Prop \(282\) and either \((R_F)\), or \((R_{\text{Ferr}})\).

3. \( t^{G_1} | \mu \mapsto r_1 \) for some \( r_1 \in \text{CONFIG}_{G_1} \cup \{\text{error}\} \). Hence \( t^G | \mu \mapsto r \) for some \( r \in \text{CONFIG}_G \cup \{\text{error}\} \) by Prop \(282\) and either \((R)\), or \((R_{\text{Eerr}})\).

Case. Other cases are similar to the app case.

\[\square\]

### A.3 Gradual Guarantee

In this section we present the proof of the conservative extensions of the static discipline and the static and the dynamic gradual guarantee.

#### A.3.1 Conservative Extensions of the Static Discipline

**Proposition 11** (Equivalence for fully-annotated terms (statics)). For any \( t \in \text{TERM} \), . \( \vdash_s t : T \) if and only if \( . \vdash t : T \)

**Proof.** By induction over the typing derivations. The proof is trivial because static types are given singleton meanings via concretization. \(\square\)

The equivalence for the dynamics of fully-annotated terms is defined in terms of a logical relation between terms of the static language \( \lambda^{REF} \), and \( \lambda^{\text{REF}}_e \) terms. The logical relation is presented in Figure A.1.

**Definition 49** (Related substitutions). We say that tuples \( \langle \sigma_1, \mu_1 \rangle \) and \( \langle \sigma_2, \mu_2 \rangle \) are related under \( \Gamma \) and \( \Sigma \), notation \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \) if \( \sigma_1 \models \Gamma \), \( \sigma_2 \models \Gamma \), \( \Sigma \vdash \mu_1 \), \( \Sigma \vdash \mu_2 \), \( \mu_1 \approx \mu_2 \), and

\[
\forall x \in \text{dom}(\Gamma), \langle \sigma_1(x), \mu_1 \rangle \approx \langle \sigma_2(x), \mu_2 \rangle : \Gamma(x)
\]
\( b = b \quad \theta(c) = B \quad \mu_1 \approx \mu_2 \)

\( \langle b, \mu_1 \rangle \approx \langle b, \mu_2 \rangle : B \)

\( o = o \quad \mu_1 \approx \mu_2 \)

\( \langle o, \mu_1 \rangle \approx \langle o^G, \mu_2 \rangle : B \)

\( \mu_1 \approx \mu_2 \quad \forall v_1', v_2', \varepsilon_1, \varepsilon_2, \mu_1', \mu_2', \)

\( \varepsilon_1 \vdash T_1 \rightarrow T_2 \sim T_1 \rightarrow T_2 \quad \varepsilon_2 \vdash T_1 \sim T_2 \)

\( \langle v_1, \mu_1 \rangle \approx \langle v_2, \mu_2 \rangle : T \)

\( \varepsilon \vdash T \sim T \)

\( \langle v_1, \mu_1 \rangle \approx \langle \varepsilon u_2 :: T, \mu_2 \rangle : T \)

\( \langle v, \mu_1 \rangle \approx \langle \varepsilon u_2 : T, \mu_2 \rangle : T \)

Figure A.1: Logical relation between \( \lambda_{\text{REF}} \) and \( \lambda_{\text{REF}}^\varepsilon \)

**Definition 50** (Semantic equivalence).

\[ \Gamma; \Sigma \vdash t \approx t^T : T \iff \forall \sigma_1, \sigma_2, \mu_1, \mu_1, \Sigma \vdash \mu_1, \Sigma \vdash \mu_2, \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle, \]  
we have \( \langle \sigma_1(t), \mu_1 \rangle \approx \langle \sigma_2(t^T), \mu_2 \rangle : T \)

**Proposition 64** (Fundamental property). For any \( t \in \text{TERM} \), \( \Gamma; \Sigma \vdash t : T \), \( \Gamma; \Sigma \vdash t \sim_\varepsilon t^T : T \), then \( \Gamma; \Sigma \vdash t \approx t^T : T \).

**Proof.** By induction on the type derivation of \( t \).

**Case (Tx).** Then \( t = x \) and therefore

\( (\text{Tx}) \quad x : T \in \Gamma \)

\( \Gamma; \Sigma \vdash x : T \)

Then we have to prove that \( x \approx x^T : T \). But the result follows directly by Prop \[68 \]

**Case (Tb).** Then \( t = b \) and therefore

\( (\text{Tc}) \quad \theta(b) = B \)

\( \Gamma; \Sigma \vdash b : B \)

and \( t^T = b \). The result follows by Prop \[67 \]

**Case (Tapp).** Then \( t = t_1 t_2 \) and \( T = T_2 \) where

\( (\text{Tapp}) \quad \Gamma; \Sigma \vdash t_1 : T_1 \rightarrow T_2 \)

\( \Gamma; \Sigma \vdash t_2 : T_1 \rightarrow T_2 \)

and

\( \Gamma; \Sigma \vdash t_1 \sim_\varepsilon t^{G_1} : T_1 \rightarrow T_2 \)

\( \Gamma; \Sigma \vdash t_2 \sim_\varepsilon t^{G_2} : T_1 \rightarrow T_2 \)

\( \varepsilon_1 = \langle T_1 \rightarrow T_2 \rangle = g_\Rightarrow(T_1 \rightarrow T_2, \overline{\text{dom}(T_1 \rightarrow T_2) \rightarrow \text{cod}(T_1 \rightarrow T_2)}) \)

\( \varepsilon_2 = \langle T_1 \rangle = g_\Rightarrow(T_1, \overline{\text{dom}(T_1 \rightarrow T_2)}) \)

\( \langle T_1 \rightarrow T_2 \rangle \sim_\varepsilon \varepsilon_1 t_1^T \rightarrow T_2 \rightarrow T_2 \varepsilon_2 t_1^T : T_2 \)
We have to prove that $\Gamma; \Sigma \vdash t_1 t_2 \approx \varepsilon_1 t_1^{T_1 \to T_2} @ T_1 \to T_2 \varepsilon_2 t_2^{T_1} : T_2$. By induction hypotheses we know that $\Gamma; \Sigma \vdash t_1 \approx t_1^{T_1 \to T_2} : T_1 \to T_2$ and that $\Gamma; \Sigma \vdash t_2 \approx t_2^{T_1} : T_1$. The result follows directly by Prop 69.

**Case (Top).** Then $t = t_1 \oplus t_2$ and $T = B_3$, where

\[
\frac{\Gamma; \Sigma \vdash t_1 : B_1 \quad \Gamma; \Sigma \vdash t_2 : B_2}{\Gamma; \Sigma \vdash t_1 \oplus t_2 : B_3} \quad \text{(Top)}
\]

and

\[
\frac{\varepsilon_1 = \langle B_1 \rangle = \mathcal{g}_- (B_1, B_1) \quad \varepsilon_2 = \langle B_2 \rangle = \mathcal{g}_- (B_2, B_2)}{\Gamma; \Sigma \vdash \varepsilon_1 t_1^{B_1} \oplus \varepsilon_2 t_2^{B_2} : B_3} \quad \text{(TRop)}
\]

We have to prove that $\Gamma; \Sigma \vdash t_1 \oplus t_2 \approx \varepsilon_1 t_1^{B_1} \oplus \varepsilon_2 t_2^{B_2} : T_2$. By induction hypotheses we know that $\Gamma; \Sigma \vdash t_1 \approx t_1^{B_1} : B_1$ and that $\Gamma; \Sigma \vdash t_2 \approx t_2^{B_2} : B_2$. The result follows directly by Prop 70.

**Case (Tif).** Then $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$, where

\[
\frac{\Gamma; \Sigma \vdash t_1 : \text{Bool} \quad \Gamma; \Sigma \vdash t_2 : T \quad \Gamma; \Sigma \vdash t_3 : T}{\Gamma; \Sigma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad \text{(Tif)}
\]

and

\[
\frac{\varepsilon_1 = \langle \text{Bool} \rangle = \mathcal{g}_- (\text{Bool}, \text{Bool}) \quad \varepsilon_2 = \langle T \rangle = \mathcal{g}_- (T, T) \quad \varepsilon = \langle T \rangle = \mathcal{g}_- (T, T)}{\Gamma; \Sigma \vdash \varepsilon_1 t_1^{\text{Bool}} \oplus \varepsilon_2 t_2^T \oplus \varepsilon t_3^T : T} \quad \text{(TRif)}
\]

We have to prove that $\Gamma; \Sigma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \approx \varepsilon_1 t_1^{\text{Bool}} \oplus \varepsilon_2 t_2^{T} \oplus \varepsilon t_3^{T} : T$. By induction hypotheses we know that $\Gamma; \Sigma \vdash t_1 \approx t_1^{\text{Bool}} : \text{Bool}$ and that $\Gamma; \Sigma \vdash t_2 \approx t_2^{T} : T$, and $\Gamma; \Sigma \vdash t_3 \approx t_3^{T} : T$. The result follows directly by Prop 71.

**Case (T\lambda).** Then $t = (\lambda x : T_1. t')$ and $T = T_1 \to T_2$, and therefore

\[
\frac{\Gamma, x : T_1; \Sigma \vdash t' : T_2}{\Gamma; \Sigma \vdash (\lambda x : T_1. t') : T_1 \to T_2} \quad \text{(T\lambda)}
\]

and

\[
\frac{\Gamma, x : T_1; \Sigma \vdash t' \approx t_1^{T_1 \cdot T_2} : T_2}{\Gamma; \Sigma \vdash \lambda x : T_1. t' : T_1 \to T_2} \quad \text{(TRA)}
\]

Then we have to prove that $\Gamma; \Sigma \vdash (\lambda x : T_1. t') \approx (\lambda x^{T_1 \cdot T_2} : T_1 \to T_2$. By induction hypothesis we already know that $\Gamma, x : T_1; \Sigma \vdash t' \approx t_2^{T_2} : T_2$. But the result follows directly by Prop 72.

**Case (T::).** Then $t = t' :: T$

\[
\frac{\Gamma; \Sigma \vdash t' : T \quad T = T}{\Gamma; \Sigma \vdash (t' :: T) : T} \quad \text{(T::)}
\]

and

\[
\frac{\varepsilon = \langle T \rangle = \mathcal{g}_- (T, T)}{\Gamma; \Sigma \vdash (t :: T) \approx \varepsilon t_1^{T} :: T : T} \quad \text{(TR::)}
\]

We have to prove that $\Gamma; \Sigma \vdash t' :: T \approx \varepsilon t_1^{T} :: T : T$. By induction hypothesis we know that $\Gamma; \Sigma \vdash t' \approx t_2^{T} : T$. The result follows directly by Prop 73.
Case \((T\text{ref})\). Then \(t = \text{ref}\ t'\) and \(T = \text{Ref}\ T\), where

\[
\begin{align*}
\text{(Tref)} & : \frac{\Gamma; \Sigma \vdash s\ t' : T}{\Gamma; \Sigma \vdash \text{ref}\ t' : \text{Ref}\ T}
\end{align*}
\]

and

\[
\begin{align*}
\text{(TRref)} & : \frac{\Gamma; \Sigma \vdash t' \sim_{\varepsilon} t^{\text{ref}} T : \text{Ref}\ T \quad \varepsilon = \langle\text{Ref}\ T\rangle \mathcal{G}(\text{Ref}\ T, \text{Ref}\ T)}{\Gamma; \Sigma \vdash \text{ref}\ t' \sim_{\varepsilon} \text{ref}^{G} \in t^{\text{ref}} T : \text{Ref}\ T}
\end{align*}
\]

We have to prove that \(\Gamma; \Sigma \vdash \text{ref}\ t' \approx \text{ref}^{G} \in t^{\text{ref}} T : \text{Ref}\ T\). By induction hypothesis we know that \(\Gamma; \Sigma \vdash t' \approx t^{\text{ref}} T : T\). The result follows directly by Prop \([74]\)

Case \((T\text{deref})\). Then \(t = \text{!t'}\), where

\[
\begin{align*}
\text{(Tderef)} & : \frac{\Gamma; \Sigma \vdash s\ t' : \text{Ref}\ T}{\Gamma; \Sigma \vdash \text{!t'} : T}
\end{align*}
\]

and

\[
\begin{align*}
\text{(TRderef)} & : \frac{\Gamma; \Sigma \vdash t' \sim_{\varepsilon} t^{\text{deref}} T : \text{Ref}\ T \quad \varepsilon = \langle\text{Ref}\ T\rangle \mathcal{G}(\text{Ref}\ T, \text{Ref}\ T)}{\Gamma; \Sigma \vdash \text{!t'} \sim_{\varepsilon} \text{!}^{\text{deref}} T \in t^{\text{deref}} T : T}
\end{align*}
\]

We have to prove that \(\Gamma; \Sigma \vdash \text{!t'} \approx \text{!}^{\text{deref}} T \in t^{\text{deref}} T : T\). By induction hypothesis we know that \(\Gamma; \Sigma \vdash t' \approx t^{\text{deref}} T : \text{Ref}\ T\). The result follows directly by Prop \([75]\)

Case \((T\text{asgn})\). Then \(t = t_{1} := t_{2}\) and \(T = \text{Unit}\), where

\[
\begin{align*}
\text{(Tasgn)} & : \frac{\Gamma; \Sigma \vdash s\ t_{1} : \text{Ref}\ T \quad \Gamma; \Sigma \vdash s\ t_{2} : T}{\Gamma; \Sigma \vdash s\ t_{1} := t_{2} : \text{Unit}}
\end{align*}
\]

and

\[
\begin{align*}
\text{(TRasgn)} & : \frac{\Gamma; \Sigma \vdash t_{1} \sim_{\varepsilon} t^{\text{asgn}} T : \text{Ref}\ T \quad \varepsilon_{1} = \langle\text{Ref}\ T\rangle \mathcal{G}(\text{Ref}\ T, \text{Ref}\ T) \quad \varepsilon_{2} = \langle\text{Ref}\ T\rangle \mathcal{G}(\text{Ref}\ T, \text{Ref}\ T)\quad \varepsilon = \langle\text{Ref}\ T\rangle \mathcal{G}(\text{Ref}\ T, \text{Ref}\ T)}{\Gamma; \Sigma \vdash t_{1} := t_{2} \sim_{\varepsilon} t^{\text{asgn}} T \in t^{\text{asgn}} T : \text{Unit}}
\end{align*}
\]

We have to prove that \(\Gamma; \Sigma \vdash t_{1} := t_{2} \approx \varepsilon^{t^{\text{asgn}} T \in t^{\text{asgn}} T : \text{Unit}}\). By induction hypotheses we know that \(\Gamma; \Sigma \vdash t_{1} \approx t^{\text{asgn}} T : \text{Ref}\ T\) and that \(\Gamma; \Sigma \vdash t_{2} \approx t^{\text{asgn}} T : \text{Ref}\ T\). The result follows directly by Prop \([76]\)

Case \((T\text{o})\). Then \(t = o\) and \(T = \text{Ref}\ T\), where

\[
\begin{align*}
\text{(To)} & : \frac{o : T \in \Sigma}{\Gamma; \Sigma \vdash o : \text{Ref}\ T}
\end{align*}
\]

and

\[
\begin{align*}
\text{(TRi)} & : \frac{o : G \in \Sigma}{\Gamma; \Sigma \vdash o \sim_{\varepsilon} o^{G} : \text{Ref}\ G}
\end{align*}
\]

Then we have to prove that \(o \approx o^{G} : \text{Ref}\ T\). But the result follows directly by Prop \([77]\)

\[
\square
\]

Lemma \(65\) (Reduction preserves relations). Consider \(\mu_{1} \approx \mu_{2}\), \(t \vdash^{*} t' \vdash \mu_{1}\), and \(t^{T} \vdash \mu_{2} \longrightarrow^{*} t'^{T} \vdash \mu'_{2}\), then \(\langle t, \mu_{1} \rangle \approx \langle t^{T}, \mu_{2} \rangle : \iff \langle t', \mu'_{1} \rangle \approx \langle t'^{T}, \mu'_{2} \rangle : \)

Proof. By definition of related terms and transitivity of \(\longrightarrow^{*}\). \(\square\)
Lemma 66. Consider \( \langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : \) and \( \mu'_1 \approx \mu'_2, \) such that \( \mu_1 \subseteq \mu'_1 \) and \( \mu_2 \subseteq \mu'_2, \) then \( \langle t, \mu'_1 \rangle \approx \langle t^T, \mu'_2 \rangle : \)

Proof. Direct as evolution of the store to related store does not alter the relation between values that not depend on new locations. \( \square \)

Proposition 67 (Compatibility \( Tb \)). If \( b \in B, \) then \( \Gamma; \Sigma \vdash b \approx b : B. \)

Proof. Trivial as \( b = b. \) \( \square \)

Proposition 68 (Compatibility \( Tx \)). If \( x : T \in \Gamma, \) then \( \Gamma; \Sigma \vdash x \approx x^T : T. \)

Proof. Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2, \) such that \( \Gamma; \Sigma \vdash \sigma_1, \mu_1 \approx \sigma_2, \mu_2. \) We are required to show that:

\[
\langle \sigma_1(x), \mu_1 \rangle \approx \langle \sigma_2(x^T), \mu_2 \rangle : T
\]

which is immediately by the definition of \( \Gamma; \Sigma \vdash \Gamma; \Sigma \vdash \sigma_1, \mu_1 \approx \sigma_2, \mu_2. \) \( \square \)

Proposition 69 (Compatibility \( Tapp \)). If \( \Gamma; \Sigma \vdash t_1 \approx t_1^{T_1 \rightarrow T_2} : T_1 \rightarrow T_2, \)\( \Gamma; \Sigma \vdash t_2 \approx t_2^{T_1} : T_1, \) \( \varepsilon_1 \vdash T_1 \rightarrow T_2 \sim T_1 \rightarrow T_2, \varepsilon_2 \vdash T_1 \sim T_1, \) then \( \Gamma; \Sigma \vdash t_1 t_2 \approx \varepsilon_1 t_1^{T_1 \rightarrow T_2} \oplus T_1 \rightarrow T_2 \varepsilon_2 t_1 : T_2. \)

Proof. Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2, \) such that \( \Gamma; \Sigma \vdash \sigma_1, \mu_1 \approx \sigma_2, \mu_2. \) We are required to show that:

\[
\langle \sigma_1(t_1 t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t_1^{T_1 \rightarrow T_2} \oplus T_1 \rightarrow T_2 \varepsilon_2 t_1), \mu_2 \rangle : T_1 \rightarrow T_2
\]

which, by definition of substitution, is equivalent to prove that

\[
\langle \sigma_1(t_1), \sigma_1(t_2), \mu_1 \rangle \approx \langle \varepsilon_1 \sigma_2(t_1^{T_1 \rightarrow T_2}) \oplus T_1 \rightarrow T_2 \varepsilon_2 \sigma_2(t_1), \mu_2 \rangle : T_1 \rightarrow T_2
\]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t_1^{T_1 \rightarrow T_2} : T_1 \rightarrow T_2 \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2. \) We then know that \( \langle t_1, \mu_1 \rangle \approx \langle t_1^{T_1 \rightarrow T_2}, \mu_2 \rangle : T_1 \rightarrow T_2. \) Then suppose \( \sigma_1(t_1) | \mu_1 \mapsto \ast v_1 \mid \mu_1' \) and \( \sigma_2(t_1^{T_1 \rightarrow T_2}) | \mu_2 \mapsto \ast v_2 \mid \mu_2'. \) (otherwise the result holds immediately). We know that \( \langle v_1, \mu_1' \rangle \approx \langle v_2, \mu_2' \rangle : T_1 \rightarrow T_2. \) Similarly we instantiate \( \Gamma; \Sigma \vdash t_2 \approx t_2^{T_1} \rightarrow T_1 \) with \( \sigma_1, \sigma_2, \mu_1' \) and \( \mu_2'. \) Notice \( \mu_1 \subseteq \mu_1' \| \mu_2 \subseteq \mu_2' \) (resp.), therefore \( \Sigma \vdash \mu_1' \) (\( \Sigma \vdash \mu_2' \) resp.). Then we know that \( \langle t_2, \mu_1' \rangle \approx \langle t_1^{T_1}, \mu_2' \rangle : T_1. \) Then suppose \( \sigma_1(t_2) | \mu_1' \mapsto \ast v_1 \mid \mu_1'' \) and \( \sigma_2(t_1^{T_1}) | \mu_2' \mapsto \ast v_2 \mid \mu_2'' \) (otherwise the result holds immediately). We know that \( \langle v_1, \mu_1'' \rangle \approx \langle v_2, \mu_2'' \rangle : T_1. \) Let us assume \( v_2 = v_2 \) (the other case is analogous modulo one trivial step of reduction). Then the result holds by definition of related lambdas instantiating with \( \varepsilon_1, \varepsilon_2, \mu_1', \mu_2', v_{12} \) and \( v_{22}. \) \( \square \)

Proposition 70 (Compatibility \( Top \)). If \( \Gamma; \Sigma \vdash t_1 \approx t^{B_1} : B_1, \) \( \Gamma; \Sigma \vdash t_2 \approx t^{B_2} : B_2, \) \( \varepsilon_1 \vdash B_1 \sim B_1, \varepsilon_2 \vdash B_2 \sim B_2, t(y) = B_1 x B_2 \rightarrow B_3, \) then \( \Gamma; \Sigma \vdash t_1 \oplus t_2 \approx \varepsilon_1 t^{B_1} \oplus \varepsilon_2 t^{B_2} : B_3. \)

Proof. Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2, \) such that \( \Gamma; \Sigma \vdash \sigma_1, \mu_1 \approx \sigma_2, \mu_2. \) We are required to show that:

\[
\langle \sigma_1(t_1 \oplus t_2), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon_1 t^{B_1} \oplus \varepsilon_2 t^{B_2}), \mu_2 \rangle : B_3
\]
which, by definition of substitution, is equivalent to prove that

\[ (\sigma_1(t_1) \oplus \sigma_1(t_2), \mu_1) \approx (\varepsilon_1 \sigma_2(t^{B_1}) \oplus \varepsilon_2 \sigma_2(t^{B_2}), \mu_2) : B_3 \]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t^{B_1} : B_1 \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2 \). We know then that \( \langle t_1, \mu_1 \rangle \approx \langle t^{B_1}, \mu_2 \rangle : B_1 \). Then suppose \( \sigma_1(t_1) | \mu_1 \longrightarrow^* v_{11} \mu_1' \) and \( \sigma_2(t^{B_1}) | \mu_2 \longrightarrow^* v_{21} \mu_2' \) (otherwise the result holds immediately). We know that \( \langle v_{11}, \mu_1' \rangle \approx \langle v_{21}, \mu_2' \rangle : B_1 \). Similarly we instantiate \( \Gamma; \Sigma \vdash t_2 \approx t^{B_2} : B_2 \) with \( \sigma_1, \sigma_2, \mu_1' \) and \( \mu_2' \). Notice \( \mu_1 \subseteq \mu_1' \) (\( \mu_2 \subseteq \mu_2' \) resp.), therefore \( \Sigma \vdash \mu_1' \) (\( \Sigma \vdash \mu_2' \) resp.). Then we know that \( \langle t_2, \mu_1' \rangle \approx \langle t^{B_2}, \mu_2' \rangle : B_2 \). Then suppose \( \sigma_1(t_2) | \mu_1' \longrightarrow^* v_{12} \mu_1'' \) and \( \sigma_2(t^{B_2}) | \mu_2' \longrightarrow^* v_{22} \mu_2'' \) (otherwise the result holds immediately). We know that \( \langle v_{21}, \mu_1'' \rangle \approx \langle v_{22}, \mu_2'' \rangle : B_2 \). Let us assume \( v_{21} = u_{21} \) and \( v_{22} = u_{22} \) (the other cases are analogous modulo one or two trivial steps of reduction). Then \( \sigma_1(t_1) \oplus \sigma_1(t_2) | \mu_1 \longrightarrow^* \varepsilon_1 u_{21} \mu_1'' \oplus \varepsilon_2 u_{22} | \mu_2'' \longrightarrow^* \varepsilon_1 u_{21} \mu_1'' \oplus \varepsilon_2 u_{22} \). But as \( v_{11} = u_{21} \) and \( v_{12} = u_{22} \), then \( v_{11} \oplus \varepsilon_1 u_{21} \mu_1'' \oplus \varepsilon_2 u_{22} \) and the result holds. 

**Proposition 71 (Compatibility Tif).** If \( \Gamma; \Sigma \vdash t_1 \approx t^{B_1} : \text{Bool} \), \( \Gamma; \Sigma \vdash t_2 \approx t^{B_2} : T \), \( \Gamma; \Sigma \vdash t_3 \approx t^T : T \), \( \varepsilon \vdash \text{Bool} \sim T \sim T \), \( \Gamma; \Sigma \vdash t_1 \) then \( t_2 \) else \( t_3 \) \( \approx \) \( \varepsilon t_1^{B_1} \) \( \text{else } \varepsilon t_2^T \) \( \text{else } \varepsilon t_3^T : T \).

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2 \), such that \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). We are required to show that:

\[ \langle \sigma_1(\text{if } t_1 \text{ then } t_2 \text{ else } t_3), \mu_1 \rangle \approx \langle \sigma_2(\text{if } \varepsilon t_1^{B_1} \text{ then } \varepsilon t_2^T \text{ else } \varepsilon t_3^T), \mu_2 \rangle : T \]

which, by definition of substitution, is equivalent to prove that

\[ \langle \text{if } \sigma_1(t_1) \text{ then } \sigma_1(t_2) \text{ else } \sigma_1(t_3), \mu_1 \rangle \approx \langle \text{if } \varepsilon \sigma_2(t_1^{B_1}) \text{ then } \varepsilon \sigma_2(t_2^T) \text{ else } \varepsilon \sigma_2(t_3^T), \mu_2 \rangle : T \]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t^{B_1} : \text{Bool} \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2 \). We know then that \( \langle t_1, \mu_1 \rangle \approx \langle t^{B_1}, \mu_2 \rangle : \text{Bool} \). Then suppose \( \sigma_1(t_1) | \mu_1 \longrightarrow^* \langle v_{11}, \mu_1' \rangle \) and \( \sigma_2(t^{B_1}) | \mu_2 \longrightarrow^* \langle v_{21}, \mu_2' \rangle \) (otherwise the result holds immediately). We know that \( \langle v_{11}, \mu_1' \rangle \approx \langle v_{21}, \mu_2' \rangle : \text{Bool} \). Let us assume \( v_{21} = u_{21} \) (the other case is analogous modulo one trivial step of reduction). Also let us assume \( v_{11} = \text{true} \) (the \text{false} case is analogous), therefore \( \langle v_{11}, \mu_1' \rangle \approx \langle u_{21}, \mu_2' \rangle : \text{Bool}, u_{21} = \text{true} \) as well. Then \( t_1 | \mu_1 \longrightarrow^* t_2 | \mu_1' \) and \( t^{B_1} | \mu_2 \longrightarrow^* \text{if } \langle \text{Bool} \rangle \text{true then } \varepsilon \sigma_2(t_2^T) \text{ else } \varepsilon \sigma_2(t_3^T) | \mu_2' \longrightarrow^* \langle T^T \rangle t_2^T \text{:: } T | \mu_2'. \)

But by instantiating \( \Gamma; \Sigma \vdash t_2 \approx t^{B_2} : T \) with \( \sigma_1, \sigma_2, \mu_1', \) and \( \mu_2' \), then \( \langle t_2, \mu_1' \rangle \approx \langle t^{B_2}, \mu_2' \rangle : T \), and then \( t_2 | \mu_1' \longrightarrow^* \langle v_{12}, \mu_1'' \rangle \) and \( t^{B_2} | \mu_2' \longrightarrow^* \langle v_{22}, \mu_2'' \rangle \), and \( \langle v_{12}, \mu_1'' \rangle \approx \langle v_{22}, \mu_2'' \rangle : T \). Let us assume \( v_{22} = u_{22} \) (the other case is analogous), then \( \langle v_{12}, \mu_1'' \rangle \approx \langle \langle T^T \rangle u_{22} :: T, \mu_2'' \rangle : T \) : the result holds. 

**Proposition 72 (Compatibility T\text{\lambda}).** If \( \Gamma, x : T_1; \Sigma \vdash t' \approx t^{T_2} : T_2 \), then \( \Gamma; \Sigma \vdash (\lambda x : T_1.t') \approx (\lambda x^{T_1}, t^{T_2}) : T_1 \rightarrow T_2 \).

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2 \), such that \( \Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle \). We are required to show that:

\[ \langle \sigma_1((\lambda x : T_1.t')), \mu_1 \rangle \approx \langle \sigma_2((\lambda x^{T_1}, t^{T_2})), \mu_2 \rangle : T_1 \rightarrow T_2 \]
which, by definition of substitution, is equivalent to prove that

\[ \langle (\lambda x : T_1.\sigma_1(t')), \mu_1 \rangle \approx \langle (\lambda x^{T_1}.\sigma_2(t^{T_2})), \mu_2 \rangle : T_1 \rightarrow T_2 \]

Consider \( v'_1, v'_2, \varepsilon_1, \varepsilon_2, \mu'_1, \mu'_2 \) such that \( \mu_1 \subseteq \mu'_1, \mu_2 \subseteq \mu'_2, \langle v'_1, \mu'_2 \rangle \approx \langle v'_2, \mu'_2 \rangle : T_1, \varepsilon_1 = \langle T_1 \rightarrow T_2 \rangle \vdash T_1 \rightarrow T_2 \sim T_1 \rightarrow T_2, \) and \( \varepsilon_2 = \langle T_1 \rangle \vdash T_1 \sim T_1. \) We have to prove that 

\[ \langle (\lambda x : T_1.\sigma_1(t')) | v'_1 | \mu'_1 \mapsto \sigma_1(t')[v'_1/x] \]

and

\[ \varepsilon_1(\lambda x^{T_1}.\sigma_2(t^{T_2})) \circ \varepsilon_2 \vdash \langle T_2 \rangle(\sigma_2(t^{T_2})|\langle T_1 \rangle u_2 :: T_1/x^{T_1}) :: T_2 \mid \mu'_2 \]

Notice that \( \sigma_1(t')[v'_1/x] = \sigma'_1(t'), \) where \( \sigma'_1 = \sigma_1[x \mapsto v'_1], \) and analogously \( \sigma_2(t^{T_2})|\langle T_1 \rangle u_2 :: T_1/x^{T_1} = \sigma'_2(t^{T_2}), \) where \( \sigma'_2 = \sigma_2[x^{T_1} \mapsto \langle T_1 \rangle u_2 :: T_1]. \) Also, as

\[ \langle v_1, \mu'_1 \rangle \approx \langle u_2, \mu'_2 \rangle : T_1 \]

\[ \langle v_1, \mu'_1 \rangle \approx \langle \langle T_1 \rangle u_2 :: T_1, \mu'_2 \rangle : T_1 \]

then \( \Gamma, x : T_1; \Sigma \vdash \langle \sigma'_1, \mu_1 \rangle \approx \langle \sigma'_2, \mu_2 \rangle. \) Then by instantiating premise \( \Gamma, x : T_1; \Sigma \vdash t' \approx t^{T_2} : T_2 \) with \( \sigma'_1, \mu_1, \sigma'_2, \) and \( \mu_2, \) we know that \( \sigma'_1(t') | \mu'_1 \mapsto^* v''_1 | \mu''_1 \iff \sigma'_2(t^{T_2}) | \mu'_2 \mapsto^* v''_2 | \mu''_2, \) where \( \langle v''_1, \mu''_1 \rangle \approx \langle v''_2, \mu''_2 \rangle : T_2. \) If \( v''_2 = \langle T_2 \rangle u''_2 :: T_2 \) (the other case is similar), \( \langle T_2 \rangle u''_2 :: T_2 \mid \mu'_2 \mapsto \langle T_2 \rangle u''_2 :: T_2 \mid \mu'_2, \) but \( \langle v''_1, \mu''_1 \rangle \approx \langle \langle T_2 \rangle u''_2 :: T_2, \mu''_2 \rangle : T_2, \) and the result holds.

\[ \Box \]

**Proposition 73** (Compatibility \( T ::). If \( \Gamma, \Sigma \vdash t \approx t^T : T, \varepsilon \vdash T \sim T, \) then \( \Gamma, \Sigma \vdash t :: T \approx \varepsilon t^T :: T : T. \)

**Proof.** Consider arbitrary \( \sigma_1, \sigma_2, \mu_1, \mu_2, \) such that \( \Gamma, \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle. \) We are required to show that:

\[ \langle \sigma_1(t :: T), \mu_1 \rangle \approx \langle \sigma_2(\varepsilon t^T :: T), \mu_2 \rangle : T \]

which, by definition of substitution, is equivalent to prove that

\[ \langle \sigma_1(t :: T), \mu_1 \rangle \approx \langle \varepsilon \sigma_2(t^T) :: T, \mu_2 \rangle : T \]

We instantiate \( \Gamma, \Sigma \vdash t \approx t^T : T \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2. \) We know then that \( \langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T. \) Then suppose \( \sigma_1(t) | \mu_1 \mapsto^* v_1 | \mu'_1 \) and \( \sigma_2(t^T) | \mu_2 \mapsto^* v_2 | \mu'_2 \) (otherwise the result holds immediately). We know that \( \langle v_1, \mu'_1 \rangle \approx \langle v_2, \mu'_2 \rangle : T. \) Let us assume \( v_2 = u_2 \) (the other case is analogous modulo one trivial step of reduction). Then \( v_1 :: T \mid \mu'_1 \mapsto v_1 \mid \mu'_1, \) so we have to prove that \( \langle v_1, \mu'_1 \rangle \approx \langle \langle T \rangle u_2 :: T, \mu'_2 \rangle : T, \) which holds because \( \langle v_1, \mu'_1 \rangle \approx \langle u_2, \mu'_2 \rangle : T. \)

\[ \Box \]
Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle$. We are required to show that:

$$\langle \sigma_1(\text{ref } t), \mu_1 \rangle \approx \langle \text{ref } t, \epsilon, \text{ref } t : T \rangle$$

which, by definition of substitution, is equivalent to prove that

$$\langle \text{ref } \sigma_1(t), \mu_1 \rangle \approx \langle \text{ref } t, \epsilon, \text{ref } t : T \rangle$$

We instantiate $\Gamma; \Sigma \vdash t \approx t^T : T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that $\langle t, \mu_1 \rangle \approx \langle t^T, \mu_2 \rangle : T$. Then suppose $\sigma_1(t) \mid \mu_1 \longrightarrow^* v_1 \mid \mu_1'$ and $\sigma_2(t^T) \mid \mu_2 \longrightarrow^* v_2 \mid \mu_2'$ (otherwise the result holds immediately). We know that $\langle v_1, \mu_1' \rangle \approx \langle v_2, \mu_2' \rangle : T$. Let us assume $v_2 = u_2$ (the other case is analogous modulo one trivial step of reduction). Then $\text{ref } v_1 \mid \mu_1' \longrightarrow o \mid \mu_1'[o \mapsto v_1]$, and $\text{ref } \epsilon v_2 \mid \mu_2' \longrightarrow o^T \mid \mu_2'[o^T \mapsto \epsilon v_2 : T]$. Let $\mu''_o = [o \mapsto v_1]$ and $\mu''_v = [o^T \mapsto \epsilon u_2 : T]$. By $(S::)$, $\langle v_1, \mu_1' \rangle \approx \langle \epsilon u_2 : T, \mu'_2 \rangle : T$, and by Lemma 66 $\langle v_1, \mu_1' \rangle \approx \langle \epsilon u_2 : T, \mu'_2 \rangle : T$, then by $(S\mu)$, $\mu''_o \approx \mu''_v$, and as $o = o$, by $(S\epsilon)$, $\langle o, \mu''_o \rangle \approx \langle o^T, \mu''_v \rangle : T$ and the result holds.

Proposition 75 (Compatibility $T\text{derf}$). If $\Gamma; \Sigma \vdash t \approx t^\text{Ref } T : \text{Ref } T, \epsilon \vdash \text{Ref } T \sim \text{Ref } T$, then $\Gamma; \Sigma \vdash !t \approx !t^\text{Ref } T \sim \text{Ref } T : T$.

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle$. We are required to show that:

$$\langle !\sigma_1(t), \mu_1 \rangle \approx \langle !t^\text{Ref } T, \mu_1 \rangle : T$$

which, by definition of substitution, is equivalent to prove that

$$\langle !\sigma_1(t), \mu_1 \rangle \approx \langle !t^\text{Ref } T, \mu_1 \rangle : T$$

We instantiate $\Gamma; \Sigma \vdash t \approx t^\text{Ref } T : \text{Ref } T$ with $\sigma_1, \sigma_2$ and arbitrary $\mu_1$ and $\mu_2$ such that $\Sigma \vdash \mu_1$ and $\Sigma \vdash \mu_2$. We know then that $\langle t, \mu_1 \rangle \approx \langle t^\text{Ref } T, \mu_2 \rangle : \text{Ref } T$. Then suppose $\sigma_1(t) \mid \mu_1 \longrightarrow^* v_1 \mid \mu_1'$ and $\sigma_2(t^\text{Ref } T) \mid \mu_2 \longrightarrow^* v_2 \mid \mu_2'$ (otherwise the result holds immediately). We know that $\langle v_1, \mu_1' \rangle \approx \langle v_2, \mu_2' \rangle : T$. Let us assume $v_1 = o$ and $v_2 = o^T$ (the other case is analogous modulo one trivial step of reduction). Then $!t_1 \mid \mu_1' \longrightarrow \mu'_1(o) \mid \mu'_1$, and $t^\text{Ref } T \mid \mu_2 \longrightarrow (T)\mu'_2(o^T) : T \mid \mu_2'$. By definition of $\mu'_1 \approx \mu'_2, \langle \mu'_1(o), \mu'_1 \rangle \approx \langle \mu'_2(o^T), \mu'_2 \rangle : T$, therefore the result follows by $(S::)$.

Proposition 76 (Compatibility $T\text{asgn}$). If $\Gamma; \Sigma \vdash t_1 \approx t^\text{Ref } T : \text{Ref } T, \Gamma; \Sigma \vdash t_2 \approx t^T : T, \epsilon_1 \vdash \text{Ref } T \sim \text{Ref } T, \epsilon_2 \vdash T \sim T$, then $\Gamma; \Sigma \vdash t_1 := t_2 \approx \epsilon_1 t^\text{Ref } T := T \epsilon_2 t^T : \text{Unit}$.

Proof. Consider arbitrary $\sigma_1, \sigma_2, \mu_1, \mu_2$, such that $\Gamma; \Sigma \vdash \langle \sigma_1, \mu_1 \rangle \approx \langle \sigma_2, \mu_2 \rangle$. We are required to show that:

$$\langle \sigma_1(t_1 := t_2), \mu_1 \rangle \approx \langle \text{ref } \sigma_2(\epsilon_1 t^\text{Ref } T := T \epsilon_2 t^T), \mu_2 ] : \text{Unit}$$

which, by definition of substitution, is equivalent to prove that
\[ \langle \sigma_1(t_1) := \sigma_1(t_2), \mu_1 \rangle \approx \langle \varepsilon_1 \sigma_2(t^\text{Ref } T) := T \varepsilon_2 \sigma_2(t^T), \mu_2 \rangle : \text{Unit} \]

We instantiate \( \Gamma; \Sigma \vdash t_1 \approx t^\text{Ref } T : \text{Ref } T \) with \( \sigma_1, \sigma_2 \) and arbitrary \( \mu_1 \) and \( \mu_2 \) such that \( \Sigma \vdash \mu_1 \) and \( \Sigma \vdash \mu_2 \). We know then that \( \langle t_1, \mu_1 \rangle \approx \langle t^\text{Ref } T, \mu_2 \rangle : \text{Ref } T \). Then suppose \( \sigma_1(t_1) \mid \mu_1 \mapsto^* v_{11} \mid \mu_1' \) and \( \sigma_2(t^\text{Ref } T) \mid \mu_2 \mapsto^* v_{21} \mid \mu_2' \) (otherwise the result holds immediately). We know that \( \langle v_{11}, \mu_1' \rangle \approx \langle v_{21}, \mu_2' \rangle : \text{Ref } T \). Similarly we instantiate \( \Gamma; \Sigma \vdash t_2 \approx t^T : B_2 \) with \( \sigma_1, \sigma_2, \mu_1' \) and \( \mu_2' \). Notice \( \mu_1 \subseteq \mu_1' \) (\( \mu_2 \subseteq \mu_2' \) resp.), therefore \( \Sigma \vdash \mu_1' \) (\( \Sigma \vdash \mu_2' \) resp.). Then we know that \( \langle t_2, \mu_1' \rangle \approx \langle t^B_2, \mu_2' \rangle : B_2 \). Then suppose \( \sigma_1(t_2) \mid \mu_1' \mapsto^* v_{12} \mid \mu_1'' \) and \( \sigma_2(t^B_2) \mid \mu_2' \mapsto^* v_{22} \mid \mu_2'' \) (otherwise the result holds immediately). We know that \( \langle v_{21}, \mu_1'' \rangle \approx \langle v_{22}, \mu_2'' \rangle : B_2 \). Let us assume \( v_{21} = u_{21} = o^T \) and \( v_{22} = u_{22} \) (the other cases are analogous modulo one or two trivial steps of reduction). Then \( \sigma_1(t_1) := \sigma_1(t_2) \mid \mu_1 \mapsto o = v_{12} \mid \mu_1'' \mapsto \text{unit} \mid \mu_1''[o \mapsto v_{12}] \) and \( \varepsilon_1 \sigma_2(t^\text{Ref } T) := T \varepsilon_2 \sigma_2(t^T) \mid \mu_2 \mapsto^* (\text{Ref } T)\sigma_2(o^T) := T \langle T \rangle u_{22} \mid \mu_2'' \mapsto \text{unit} \mid \mu_2''[o^T \mapsto \langle T \rangle u_{22} \vdash T] \).

Let \( \mu_1''' = [o \mapsto v_{12}] \) and \( \mu_2''' = [o^T \mapsto \langle T \rangle u_{22} \vdash T] \). By (S::), \( \langle v_{12}, \mu_1''' \rangle \approx \langle \langle T \rangle u_{22} \vdash T, \mu_2''' \rangle : T \), and by Lemma 66, \( \langle v_1, \mu_1''' \rangle \approx \langle \langle T \rangle u_{22} \vdash T, \mu_2''' \rangle : T \), then by (S\mu), \( \mu_1''' \approx \mu_2''' \), and as unit = unit, by (Sb), \( \langle \text{unit}, \mu_1''' \rangle \approx \langle \text{unit}, \mu_2''' \rangle : T \) and the result holds.

\[ \square \]

**Proposition 77** (Compatibility To). If \( o : T \in \Sigma \), then \( \Gamma; \Sigma \vdash o \approx o^T : T \).

**Proof.** Direct by definition of related stores. \( \square \)

**Proposition 78** (Equivalence for fully-annotated terms (dynamics)). For any \( t \in \text{TERM}, \vdash_s t : T \), \( \vdash_t t^\epsilon^T : T \), then \( t \mid \cdot \mapsto^* v \mid \mu \iff t^T \mid \cdot \mapsto^* v' \mid \mu' \), for some \( \mu, \mu' \) such that \( \langle v, \mu \rangle \approx \langle v', \mu' \rangle : T \).

**Proof.** As a special case of the fundamental property 64 and the unfolding of related computations. \( \square \)
A.3.2 Static Gradual Guarantee

Definition 51 (Term precision).

\begin{align*}
\text{(Px)} \quad & x \sqsubseteq x \\
\text{(Pc)} \quad & c \sqsubseteq c \\
\text{(P\oplus)} \quad & t_1 \sqsubseteq t'_1 \quad t_2 \sqsubseteq t'_2 \\
& t_1 \oplus t_2 \sqsubseteq t'_1 \oplus t'_2 \\
\text{(Pif)} \quad & t \sqsubseteq t \quad t_1 \sqsubseteq t'_1 \quad t_2 \sqsubseteq t'_2 \\
& \text{if } t \text{ then } t_1 \text{ else } t_2 \sqsubseteq \text{if } t' \text{ then } t'_1 \text{ else } t'_2 \\
\text{(Pref)} \quad & t \sqsubseteq t' \quad G \sqsubseteq G' \\
& \text{ref}^G_t \sqsubseteq \text{ref}^{G'}_{t'} \\
\text{(Pderref)} \quad & t \sqsubseteq t' \\
& t \sqsubseteq t' \Rightarrow \text{ref}^G_t \sqsubseteq \text{ref}^{G'}_{t'} \\
\text{(P\lambda)} \quad & t \sqsubseteq t' \quad G_1 \sqsubseteq G'_1 \\
& (\lambda x : G_1.t) \sqsubseteq (\lambda x : G'_1.t') \\
\text{(Papp)} \quad & t_1 \sqsubseteq t'_1 \quad t_2 \sqsubseteq t'_2 \\
& t_1 \cdot t_2 \sqsubseteq t'_1 \cdot t'_2 \\
\text{(P:\:)} \quad & t \sqsubseteq t' \\
& G \sqsubseteq G' \\
& t : G \sqsubseteq t' : G' \\
\text{(Passing)} \quad & t_1 \sqsubseteq t'_1 \quad t_2 \sqsubseteq t'_2 \\
& t_1 : t_2 \sqsubseteq t'_1 : t'_2 \\
\text{(Po)} \quad & o \sqsubseteq o \\
\end{align*}

Definition 52 (Type environment precision).

\[ \Gamma \sqsubseteq \Gamma' \quad G \sqsubseteq G' \]
\[ \Gamma, x : G \sqsubseteq \Gamma', x : G' \]

Definition 53 (Store typing precision).

\[ \Sigma \sqsubseteq \Sigma' \quad G \sqsubseteq G' \]
\[ \Sigma, o : G \sqsubseteq \Sigma', o : G' \]

Lemma 79. If \( \Gamma; \Sigma \vdash t : G \), \( \Gamma \sqsubseteq \Gamma' \) and \( \Sigma \sqsubseteq \Sigma' \), then \( \Gamma'; \Sigma' \vdash t : G' \) for some \( G \sqsubseteq G' \).

Proof. Simple induction on typing derivations. \( \square \)

Lemma 80. If \( G_1 \sim G_2 \) and \( G_1 \sqsubseteq G'_1 \) and \( G_2 \sqsubseteq G'_2 \) then \( G'_1 \sim G'_2 \).

Proof. By definition of \( \sim \), there exists \( \langle T_1, T_2 \rangle \in \langle \widehat{T}_1, \widehat{T}_2 \rangle \in \gamma^2(G_1, G_2) \) such that \( T_1 = T_2 \). \( G_1 \sqsubseteq G'_1 \) and \( G_2 \sqsubseteq G'_2 \) mean that \( \gamma(G_1) \subseteq \gamma(G'_1) \) and \( \gamma(G_2) \subseteq \gamma(G'_2) \), therefore \( \langle T_1, T_2 \rangle \in \langle \widehat{T}_1, \widehat{T}_2 \rangle \in \gamma^2(G'_1, G'_2) \). \( \square \)

Proposition 13 (Static gradual guarantee). If \( \vdash t_1 : G_1 \) and \( t_1 \sqsubseteq t_2 \), then \( \vdash t_2 : G_2 \), for some \( G_2 \) such that \( G_1 \sqsubseteq G_2 \).

Proof. We prove the property on opens terms instead of closed terms: If \( \Gamma; \Sigma \vdash t_1 : G_1 \) and \( t_1 \sqsubseteq t_2 \) then \( \Gamma; \Sigma \vdash t_2 : G_2 \) and \( G_1 \sqsubseteq G_2 \).

The proof proceed by induction on the typing derivation.

Case (Gx, Gb). Trivial by definition of \( \sqsubseteq \) using (Px), (Pb) respectively.

Case (G\lambda). Then \( t_1 = (\lambda x : G_1.t) \) and \( G_1 = G'_1 \rightarrow G'_2 \). By (G\lambda) we know that:

\[ (G\lambda) \quad \vdash \Gamma, x : G'_1; \Sigma \vdash t : G'_2 \]
\[ \Gamma \vdash (\lambda x : G'_1.t) : G'_1 \rightarrow G'_2 \]

(A.1)
Consider $t_2$ such that $t_1 \sqsubseteq t_2$. By definition of term precision $t_2$ must have the form $t_2 = (\lambda x : G_1'' t')'$ and therefore

$$
(G\lambda) \frac{t \sqsubseteq t' \quad G_1' \sqsubseteq G_1''}{(\lambda x : G_1', t) \sqsubseteq (\lambda x : G_1'', t')}
$$

Using induction hypotheses on the premise of $[D.1]$ $\Gamma; x : G_1'; \Sigma \vdash t' : G''_2$ with $G_2' \sqsubseteq G_2''$. By Lemma 267, $\Gamma; x : G_1'' \vdash t' : G''_2$ where $G_2'' \sqsubseteq G_2''$. Then we can use rule $(G\lambda)$ to derive:

$$
(G\lambda) \frac{\Gamma; x : G_1'; \Sigma' \vdash t' : G_2''}{\Gamma; (\lambda x : G_1', t') : G_1' \to G_2''}
$$

Where $G_2' \sqsubseteq G_2''$. Using the premise of $[D.2]$ and the definition of type precision we can infer that

$$
G_1' \to G_2' \sqsubseteq G_1' \to G_2''
$$

and the result holds.

Case $(G\oplus)$. Then $t_1 = t_1' \oplus t_2'$ and $G_1 = \text{Int}$. By $(G\oplus)$ we know that:

$$
(T\oplus) \frac{\Gamma; \Sigma \vdash t_1 : G_1 \\ \Gamma; \Sigma \vdash t_2 : G_2 \quad G_1 \sim B_1 \quad G_2 \sim B_2}{\Gamma; \Sigma \vdash t_1 \oplus t_2 : B_3}
$$

Consider $t_2$ such that $t_1 \sqsubseteq t_2$. By definition of term precision $t_2$ must have the form $t_2 = t_1'' \oplus t_2''$ and therefore

$$
(P\oplus) \frac{t_1' \sqsubseteq t_1'' \quad t_2' \sqsubseteq t_2''}{t_1' \oplus t_1'' \sqsubseteq t_1'' \oplus t_2''}
$$

Using induction hypotheses on the premises of $[D.9]$ $\Gamma; \Sigma \vdash t_1' : G_1'$ and $\Gamma; \Sigma \vdash t_2' : G_2'$, where $G_1' \sqsubseteq G_1''$ and $G_2' \sqsubseteq G_2''$. By Lemma 268, $G_1' \sim B_1$ and $G_2' \sim B_2$. Therefore we can use rule $(G\oplus)$ to derive:

$$
(T\oplus) \frac{\Gamma'; \Sigma' \vdash t_1' : G_1' \\ \Gamma'; \Sigma' \vdash t_2' : G_2' \quad G_1' \sim B_1 \quad G_2' \sim B_2}{\Gamma'; \Sigma' \vdash t_1' \oplus t_2' : B_3}
$$

and the result holds.

Case $(\text{Gapp})$. Then $t_1 = t_1' t_2'$ and $G_1 = G_{12}$. By $(\text{Gapp})$ we know that:

$$
(G\text{app}) \frac{\Gamma; \Sigma \vdash t_1' : G_1' \quad \Gamma; \Sigma \vdash t_2' : G_2' \quad G_2' \sim \text{dom}(G_1')}{\Gamma; \Sigma \vdash t_1' t_2' : \text{cod}(G_1')}
$$

Consider $t_2$ such that $t_1 \sqsubseteq t_2$. By definition of term precision $t_2$ must have the form $t_2 = t_1'' t_2''$ and therefore

$$
(P\text{app}) \frac{t_1' \sqsubseteq t_1'' \quad t_2' \sqsubseteq t_2''}{t_1' t_1'' \sqsubseteq t_1'' t_2''}
$$

Using induction hypotheses on the premises of $[D.11]$ $\Gamma; \Sigma \vdash t_1' : G_1''$ and $\Gamma; \Sigma \vdash t_2' : G_2''$, where $G_1' \sqsubseteq G_1''$ and $G_2' \sqsubseteq G_2''$. By definition precision (Def. 2) and the definition of $\text{dom}$, $\text{dom}(G_1') \sqsubseteq \text{dom}(G_1'')$ and, therefore by Lemma 268, $G_2' \sim \text{dom}(G_1'')$. Also, by the previous argument $\text{cod}(G_1') \sqsubseteq \text{cod}(G_1'')$ Then we can use rule $(\text{Gapp})$ to derive:

$$
(G\text{app}) \frac{\Gamma'; \Sigma' \vdash t_1' : G_1'' \quad \Gamma'; \Sigma' \vdash t_2' : G_2'' \quad G_2' \sim \text{dom}(G_1'')}{\Gamma'; \Sigma' \vdash t_1' t_2' : \text{cod}(G_1'')}
$$

and the result holds.
Case (Gif). Then \( t_1 = \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 \) and \( G_1 = (G'_2 \cap G'_3) \). By (Gif) we know that:

\[
\frac{\Gamma; \Sigma \vdash t'_1 : G'_1, \Gamma; \Sigma \vdash t'_2 : G'_2, \Gamma; \Sigma \vdash t'_3 : G'_3}{\Gamma; \Sigma \vdash \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 : (G'_2 \cap G'_3)} \tag{A.7}
\]

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{if } t''_1 \text{ then } t''_2 \text{ else } t''_3 \) and therefore

\[
\frac{t'_1 \subseteq t''_1 \quad t'_1 \subseteq t''_1 \quad t'_2 \subseteq t''_2 \quad t'_3 \subseteq t''_3}{\text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 \subseteq \text{if } t''_1 \text{ then } t''_2 \text{ else } t''_3} \tag{A.8}
\]

Then we can use induction hypotheses on the premises of C.7 and derive:

\[
\frac{\Gamma'; \Sigma' \vdash t'_1 : G''_1, \Gamma'; \Sigma' \vdash t'_2 : G''_2, \Gamma'; \Sigma' \vdash t'_3 : G''_3}{\Gamma'; \Sigma' \vdash \text{if } t'_1 \text{ then } t'_2 \text{ else } t'_3 : (G''_2 \cap G''_3)} \tag{A.9}
\]

Where \( G'_1 \subseteq G''_1 \) and \( G'_2 \subseteq G''_2 \). Using the definition of type precision (Def. 2) we can infer that

\[
(G'_1 \cap G'_2) \subseteq (G''_1 \cap G''_2)
\]

and the result holds.

Case (G::). Then \( t_1 = t :: G_1 \). By (G::) we know that:

\[
\frac{\Gamma \vdash t : G'_1, G'_1 \sim G_1}{\Gamma \vdash t :: G'_1 : G_1} \tag{A.10}
\]

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t' :: G_2 \) and therefore

\[
\frac{t \subseteq t' \quad G_1 \subseteq G_2}{t :: G_1 \subseteq t' :: G_2} \tag{A.11}
\]

Using induction hypotheses on the premises of D.7 \( \Gamma \vdash t' : G'_2 \) where \( G'_1 \subseteq G'_2 \). We can use rule (G::) and Lemma 268 to derive:

\[
\frac{\Gamma \vdash t' : G'_2, G'_2 \sim G_2}{\Gamma \vdash t' :: G_2 : G_2} \tag{A.12}
\]

Where \( G_1 \subseteq G_2 \) and the result holds.

Case (Gref). Then \( t_1 = \text{ref}^G t'_1 \). By (Gref) we know that:

\[
\frac{\Gamma; \Sigma \vdash t'_1 : G'_1, G'_1 \sim G}{\Gamma; \Sigma \vdash \text{ref}^G t'_1 : \text{Ref}^G G} \tag{A.13}
\]

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{ref}^G t'_2 \) and therefore

\[
\frac{t'_1 \subseteq t'_2 \quad G \subseteq G'}{\text{ref}^G t'_2 \subseteq \text{ref}^G t'_2} \tag{A.14}
\]

Using induction hypotheses on the premise of B.13 \( \Gamma; \Sigma \vdash t'_1 : G'_1 \), where \( G'_1 \subseteq G'_2 \). By definition of precision on types \( \text{Ref} G \subseteq \text{Ref} G' \). Then we can use rule (Gref) to derive:

\[
\frac{\Gamma'; \Sigma' \vdash t'_2 : G''_2, G''_2 \sim G'}{\Gamma' \vdash \text{ref}^G t'_2 : \text{Ref} G'} \tag{A.15}
\]

and the result holds.
Case (Gderef). Then \( t_1 = !t'_1 \). By (Gderef) we know that:

\[
\frac{\Gamma; \Sigma \vdash t'_1 : G}{\Gamma; \Sigma \vdash !t'_1 : \text{tref}(G)} \tag{A.13}
\]

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = !t'_2 \) and therefore

\[
\frac{t'_1 \sqsubseteq t'_2}{!t'_1 \sqsubseteq !t'_2} \tag{A.14}
\]

Using induction hypotheses on the premise of \( \text{B.15} \), \( \Gamma; \Sigma \vdash t'_1 : G'_1 \). By definition of \( \text{tref} \), \( \text{tref} G \sqsubseteq \text{tref} G' \). Then we can use rule (Gderef) to derive:

\[
\frac{\Gamma'; \Sigma' \vdash t'_2 : G'}{\Gamma'; \Sigma' \vdash !t'_2 : \text{tref}(G')} \tag{A.15}
\]

and the result holds.

Case (Gassign). Then \( t_1 = t'_1 := t'_2 \) and \( G_1 = G_{12} \). By (Gassign) we know that:

\[
\frac{\Gamma; \Sigma \vdash t'_1 : G'_1 \quad \Gamma; \Sigma \vdash t'_2 : G'_2 \quad G'_2 \sim \text{tref}(G'_1)}{\Gamma; \Sigma \vdash t'_1 := t'_2 : \text{cod}(G'_1)} \tag{A.16}
\]

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 := t''_2 \) and therefore

\[
\frac{t'_1 \sqsubseteq t''_1 \quad t'_2 \sqsubseteq t''_2}{t'_1 := t'_2 \sqsubseteq t''_1 := t''_2} \tag{A.17}
\]

Using induction hypotheses on the premises of \( \text{A.15} \), \( \Gamma; \Sigma \vdash t''_1 : G''_1 \) and \( \Gamma; \Sigma \vdash t''_2 : G''_2 \), where \( G'_1 \sqsubseteq G''_1 \) and \( G'_2 \sqsubseteq G''_2 \). By definition precision (Def. 2) and the definition of \( \text{tref} \), \( \text{tref}(G'_1) \sqsubseteq \text{tref}(G''_1) \) and, therefore by Lemma \( \text{268} \), \( G''_2 \sim \text{tref}(G''_1) \). Then we can use rule (Gassign) to derive:

\[
\frac{\Gamma'; \Sigma' \vdash t''_1 : G''_1 \quad \Gamma'; \Sigma' \vdash t''_2 : G''_2 \quad G''_2 \sim \text{tref}(G''_1)}{\Gamma'; \Sigma' \vdash t'_1 := t'_2 : \text{Unit}} \tag{A.18}
\]

and the result holds.

\[\square\]

### A.3.3 Dynamic Gradual Guarantee

In this section we present the proof the Dynamic Gradual Guarantee for \( \lambda^{\text{REF}}_\epsilon \).

**Definition 54** (Intrinsic term precision). Let \( \Omega \in \mathcal{P}(\mathbb{V}[\epsilon] \times \mathbb{V}[\epsilon]) \cup \mathcal{P}(\mathbb{L}[\epsilon] \times \mathbb{L}[\epsilon]) \) be defined as \( \Omega := \{ \epsilon^{G_{i1}} \subseteq \epsilon^{G_{i2}}, \epsilon^{O_{ij1}} \subseteq \epsilon^{O_{ij2}} \} \). We define an ordering relation (\( \cdot \sqsubseteq \cdot \)) \( \in (\mathcal{P}(\mathbb{V}[\epsilon] \times \mathbb{V}[\epsilon]) \cup \mathcal{P}(\mathbb{L}[\epsilon] \times \mathbb{L}[\epsilon])) \times \mathbb{T}[\epsilon] \times \mathbb{T}[\epsilon] \) shown in Figure \( \text{C.10} \).
Figure A.2: Intrinsic term precision
Definition 55 (Well Formedness of Ω). We say that Ω is well formed iff \( \forall \{ x^{G_1} \sqsubseteq x^{G_2} \} \in \Omega, G_{i1} \sqsubseteq G_{i2} \) is systematically a premise (either directly or transitively).

Before proving the gradual guarantee, we first establish some auxiliary properties of precision. For the following propositions, we assume Well Formedness of Ω (Definition 55).

Proposition 81. If \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \) for some \( \Omega \in \mathcal{P}(V[\ast] \times V[\ast]) \cup \mathcal{P}(\text{Loc}_s \times \text{Loc}_s) \), then \( G_1 \sqsubseteq G_2 \).

Proof. Straightforward induction on \( \Omega \vdash t^{G_1} \sqsubseteq t^{G_2} \), since the corresponding precision on types is systematically a premise (either directly or transitively).

Proposition 82 (Substitution preserves precision). If \( \Omega \cup \{ x^{G_3} \sqsubseteq x^{G_4} \} \vdash t^{G_1} \sqsubseteq t^{G_2} \) and \( \Omega \vdash t^{G_3} \sqsubseteq t^{G_4} \), then \( \Omega \vdash \frac{t^{G_3}/x^{G_3}}{t^{G_4}/x^{G_4}} t^{G_1} \sqsubseteq \frac{t^{G_4}/x^{G_4}}{} t^{G_2} \).

Proof. By induction on the derivation of \( t^{G_1} \sqsubseteq t^{G_2} \), and case analysis of the last rule used in the derivation. All cases follow either trivially (no premises) or by the induction hypotheses.

Proposition 83 (Monotonicity of evidence). If \( \varepsilon_1 \sqsubseteq \varepsilon_2, \varepsilon_3 \sqsubseteq \varepsilon_4 \), and \( \varepsilon_1 \circ \varepsilon_3 \sqsubseteq \varepsilon_2 \circ \varepsilon_4 \) is defined, then \( \varepsilon_1 \circ \varepsilon_3 \sqsubseteq \varepsilon_2 \circ \varepsilon_4 \).

Proof. By definition of consistent transitivity for \( = \) and the definition of precision.

Proposition 84. If \( G_{11} \sqsubseteq G_{12} \) and \( G_{21} \sqsubseteq G_{22} \), then \( G_{11} \cap G_{21} \sqsubseteq G_{12} \cap G_{22} \).

Proof. By induction on the type derivation of the types and meet.

Proposition 85 (Dynamic guarantee for \( \longrightarrow \)). Suppose \( \Omega \vdash t^{G_1}_1 \sqsubseteq t^{G_2}_1 \) and \( \mu_1 \sqsubseteq \mu_2 \). If \( t^{G_1}_1 \mid \mu_1 \longrightarrow t^{G_1}_2 \mid \mu'_1 \) then \( t^{G_2}_1 \mid \mu_2 \longrightarrow t^{G_2}_2 \mid \mu'_2 \), where \( \Omega' \vdash t^{G_1}_2 \sqsubseteq t^{G_2}_2, \mu'_1 \sqsubseteq \mu'_2 \) for some \( \Omega' \supseteq \Omega \).

Proof. By induction on reduction \( t^{G_1}_1 \mid \mu_1 \longrightarrow t^{G_1}_2 \mid \mu'_1 \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

Case (r1). We know that \( t^{G_1}_1 = (\varepsilon_1 (c_1) \circ \varepsilon_2 (c_2)) \) then by \( (\sqsubseteq \circ \circ) \) \( t^{G_2}_1 = (\varepsilon_1 (c_1) \circ \varepsilon_2 (c_2)) \) for some \( \varepsilon_2 \sqsubseteq \varepsilon_2 \) such that \( \varepsilon_1 \sqsubseteq \varepsilon_2 \) and \( \varepsilon_1 \sqsubseteq \varepsilon_2 \).

If \( t^{G_1}_1 \mid \mu_1 \longrightarrow c_3 \mid \mu_1 \) where \( c_3 = (c_1 \mid \varepsilon) \mid c_2 \), then \( t^{G_1}_2 \mid \mu_2 \longrightarrow c_3' \mid \mu_2 \) where \( c_3' = (c_1 \mid \varepsilon) \mid c_2 \).

Case (r2). We know that \( t^{G_1}_1 = \varepsilon_1 (\lambda x^{G_1}_1, t^{G_1}_1) \circ \varepsilon_2 \circ \varepsilon_3 u \) then by \( (\sqsubseteq \circ \circ \circ) \) \( t^{G_2}_1 \) must have the form \( \varepsilon_2 (\varepsilon_1 (\lambda x^{G_2}, t^{G_2}) \circ \varepsilon_3 u) \) for some \( \varepsilon_2, x^{G_2}, t^{G_2}, G_3, G_4, \varepsilon_3, u \).

Let us pose \( \varepsilon_1 = \varepsilon_2 \circ \varepsilon_3 = \text{idom}(\varepsilon_1) \). Then \( t^{G_1}_1 \mid \mu_1 \longrightarrow \text{icod}(\varepsilon_1) t'_1 :: G_2 \mid \mu_1 \) with \( t'_1 = \frac{[(\varepsilon_1 u_1 :: G_11)/x^{G_11}]}{t^{G_1}_1} \).

Also, by \( \text{idom}(\varepsilon_2) \) \( \varepsilon_2 = \varepsilon_2 \circ \varepsilon_3 = \text{idom}(\varepsilon_2) \) is defined. Then \( t^{G_2}_1 \mid \mu_2 \longrightarrow \text{icod}(\varepsilon_2) t'_2 :: G_4 \mid \mu_2 \) with \( t'_2 = \frac{[(\varepsilon_2 u_2 :: G_21)/x^{G_21}]}{t^{G_2}_1} \).

198
As $\Omega \vdash t_{G_1}^1 \subseteq t_{G_2}^1$, then $u_1 \subseteq u_2$, $\varepsilon_{12} \subseteq \varepsilon_{22}$ and $\text{idom}(\varepsilon_{11}) \subseteq \text{idom}(\varepsilon_{21})$ as well, then by Prop\,253 $\varepsilon_{1} \preceq \varepsilon_{2}$. Then $\varepsilon_{1} u_1 :: G_{11} \subseteq \varepsilon_{2} u_2 :: G_{21}$ by $(\subseteq \cdot)$. We also know by $(\subseteq \cdot \text{APP})$ and $(\subseteq \cdot \lambda)$ that $\Omega \cup \{x_{G_21} \subseteq x_{G_21}\} \vdash t_{G_{12}}^2 \subseteq t_{G_{22}}^2$. By Substitution preserves precision (Prop\,252) $t_1^1 \subseteq t_2^1$, therefore $\text{icod}(\varepsilon_{11}) t_1^1 :: G_{2} \supseteq \text{icod}(\varepsilon_{21}) t_2^1 :: G_{4}$ by $(\subseteq \cdot)$. Then $t_{G_1}^2 \subseteq t_{G_2}^2$.

**Case** ($r3 - \text{true}$). $t_{G_1}^1 = \text{ref}^{G_1} \varepsilon_{1} u_1$ where $G_1 = \text{Ref} \ G_1'$, then by $(\subseteq \cdot \text{REF})$ $t_{G_1}^1$ must have the form $t_{G_2}^1 = \text{ref}^{G_2} \varepsilon_{2} u_2$ for some $\varepsilon_{2}, u_2, G_2'$ such that $\varepsilon_{1} \subseteq \varepsilon_{2}, u_1 \subseteq u_2$, and $G_1' \subseteq G_2'$. Then $t_{G_1}^1 \mid \mu_1 \rightarrow o^{G_1} \mid \mu_1 [o^{G_1} \mapsto \varepsilon_{1} u_1 :: G_1']$. Also, $t_{G_2}^1 \mid \mu_2 \rightarrow o^{G_2} \mid \mu_2 [o^{G_2} \mapsto \varepsilon_{2} u_2 :: G_2']$. Then by $(\subseteq \cdot)$, $\varepsilon_{1} u_1 :: G_1' \subseteq \varepsilon_{2} u_2 :: G_2'$, and then $\mu_1 [o^{G_1} \mapsto \varepsilon_{1} u_1 :: G_1'] \subseteq \mu_2 [o^{G_2} \mapsto \varepsilon_{2} u_2 :: G_2']$. Also by $(\subseteq \cdot)$, as $G_1' \subseteq G_2'$, $o^{G_1} \subseteq o^{G_2}$ and the result holds.

**Case** ($r3 - \text{false}$). Same as case $\rightarrow$-true, using the fact that $\varepsilon_{13} \subseteq \varepsilon_{23}$ and $t_{G_1}^3 \subseteq t_{G_2}^3$.

**Case** ($r4$). We know that $t_{G_1}^1 = \text{ref}^{G_1} \varepsilon_{1} u_1$ where $G_1 = \text{Ref} \ G_1'$, then by $(\subseteq \cdot \text{REF})$ $t_{G_1}^1$ must have the form $t_{G_2}^1 = \text{ref}^{G_2} \varepsilon_{2} u_2$ for some $\varepsilon_{2}, u_2, G_2'$ such that $\varepsilon_{1} \subseteq \varepsilon_{2}, u_1 \subseteq u_2$, and $G_1' \subseteq G_2'$. Then $t_{G_1}^1 \mid \mu_1 \rightarrow o^{G_1} \mid \mu_1 [o^{G_1} \mapsto \varepsilon_{1} u_1 :: G_1']$. Also, $t_{G_2}^1 \mid \mu_2 \rightarrow o^{G_2} \mid \mu_2 [o^{G_2} \mapsto \varepsilon_{2} u_2 :: G_2']$. Then by $(\subseteq \cdot)$, $\varepsilon_{1} u_1 :: G_1' \subseteq \varepsilon_{2} u_2 :: G_2'$, and then $\mu_1 [o^{G_1} \mapsto \varepsilon_{1} u_1 :: G_1'] \subseteq \mu_2 [o^{G_2} \mapsto \varepsilon_{2} u_2 :: G_2']$. Also by $(\subseteq \cdot)$, as $G_1' \subseteq G_2'$, $o^{G_1} \subseteq o^{G_2}$ and the result holds.

**Case** ($r5$). We know that $t_{G_1}^1 = !^{G_1} \varepsilon_{1} o^{G_1}$, then by $(\subseteq \cdot)$ $t_{G_2}^1$ must have the form $t_{G_2}^1 = !^{G_2} \varepsilon_{2} o^{G_2}$ for some $\varepsilon_{2}, o^{G_2}, G_2'$ such that $\varepsilon_{1} \subseteq \varepsilon_{2}, o^{G_1} \subseteq o^{G_2}$, and $G_1' \subseteq G_2'$. Then $t_{G_1}^1 \mid \mu_1 \rightarrow \varepsilon_{1} \mu_1 (o^{G_1}) :: G_1 \mid \mu_1$. Also, $t_{G_2}^1 \mid \mu_2 \rightarrow \varepsilon_{2} \mu_2 (o^{G_2}) :: G_2 \mid \mu_2$. As $\mu_1 \preceq \mu_2$, then $\varepsilon_{1} (o^{G_1}) \subseteq \mu_2 (o^{G_2})$. Then by $(\subseteq \cdot)$, $\varepsilon_{1} \mu_1 (o^{G_1}) :: G_1' \subseteq \varepsilon_{2} \mu_2 (o^{G_2}) :: G_2'$, and the result holds.

**Case** ($r6$). We know that $t_{G_1}^1 = \varepsilon_{11} o^{G_{11}} := \varepsilon_{12} u_1$ where $G_1 = \text{Unit}$, then by $(\subseteq \cdot)$ $t_{G_2}^1$ must have the form $t_{G_2}^1 = \varepsilon_{21} o^{G_{21}} := \varepsilon_{22} u_2$ for some $\varepsilon_{21}, \varepsilon_{22}, u_2, G_{21}, G_{22}$ such that $\varepsilon_{11} \subseteq \varepsilon_{21}, \varepsilon_{12} \subseteq \varepsilon_{22}, u_1 \subseteq u_2, G_{11} \subseteq G_{21}, G_{12} \subseteq G_{22}$. Let us pose $\varepsilon_{1} = \varepsilon_{12} o^= \text{iref} (\varepsilon_{11})$. Then $t_{G_1}^1 \mid \mu_1 \rightarrow \text{unit} \mid \mu_1 [o^{G_{11}} \mapsto \varepsilon_{1} u_1 :: G_{11}]$. By inspection of evidence and inversion lemma, as $\varepsilon_{12} \subseteq \varepsilon_{21}$ then $\text{iref} (\varepsilon_{12}) \subseteq \text{iref} (\varepsilon_{21})$. Also, by 253 $\varepsilon_{2} = \varepsilon_{22} o^= \text{iref} (\varepsilon_{21})$ is defined and $\varepsilon_{1} \preceq \varepsilon_{2}$. Then, $t_{G_1}^1 \mid \mu_2 \rightarrow \text{unit} \mid \mu_2 [o^{G_{21}} \mapsto \varepsilon_{2} u_2 :: G_{21}]$. Then by $(\subseteq \cdot)$, $\varepsilon_{1} u_1 :: G_{11} \subseteq \varepsilon_{2} u_2 :: G_{21}$, and then $\mu_1 [o^{G_{21}} \mapsto \varepsilon_{1} u_1 :: G_{11}] \subseteq \mu_2 [o^{G_{21}} \mapsto \varepsilon_{2} u_2 :: G_{21}]$ and the result holds.

\[
\square
\]

**Proposition 256 (Dynamic gradual guarantee).** Suppose $t_{G_1}^1 \subseteq t_{G_2}^2$ and $\mu_1 \preceq \mu_2$. Then if $t_{G_1}^1 \mid \mu_1 \rightarrow t_{G_1}^2 \mid \mu_1'$ then $t_{G_1}^2 \mid \mu_2 \rightarrow t_{G_2}^2 \mid \mu_2'$ where $t_{G_1}^2 \subseteq t_{G_2}^2$ and $\mu_1' \preceq \mu_2'$.

**Proof.** We prove the following property instead: Suppose $\Omega \vdash t_{G_1}^1 \subseteq t_{G_2}^2$ and $\mu_1 \preceq \mu_2$. If $t_{G_1}^1 \mid \mu_1 \rightarrow t_{G_2}^1 \mid \mu_1'$ then $t_{G_1}^2 \mid \mu_2 \rightarrow t_{G_2}^2 \mid \mu_2'$ where $\Omega' \vdash t_{G_1}^2 \subseteq t_{G_2}^2$, and $\mu_1' \preceq \mu_2'$ for some $\Omega' \supseteq \Omega$. 199
By induction on reduction \( t_{G_1}^{t_1} | \mu_1 \mapsto t_{G_2}^{t_2} | \mu'_1 \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case** \((t_{G_1}^{t_1} | \mu_1 \mapsto t_{G_2}^{t_2} | \mu'_1)\). By dynamic guarantee of \( \rightarrow \) (Prop. 255), \( t_{G_2}^{t_2} | \mu_2 \mapsto t_{G_2}^{t_2} | \mu'_2 \) where \( \Omega \vdash t_{G_2}^{t_2} \subseteq t_{G_2}^{t_2}, \mu_1 \subseteq \mu_2 \) for some \( \Omega' \supseteq \Omega \). And the result holds immediately.

**Case** \((\varepsilon_{11}^{22} t_{G_1}^{t_1} \vdash_{G_1} \varepsilon_{12}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} | \mu_1)\). By inspection of \( (\equiv APP) \) \( t_{G_2} = t_{G_2}^{t_2} = \varepsilon_{21} t_{G_1}^{t_1} \vdash_{G_1} \varepsilon_{12}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( \varepsilon_{11} \subseteq \varepsilon_{21}, \varepsilon_{12} \subseteq \varepsilon_{22} G_1 \subseteq G_2, t_{G_1}^{t_1} \subseteq t_{G_2}^{t_2} \), and \( \varepsilon_{11} \subseteq \varepsilon_{12} \subseteq \varepsilon_{21} \vdash_{G_2} t_{G_2}^{t_2} \). By induction hypothesis on \( \varepsilon_{11} t_{G_1}^{t_1} :: G_1 \vdash_{G_1} \mu_1 \), then \( \varepsilon_{11} t_{G_1}^{t_1} :: G_1 \vdash_{G_1} \mu_1 \), then \( \varepsilon_{21} t_{G_1}^{t_1} :: G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and the result holds.

**Case** \((\varepsilon_{11}^{22} u_{11}^{11} \vdash_{G_1} \varepsilon_{22}^{11} G_2 \vdash_{G_2} t_{G_2}^{t_2} | \mu_1)\). By inspection of \( (\equiv APP) \) \( t_{G_2} = t_{G_2}^{t_2} = \varepsilon_{21} u_{11}^{11} \vdash_{G_1} \varepsilon_{22}^{11} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( \varepsilon_{11} \subseteq \varepsilon_{21}, \varepsilon_{12} \subseteq \varepsilon_{22} G_1 \subseteq G_2, u_1 \subseteq u_2, \) and \( t_{G_1}^{t_1} \subseteq t_{G_2}^{t_2} \). By induction hypothesis on \( \varepsilon_{11} u_{11}^{11} :: G_1 \vdash_{G_1} \mu_1 \), then \( \varepsilon_{11} u_{11}^{11} :: G_1 \vdash_{G_1} \mu_1 \), then \( \varepsilon_{21} u_{11}^{11} :: G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and the result holds.

**Case** \((\ref{G_2}^{G_1} \ref{G_2}^{G_1} \vdash_{G_1} \varepsilon_{11}^{22} t_{G_1}^{t_1} | \mu_1 \mapsto \varepsilon_{21}^{11} t_{G_1}^{t_1} | \mu'_1)\). By inspection of \( (\equiv REF) \) \( t_{G_2} = t_{G_2}^{t_2} = \varepsilon_{21}^{11} t_{G_1}^{t_1} | \mu'_1 \). By induction hypothesis on \( \varepsilon_{11}^{22} t_{G_1}^{t_1} :: G_1 \vdash_{G_1} \mu_1 \), then \( \varepsilon_{11}^{22} t_{G_1}^{t_1} :: G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( t_{G_1}^{t_1} \subseteq t_{G_2}^{t_2} \), \( \varepsilon_{11} \subseteq \varepsilon_{21} \subseteq \varepsilon_{22} G_1 \subseteq G_2, \) and \( \mu'_1 \subseteq \mu'_2 \). Then by \( (\equiv REF) \) and Lemma 105 \( \varepsilon_{11}^{22} t_{G_1}^{t_1} \subseteq \varepsilon_{21}^{11} t_{G_1}^{t_1} \subseteq \varepsilon_{22}^{11} t_{G_1}^{t_1} t_{G_2}^{t_2} \) and the result holds.

**Case** \((t_{G_2} = \varepsilon_{21}^{11} t_{G_1}^{t_1} \vdash_{G_1} \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} | \mu_1)\). By induction hypothesis on \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), then \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( \varepsilon_{11} \subseteq \varepsilon_{21} \subseteq \varepsilon_{22} G_1 \subseteq G_2, \) and \( \mu'_1 \subseteq \mu'_2 \). Then by \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and Lemma 105 \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \subseteq \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and the result holds.

**Case** \((t_{G_2} = \varepsilon_{21}^{11} t_{G_1}^{t_1} \vdash_{G_1} \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} | \mu_1)\). By induction hypothesis on \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), then \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( \varepsilon_{11} \subseteq \varepsilon_{21} \subseteq \varepsilon_{22} G_1 \subseteq G_2, \) and \( \mu'_1 \subseteq \mu'_2 \). Then by \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and Lemma 105 \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \subseteq \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and the result holds.

**Case** \((t_{G_2} = \varepsilon_{21}^{11} t_{G_1}^{t_1} \vdash_{G_1} \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} | \mu_1)\). By induction hypothesis on \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), then \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \), where \( \varepsilon_{11} \subseteq \varepsilon_{21} \subseteq \varepsilon_{22} G_1 \subseteq G_2, \) and \( \mu'_1 \subseteq \mu'_2 \). Then by \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and Lemma 105 \( \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \subseteq \varepsilon_{11}^{22} G_2 \vdash_{G_2} t_{G_2}^{t_2} \) and the result holds.
We prove both sides of the proposition by induction on the premise, i.e. by induction on $G_1 \cap G_2$.

**Case ($\varepsilon_{12} (\varepsilon_{11} u_1 :: G_{11}) :: G_{12} | \mu_1 \mapsto \varepsilon_{11} u_1 :: G_{12} | \mu_1$).** By inspection of ($\sqsubseteq$)

$t^{G_{12}} = \varepsilon_{22} (\varepsilon_{21} u_2 :: G_{21}) :: G_{22},$ where $\varepsilon_{11} \sqsubseteq \varepsilon_{21}, \varepsilon_{12} \sqsubseteq \varepsilon_{22}, G_{11} \sqsubseteq G_{21}, G_{12} \sqsubseteq G_{22}, u_1 \sqsubseteq u_2$. If $\varepsilon_{11}' = \varepsilon_{11} \circ \varepsilon_{12}$ is defined, then by Prop 253 $\varepsilon_{21}' = \varepsilon_{21} \circ \varepsilon_{22}$ is also defined, and furthermore $\varepsilon_{11}' \sqsubseteq \varepsilon_{21}'$. Then $\varepsilon_{22} (\varepsilon_{21} u_2 :: G_{21}) :: G_{22} | \mu_2 \mapsto \varepsilon_{21} u_2 :: G_{22} | \mu_2$, and the result holds directly by ($\sqsubseteq_{\text{REF}}$).

\[ \square \]

### A.4 Relation to the coercion calculus

**Lemma 86.** $G_1 \cap G_2 = G \iff G_2 \cap G_1 = G$

**Proof.** We prove both sides of the proposition by induction on the premise, i.e. by induction on $G_1 \cap G_2$ for the $\Rightarrow$ case, and induction on $G_2 \cap G_1$ for the $\Leftarrow$ (both cases as identical).

\[ \square \]

**Lemma 87.** $\varepsilon_1 \circ^= \varepsilon_2 = \varepsilon \iff \varepsilon_2 \circ^= \varepsilon_1 = \varepsilon$

**Proof.** Direct by Prop 86

\[ \square \]

**Lemma 88.** $c = \langle \varepsilon \vdash G_1 \sim G_2 \rangle$, then $\text{nm } c$

**Proof.** Straightforward induction on judgment $\varepsilon \vdash G_1 \sim G_2$.

\[ \square \]

**Proposition 89.** Let $c_1 = \langle \varepsilon_1 \vdash G_1 \sim G_2 \rangle$ and $c_2 = \langle \varepsilon_2 \vdash G_2 \sim G_3 \rangle$. Then

1. $(c_1; c_2) \mapsto^* \text{Fail} \iff \varepsilon_1 \circ^= \varepsilon_2$ is undefined

2. $(c_1; c_2) \mapsto^* \text{c} \land \text{nm c} \iff \varepsilon_1 \circ^= \varepsilon_2$ is defined. Furthermore $c = \langle (\varepsilon_1 \circ^= \varepsilon_2) \vdash G_1 \sim G_3 \rangle$.

**Proof.** Direct by induction on types $G_1, G_2$ and $G_3$, inspection on the coercion reduction rules and transitivity of evidence. We only present interesting cases.

**Case ($G_1 = R_1, G_2 = ? , G_3 = R_2$).** Then $\varepsilon_1 = \langle R_1 \rangle, \varepsilon_2 = \langle R_2 \rangle, c_1 = R_1!$, and $c_2 = R_2?$. If $R_1 \neq R_2$ then $R_1 \cap R_2$ is not defined and then $\varepsilon_1 \circ^= \varepsilon_2$ is not defined. Also $R_1!; R_2? \mapsto \text{Fail}$ and the result holds.

If $R_1 = R_2 = R$ then $R_1 \cap R_2 = R$ and then $\varepsilon_1 \circ^= \varepsilon_2 = \langle R \rangle$. Also $R!; R? \mapsto i_R$, but $i_R = \langle \langle R \rangle \vdash R \sim R \rangle$ and the result holds.

**Case ($G_1 = ?, G_2 = R, G_3 = ?$).** Then $\varepsilon_1 = \langle R \rangle, \varepsilon_2 = \langle R \rangle, c_1 = R?$, and $c_2 = R!$. As $R \cap R = R$ then $\varepsilon_1 \circ^= \varepsilon_2 = \langle R \rangle$. Also $R?; R!$ is in normal form, but $R?; R! = \langle \langle R \rangle \vdash ? \sim ? \rangle$ and the result holds.

**Case ($G_1 = ?, G_2 = ?, G_3 = ?$).** Then we proceed on cases for $\varepsilon_1$ and $\varepsilon_2$.

201
1. \((\varepsilon_1 = \langle ? \rangle, \varepsilon_2 = \langle ? \rangle)\). Then \(c_1 = i_?\) and \(c_2 = i_?\). But \(i_?; i_? \rightarrow i_?, \text{ nm } i_?, \varepsilon_1 \circ \varepsilon_2 = \langle ? \rangle\), and \(i_? = \langle \langle ? \rangle \vdash ? \sim ? \rangle\) and the result holds.

2. \((\varepsilon_1 = \langle R \rangle, \varepsilon_2 = \langle ? \rangle)\). Then \(c_1 = R?; R!\) and \(c_2 = i_?\). But \(R?; R!; i_? \rightarrow R?; R!\), \(\text{ nm } R?; R!\), \(\varepsilon_1 \circ \varepsilon_2 = \langle R \rangle\), and \(R?; R! = \langle \langle R \rangle \vdash ? \sim ? \rangle\) and the result holds.

3. \((\varepsilon_1 = \langle ? \rangle, \varepsilon_2 = \langle R \rangle)\). Analogous to previous sub-case.

4. \((\varepsilon_1 = \langle R_1 \rangle, \varepsilon_2 = \langle R_2 \rangle)\). Then \(c_1 = R_1?; R_1!\) and \(c_2 = R_2?; R_2!\). If \(R_1 \neq R_2\), then \(R_1?; R_1!; R_2?; R_2! \rightarrow \text{ 3Fail}\), but also \(\varepsilon_1 \circ \varepsilon_2 = \langle R_1 \rangle \circ \varepsilon_2 = \langle R_2 \rangle\) is undefined as \(R_1 \cap R_2\) is not defined.

If \(R_1 = R_2 = R\), then

\[
R_1?; R_1!; R_2?; R_2!
\]

\[
= R?; R!; R?; R!
\]

\[
\rightarrow R?; i_R; R!
\]

\[
\rightarrow R?; R!
\]

\[
\rightarrow i_R
\]

where \(\text{ nm } i_R\). But also \(\varepsilon_1 \circ \varepsilon_2 = \langle R_1 \rangle \circ \varepsilon_2 = \langle R \rangle \circ \varepsilon_2 = \langle R \rangle\), and \(i_R = \langle \langle R \rangle \vdash ? \sim ? \rangle\) and the result holds.

Case \((G_1 = R_1, G_2 = ? , G_3 = ?)\). Then we proceed on cases for \(\varepsilon_2\).

1. \((\varepsilon_1 = \langle R_1 \rangle, \varepsilon_2 = \langle ? \rangle)\). Then \(c_1 = R_1?; R_1!\) and \(c_2 = i_?\). But \(R_1?; R_1!; i_? \rightarrow R_1?; R_1!\), \(\text{ nm } R_1?; R_1!\), \(\varepsilon_1 \circ \varepsilon_2 = \langle R_1 \rangle\), and \(R_1?; R_1! = \langle \langle R_1 \rangle \vdash ? \sim ? \rangle\) and the result holds.

2. \((\varepsilon_1 = \langle R_1 \rangle, \varepsilon_2 = \langle R_2 \rangle)\). Then \(c_1 = R_1?; R_1!\) and \(c_2 = R_2?; R_2!\). If \(R_1 \neq R_2\), then \(R_1?; R_1!; R_2?; R_2! \rightarrow \text{ 3Fail}\), but also \(\varepsilon_1 \circ \varepsilon_2 = \langle R_1 \rangle \circ \varepsilon_2 = \langle R_2 \rangle\) is undefined as \(R_1 \cap R_2\) is not defined.

If \(R_1 = R_2 = R\), then

\[
R_1?; R_1!; R_2?; R_2!
\]

\[
= R?; R!; R?; R!
\]

\[
\rightarrow R?; i_R; R!
\]

\[
\rightarrow R?; R!
\]

\[
\rightarrow i_R
\]

where \(\text{ nm } i_R\). But also \(\varepsilon_1 \circ \varepsilon_2 = \langle R_1 \rangle \circ \varepsilon_2 = \langle R \rangle \circ \varepsilon_2 = \langle R \rangle\), and \(i_R = \langle \langle R \rangle \vdash ? \sim ? \rangle\) and the result holds.

Case \((G_1 = ?, G_2 = ?, G_3 = R_2)\). Analogous to previous case.

Case \((G_1 = \text{ Ref } G_1' \neq R_1, G_2 = \text{ Ref } G_2' \neq R_2, G_3 = \text{ Ref } G_3' \neq R_3)\). Then \(\varepsilon_1 = \langle \text{ Ref } G_1' \rangle, \varepsilon_2 = \langle \text{ Ref } G_2' \rangle\), \(c_1 = \text{ Ref } c_{21}c_{12}\), where \(c_{21} = \langle \langle G_1' \rangle \vdash G_2' \sim G_1' \rangle\) and \(c_{12} = \langle \langle G_1' \rangle \vdash G_1' \sim G_2' \rangle\), and \(c_2 = \text{ Ref } c_{32}c_{23}\), where \(c_{32} = \langle \langle G_2' \rangle \vdash G_3' \sim G_2' \rangle\) and \(c_{23} = \langle \langle G_2' \rangle \vdash G_3' \sim G_3' \rangle\).
But,
\[
c_1; c_2 = (\text{Ref } c_{21} \; c_{12}); (\text{Ref } c_{32} \; c_{23})
\]
\[
\rightarrow \text{Ref } (c_{32}; c_{21})(c_{12}; c_{23})
\]
and \(\varepsilon_1 \circ= \varepsilon_2 = (\text{Ref } G'_{12}) \circ= (\text{Ref } G'_{23}) = (\text{Ref } G'_{23}) \circ= (\text{Ref } G'_{12})\) (Prop \[87\]).

By induction hypothesis on \(c_{32} = (\langle G'_{23} \mid G'_{3} \sim G'_{2} \rangle)\) and \(c_{21} = (\langle G'_{12} \mid G'_{2} \sim G'_{1} \rangle)\), if \(c_{32}; c_{21} \rightarrow^* \text{Fail} \), and \(G'_{23} \cap G'_{12}\) is not defined, therefore \(\text{Ref } (c_{32}; c_{21})(c_{12}; c_{23}) \rightarrow^* \text{Fail} \), and \(\text{Ref } G'_{12} \circ= \text{Ref } G'_{23}\) is not defined and the result holds. Similarly by induction hypothesis on \(c_{12} = (\langle G'_{12} \mid G'_{1} \sim G'_{2} \rangle)\) and \(c_{23} = (\langle G'_{23} \mid G'_{2} \sim G'_{3} \rangle)\), if \(c_{32}; c_{21} \rightarrow^* \text{Fail} \) and \(G'_{12} \cap G'_{23}\) is not defined, therefore the result holds.

The only case left is that if we apply both induction hypotheses and we know that \(\text{Prop } \[87\], c_{12}; c_{23} \rightarrow^* c_{13}, \text{nm } c_{13}\), and \(c_{13} = (\langle G'_{12} \cap G'_{23} \rangle \mid G'_{1} \sim G'_{3} \rangle)\). Then \(\text{Ref } (c_{32}; c_{21})(c_{12}; c_{23}) \rightarrow^* \text{Ref } c_{31} \; c_{13}, \text{nm Ref } c_{31} \; c_{13}\), and \(\varepsilon_1 \circ= \varepsilon_2 = (\text{Ref } G'_{12}) \circ= (\text{Ref } G'_{23}) = (\text{Ref } G'_{12} \cap G'_{23})\).

But \((\langle \text{Ref } G'_{12} \cap G'_{23} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } G'_{2} \rangle =
\text{Ref } (\langle G'_{12} \cap G'_{23} \rangle \mid G'_{3} \equiv G'_{1} \rangle) \langle (G'_{12} \cap G'_{23}) \mid G'_{1} \equiv G'_{2} \rangle = \text{Ref } c_{31} \; c_{13}\) and the result holds.

Case \((G'_{1} = \text{Ref } G'_{1} \neq R_{1}, G'_{2} = \text{Ref } G'_{2} \neq R_{2}, G_{2} \neq G_{3} = ?)\). Then \(\varepsilon_{1} = (\text{Ref } G'_{12}), \varepsilon_{2} = (\text{Ref } G'_{23}), c_{1} = \text{Ref } c_{21} \; c_{12}\), where \(c_{21} = (\langle G'_{12} \mid G'_{2} \sim G'_{1} \rangle)\) and \(c_{12} = (\langle G'_{12} \mid G'_{1} \sim G'_{2} \rangle)\), and \(c_{2} = \text{Ref } c_{32} \; c_{23}\), where \(c_{32} = (\langle G'_{23} \mid G'_{3} \sim G'_{2} \rangle)\) and \(c_{23} = (\langle G'_{23} \mid G'_{2} \sim G'_{3} \rangle)\), and we proceed analogous to previous case.

Case \((G'_{1} = \text{Ref } G'_{1} \neq R_{1}, G_{2} = ? , G_{2} = ?)\).

Then \(\varepsilon_{1} = (\text{Ref } G'_{1}), c_{1} = (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle; (\text{Ref } ?)!\), and we proceed by cases for \(\varepsilon_{2}\).

1. \(\varepsilon_{2} = (\text{Ref } ?)\). Then \(c_{2} = i_{?}\), and \(c_{1}; c_{2} \rightarrow c_{1}\). We also know \(\text{nm } c_{1}\) (by Lemma \[88\]), and \(\varepsilon_{1} \circ= \varepsilon_{2} = (\text{Ref } G'_{12} \cap ?) = (\text{Ref } G'_{12}), \text{but } c_{1} = (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv ? \rangle\), and the result holds.

2. \(\varepsilon_{2} = (\text{Ref } ?)\). Then \(c_{2} = (\text{Ref } ?)\); (Ref ?)!, and
\[
c_{1}; c_{2} = (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle; (\text{Ref } ?)!; (\text{Ref } ?)\rangle; (\text{Ref } ?)\rangle!
\]
\[
\rightarrow (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle; (\text{Ref } ?)!\rangle!
\]
\[
\rightarrow (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle); i_{\text{Ref } ?}; (\text{Ref } ?)!\rangle!
\]
\[
eq c_{1}
\]
i.e. \(c_{1}; c_{2} \rightarrow^* c_{1}\). We also know \(\text{nm } c_{1}\) (by Lemma \[88\]), and \(\varepsilon_{1} \circ= \varepsilon_{2} = (\langle \text{Ref } G'_{12} \rangle \cap (\text{Ref } ?)\rangle = (\text{Ref } (G'_{12} \cap ?)) = (\text{Ref } G'_{12}), \text{but } c_{1} = (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv ? \rangle\), and the result holds.

3. \(\varepsilon_{2} = (\text{Ref } R)\), \(R \neq \text{Ref } ?\). Then \(c_{2} = \text{Ref } R?; \text{R}!\), and
\[
c_{1}; c_{2} = (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle; (\text{Ref } ?)!; \text{Ref } R?; \text{R}!
\]
\[
\rightarrow (\langle \text{Ref } G'_{12} \rangle \mid \text{Ref } G'_{1} \equiv \text{Ref } ? \rangle); \text{Fail}
\]
\[
\rightarrow^* \text{Fail}
\]
We also know that $\varepsilon_1 \circ= \varepsilon_2 = \varepsilon$ is not defined as $\text{Ref} G'_{12}) \cap (R)$ is undefined, and the result holds immediately.

Lemma 90. If $\varepsilon_1 \circ= \varepsilon_2$ is not defined then $\forall \varepsilon_2' \subseteq \varepsilon_2$, $\varepsilon_1 \circ= \varepsilon_2'$ is not defined.

Proof. By induction on $\varepsilon_1$ and $\varepsilon_2$ subject to consistent transitivity being not defined.

Lemma 91. If $\varepsilon_1 \circ= \varepsilon_2$ is not defined then $\forall \varepsilon_1' \subseteq \varepsilon_1$, $\varepsilon_1' \circ= \varepsilon_2$ is not defined.

Proof. Direct by Lemma 90 and Lemma 87.

Proposition 92. If $G = G_1 \cap G_2$ is defined, then $G \subseteq G_1$ and $G \subseteq G_2$.

Proof. Straightforward induction on $G_1 \cap G_2$.

Proposition 93. If $\langle \text{Ref } G \rangle \vdash \text{Ref } G_1 \sim \text{Ref } G_2$ then $G \vdash G_1 \sim G_2$.

Proof. Straightforward induction on $\langle \text{Ref } G \rangle \vdash \text{Ref } G_1 \sim \text{Ref } G_2$.

Proposition 94. If $\langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}$ then $G_1 \vdash G_{21} \sim G_{11}$.

Proof. Straightforward induction on $\langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}$.

Proposition 95. If $\langle G_1 \rightarrow G_2 \rangle \vdash G_{11} \rightarrow G_{12} \sim G_{21} \rightarrow G_{22}$ then $G_2 \vdash G_{12} \sim G_{22}$.

Proof. Straightforward induction on $\langle \text{Ref } G \rangle \vdash \text{Ref } G_1 \sim \text{Ref } G_2$.

Proposition 96 (Optimality). If $\varepsilon = \varepsilon_1 \circ= \varepsilon_2$ is defined, then $\pi_1(\varepsilon) \subseteq \pi_1(\varepsilon_1)$ and $\pi_2(\varepsilon) \subseteq \pi_2(\varepsilon_2)$.

Proof. Direct by Lemma 92 as evidences can be represented as singletons.

Lemma 97. $(c_1; c_2 \leftarrow \ast c \land c; c_3 \leftarrow \ast c') \iff (c_1; c_2); c_3 \leftarrow \ast c'$

Proof. Straightforward induction on $(c_1; c_2)$ and then induction on $c_3$.

Lemma 98. $(c_2; c_3 \leftarrow \ast c \land c_1; c \leftarrow \ast c') \iff (c_1; c_2); c_3 \leftarrow \ast c'$

Proof. For proving ($\Rightarrow$), we use straightforward induction on $(c_2; c_3)$ and then induction on $c_1$. For proving the other direction ($\Leftarrow$), we use induction on $(c_1; c_2); c_3$.
Proposition 99 (Associativity). Let $\varepsilon_1 \vdash G_1 \sim G_2$, $\varepsilon_2 \vdash G_2 \sim G_3$ and $\varepsilon_3 \vdash G_3 \sim G_4$. Then $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = \varepsilon_1 \circ^n (\varepsilon_2 \circ^n \varepsilon_3)$ or both are undefined.

Proof. By straightforward induction on evidences $\varepsilon_1, \varepsilon_2,$ and $\varepsilon_3$, noticing that if $\varepsilon_1 = \langle G_1 \rangle, \varepsilon_2 = \langle G_2 \rangle, \varepsilon_3 = \langle G_3 \rangle$, then it is equivalent to prove that $(G_1 \cap G_2) \cap G_3 = G_1 \cap (G_2 \cap G_3)$ or both are undefined. We only present interesting cases.

Case $(\varepsilon_1 = \langle G \rangle, \varepsilon_1 = \langle G \rangle, \varepsilon_1 = \langle G \rangle)$. Then the result is trivial as $\langle G \rangle \circ^n \langle G \rangle = \langle G \rangle$.

Case $(\varepsilon_1 = \langle G_1 \rangle, \varepsilon_1 = \langle G_2 \rangle)$. As $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle)$, the result holds immediately.

Case $(\varepsilon_1 = \langle G \rangle, \varepsilon_1 = \langle G \rangle)$. As $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = (\langle G \rangle \circ^n \langle G \rangle) \circ^n \varepsilon_3 = (\langle G \rangle \circ^n \langle G \rangle)$, the result holds immediately.

Case $(\varepsilon_1 = \langle G_1 \rangle, \varepsilon_1 = \langle G_2 \rangle)$. As $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle)$, the result holds immediately.

Case $(\varepsilon_1 = \langle G_1 \rangle, \varepsilon_1 = \langle G_2 \rangle, \varepsilon_1 = \langle G_3 \rangle)$, $G_1 \cap G_2$ not defined. We know that $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle$ is not defined, and that $\varepsilon_1 \circ^n \varepsilon_2 \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle = (\langle G_1 \rangle \circ^n \langle G_2 \cap G_3 \rangle) \circ^n \langle G_3 \rangle$ if $G_1 \cap G_2$ is not defined the result holds immediately. By Prop 96 $(\langle G_2 \cap G_3 \rangle \subseteq \langle G_2 \rangle)$. Then by Prop 90 $(\langle G_1 \rangle \circ^n \langle G_2 \cap G_3 \rangle)$ is not defined, and the result holds.

Case $(\varepsilon_1 = \langle G_1 \rangle, \varepsilon_1 = \langle G_2 \rangle, \varepsilon_1 = \langle G_3 \rangle)$, $G_1 \cap G_2$ not defined. We know that $\varepsilon_1 \circ^n (\varepsilon_2 \circ^n \varepsilon_3) = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle$ is not defined, and that $\varepsilon_1 \circ^n \varepsilon_2 \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle = (\langle G_1 \cap G_2 \rangle \circ^n \langle G_3 \rangle)$ if $G_1 \cap G_2$ is not defined the result holds immediately. By Prop 96 $(\langle G_1 \cap G_2 \rangle \subseteq \langle G_2 \rangle)$. Then by Prop 91 $(\langle G_1 \cap G_2 \rangle \circ^n \langle G_3 \rangle$ is not defined, and the result holds.

Case $(\varepsilon_1 = \langle G_1 \rangle, \varepsilon_1 = \langle G_2 \rangle, \varepsilon_1 = \langle G_3 \rangle)$, $G_1 \cap G_2$ not defined. We know that $(\varepsilon_1 \circ^n \varepsilon_2) \circ^n \varepsilon_3 = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle$, and that $\varepsilon_1 \circ^n (\varepsilon_2 \circ^n \varepsilon_3) = (\langle G_1 \rangle \circ^n \langle G_2 \rangle) \circ^n \langle G_3 \rangle$ (if $G_1 \cap G_2$ or $G_2 \cap G_3$ are not defined then the result holds immediately by next argument).

Then by Prop 96 $(\langle G_1 \cap G_2 \rangle \subseteq \langle G_1 \rangle$ and $(\langle G_2 \cap G_3 \rangle \subseteq \langle G_3 \rangle$, therefore by Prop 91 $(\langle G_1 \cap G_2 \rangle \circ^n \langle G_3 \rangle$ is not defined, and by Prop 90 $(\langle G_1 \rangle \circ^n \langle G_2 \cap G_3 \rangle$ is not defined, and the result holds as both combinations of evidence fail, regardless if $G_1 \cap G_2$ or $G_2 \cap G_3$ are defined or not.

Case $(\varepsilon_1 = \langle \text{Ref} G'_1 \rangle, \varepsilon_1 = \langle \text{Ref} G'_2 \rangle, \varepsilon_1 = \langle \text{Ref} G'_3 \rangle)$. Notice that $(\langle \text{Ref} G'_1 \rangle \circ^n \langle \text{Ref} G'_2 \rangle) \circ^n \langle \text{Ref} G'_3 \rangle = \langle \text{Ref} G'_1 \rangle$ is defined if and only if $G'_1 = \langle G'_1 \rangle \circ^n \langle G'_2 \rangle \circ^n \langle G'_3 \rangle$. Similarly $(\langle \text{Ref} G'_1 \rangle \circ^n \langle \text{Ref} G'_2 \rangle \circ^n \langle \text{Ref} G'_3 \rangle) = \langle \text{Ref} G'_1 \rangle$ is defined if and only if $G'_1 = \langle G'_1 \rangle \circ^n \langle G'_2 \rangle \circ^n \langle G'_3 \rangle$ is defined. Also $(\langle \text{Ref} G'_1 \rangle \vdash \text{Ref} G''_1 \sim \text{Ref} G''_2$, $(\langle \text{Ref} G'_2 \rangle \vdash \text{Ref} G''_2 \sim \text{Ref} G''_3$, and $(\langle \text{Ref} G'_3 \rangle \vdash \text{Ref} G''_3 \sim \text{Ref} G''_4$, then by inversion lemma (Lemma 93), $(\langle \text{Ref} G'_1 \rangle \vdash G''_1 \sim G''_2$, $(\langle \text{Ref} G'_2 \rangle \vdash G''_2 \sim G''_3$, and $(\langle \text{Ref} G'_3 \rangle \vdash G''_3 \sim G''_4$.

Then the result holds immediately as by induction hypothesis $(\langle \text{Ref} G'_1 \rangle \circ^n \langle G'_2 \rangle) \circ^n \langle G'_3 \rangle = (\langle \text{Ref} G'_1 \rangle \circ^n \langle G'_2 \rangle) \circ^n \langle G'_3 \rangle$ or both are undefined.

Case $(\varepsilon_1 = \langle G'_{11} \rightarrow G'_{12} \rangle, \varepsilon_1 = \langle G'_{21} \rightarrow G'_{22} \rangle, \varepsilon_1 = \langle G'_{31} \rightarrow G'_{32} \rangle)$. Analogous to the previous case but using inversion lemmas 93 and 94.
Proposition 100. Let \( c_1 \vdash G_1 \Rightarrow G_2, c_2 \vdash G_2 \Rightarrow G_3 \) and \( c_3 \vdash G_3 \Rightarrow G_4 \). Then \((c_1;c_2);c_3 = c_1;(c_2;c_3)\) or both are undefined.

Proof. By induction on \( c_1, c_2, \) and \( c_3 \). Alternatively, by Prop\[89 and Prop\[99 noticing that \( c_1 = \langle e_1 \vdash G_1 \sim G_{i+1} \rangle \) for some \( e_1 \).

Lemma 101. \[ \epsilon_1 t_1 \oplus @G_1 \rightarrow G_2 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1' t_1' \oplus @G_1 \rightarrow G_2 \varepsilon_2' t_2 \mid \mu' \] if and only if \( \epsilon_1 t_1 \vdash G_1 \rightarrow G_2 \mid \mu \mapsto \epsilon_1' t_1' \vdash G_1 \rightarrow G_2 \mid \mu' \).

Proof. We start by proving \( \Rightarrow \) by case analysis on \( t_1 \) (the other direction is analogous).

- If \( t_1 = \epsilon_3 t_3 \vdash G_3 \) where \( \epsilon_1(\epsilon_3 t_3 \vdash G_3) @G_1 \rightarrow G_2 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1(\epsilon_3' t_3' \vdash G_3) @G_1 \rightarrow G_2 \varepsilon_2' t_2 \mid \mu' \), then it is easy to see that \( \epsilon_1(\epsilon_3 t_3 \vdash G_3) \vdash G_1 \rightarrow G_2 \mid \mu \mapsto \epsilon_1(\epsilon_3' t_3' \vdash G_3) \vdash G_1 \rightarrow G_2 \mid \mu' \) and the result holds.

- If \( t_1 = \epsilon_3 u \vdash G_3 \) where \( \epsilon_1(\epsilon_3 u \vdash G_3) @G_1 \rightarrow G_2 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1' u @G_1 \rightarrow G_2 \varepsilon_2 t_2 \mid \mu \) and \( \epsilon_1' = \epsilon_3 \circ \epsilon_1 \), then also \( \epsilon_1(\epsilon_3 u \vdash G_3) \vdash G_1 \rightarrow G_2 \mid \mu \mapsto \epsilon_1' u \vdash G_1 \rightarrow G_2 \mid \mu \) and the result holds.

Lemma 102. \[ \epsilon_1 t_1 @G_1 \rightarrow G_2 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1' t_1' @G_1 \rightarrow G_2 \varepsilon_2' t_2 \mid \mu' \] if and only if \( \epsilon_2 t_2 \vdash G_1 \mid \mu \mapsto \epsilon_2' t_2' \vdash G_1 \mid \mu' \).

Proof. Analogous to Lemma \[101 \]

Lemma 103. \[ \epsilon_1 t_1 := G_3 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1' t_1' := G_3 \varepsilon_2' t_2 \mid \mu' \] if and only if \( \epsilon_1 t_1 \vdash \text{Ref } G_3 \mid \mu \mapsto \epsilon_1' t_1' \vdash \text{Ref } G_3 \mid \mu' \).

Proof. Similar to Lemma \[101 \]

Lemma 104. \[ \epsilon_1 t_1 := G_3 \varepsilon_2 t_2 \mid \mu \mapsto \epsilon_1 t_1 := G_3 \varepsilon_2' t_2 \mid \mu' \] if and only if \( \epsilon_2 t_2 \vdash G_3 \mid \mu \mapsto \epsilon_2' t_2' \vdash G_3 \mid \mu' \).

Proof. Analogous to Lemma \[103 \]

Lemma 105. \[ \text{ref}^G \varepsilon t \mid \mu \mapsto \text{ref}^G \varepsilon' t' \mid \mu' \] if and only if \( \varepsilon t \vdash G \mid \mu \mapsto \varepsilon' t' \vdash G \mid \mu' \).

Proof. Similar to Lemma \[101 \]

Lemma 106. \[ \text{!}^G \varepsilon t \mid \mu \mapsto \text{!}^G \varepsilon' t' \mid \mu' \] if and only if \( \varepsilon t \vdash \text{Ref } G \mid \mu \mapsto \varepsilon' t' \vdash \text{Ref } G \mid \mu' \).

Proof. Similar to Lemma \[101 \]
Lemma 107. If ε₁t₁ \(\vDash\) ε₂t₂ else ε₃t₃ \(\mid\) μ \(\longmapsto\) if ε₁t₁ then ε₂t₂ else ε₃t₃ \(\mid\) μ′ if and only if ε₁t₁ \(\vDash\) Bool \(\mid\) μ \(\longmapsto\) ε₁t₁ ′ \(\vDash\) Bool \(\mid\) μ′.

Proof. Similar to Lemma 101

Lemma 108. If \(c₁ \rightarrow c₂ = (ε \vDash G'₁ \rightarrow G'₂ \approx G₁ \rightarrow G₂), then c₁ = \{idom(ε) \vDash G₁ \approx G'₁\}\) and \(c₂ = \{icod(ε) \vDash G'₂ \approx G₂\}\).

Proof. By definition of the map function between evidence augmented consistent judgments and coercions we know that \(\{ε \vDash G₁₁ \rightarrow G₁₂ \approx G₂₁ \rightarrow G₂₂\} = \{idom(ε) \vDash G₁ \approx G'₁\}\) which is equal to \(c₁ \rightarrow c₂\), and the result holds immediately.

Lemma 109. If Ref \(c₁ \rightarrow c₂ = (ε \vDash \text{Ref} G' \approx \text{Ref} G), then c₁ = \{iref(ε) \vDash \text{Ref} G \approx \text{Ref} G'\}\) and \(c₂ = \{iref(ε) \vDash \text{Ref} G' \approx \text{Ref} G\}\).

Proof. By definition of the map function between evidence augmented consistent judgments and coercions we know that \(\{ε \vDash \text{Ref} G' \approx \text{Ref} G\} = \{iref(ε) \vDash G \approx G'\}\) which is equal to \(\text{Ref} c₁ \rightarrow c₂\), and the result holds immediately.

Lemma 110. If \(t₁ \approx t₂, v₁ \in T[G], and v₁ \approx v₂\), then \(t₁[v₁/x₁] \approx t₂[v₂/x]\)

Proof. By induction on the derivation \(t₁ \approx t₂\).

Lemma 111. Consider \(\langle G \rangle \vDash \) G \(\approx\) G, then

1. \(∀ε \vDash G \approx G', \langle G \rangle \circ ε = ε, and\)
2. \(∀ε \vDash G! \approx G, ε \circ \langle G \rangle = ε\)

Proof. By induction on evidence augmented consistent judgment \(\langle G \rangle \vDash \) G \(\approx\) G.

Lemma 15 (Bisimulation between \(\lambda^{\text{REF}}\ε \) and HCC⁺). If \(t₁ \in T[G], Γ; Σ \vDash_{H} t₂ : G, μ₂ \models Σ, µ₁ \approx µ₂, and t₁ \approx t₂, then

1. If \(t₁ \mid µ₁ \longmapsto t₁' \mid µ₁'\), then \(t₂ \mid µ₂ \longmapsto t₂' \mid µ₂'\) such that \(t₁' \approx t₂'\) and \(µ₁' \approx µ₂'\).
2. \(∃j, 1 \leq j \leq 3. \) if \(t₂ \mid µ₂ \longmapsto j \ t₂' \mid µ₂'\), then \(t₁ \mid µ₁ \longmapsto t₁' \mid µ₁'\) such that \(t₁' \approx t₂'\) and \(µ₁' \approx µ₂'\).

Proof. 1. We proceed by induction on \(t₁ \mid µ₁ \longmapsto t₁' \mid µ₁'\).

Case \((ε₁(λx^{G₁₁}.t₁') \mid @G₁ \rightarrow G₂ \vDash ε₂u₁ \mid µ₁ \longmapsto ε₁t₁'[ε₂u₁] \mid G₁₁/x^{G₁₁} \vDash G₂ \mid µ₁).\) Where \(ε₁ = icod(ε₁)\) and \(ε₂ = ε₂ \circ idom(ε₁)\). By inspection of (bapp), we know that \(t₂ = t₂₁ t₂₂\), for some \(t₂₁, t₂₂\) such that \(ε₁(λx^{G₁₁}.t₁') \vDash G₁ \rightarrow G₂ \approx t₂₁\) and \(ε₂u₁ \vDash G₁ \approx t₂₂\). We proceed by case analysis on \(t₂₁ t₂₂\).
• If \( t_{21} \rightarrow t_{22} = ((c_1 \rightarrow c_2)(\lambda x : G_{11}.t_2'))v_{22} \). Where \\
c_1 \rightarrow c_2 = (\varepsilon_1 \vdash G_{11} \rightarrow G_{12} \sim G_1 \rightarrow G_2). By Lemma \[108 \]
c_1 = \{\text{idom}(\varepsilon_1) \vdash G_1 \sim G_{11}\}, and \( c_2 = \{\text{idom}(\varepsilon_1) \vdash G_{12} \sim G_2\} \).

- If \( v_{22} = c_vu_{22} \), where \( c_v = (\varepsilon_2 \vdash G_u \sim G_1) \) and \( u_1 \approx u_{22} \), then by Prop \[89 \] \( c'_1 = c_v; c_1 = (\varepsilon'_2 \vdash G_u \sim G_{11}) \). Then if we assume \( c'_1 \neq i_{G_u} \) (the other case is analogous)

\[
((c_1 \rightarrow c_2)(\lambda x : G_{11}.t'_2))c_vu_{22} \mid \mu_2 \mapsto c_1((\lambda x : G_{11}.t'_2)c_1(c_vu_{22})) \mid \mu_2 \\
\mapsto c_1((\lambda x : G_{11}.t'_2)c'_1u_{22}) \mid \mu_2 \\
\mapsto c_1(t'_2[c'_1u_{22}/x]) \mid \mu_2
\]

But we know that \( t'_1 \approx t'_2 \), and that by \( \text{(b::eq)} \) \( \varepsilon'_2u_1 :: G_{11} \approx c'_1u_{22} \), by Lemma \[110 \] and \( \text{(b::eq)} \), \( \text{idom}(\varepsilon_1)t'_1[\varepsilon'_2u_1 :: G_{11}/x^{G_{11}}] :: G_2 \approx c_1(t'_2[c'_1u_{22}/x]) \)
and the result holds.

- If \( v_{22} = u_{22} \) where \( u_1 \approx u_{22} \), then by \( \text{(b::id)} \) \( \varepsilon_2 = (G_1) \) and \( u_1 \in T[G_1] \), therefore \( (G_1) \vdash G_1 \sim G_1 \). Therefore by Lemma \[111 \] \( \varepsilon'_2 = \text{idom}(\varepsilon_1) \). Then

\[
((c_1 \rightarrow c_2)(\lambda x : G_{11}.t'_2))u_{22} \mid \mu_2 \mapsto c_1((\lambda x : G_{11}.t'_2)c_1(u_{22})) \mid \mu_2 \\
\mapsto c_1(t'_2[c_1u_{22}/x]) \mid \mu_2
\]

But we know that \( t'_1 \approx t'_2 \), and that by \( \text{(b::eq)} \) \( \varepsilon'_2u_1 :: G_{11} \approx c_1u_{22} \), by Lemma \[110 \] and \( \text{(b::eq)} \), \( \text{idom}(\varepsilon_1)t'_1[\varepsilon'_2u_1 :: G_{11}/x^{G_{11}}] :: G_2 \approx c_1(t'_2[c_1u_{22}/x]) \)
and the result holds.

• If \( t_{21} \rightarrow t_{22} = (\lambda x : G_1.t'_2)v_{22} \). Then \( G_{11} = G_1, t'_1 \in T[G_2], \varepsilon_1 = (G_1 \rightarrow G_2) \), and \( (G_1 \rightarrow G_2) \vdash G_1 \rightarrow G_2 \sim G_1 \rightarrow G_2 \). By the inversion lemmas \( \text{idom}(\varepsilon_1) = (G_1) \vdash G_1 \sim G_1 \) and \( \text{idom}(\varepsilon_1) = (G_2) \vdash G_2 \sim G_2 \). But we know that \( t'_1 \approx t'_2 \), therefore by Lemma \[111 \] \( \varepsilon'_2 = \varepsilon_2 \). Finally by \( \text{(b::id)} \) and as \( \varepsilon_2u_1 :: G_1 \approx v_{22} \), by Lemma \[110 \] \( \text{idom}(\varepsilon_1)t'_1[\varepsilon'_2u_1 :: G_{11}/x^{G_{11}}] :: G_2 \approx (t'_2[v_{22}/x]) \)
and the result holds.

Case \( \text{(ref}^{G_1} \varepsilon_u_1 \mid \mu_1 \mapsto o^{G_1} \mid \mu_1[o^{G_1} \mapsto \varepsilon u_1 :: G_1]) \). We know by \( \text{(b::ref)} \) that \( t_2 = \text{ref} v_2 \), for some \( v_2 \) such that \( \varepsilon u_1 :: G_1 \approx v_2 \). But \( \text{ref} v_2 \mid \mu_2 \mapsto o \mid \mu_2[o \mapsto v_2] \). As \( o^{G_1} \approx o \), and \( \mu_1 \approx \mu_2 \), we only have to prove that \( \varepsilon u_1 :: G_1 \approx v_2 \), but we already know that by \( \text{(b::ref)} \), and the result holds immediately.

Case \( \text{(!}^{G} (\varepsilon o^{G_2}) \mid \mu_1 \mapsto \text{iref}(\varepsilon)v_1 :: G \mid \mu_1) \). Where \( \mu_1(x^{G_2}) = v_1 \). By inspection of \( \text{(b)} \), we know that \( t_2 = !v_2 \), for some \( v_2 \) such that \( \varepsilon o^{G_2} :: \text{Ref} G \approx v_2 \). We proceed by case analysis on \( v_2 \).

• If \( v_2 = (\text{Ref} c_1 c_2) o \). Where \( \text{Ref} c_1 c_2 = (\varepsilon \vdash \text{Ref} G_2 \sim \text{Ref} G) \), and \( o^{G_2} \approx o \). By Lemma \[109 \] \( c_1 = \{\text{iref}(\varepsilon) \vdash G \sim G_2\} \), and \( c_2 = \{\text{iref}(\varepsilon) \vdash G_2 \sim G\} \).

\[
!(\text{Ref} c_1 c_2) o \mid \mu_2 \mapsto c_2 o \mid \mu_2 \\
\mapsto c_2 v'_2 \mid \mu_2
\]

where \( \mu_2(o) = v'_2 \), and \( v'_2 \approx v_1 \). The result follows by applying \( \text{(b::eq)} \).
If \( v_2 = o \). Then \( G_2 = G \) and \( \varepsilon = \langle \text{Ref } G \rangle \). By the inversion lemma on evidence \( \text{iref}(\varepsilon) = \langle G \rangle \vdash G \sim G \). Then \( o | \mu_2 \mapsto v'_2 | \mu_2 \), where \( \mu_2(o) = v'_2 \), and \( v'_2 \approx v_1 \). By (b:id) we know that \( \langle G \rangle v_1 :: G \approx v'_2 \) and the result holds.

**Case** \( \varepsilon_1^G_3 := G_3 \varepsilon_2 u_{12} | \mu_1 \mapsto \text{unit } | \mu'_1 \). Where \( \mu'_1 = \mu_1[\sigma^G_1 \mapsto \varepsilon'_2 u_{12} :: G_1] \), and \( \varepsilon'_2 = \varepsilon_2 \circ \text{iref}(\varepsilon_1) \). By inspection of \( (b::=) \), we know that \( t_2 = t_{21} := t_{22} \), for some \( t_{21}, t_{22} \) such that \( \varepsilon_1^G_3 :: \text{Ref } G_3 \approx t_{21} \) and \( \varepsilon_2 u_{12} :: G_3 \approx t_{22} \). We proceed by case analysis on \( t_{21} := t_{22} \).

If \( t_{21} := t_{22} = (\langle \text{Ref } c_1 c_2 \rangle o) v_{22} \). Where \( \text{Ref } c_1 c_2 = \langle \varepsilon_1 \vdash \text{Ref } G_1 \sim \text{Ref } G_3 \rangle \).

By Lemma \[109\], \( c_1 = \langle \text{iref}(\varepsilon_1) \vdash G_3 \sim G_1 \rangle \), and \( c_2 = \langle \text{iref}(\varepsilon_1) \vdash G_1 \sim G_3 \rangle \).

- If \( v_{22} = c_v u_{22} \), where \( c_v = \langle \varepsilon_2 \vdash G_u \sim G_3 \rangle \) and \( u_1 \approx u_{22} \), then by Prop \[89\], \( c'_1 = c_v ; c_1 = \langle \varepsilon'_2 \vdash G_u \sim G_3 \rangle \). Then if we assume \( c'_1 \neq \iota_{G_u} \) (the other case is analogous)

\[
\begin{align*}
\langle \text{Ref } c_1 c_2 \rangle o := c_v u_{22} | \mu_2 \mapsto o := c_1(c_v u_{22}) | \mu_2 \\
\mapsto o := c'_1 u_{22} | \mu_2 \\
\mapsto \text{unit } | \mu_2 o \mapsto c_1 u_{22}
\end{align*}
\]

But we know that \( \text{unit } \approx \text{unit } \), and that by \( (b::eq) \) \( \varepsilon'_2 u_1 :: G_1 \approx c'_1 u_{22} \), and so \( \mu_1[\sigma^G_1 \mapsto \varepsilon'_2 u_{12} :: G_1] \approx \mu_2[\sigma \mapsto c'_1 u_{22}] \), and the result holds.

- If \( v_{22} = u_{22} \) where \( u_1 \approx u_{22} \), then by \( (b::id) \) \( \varepsilon_2 = \langle G_3 \rangle \) and \( u_1 \in T[G_3] \), therefore \( \langle G_3 \rangle \vdash G_3 \sim G_3 \). Therefore by Lemma \[111\], \( \varepsilon'_2 = \text{iref}(\varepsilon_1) \). Then

\[
\begin{align*}
\langle \text{Ref } c_1 c_2 \rangle o := u_{22} | \mu_2 \mapsto o := c_1(u_{22}) | \mu_2 \\
\mapsto \text{unit } | \mu_2 o \mapsto c_1 u_{22}
\end{align*}
\]

But we know that \( \text{unit } \approx \text{unit } \), and that by \( (b::eq) \) \( \varepsilon'_2 u_1 :: G_1 \approx c_1 u_{22} \), and so \( \mu_1[\sigma^G_1 \mapsto \varepsilon'_2 u_{12} :: G_1] \approx \mu_2[\sigma \mapsto c_1 u_{22}] \), and the result holds.

If \( t_{21} := t_{22} = o := v_{22} \). Then \( G_1 = G_3, \varepsilon_1 = \langle \text{Ref } G_3 \rangle \), and \( \langle G_3 \rangle \vdash \text{Ref } G_3 \sim \text{Ref } G_3 \). By the inversion lemmas \( \text{iref}(\varepsilon_1) = \langle G_3 \rangle \vdash G_3 \sim G_3 \) and \( \text{iref}(\varepsilon_1) = \langle G_3 \rangle \vdash G_3 \sim G_3 \). We also know that \( o := v_{22} | \mu_2 \mapsto o := v_{22} | \mu_2 o \mapsto v_{22} \), and \( \text{unit } \approx \text{unit } \). Also by Lemma \[111\], \( \varepsilon'_2 = \varepsilon_2 \). Finally by premise \( \varepsilon_2 u_1 :: G_1 \approx v_{22} \), and so \( \mu_1[\sigma^G_1 \mapsto \varepsilon_2 u_{12} :: G_1] \approx \mu_2[o \mapsto v_{22}] \), and the result holds.

**Case** (if \( \varepsilon b \) then \( \varepsilon_2 t_{12} \) else \( \varepsilon_3 t_{13} \) | \( \mu_1 \mapsto \varepsilon_2 t_{12} :: G | \mu_1 \)). Where \( b = \text{true} \) and \( \varepsilon_1 = \langle \text{Bool} \rangle \). By inspection of \( (b) \), we know that \( t_2 = \text{if } t_{21} \text{ then } t_{22} \text{ else } t_{23} \), for some \( t_{21}, t_{22}, t_{23} \), such that \( \langle \text{Bool} \rangle b :: \text{Bool} \approx t_{21}, \varepsilon_2 t_{12} :: G \approx t_{22}, \text{ and } \varepsilon_3 t_{13} :: G \approx t_{23} \). By \( (b::eq) \) or \( (b::id) \) and \( (bb) \), we know that either \( t_{21} = \text{true} \) or \( t_{21} = i_{\text{Booltrue}} \). Let us assume \( t_{21} = \text{true} \) (the other case is analogous modulo one extra step of evaluation). Then \( t_2 | \mu_2 \mapsto t_{22} | \mu_2 \), but \( \varepsilon_2 t_{12} :: G \approx t_{22} \) and the result holds immediately.

**Case** (if \( \varepsilon b \) then \( \varepsilon_2 t_{12} \) else \( \varepsilon_3 t_{13} \) | \( \mu_1 \mapsto \varepsilon_3 t_{13} :: G | \mu_1 \)). Where \( b = \text{false} \) and \( \varepsilon_1 = \langle \text{Bool} \rangle \). By inspection of \( (b) \), we know that \( t_2 = \text{if } t_{21} \text{ then } t_{22} \text{ else } t_{23} \), for some \( t_{21}, t_{22}, t_{23} \), such that \( \langle \text{Bool} \rangle b :: \text{Bool} \approx t_{21}, \varepsilon_2 t_{12} :: G \approx t_{22}, \text{ and } \varepsilon_3 t_{13} :: G \approx t_{23} \). By \( (b::eq) \) or \( (b::id) \) and \( (bb) \), we know that either \( t_{21} = \text{false} \) or \( t_{21} = i_{\text{Boolfalse}} \). Let us assume \( t_{21} = \text{false} \) (the other case is analogous modulo one extra step of evaluation). Then \( t_2 | \mu_2 \mapsto t_{23} | \mu_2 \), but \( \varepsilon_3 t_{13} :: G \approx t_{23} \) and the result holds immediately.
Case \((B_1)b_1 \oplus (B_2)b_2 \mid \mu_1 \mapsto b_{13} \mid \mu_1\). Where \(b_3 = b_1 \oplus b_2\). Then either \(t_2 = t_{21} \oplus t_{22}\). Where by (b::leq) or (b::id) and (bb) \(t_{21} = c[1]\) or \(t_{21} = \epsilon b_1\), and \(t_{22} = b_2\) or \(t_{22} = \epsilon b_2 b_2\). Let us assume \(t_{21} = b_1\) and \(t_{22} = b_2\) (the other cases is analogous modulo one or two extra steps of evaluation). Then \(b_1 \oplus b_2 = b_3\), where \(b_3 = b_1 \oplus b_2\), and the result holds immediately by (b::b).

Case \((\epsilon t_{11} \oplus G^2 \epsilon t_{12} \mid \mu_1 \mapsto \epsilon' t_{11} \oplus G^2 \epsilon t_{12} \mid \mu_1')\). Then by (bapp) \(t_2 = t_{21} t_{22}\). By Lemma \[101\] we know that \(\epsilon_1 t_{11} :: G_1 \rightarrow G_2 \mid \mu_1 \mapsto \epsilon_1 t_{11} :: G_1 \rightarrow G_2 \mid \mu_1'\). Also by (bapp) we know that \(\epsilon_1 t_{11} :: G_1 \rightarrow G_2 \mapsto t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* t_{21}' \mid \mu_2'\), and that \(\epsilon_1 t_{11} :: G_1 \rightarrow G_2 \approx t_{21}' \mid \mu_2'. The result follows directly by (bapp).

Case \((\epsilon_1 u_{11} \oplus G^2 \epsilon_1 u_{12} \mid \mu_1 \mapsto \epsilon_1 u_{11} \oplus G^2 \epsilon_1 u_{12} \mid \mu_1')\). Analogous to previous case but using Lemma \[102\]

Case \((\epsilon_1 t_{11} \mapsto G_3 \epsilon_2 t_{12} \mid \mu_1 \mapsto \epsilon_1 t_{11} :: G_3 \epsilon_2 t_{12} \mid \mu_1')\). Then by (b::ref) \(t_2 = t_{21} := t_{22}\). By Lemma \[103\] we know that \(\epsilon_1 t_{11} :: G_3 \mid \mu_1 \mapsto \epsilon_1 t_{11} :: G_3 \mid \mu_1'\). Also by (b::ref) we know that \(\epsilon_1 t_{11} :: G_3 \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* t_{21}' \mid \mu_2'\), and that \(\epsilon_1 t_{11} :: G_3 \approx t_{21}' \mid \mu_2'. The result follows directly by (b::ref).

Case \((\epsilon_1 t_{11} :: G^2 \mid \mu_1 \mapsto \epsilon_1 t_{11} :: G^2 \mid \mu_1')\). Then by (bb) \(t_2 = \epsilon t_{21}\). By Lemma \[105\] we know that \(\epsilon_1 t_{11} :: G^2 \mid \mu_1 \mapsto \epsilon_1 t_{11} :: G^2 \mid \mu_1'\). Also by (bb) we know that \(\epsilon_1 t_{11} :: G^2 \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* t_{21}' \mid \mu_2'\), and that \(\epsilon_1 t_{11} :: G^2 \approx t_{21}' \mid \mu_2'. The result follows directly by (bb).

Case \((!G \epsilon_1 t_{11} \mid \mu_1 \mapsto !G \epsilon_1 t_{11} \mid \mu_1')\). Then by (b!) \(t_2 = !t_{21}\). By Lemma \[106\] we know that \(\epsilon_1 t_{11} :: !G \mid \mu_1 \mapsto \epsilon_1 t_{11} :: !G \mid \mu_1'\). Also by (b!) we know that \(\epsilon_1 t_{11} :: !G \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* t_{21}' \mid \mu_2'\), and that \(\epsilon_1 t_{11} :: !G \approx t_{21}' \mid \mu_2'. The result follows directly by (b!).

Case \((\epsilon t_{11} :: G \mid \mu_1 \mapsto \epsilon t_{11} :: G \mid \mu_1')\). Without loosing generality let us assume that \(\epsilon t_{11} :: G = \epsilon t_{1}(...) (\epsilon n t_{1n} :: G_n) \mid \mu_1\), where \(t_{1n} \in T[G_{n-1}]\) is not an ascribed term.

Then by (b::eq), (b::id) and (b::leq), either \(t_2 = c_1 (...) (c_m t_{2m})\) (and \(m \leq n\)) or \(t_2 = t_{21}\), where \(t_{2m}\) and \(t_{21}\) are coered terms, such that \(t_{1n} \approx t_{2m}\) or \(t_{1n} \approx t_{21}\) respectively.

Let us assume that \(t_2 = c_1 (...) (c_m t_{2m})\) (the other case is analogous to the second sub-case below). Then we know that \(c_1 (...) (c_m t_{2m}) \mapsto c_1 t_{2m}\), where \(c_m :: c_{m-1} :: \ldots :: c_1 \mapsto^* c_1'\) and \(nm c_1'\). Also by repeatedly applying (b::eq) and (b::leq), \(\epsilon t_{1} (...) (\epsilon n t_{1n} :: G_n) \mid \mu_1\). Additionally, by inspection of (b::eq) and (b::leq), \(\exists \epsilon t_{11}, \ldots \epsilon t_{1n}\), such that \(c_1' t_{2m}\). We now proceed by case analysis on \(t_{1n}\).
• If \( t_{1n} = u_1 \). Then 
  \[ \varepsilon_1(\ldots(\varepsilon_{n-1}(\varepsilon_n u_1 \vdash G_n) \vdash G_{n-1})\ldots) \vdash G \mapsto \varepsilon_1(\ldots(\varepsilon_{n-1}(u_1 \vdash G_{n-1})\ldots) \vdash G \text{, where } \varepsilon_{n-1} = (\varepsilon_n \circ \varepsilon_{n-1}). \]
  Then \( c'_{1n}; c'_{1n-1} \mapsto * c''_{1n-1} \), for some \( c''_{1n-1} \), therefore by Lemma 89 \( c''_{1n-1} = (\varepsilon_{n-1} \vdash G_{n+1} \sim G_{n-1}) \), and then by (b:eq), \( \varepsilon_{n-1} u_1 \vdash G_{n-1} \approx c''_{1n-1} u_2 \). Then the result holds by using (b::leq) and (b::eq) repeatedly and using \( c'_{11}, \ldots, c'_{1n-2}, c''_{1n-1} \) and Lemma 100.

• If \( t_{1n} \) is not a simple value, and therefore \( t_{1n} \mid \mu_1 \mapsto t'_{1n} \mid \mu'_1 \). By induction hypothesis \( t_{2m} \mid \mu_2 \mapsto * t'_{2m} \mid \mu'_2 \), such that \( t'_{1n} \approx t'_{2m} \) and \( \mu'_{1} \approx \mu'_{2} \). Therefore by (b::leq) and (b::eq), \( \varepsilon_1(\ldots(\varepsilon_{n} t'_{1n} \vdash G_n)\ldots) \vdash G \approx c'_1 t'_{2m} \) and the result holds.

The proof of (2) is similar but choosing sometimes \( j = 2 \) or \( j = 3 \) in cases for application, dereference or assignment.

\[ \square \]

**Lemma 112.** If \( c = (\varepsilon \vdash G' \sim G) \), then \( \text{nm } c \).

*Proof.* Direct by induction on \( \varepsilon \vdash G' \sim G \) and definition of \( \text{nm } (\|) (\text{Fig. 3.12}). \)

\[ \square \]

**Lemma 113.** If \( t_1 \in T[G], \Gamma; \Sigma \vdash t_2 : G, \mu_2 = \Sigma, \mu_1 \approx \mu_2 \), and \( t_1 \approx t_2 \), then \( t_1 \mid \mu_1 \Downarrow \iff t_2 \mid \mu_2 \Downarrow \).

*Proof.*

*Case* \((\Rightarrow)\). We proceed by induction on \( t_1 \mid \mu_1 \mapsto * v_1 \mid \mu'_1 \).

*Case* \((t_1 = v_1)\). As \( t_1 \approx t_2 \) and by Lemma 15 then \( t_2 \mid \mu_2 \mapsto * t'_2 \mid \mu'_2 \), and \( v_1 \approx t'_2 \) and \( \mu_1 \approx \mu'_2 \). If \( v_1 = u_1 \), then the result holds immediately by inspection of (Bb), (b\( \lambda \)), and (bo). If \( v_1 = \varepsilon u \vdash G \) then either by (b::id) \( t'_2 = u_2 \) and the result holds, or by (b::eq) \( t'_2 = c u_2 \), where \( c = (\varepsilon \vdash G \sim G) \) (and therefore \( c \neq \text{Fail} \)) and by Lemma 112 the result holds.

*Case* \((t_1 \mid \mu_1 \mapsto t'_1 \mid \mu''_1 \) and \( t'_1 \mid \mu''_1 \mapsto * v_1 \mid \mu'_1 \)). By Lemma 15 then \( t_2 \mid \mu_2 \mapsto * t'_2 \mid \mu'_2 \) and \( t'_1 \approx t'_2 \) and \( \mu'_1 \approx \mu_2 \). Then by induction hypothesis, \( t'_2 \mid \mu'_2 \Downarrow \), and therefore \( t_2 \mid \mu_2 \Downarrow \) and the result holds.

*Case* \((\Leftarrow)\). We proceed similarly by induction on \( t_2 \mid \mu_2 \mapsto * v_2 \mid \mu'_2 \).

*Case* \((t_2 = v_2)\). Similar to the \((t_1 = v_1)\) case.

*Case* \((t_2 \mid \mu_2 \mapsto k t'_{2} \mid \mu''_2 \) and \( t'_{2} \mid \mu''_2 \mapsto n-k v_2 \mid \mu'_2 \)). By Lemma 15 we know that there exists some \( j \in \{1,2,3\} \), such that \( t_1 \mid \mu_1 \mapsto * t'_1 \mid \mu''_1 \) and \( t'_1 \approx t'_2 \) and \( \mu'_1 \approx \mu_2 \). We choose \( k = j \), and by induction hypothesis, \( t'_2 \mid \mu'_2 \Downarrow \), and therefore \( t'_1 \mid \mu_1 \Downarrow \) and the result holds.

\[ \square \]

**Lemma 114.** If \( t_1 \in T[G], \Gamma; \Sigma \vdash t_2 : G, \mu_2 = \Sigma, \mu_1 \approx \mu_2 \), and \( t_1 \approx t_2 \), then

1. If \( t_1 \mid \mu_1 \mapsto \text{error} \), then \( t_2 \mid \mu_2 \mapsto * \text{error} \).
2. \( \exists j, 1 \leq j \leq 3 \). If \( t_2 \mid \mu_2 \mapsto j \text{ error} \), then \( t_1 \mid \mu_1 \mapsto * \text{error} \).

211
Proof.  1. We proceed by induction on $t_1 | \mu_1 \mapsto \text{error}$.
Case $\varepsilon_1(\lambda x : G_{11}, t'_1) @^{G_1 \rightarrow G_2} \varepsilon_2 u_1 | \mu_1 \mapsto \text{error}$. Where $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is not defined. By inspection of (bapp), we know that $t_2 = t_{21} t_{22}$, for some $t_{21}, t_{22}$ such that $\varepsilon_1(\lambda x : G_{11}, t'_1) :: G_1 \rightarrow G_2 \approx t_{21}$ and $\varepsilon_2 u_1 :: G_1 \approx t_{22}$. We proceed by case analysis on $t_{21} t_{22}$.

- If $t_{21} t_{22} = ((c_1 \rightarrow c_2) (\lambda x : G_{11}, t'_2)) v_{22}$. Where $c_1 \rightarrow c_2 = (\varepsilon_2 \vdash G_u \sim G_1)$ and $u_1 \approx u_{22}$, then by Prop[89] $c_1' = c_v; c_1 = \text{Fail}$.

\[
\begin{align*}
((c_1 \rightarrow c_2) (\lambda x : G_{11}, t'_2)) c_v u_{22} | \mu_2 \mapsto & c_1 ((\lambda x : G_{11}, t'_2) c_1 (c_v u_{22})) | \mu_2 \\
\mapsto & c_1 ((\lambda x : G_{11}, t'_2) c_1 u_{22}) | \mu_2 \\
\mapsto & \text{error}
\end{align*}
\]

and the result holds.

- If $v_{22} = u_{22}$. This case cannot happen: as $u_1 \approx u_{22}$, then by (b::id) $\varepsilon_2 = (\langle G_1 \rangle)$ and $u_1 \in \mathcal{T}[G_1]$, therefore $\langle G_1 \rangle \vdash G_1 \sim G_1$. Therefore by Lemma[111] $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is defined, which is a contradiction.

- If $t_{21} t_{22} = (\lambda x : G_1, t'_2) v_{22}$. This case cannot happen as $\varepsilon_1 = (\langle G_1 \rightarrow G_2 \rangle)$ and as $\varepsilon_2 \vdash G_u \sim G_1$, $\varepsilon_2 \circ \text{dom}(\varepsilon_1) = \varepsilon_2 \circ \langle G_1 \rangle$ by Lemma[111] it would never fail.

Case $\varepsilon_1 o^{G_1} := \varepsilon_2 u_{12} | \mu_1 \mapsto \text{error}$. Where $\varepsilon_2 \circ \text{iref}(\varepsilon_1)$ is not defined. By inspection of (b::=), we know that $t_2 = t_{21} := t_{22}$, for some $t_{21}, t_{22}$ such that $\varepsilon_1 o^{G_1} :: \text{Ref} G_3 \approx t_{21}$ and $\varepsilon_2 u_{12} :: G_3 \approx t_{22}$. We proceed by case analysis on $t_{21} := t_{22}$.

- If $t_2 := t_{21} = ((\text{Ref} c_1 c_2) o) v_{22}$. Where $\text{Ref} c_1 c_2 = (\langle \varepsilon_1 \vdash \text{Ref} G_1 \sim \text{Ref} G_3 \rangle)$. By Lemma[109] $c_1 = (\langle \text{iref}(\varepsilon_1) \vdash G_3 \sim G_1 \rangle)$, and $c_2 = (\langle \text{iref}(\varepsilon_1) \vdash G_1 \sim G_3 \rangle)$.

\[
\begin{align*}
((\text{Ref} c_1 c_2) o) := & c_v u_{22} | \mu_2 \mapsto o := c_1 (c_v u_{22}) | \mu_2 \\
\mapsto & o := c_1' u_{22} | \mu_2 \\
\mapsto & \text{error}
\end{align*}
\]

And the result holds.

- If $v_{22} = u_{22}$. This case cannot happen: as $u_1 \approx u_{22}$, then by (b::id) $\varepsilon_2 = (\langle G_3 \rangle)$ and $u_1 \in \mathcal{T}[G_3]$, therefore $\langle G_3 \rangle \vdash G_3 \sim G_3$. Therefore by Lemma[111] $\varepsilon_2 \circ \text{dom}(\varepsilon_1)$ is defined, which is a contradiction.

- If $t_2 := t_{21} = o := v_{22}$. This case cannot happen as $\varepsilon_1 = (\langle \text{Ref} G_3 \rangle)$ and as $\varepsilon_2 \vdash G_u \sim G_3$, $\varepsilon_2 \circ \text{iref}(\varepsilon_1) = \varepsilon_2 \circ (\langle G_3 \rangle)$ by Lemma[111] it would never fail.
Case \((\varepsilon_1 t_{11} \cdot G_1 \rightarrow G_2 \varepsilon_2 t_{12} \mid \mu_1 \mapsto \text{error}, t_{11} \neq u)\). Then by (bapp) \(t_2 = t_{21} \ t_{22}\). By Lemma 101 we know that \(\varepsilon_1 t_{11} :: G_1 \rightarrow G_2 \mid \mu_1 \mapsto \text{error}\). Also by (bapp) we know that \(\varepsilon_1 t_{11} :: G_1 \rightarrow G_2 \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* \text{error}\), and the result holds.

Case (if \(\varepsilon_1 t_{11} \text{ then } \varepsilon_2 t_{12} \text{ else } \varepsilon_3 t_{13} \mid \mu_1 \mapsto \text{error}, t_{11} \neq u\)). Then by (bif) \(t_2 = \text{if } t_{21} \text{ then } t_{22} \text{ else } t_{23}\). By Lemma 107 we know that \(\varepsilon_1 t_{11} :: \text{Bool} \mid \mu_1 \mapsto \text{error}\). Also by (bif) we know that \(\varepsilon_1 t_{11} :: \text{Bool} \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* \text{error}\), and the result holds.

Case \((\varepsilon_1 u_{11} @ G_1 \rightarrow G_2 \varepsilon_2 t_{12} \mid \mu_1 \mapsto \text{error}, t_{12} \neq u)\). Analogous to previous case but using Lemma 102.

Case \((\varepsilon_1 t_{11} :: G_3 \varepsilon_2 t_{12} \mid \mu_1 \mapsto \text{error}, t_{11} \neq u)\). Then by (b:=) \(t_2 = t_{21} := t_{22}\). By Lemma 103 we know that \(\varepsilon_1 t_{11} :: \text{Ref } G_3 \mid \mu_1 \mapsto \text{error}\). Also by (b:=) we know that \(\varepsilon_1 t_{11} :: \text{Ref } G_3 \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* \text{error}\), and the result follows.

Case \((\varepsilon_1 u_{11} :: G_3 \varepsilon_2 t_{12} \mid \mu_1 \mapsto \text{error}, t_{12} \neq u)\). Analogous to previous case but using Lemma 104.

Case \((\text{ref}^G \varepsilon_1 t_{11} \mid \mu_1 \mapsto \text{error}, t_{11} \neq u)\). Then by (bref) \(t_2 = \text{ref } t_{21}\). By Lemma 105 we know that \(\varepsilon_1 t_{11} :: G' \mid \mu_1 \mapsto \text{error}\). Also by (bref) we know that \(\varepsilon_1 t_{11} :: G' \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* \text{error}\), and the result follows.

Case \((t^G \varepsilon_1 t_{11} \mid \mu_1 \mapsto \text{error}, t_{11} \neq u)\). Then by (b!) \(t_2 = t_{21}\). By Lemma 106 we know that \(\varepsilon_1 t_{11} :: \text{Ref } G' \mid \mu_1 \mapsto \text{error}\). Also by (b!) we know that \(\varepsilon_1 t_{11} :: \text{Ref } G' \approx t_{21}\). Then by induction hypothesis we know that \(t_{21} \mid \mu_2 \mapsto^* \text{error}\), and the result follows.

Case \((\varepsilon_1 t_{11} :: G \mid \mu_1 \mapsto \varepsilon_1 t_{11}' :: G \mid \mu_1'\)\). Without loosening generality let us assume that \(\varepsilon_1 t_{11} :: G = \varepsilon_1(...)\varepsilon_{n t_{1n}} :: G_n(...) :: G\), where \(t_{1n} \in T[G_{n-1}]\) is not an ascribed term.

Then by (b:eq), (b:id) and (b:len), either \(t_2 = c_1(...)c_{m t_{2m}}(...)\) (and \(m \leq n\)) or \(t_2 = t_{21}\), where \(t_{2m}\) and \(t_{21}\) are not coerced terms, such that \(t_{1n} \approx t_{2m}\) or \(t_{1n} \approx t_{21}\) respectively.

Let us assume that \(t_2 = c_1(...)c_{m t_{2m}}(...)\) (the other case is analogous to the second sub-case below). Then we know that \(c_1(...)c_{m t_{2m}}(...) \mapsto c_1' t_{2m}\), where \(c_m::c_{m-1}::c_1 \mapsto^* c_1'\) and \(nm c_1'.\) Also by repeatedly applying (b:eq) and (b:len), \(\varepsilon_1(...)\varepsilon_{n t_{1n}} :: G_n(...) :: G \approx c_1' t_{2m}\). Additionally, by inspection of (b:eq) and (b:len), \(\exists c_1', ...c_{1n}'\), such that \(c_1' = (\exists i \vdash G_{i+1} \sim G_i)\), and that \(c_1';...c_{11}' \mapsto^* c_1'.\)

We now proceed by case analysis on \(t_{1n}\).

- If \(t_{1n} = u_1\). Then \(t_{2m} = u_2\) for some \(u_2\), also \(\varepsilon_1(...)\varepsilon_{n-1}(\varepsilon_n u_1 :: G_n :: G_{n-1}(...)) :: G \mapsto \text{error}\), where \((\varepsilon_n \circ \varepsilon_{n-1})\) is not defined. Then by Lemma 89 \(c_1' \mapsto^* \text{Fail}\), therefore by Lemma 100 \(c_1' = \text{Fail}\), but \(\text{Fail } u_2 \mapsto \text{error}\) and the result holds.

- If \(t_{1n}\) is not a simple value, and therefore \(t_{1n} \mid \mu_1 \mapsto \text{error}\). By induction hypothesis \(t_{2m} \mid \mu_2 \mapsto^* \text{error}\) and the result holds.

The proof of (2) is similar but choosing sometimes \(j = 2\) or \(j = 3\) in cases for application, dereference or assignment. □
Lemma 115. Let $G_1 \neq G_2$ such that $G_1 \sim G_2$, then $\langle G_1 \Rightarrow G_2 \rangle = \langle \mathcal{R}(G_1, G_2) \vdash G_1 \sim G_2 \rangle$

Proof. Straightforward induction on $G_1 \sim G_2$. □

Lemma 116. If $t_1 \in T[G], \Gamma; \Sigma \vdash_M t_2 : G$, $\mu_2 \models \Sigma$, $\mu_1 \approx \mu_2$, and $t_1 \approx t_2$, then $t_1 \mid \mu_1 \Downarrow \text{error} \iff t_2 \mid \mu_2 \Downarrow \text{error}$.

Proof. Similar to Lemma 113 □

Proposition 16 (Translations are bisimilar). If $\varnothing; \varnothing \vdash t : G$, $\varnothing; \varnothing \vdash t \sim \varepsilon t_1 : G$, and $\varnothing; \varnothing \vdash t \sim t_2 : G$, then $t_1 \approx t_2$.

Proof. We prove the proposition on open terms: If $\Gamma; \varnothing \vdash t : G$, $\Gamma; \varnothing \vdash t \sim \varepsilon t_1 : G$, and $\Gamma; \varnothing \vdash t \sim t_2 : G$, then $t_1 \approx t_2$.

We proceed by induction on $\Gamma; \varnothing \vdash t : G$ (we only show some cases as the others are analogous).

Case ($\Gamma; \varnothing \vdash t' : G : G$). Then

$$\Gamma; \varnothing \vdash t' \sim \varepsilon t' : G' \quad \varepsilon = \mathcal{R}(G', G)$$

By ($G :)$ we know that $\Gamma; \varnothing \vdash t' : G'$, then by induction hypothesis $t' \approx t'$. If $G' = G$, then $\varepsilon = \mathcal{R}(G)$ and $\mathcal{R}(G \Rightarrow G) t' = t'$, therefore the result holds immediately by (b:id). If $G' \neq G$, then by Lemma 115 $\langle G' \Rightarrow G \rangle = \langle G' \Rightarrow G \rangle = \langle \varepsilon \vdash G' \sim G \rangle$, and the result holds immediately by (b:).eq.

Case ($\Gamma; \varnothing \vdash \lambda x : G_1 : t' : G_1 \rightarrow G_2$). Then

$$\Gamma, x : G_1 \vdash t' \sim \varepsilon t' : G_2$$

By ($G\lambda$) we know that $\Gamma, x : G_1 \vdash t' : G_2$, then by induction hypothesis $t' \approx t'_2$. Then the result holds immediately by ($b\lambda$).

Case ($\Gamma; \varnothing \vdash t_1 := t_2 : \text{Unit}$).

$$\begin{align*}
\Gamma; \varnothing \vdash t_1 \sim \varepsilon t^G_1 : G_1 \\
\Gamma; \varnothing \vdash t_2 \sim \varepsilon t^G_2 : G_2 \\
\Gamma; \varnothing \vdash t_1 := t_2 \sim \varepsilon_1 t^G_1 := t^G_2 : \text{Unit}
\end{align*}$$
By \((G :=) \Gamma; \emptyset \vdash t_1 : G_1\) and \(\Gamma; \emptyset \vdash t_2 : G_2\), therefore by induction hypothesis \(t^{G_1} \approx t'_1\) and \(t^{G_2} \approx t'_2\). Let us consider \(G_1 \neq \text{Ref } G_3\) and \(G_2 \neq G_3\) (the other cases are similar – see case for ascription). By Lemma 115 \(\langle G_1 \Rightarrow \text{Ref } G_3 \rangle = \langle G_1 \Rightarrow \text{Ref } G_3 \rangle = \langle \varepsilon_1 \vdash G_1 \Rightarrow \text{Ref } G_3 \rangle\), therefore by \((b::eq)\), \(\varepsilon_1 t^{G_1} :: \text{Ref } G_3 \approx \langle G_1 \Rightarrow \text{Ref } G_3 \rangle t'_1\). By using similar argument, \(\varepsilon_2 t^{G_2} :: G_3 \approx \langle G_2 \Rightarrow G_3 \rangle t'_2\). Then the result holds by \((b::=)\).

\[\square\]

**Lemma 117.** Consider \(t_1 \in T[G]\), \(\Gamma; \Sigma \vdash_H t_2 : G\), \(\emptyset; \Sigma \vdash C[\Gamma; \Sigma \vdash . : G] : G'\), \(C \rightsquigarrow (C, C)\). If \(t_1 \approx t_2\), then 
\(C[t_1] \approx C[t_2]\).

**Proof.** Straightforward induction on \(C\), using Prop 16 \[\square\]

**Proposition 17.** If \(t_1 \in T[G]\), \(\Gamma; \Sigma \vdash t_2 : G\), and \(t_1 \approx t_2\), then \(\Gamma; \Sigma \vdash t_1 \approx_{\text{ctx}} t_2 : G\).

**Proof.** Given \(\emptyset; \Sigma \vdash C[\Gamma; \Sigma \vdash . : G] : G'\) and \(C \rightsquigarrow (C, C)\), by Lemma 117, we know that \(C[t_1] \approx C[t_2]\). Then the result holds by Lemmas 113 and 116 \[\square\]

**Corollary 18 (Contextual equivalence).** If \(\Gamma; \Sigma \vdash t : G\), \(\Gamma; \Sigma \vdash t \sim_e t_1 : G\), and \(\Gamma; \Sigma \vdash t \sim_c t_2 : G\), then \(\Gamma; \Sigma \vdash t_1 \approx_{\text{ctx}} t_2 : G\).

**Proof.** By combining Props 16 and 17 \[\square\]

### A.5 Encoding Permissive and Monotonic References in \(\lambda\text{REF}\)

**Proposition 19 (Monotonicity of the heap).** If \(t^G \mid \mu \mapsto t'^G \mid \mu'\), then 
\(\forall o^{G'}_m \in \text{dom}(\mu), \mu(o^{G'}_m) = \varepsilon u :: G', \text{then } \mu'(o^{G'}_m) = \varepsilon' u' :: G'\) and \(\varepsilon' \subseteq \varepsilon\).

**Proof.** We proceed by induction on \(t^G \mid \mu \mapsto t'^G \mid \mu'\). We only illustrate representative cases.

**Case (RE and r4).** Then \(\text{ref}^{G_2} : \langle G_1 \rangle u \mid \mu \mapsto \langle \text{Ref } G_1 \rangle o^{G_2} :: \text{Ref } G_2 \mid \mu'\), where 
\(\mu' = \mu[o^{G_2} \mapsto \langle G_1 \rangle u :: G_2]\) and \(o^{G_2} \notin \text{dom}(\mu)\). But then \(\forall o^{G'}_m \in \text{dom}(\mu), \mu(o^{G'}_m) = \mu'(o^{G'}_m)\) and the result holds immediately.

**Case (RE and r6).** Then \(t^G = \langle \text{Ref } G_1 \rangle o^{G_4} := G_3 \langle G_2 \rangle u\). Then 
\(t^G \mid \mu \mapsto \text{unit} \mid \mu[o^{G_4}_m \mapsto \langle G_2 \cap G' \cap G_1 \rangle u :: G_4]\), where \(\mu(o^{G_4}_m) = \langle G' \rangle u :: G_4\). Then we have to prove that \(G_2 \cap G' \cap G_1 \subseteq \langle G' \rangle\), but the result holds immediately by Prop 92.

**Case (RE and r6).** Then \(t^G = \langle \text{Ref } G_1 \rangle o^{G_4} := G_3 \langle G_2 \rangle u, z \neq m\). The result is immediate as the updated location is not monotonic.

**Case (RF).** Then \(t^G \mid \mu \mapsto F[\langle G_3 \rangle o^{G_5}_m] \mid \mu[o^{G_5}_m \mapsto \langle G_4 \rangle u' :: G_5]\), where \(\mu(o^{G_5}_m) = \langle G' \rangle u' :: G_5\). We have to prove that \(G_4 \subseteq G'\), but the result holds immediately by Prop 92.
Proposition 118 (\(\quad\rightarrow\) is well defined). If \(t^G \vdash \mu'\) and \(t^G \rightarrow r\), then \(r \in \text{CONFIG}_G \cup \{\text{error}\}\), and if \(r = t^G \mid \mu'\), then also \(t^G \vdash \mu'\) and \(\text{dom}(\mu) \subseteq \text{dom}(\mu')\).

Proof. By induction on the structure of a derivation of \(t^G \rightarrow r\), considering the last rule used in the derivation. The proof is analogous to some cases considered in Prop 19. We only illustrate representative cases.

Case (\(r_2\)). Then \(t^G \vdash (G_1)u \mid \mu \rightarrow \langle \text{Ref } G_1 \rangle o^G_2; \text{Ref } G_2 \mid \mu'\), where \(\mu' = \mu[o^G_2 \mapsto \langle G_1 \rangle u :: G_2]\) and \(o^G_2 \notin \text{dom}(\mu)\).

Case (r4). Then \(t^G = \varepsilon_1 o^G_2 \mapsto \varepsilon_2 u\). Then

\[
\begin{array}{c}
\text{(Igref)} & u \in T[G_1] & \langle G_1 \rangle \vdash G_1' \sim G_2 \\
\text{ref}^G_2 & \langle G_1 \rangle u \in T[\text{Ref } G_2]
\end{array}
\]

Then

\[
\text{ref}^G_2 \quad \langle G_1 \rangle u \mid \mu \rightarrow \langle \text{Ref } G_2 \rangle o^G_2; \text{Ref } G_2 \mid \mu[o^G_2 \mapsto \langle G_1 \rangle u :: G_2]
\]

where \(o \notin \text{dom}(\mu)\). But as \(\langle G_1 \rangle u :: G_2 \in T[G_2]\), then \(\langle \text{Ref } G_2 \rangle o^G_2; \text{Ref } G_2 \vdash \mu[o^G_2 \mapsto \langle G_1 \rangle u :: G_2]\). Also as \(\langle \text{Ref } G_2 \rangle \vdash \text{Ref } G_2 \sim \text{Ref } G_2\), \(\langle \text{Ref } G_2 \rangle o^G_2; \text{Ref } G_2 \in T[\text{Ref } G_2]\) and the result holds.

Case (r6). Then \(t^G = \varepsilon_1 o^G_1 := \varepsilon_3 \varepsilon_2 u\). Then

\[
\begin{array}{c}
\text{(Igasgn)} & o^G_1 \in T[\text{Ref } G_1] & \varepsilon_1 \vdash \text{Ref } G_1 \sim \text{Ref } G_3 \\
u \in T[G_2] & \varepsilon_2 \vdash G_2 \sim G_3
\end{array}
\]

If \(\varepsilon' = (\varepsilon_2 \cap \text{iref}(\varepsilon_1))\) is not defined, then \(t^G \rightarrow \text{error}\), and then the result hold immediately. Suppose \(\mu(o^G_1) = \varepsilon_3 u':: G_1\). Suppose \(z = m\) (the other case is similar to Prop 19) and \(\varepsilon'' = \varepsilon_2 \cap \text{iref}(\varepsilon_1) \cap \varepsilon_3\) is not defined, then \(t^G \rightarrow \text{error}\), and then the result hold immediately. Suppose that consistent transitivity does hold, then

\[
\varepsilon_1 o^G_1 := \varepsilon_3 \varepsilon_2 u \mid \mu \rightarrow \text{unit} \mid \mu[o^G_1 \mapsto \varepsilon'' u :: G_1]
\]

As \(\varepsilon_2 \vdash G_2 \sim G_3\) and by inversion lemma \(\text{iref}(\varepsilon_1) \vdash G_1 \sim G_3\), and as evidence is symmetrical \(\text{iref}(\varepsilon_1) \vdash G_3 \sim G_1\), then \(\varepsilon' \vdash G_2 \sim G_1\). Also \(\varepsilon_3 \vdash G_3 \sim G_1\) and \(\varepsilon_3 \vdash G_1 \sim G_2\), therefore \(\varepsilon'' \vdash G_2 \sim G_1\). Then \(\varepsilon'' = \varepsilon' \circ \varepsilon_3 \vdash G_2 \sim G_1\). Therefore \(\varepsilon'' u :: G_1 \in T[G_1]\), and therefore \(\text{unit} \vdash \mu[o^G_1 \mapsto \varepsilon'' u :: G_1]\). Also

\[
\theta(\text{unit}) = \text{Unit}
\]

\[
\text{unit} \in T[\text{Unit}]
\]

and the result holds.

□

Proposition 119 (\(\quad\rightarrow\) is well defined). If \(t^G \vdash \mu\) and \(t^G \mid \mu \rightarrow r\), then \(r \in \text{CONFIG}_G \cup \{\text{error}\}\), and if \(r = t^G \mid \mu'\), then also \(t^G \vdash \mu'\) and \(\text{dom}(\mu) \subseteq \text{dom}(\mu')\).
Proof. By induction on the structure of a derivation of \( t^G \rightarrow r \).
We proceed almost identical to \( \text{282} \) therefore we only illustrate main differences.

Case (RF). Let \( \text{EvTerm}_{G_2} \) be notation for the family of evidence terms \( \varepsilon t^{G_1} \) such that \( \varepsilon \vdash G_1 \sim G_2 \). Then \( t^{G_2} = F[et], F[et] \in T[G] \), and \( F : \text{EvTerm}_{G_2} \rightarrow T[G] \), and \( et \mid \mu \rightarrow c \). Let \( t_e = t_{e_2} \), and \( \mu ^{e_2} \). Then there exists \( G_e, G_x \) such that \( et = e_{et}^{G_e} \), and \( 
abla e \vdash G_e \sim G_x \). Also, \( t_e = \varepsilon_{et} : G_e \), with \( u \in T[G_e] \), and \( \varepsilon u \vdash G_v \sim G_e \).

We know that \( \varepsilon_{et}^{G_e} = \varepsilon u \) is defined, and \( et = \varepsilon_{et} t_e \mid \mu \rightarrow c \varepsilon u = et' \mid \mu ^{e_2} \). By definition of \( \varepsilon \) we have \( \varepsilon_{et} \vdash G_v \sim G_x \), so \( F[et] \in T[G] \).

As \( \text{freeLocs}(et) = \text{freeLocs}(et') \) to conclude that \( F[et] \vdash \mu \) we have to prove that if \( u = o^{G_5}_m \) and \( \mu (u) = \langle G' \rangle u' : G_5 \), then \( \mu ^{e_2}(u) = \langle G' \cap tref(G_1 \cap G_2) \rangle u' : G_5 \in T[G_5] \).

We know that \( \langle G_1 \rangle \vdash \text{Ref} \ G_5 \sim \ G_1 \vdash \ G_2 \), then \( \langle G_1 \rangle \vdash \text{Ref} \ G_5 \sim \ G_1 \vdash \ G_2 \). Then \( \langle G_1 \rangle \vdash \text{Ref} \ G_5 \sim \ G_1 \vdash \ G_2 \), therefore \( \langle \text{tref} \ G_1 \cap G_2 \rangle \vdash \ G_5 \sim \ G_5 \). Notice that references are invariant therefore \( \langle \text{tref} \ G_1 \cap G_2 \rangle \vdash \ G_u \sim \ G_5 \), therefore \( \langle G' \rangle \vdash \ G_u \sim \ G_5 \), then \( \langle G' \cap \text{tref} \ G_1 \cap G_2 \rangle \vdash \ G_u \sim \ G_5 \), and the result holds.

\[ \Box \]

\textbf{Proposition 120} (Dynamic guarantee for \( \rightarrow \)). Suppose \( \Omega \vdash t^{G_1}_1 \subseteq t^{G_2}_1 \) and \( \mu _1 \subseteq \mu _2 \). If \( t^{G_1}_1 \mid \mu _1 \rightarrow t^{G_1}_2 \mid \mu _1' \) then \( t^{G_2}_1 \mid \mu _2 \rightarrow t^{G_2}_2 \mid \mu _2' \), where \( \Omega ^{G_2}_1 \vdash t^{G_2}_1 \subseteq t^{G_2}_2 \), \( \mu _1' \subseteq \mu _2' \) for some \( \Omega ' \supseteq \Omega \).

Proof. By induction on reduction \( t^{G_1}_1 \mid \mu _1 \rightarrow t^{G_2}_1 \mid \mu _1' \). We proceed almost identical to \( \text{255} \) therefore we only illustrate main differences. For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

Case (r4). We know that \( t^{G_1}_1 = \text{ref}_{G_1}^{G_1} \varepsilon _1 u_1 \) where \( G_1 = \text{Ref} \ G'_1 \), then by \( ( \subseteq \text{REF} ) t^{G_2}_1 \) must have the form \( t^{G_2}_1 = \text{ref}_{G_2}^{G_2} \varepsilon _2 u_2 \) for some \( \varepsilon _2, u_2, G'_2 \) such that \( \varepsilon _1 \subseteq \varepsilon _2, u_1 \subseteq u_2, \) and \( G'_1 \subseteq G'_2 \).

Then \( t^{G_1}_1 \mid \mu _1 \rightarrow \langle \text{Ref} \ G'_1 \rangle o_{G'_1}^{G'_1} \subseteq \text{Ref} \ G'_1 \mid \mu _1 [o_{G'_1}^{G'_1} \mapsto \varepsilon _1 u_1 : G'_1] \).

Also, \( t^{G_2}_1 \mid \mu _2 \rightarrow \langle \text{Ref} \ G'_2 \rangle o_{G'_2}^{G'_2} \subseteq \text{Ref} \ G'_2 \mid \mu _2 [o_{G'_2}^{G'_2} \mapsto \varepsilon _2 u_2 : G'_2] \).

Then by \( ( \subseteq \cap ) \), \( \varepsilon _1 u_1 : G'_1 \subseteq \varepsilon _2 u_2 \subseteq G'_2 \), and then \( \mu _1 [o_{G'_1}^{G'_1} \mapsto \varepsilon _1 u_1 : G'_1] \subseteq \mu _2 [o_{G'_2}^{G'_2} \mapsto \varepsilon _2 u_2 : G'_2] \). Also by \( ( \subseteq \cap ) \), \( G'_1 \subseteq G'_2 \) and by \( ( \subseteq \cap ) \), \( \langle \text{Ref} \ G'_1 \rangle o_{G'_2}^{G'_2} \subseteq \text{Ref} \ G'_1 \subseteq \langle \text{Ref} \ G'_2 \rangle o_{G'_2}^{G'_2} \subseteq \text{Ref} \ G'_2 \) and the result holds.

Case (r6 where \( z = m \)). We know that \( t^{G_1}_1 = \varepsilon _{11} o_{z_{G_1}}^{G_1} := G_{12} \varepsilon _{12} u_1 \) where \( G_1 = \text{Unit} \), then by \( ( \subseteq \cap ) \) \( t^{G_2}_1 \) must have the form \( t^{G_2}_1 = \varepsilon _{21} o_{G_2}^{G_2} := G_{22} \varepsilon _{22} u_2 \) for some \( \varepsilon _{21}, \varepsilon _{22}, u_2, G_2, G_22 \) such that \( \varepsilon _{11} \subseteq \varepsilon _{21}, \varepsilon _{12} \subseteq \varepsilon _{22}, u_1 \subseteq u_2, G_{11} \subseteq G_{21}, G_{12} \subseteq G_{22} \).

Suppose \( \mu _1 (o_{z_{G_1}}^{G_1}) = \varepsilon _{11} u_1 : G_{11} \), and as \( \mu _1 \subseteq \mu _2 \), then \( \mu _2 (o_{G_2}^{G_2}) = \varepsilon _{22} u_2 : G_2 \), such that \( \varepsilon _{11} \subseteq \varepsilon _{21}, u_1 \subseteq u_2 \). Let us pose \( \varepsilon _1 = \varepsilon _{12} o = \varepsilon _{11} o = \text{iref} (\varepsilon _{11}) \).

Then \( t^{G_1}_1 \mid \mu _1 \rightarrow \text{unit} \mid \mu _1 [o_{z_{G_1}}^{G_1} \mapsto \varepsilon _1 u_1 : G_{11}] \).

By inspection of evidence and inversion lemma, as \( \varepsilon _{12} \subseteq \varepsilon _{21} \) then \( \text{iref} (\varepsilon _{12}) \subseteq \text{iref} (\varepsilon _{21}) \). Also, by \( \text{253} \) \( \varepsilon _2 = \varepsilon _{22} o = \varepsilon _{12} o = \text{iref} (\varepsilon _{21}) \) is defined and \( \varepsilon _1 \subseteq \varepsilon _2 \). Then, \( t^{G_2}_1 \mid \mu _2 \rightarrow \text{unit} \mid \mu _2 [o_{z_{G_2}}^{G_2} \mapsto \varepsilon _2 u_2 : G_{21}] \).

217
Then by $(\subseteq)$, $\varepsilon_1 u_1 :: G_{11} \subseteq \varepsilon_2 u_2 :: G_{21}$, and then $\mu_1[\sigma_{G_{21}} \mapsto \varepsilon_1 u_1 :: G_{11}] \sqsubseteq \mu_2[\sigma_{G_{21}} \mapsto \varepsilon_2 u_2 :: G_{21}]$ and the result holds.

\[ \square \]

**Proposition 121** (Dynamic guarantee). Suppose $t^{G_1}_1 \sqsubseteq t^{G_2}_1$ and $\mu_1 \sqsubseteq \mu_2$. Then if $\mu_1 \rightarrow t^{G_1}_1 | \mu'_1$ then $t^{G_2}_1 | \mu_2 \rightarrow t^{G_2}_2 | \mu'_2$ where $t^{G_1}_2 \sqsubseteq t^{G_2}_2$ and $\mu'_2 \sqsubseteq \mu_2$.

**Proof.** We prove the following property instead: Suppose $\Omega \vdash t^{G_1}_1 \sqsubseteq t^{G_2}_1$ and $\mu_1 \sqsubseteq \mu_2$. If $\mu_1 \rightarrow t^{G_1}_1 | \mu'_1$ then $t^{G_2}_1 | \mu_2 \rightarrow t^{G_2}_2 | \mu'_2$ where $\Omega' \vdash t^{G_1}_2 \sqsubseteq t^{G_2}_2$, and $\mu'_2 \sqsubseteq \mu_2'$ for some $\Omega' \supseteq \Omega$.

By induction on reduction $t^{G_1}_1 | \mu_1 \rightarrow t^{G_2}_1 | \mu'_1$. We proceed almost identical to 256, therefore we only illustrate main differences. For simplicity we omit the $\Omega \vdash$ notation on precision relations when it is not relevant for the argument.

---

**Case** $(\varepsilon_{12}(\varepsilon_{11} u_1 :: G_{11}) :: G_{12} | \mu_1 \mapsto \varepsilon'_{11} u_1 :: G_{12} | \mu'_1$, where $u_1 = \sigma_{u_1}^{G_{11}}$). By inspection of $(\subseteq)$ $t^{G_2} = \varepsilon_{22}(\varepsilon_{21} u_2 :: G_{21}) :: G_{22}$, where $\varepsilon_{11} \subseteq \varepsilon_{21}$, $\varepsilon_{12} \subseteq \varepsilon_{22}$, $G_{11} \sqsubseteq G_{21}$, $G_{12} \sqsubseteq G_{22}$, $u_1 \sqsubseteq u_2$. Therefore by $(\subseteq)$, $u_2 = \sigma_{u_2}^{G_{22}}$; for some $G_{u_2}$ such that $G_{u_1} \sqsubseteq G_{u_2}$. Suppose $\mu_1(u_1) = \varepsilon_{u_1} u'_1 :: G_{u_1}$, then as $\mu_1 \sqsubseteq \mu_2$, then $\mu_2(u_2) = \varepsilon_{u_2} u'_2 :: G_{u_2}$, where $\varepsilon_{u_1} \subseteq \varepsilon_{u_2}$, $u'_1 \subseteq u'_2$. If $\varepsilon'_{11} = \varepsilon_{11} \circ \varepsilon_{12}$ is defined, and $\varepsilon'_{12} = \text{iref}(\varepsilon'_{11}) \circ \varepsilon_{u_1}$ is defined, then $\mu'_1 = \mu_1[u_1 \mapsto \varepsilon'_{12} u'_1 :: G_{u_1}$. By Prop 253 $\varepsilon'_{21} = \varepsilon_{21} \circ \varepsilon_{22}$ is defined and $\varepsilon'_{21} \subseteq \varepsilon'_{22}$, $\varepsilon'_{22} = \text{iref}(\varepsilon'_{21}) \circ \varepsilon_{u_2}$ is also defined and $\varepsilon'_{12} \subseteq \varepsilon'_{22}$. Then $\varepsilon_{22}(\varepsilon_{21} u_2 :: G_{21}) :: G_{22} | \mu_2 \mapsto \varepsilon'_{21} u_2 :: G_{22} | \mu_2[u_2 \mapsto \varepsilon'_{22} u'_2 :: G_{u_2}]$, and the result holds directly by $(\subseteq_{REF})$.

\[ \square \]
Appendix B

Type-driven Gradual Security Typing

In this appendix we present additional definitions and proofs that were not included in the main body of §4.

B.1 Additional Definitions

B.1.1 SSL_{Ref}: Static Semantics

In this section we present additional definitions of the static semantics of SSL_{Ref}. The join between types and labels is defined as follows

\[
\begin{align*}
\text{Bool}_\ell \otimes \ell' &= \text{Bool}(\ell'') \\
(S_1 \xrightarrow{\ell_1 \to \ell_2} S_2) \otimes \ell' &= S_1 \xrightarrow{\ell''} (\ell'') S_2 \\
\text{Ref}_\ell S \otimes \ell' &= \text{Ref}(\ell'') S
\end{align*}
\]

Figure B.1 presents the join and meet type functions.

Definition 56 (Valid Type Sets).

\[
\begin{array}{c}
\text{valid}(\{\text{Bool}_\ell\}) \\
\text{valid}(\{S_{i1}\}) \quad \text{valid}(\{S_{i2}\}) \\
\text{valid}(\{S_{i1} \xrightarrow{\ell_{i1} \to \ell_{i2}} S_{i2}\}) \\
\text{valid}(\{\text{Ref}_\ell S_i\}) \\
\text{valid}(\{\text{Unit}_\ell\})
\end{array}
\]

B.1.2 SSL_{Ref}: Noninterference Definitions

In this section we present definitions and properties of noninterference for SSL_{Ref}. Figure B.2 presents the full definition of step-indexed logical relations.
\( \texttt{S} \lor \texttt{S}, \texttt{S} \land \texttt{S} \)

\[
\begin{align*}
\tilde{\vee} : \text{TYPE} \times \text{TYPE} & \rightarrow \text{TYPE} \\
\text{Bool}_r \tilde{\vee} \text{Bool}_r' & = \text{Bool}_r(\ell(\ell')) \\
(S_{11} \xrightarrow{\ell_r} t S_{12}) \tilde{\vee} (S_{21} \xrightarrow{\ell_r'} t S_{22}) & = (S_{11} \xrightarrow{\ell_r} t S_{21}) \tilde{\vee} (\ell(\ell')) (S_{12} \tilde{\vee} S_{22}) \\
\text{Ref}_\ell S \tilde{\lor} \text{Ref}_\ell' S & = \text{Ref}_r(\ell(\ell')) S \\
S \tilde{\lor} S & \text{ undefined otherwise}
\end{align*}
\]

\[
\begin{align*}
\tilde{\land} : \text{TYPE} \times \text{TYPE} & \rightarrow \text{TYPE} \\
\text{Bool}_r \land \text{Bool}_r' & = \text{Bool}_r(\ell(\ell')) \\
(S_{11} \xrightarrow{\ell_r} t S_{12}) \land (S_{21} \xrightarrow{\ell_r'} t S_{22}) & = (S_{11} \tilde{\lor} S_{21}) \xrightarrow{\ell(\ell')} (S_{12} \land S_{22}) \\
\text{Ref}_\ell S \mid \text{Ref}_\ell' S & = \text{Ref}_r(\ell(\ell')) S \\
S \land S & \text{ undefined otherwise}
\end{align*}
\]

Figure B.1: SSL\(_{\text{Ref}}\): Join and meet type functions

**Definition 57.** Let \( \sigma \) be a substitution, \( \Gamma \) and \( \Sigma \) a type substitutions. We say that substitution \( \sigma \) satisfy environment \( \Gamma \) and \( \Sigma \), written \( \sigma \models \Gamma; \Sigma \), if and only if \( \text{dom}(\sigma) = \Gamma \) and \( \forall x \in \text{dom}(\Gamma), \forall \ell_c, \Gamma; \Sigma; \ell_c \vdash \sigma(x) : S', \) where \( S' \prec S(\ell) \).

**Definition 58 (Related substitutions).** Tuples \( \langle \ell_1, \sigma_1, \mu_1 \rangle \) and \( \langle \ell_2, \sigma_2, \mu_2 \rangle \) are related on \( k \) steps, notation \( \Gamma; \Sigma \vdash \langle \ell_1, \sigma_1, \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_2, \sigma_2, \mu_2 \rangle \), if \( \sigma_1 \models \Gamma; \Sigma, \Sigma \vdash \langle \ell_1, \sigma_1(t), \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_2, \sigma_2(t), \mu_2 \rangle : \Gamma(x) \).

**Definition 59 (Semantic Security Typing).**

\[
\Gamma; \Sigma; \ell_c \vdash t : S \iff \forall ol \in \text{LABEL}, k \geq 0, \sigma_1, \sigma_2 \in \text{SUBST} \text{ and } \mu_1, \mu_2 \in \text{STORE} \text{ such that } \Sigma \vdash \mu_i \text{ and } \Gamma; \Sigma \vdash \langle \ell_c, \sigma_1, \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_c, \sigma_2, \mu_2 \rangle, \text{ we have } \\
\Sigma \vdash \langle \ell_c, \sigma_1(t), \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_c, \sigma_2(t), \mu_2 \rangle : C(S) \]

**Proposition 122 (Security Type Soundness).** If \( \Gamma; \Sigma; \ell_c \vdash t : S \) \( \implies \forall S, S' \prec S, \Gamma; \Sigma; \ell_c \vdash t : S \)
\( \Sigma \vdash (\ell_1, v_1, \mu_1) \approx^k_{ol} (\ell_2, v_2, \mu_2) : S \iff \ell_1 \approx_{ol} \ell_2 \land \Sigma \vdash \mu_1 \approx^k_{ol} \mu_2 \land \Sigma; \ell_1 \vdash v_1 : S'_1, S'_1 <: S, \) \\
\( \quad \land (\text{obs}_{ol}(\ell_1, S) \implies \Sigma \vdash \text{obsRel}_{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S) \)

\( \Sigma \vdash \text{obsRel}_{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S \iff (\text{rval}(v_1) = \text{rval}(v_2)) \)

if \( S \in \{ \text{Bool}_g, \text{Unit}_g, \text{Ref}_g S' \} \)

\( \Sigma \vdash \text{obsRel}_{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S \overset{\ell}{\rightarrow} S_2 \iff \\
\quad \forall j \leq k, \forall \Sigma \subseteq \Sigma', \Sigma' \vdash (\ell_1, v'_1, \mu'_1) \approx^j_{ol} (\ell_2, v'_2, \mu'_2) : S_1, \\
\quad \Sigma' \vdash (\ell_1, v_1, v'_1, \mu_1) \approx^j_{ol} (\ell_2, v_2, v'_2, \mu_2) : C(S_2 \uplus g) \)

\( \Sigma \vdash (\ell_1, t_1, \mu_1) \approx^k_{ol} (\ell_2, t_2, \mu_2) : C(S) \iff \ell_1 \approx_{ol} \ell_2 \land \Sigma \vdash \mu_1 \approx^k_{ol} \mu_2 \land \Sigma; \ell_1 \vdash t_1 : S'_1, S'_1 <: S, \forall j < k \\
\quad (t_1 | \mu_1 \overset{\ell_1}{\rightarrow} t'_1 | \mu'_1 \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu'_1 \approx^{k-j}_{ol} \mu'_2 \land \Sigma' \vdash (\ell_1, t'_1, \mu'_1) \approx^{k-j}_{ol} (\ell_2, t'_2, \mu'_2) : S) ) \)

\( \Sigma \vdash \mu_1 \approx^k_{ol} \mu_2 \iff \Sigma \vdash \mu_i \land \forall \ell_1, \ell_1 \approx_{ol} \ell_2, j < k, \forall o \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \\
\quad \Sigma \vdash (\ell_1, \mu_1(o), \mu_1) \approx^j_{ol} (\ell_2, \mu_2(o), \mu_2) : \Sigma(o) \)

\( \ell_1 \approx_{ol} \ell_2 \iff \text{obs}_{ol}(\ell_1) \lor \neg \text{obs}_{ol}(\ell_1) \)

\( \mu_1 \approx_{ol} \mu_2 \iff \text{dom}(\mu_1) \subseteq \text{dom}(\mu_2) \)

\( \text{obs}_{ol}(\ell, S) \iff \text{obs}_{ol}(\ell) \land \text{obs}_{ol}(\text{label}(S)) \)

\( \text{obs}_{ol}(\ell) \iff \ell \not\approx_{ol} \)

Figure B.2: Security logical relations
B.1.3 \textit{GSL}_{\text{Ref}}: Static Semantics

In this section we present some additional definitions needed in gradualizing SSL_{Ref}.

**Definition 60** (Type Concretization). \( \gamma_S : \text{GTYPE} \rightarrow \mathcal{P}(\text{TYPE}) \)

\[
\gamma_S(\text{Bool}_g) = \{ \text{Bool}_\ell \mid \ell \in \gamma(g) \} \quad \gamma_S(U_1 \rightarrow_{g'} U_2) = \gamma_S(U_1) \rightarrow_{(g')\gamma(g)} \gamma_S(U_2) \\
\gamma_S(\text{Unit}_g) = \{ \text{Unit}_\ell \mid \ell \in \gamma(g) \} \quad \gamma_S(\text{Ref}_g U) = \{ \text{Ref}_\ell S \mid \ell \in \gamma(g), S \in \gamma_S(U) \}
\]

Type concretization induces notions of precision and abstraction.

**Definition 61** (Type Abstraction). \( U_1 \sqsubseteq U_2 \), if and only if \( \gamma_S(U_1) \sqsubseteq \gamma_S(U_2) \).

**Definition 62** (Type Precision). \( \alpha_S : \mathcal{P}(\text{TYPE}) \rightarrow \text{GTYPE} \)

\[
\alpha_S(\{ \text{Bool}_\ell \}) = \text{Bool}_{\alpha(\{ \ell \})} \quad \alpha_S(\{ \text{Unit}_\ell \}) = \text{Unit}_{\alpha(\{ \ell \})} \\
\alpha_S(\{ S_{11} \rightarrow_{\ell_1} S_{12} \}) = \alpha_S(\{ S_{11} \})^{-1} \circ \alpha_S(\{ S_{12} \}) \\
\alpha_S(\{ S_{11} \}) \alpha_S(\{ S_{12} \}) = \alpha_S(\{ \text{Ref}_\ell S_{11} \}) \alpha_S(\{ S_{12} \})
\]

\( \alpha_S(S) \) is undefined otherwise.

**Proposition 123** \((\alpha_S \) is Sound and Optimal). Assuming \( \hat{S} \) valid:

(i) \( S \sqsubseteq \gamma_S(\alpha_S(S)) \)  
(ii) If \( \hat{S} \sqsubseteq \gamma_S(U) \) then \( \alpha_S(S) \sqsubseteq U \).

**Definition 63** (Gradual label meet).

\( g_1 \sqcap g_2 = \alpha(\{ \ell_1 \sqcap \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \}) \).

\[
\begin{array}{c}
\hat{\sqcap} : \text{TYPE} \times \text{TYPE} \rightarrow \text{TYPE} \\
\text{Bool}_g \sqcap \text{Bool}_{g'} = \text{Bool}_{(g \sqcap g')} \\
(U_{11} \rightarrow_{g} U_{12}) \sqcap (U_{21} \rightarrow_{g'} U_{22}) = (U_{11} \hat{\sqcap} U_{12}) \rightarrow_{(g \sqcap g')} (U_{21} \hat{\sqcap} U_{22}) \\
\text{Ref}_g U \sqcap \text{Ref}_{g'} U' = \text{Ref}_{(g \sqcap g')} U \sqcap U' \\
U \sqcap U \text{ undefined otherwise}
\end{array}
\]

Figure B.3: \textit{GSL}_{\text{Ref}}: consistent meet

**Definition 64** (Gradual label join).

\( g_1 \hat{\sqcup} g_2 = \alpha(\{ \ell_1 \sqcup \ell_2 \mid (\ell_1, \ell_2) \in \gamma(g_1) \times \gamma(g_2) \}) \)

**Definition 65** (Label Meet).

\( g_1 \sqcap g_2 = \alpha(\gamma(g_1) \sqcap \gamma(g_2)) \).

**Definition 66** (Type Meet).

\( U_1 \sqcap U_2 = \alpha_S(\gamma_S(U_1) \sqcap \gamma_S(U_2)) \).

222
(Ix) \[ \frac{x : U \in \Gamma}{\Gamma; \Sigma; \epsilon g_c \vdash x : U} \]

(II) \[ \frac{o : U \in \Sigma}{\Gamma; \Sigma; \epsilon g_c \vdash o_g : \text{Ref}_g U} \]

(Ia) \[ \frac{\Gamma, x : U_1; \Sigma; \epsilon g' \vdash t : U_2}{\epsilon = g' \prec (g' \epsilon)} \]

(Ib) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash b_g : \text{Bool}_g}{\Gamma; \Sigma; \epsilon g_c \vdash \text{unit}_g : \text{Unit}_g} \]

(Iprot) \[ \frac{\Gamma; \Sigma; \epsilon g' \vdash t : U'; \epsilon_1 \vdash U' \preceq U \quad \epsilon_2 \vdash g' \preceq g}{\Gamma; \Sigma; \epsilon g_c \vdash \text{prot}_{o \epsilon g'} \epsilon_2 \epsilon'(\epsilon_1) t : U \gamma g} \]

(Iapp) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash t_1 : U_1 \quad \epsilon_1 \vdash U_1 \preceq U_1 \quad \epsilon_2 \vdash U_2 \preceq U_2 \quad \epsilon_3 \vdash U_3 \preceq U_3}{\Gamma; \Sigma; \epsilon g_c \vdash \epsilon_1 t_1 \oplus \epsilon_2 t_2 : \text{Bool}_g \gamma g} \]

(II) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash \epsilon_1 t_1 : U_1}{\Gamma; \Sigma; \epsilon g_c \vdash \epsilon_1 t_1 \oplus \epsilon_2 t_2 : \text{Bool}_g \gamma g} \]

(Iref) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash t : U'}{\epsilon_1 \vdash U' \preceq U \quad \epsilon_2 \vdash g' \preceq \text{label}(U)} \]

(Ideref) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash t : U' \quad \epsilon' \vdash U' \preceq \text{Ref}_g U}{\Gamma; \Sigma; \epsilon g_c \vdash \epsilon' t : U \gamma g} \]

(Iassgn) \[ \frac{\Gamma; \Sigma; \epsilon g_c \vdash t_2 : U_2 \quad \epsilon_2 \vdash U_2 \preceq U_1 \quad \epsilon_3 \vdash g' \gamma g \preceq \text{label}(U_1)}{\Gamma; \Sigma; \epsilon g_c \vdash \epsilon_1 t_1 := \epsilon_3 \epsilon_2 t_2 : \text{Unit}_g} \]

Every type rule has the extra premise \( \epsilon \vdash g_c \preceq g_c' \).

Figure B.4: GSL\(^\varepsilon\)_Ref: Static Semantics

Also, we introduce a function \( \text{label} \), which yields the security label of a given type:

\[ \text{label} : \text{GType} \to \text{LABEL} \]

\[ \text{label}(\text{Bool}_g) = g \quad \text{label}(\text{Unit}_g) = g \quad \text{label}(U_1 \to_g U_2) = g \quad \text{label}(\text{Ref}_g U) = g \]

B.1.4 GSL\(^\varepsilon\)_Ref: Static Semantics

The static semantics of GSL\(^\varepsilon\)_Ref is presented in Figure B.4.
\begin{align*}
\langle t_1, t_2 \rangle \cap \langle t'_1, t'_2 \rangle &= \langle t_1 \cap t'_1, t_2 \cap t'_2 \rangle \\
\langle t_1, t_2 \rangle \cap \langle t'_1, t'_2 \rangle &= \langle t_1 \cap t'_1, t_2 \cap t'_2 \rangle
\end{align*}

Figure B.5: GSL_{Ref}^\varepsilon: Auxiliary functions for the dynamic semantics (Labels)

**B.1.5 GSL_{Ref}^\varepsilon: Dynamic Semantics**

In this section we present additional definition of the dynamic semantics of GSL_{Ref}^\varepsilon. Auxiliary functions for evidence for labels are presented in Figure B.5. Auxiliary functions for evidence for types are shown in Figure B.6, and the inversion functions for evidence are in Figure B.7.

**Definition 67 (Type Evidence Concretization).** Let $\gamma_E : \text{GETYPE} \to \mathcal{P}(\text{TYPE})$ be defined as follows:

\begin{align*}
\gamma_E(\text{Bool}_i) &= \{ \text{Bool}_I \mid I \in \gamma_i(t) \} \\
\gamma_E(E_1 \xrightarrow{t_2} t_1 E_2) &= \gamma_E(E_1)^{\gamma(t_2)} \gamma_i(t_1) \gamma_E(E_2) \\
\gamma_E(\text{Ref}_I E) &= \{ \text{Ref}_I S \mid I \in \gamma_i(t), S \in \gamma_E(E) \}
\end{align*}

where $\to$ is the set of all possible combinations of function types, using each member of the sets obtained by the $\gamma_E$ and $\gamma_i$ functions.

**Definition 68 (Evidence Concretization).** Let $\gamma_{\varepsilon_I} : \text{GETYPE}^2 \to \mathcal{P}(\text{TYPE}^2)$ be defined as follows:

\[ \gamma_{\varepsilon_I}(\langle E_1, E_2 \rangle) = \{ \langle S_1, S_2 \rangle \mid S_1 \in \gamma_E(E_1), S_2 \in \gamma_E(E_2) \} \]

**Definition 69 (Type Evidence Abstraction).** Let the abstraction function $\alpha_E : \mathcal{P}(\text{TYPE}) \to \text{GETYPE}$ be defined as:

\begin{align*}
\alpha_E(\{ \text{Bool}_I \}) &= \text{Bool}_{\alpha_i(I)} \\
\alpha_E(\{ S_{1i} \xrightarrow{t_2} \text{Ref}_I S_{12} \}) &= \alpha_E(\{ S_{1i} \})^{\alpha_i(I)} \alpha_i(\{ S_{12} \}) \\
\alpha_E(\{ \text{Ref}_I S_i \}) &= \text{Ref}_{\alpha_i(I)}(\{ S_i \}) \\
\alpha_E(S) &\text{ is undefined otherwise}
\end{align*}

**Definition 70 (Evidence Abstraction).** Let $\alpha_\varepsilon : \mathcal{P}(\text{TYPE}^2) \to \text{GETYPE}^2$ be defined as follows:

\[ \alpha_\varepsilon(\emptyset) \text{ is undefined} \]

\[ \alpha_\varepsilon(\{ \langle S_{1i}, S_{2i} \rangle \}) = \langle \alpha_E(\{ S_{1i} \}), \alpha_E(\{ S_{2i} \}) \rangle \text{ otherwise} \]

**Proposition 124** ($\alpha_i$ is Sound). If $\hat{I}$ is not empty, then $\hat{I} \subseteq \gamma_i(\alpha_i(\hat{I}))$.

**Proposition 125** ($\alpha_i$ is Optimal). If $\hat{I}$ is not empty, and $\hat{I} \subseteq \gamma_i(t)$ then $\alpha_i(\hat{I}) \subseteq t$.
\( \text{Bool}_1 \cap \text{Bool}_{i'} = \text{Bool}_{i \cap i'} \)

\[
(E_{11} \rightarrow_{\gamma_i} E_{12}) \cap (E_{21} \rightarrow_{\gamma_i'} E_{22}) = (E_{11} \cap E_{21}) \rightarrow_{\gamma_i \cap \gamma_i'} (E_{12} \cap E_{22})
\]

\( E \cap E' \text{ undefined otherwise} \)

\[
\text{Bool}_1 \, \widetilde{\gamma} \, \iota_2 = \text{Bool}_{(i_1 \, \widetilde{\gamma} \, \iota_2)} \quad E_1 \rightarrow_{\gamma_i} E_2 \, \widetilde{\gamma} \, \iota_3 = E_1 \rightarrow_{\gamma_i} ((i_1 \, \widetilde{\gamma} \, \iota_3) \, \text{Ref}_1) \quad \text{Ref}_1 \, E \, \widetilde{\gamma} \, \iota_2 = \text{Ref}_{(i_1 \, \widetilde{\gamma} \, \iota_2)} \, E
\]

\[
\text{Bool}_1 \, \widetilde{\alpha} \, \iota_2 = \text{Bool}_{(i_1 \, \widetilde{\alpha} \, \iota_2)} \quad E_1 \rightarrow_{\alpha_i} E_2 \, \widetilde{\alpha} \, \iota_3 = E_1 \rightarrow_{\alpha_i} ((i_1 \, \widetilde{\alpha} \, \iota_3) \, \text{Ref}_1) \quad \text{Ref}_1 \, E \, \widetilde{\alpha} \, \iota_2 = \text{Ref}_{(i_1 \, \widetilde{\alpha} \, \iota_2)} \, E
\]

\[
\langle E_1, E_2 \rangle \, \widetilde{\gamma} \, \iota (i_1, i_2) = \langle E_1 \, \widetilde{\gamma} \, \iota_1, E_2 \, \widetilde{\gamma} \, \iota_2 \rangle \quad \langle E_1, E_2 \rangle \, \widetilde{\alpha} \, \iota (i_1, i_2) = \langle E_1 \, \widetilde{\alpha} \, \iota_1, E_2 \, \widetilde{\alpha} \, \iota_2 \rangle
\]

\[
\Delta^{\prec}(i_1, i_2, i_3) = \langle i'_1, i'_3 \rangle
\]

\[
\Delta^{\prec}( \text{Bool}_{i_1}, \text{Bool}_{i_2}, \text{Bool}_{i_3} ) = \langle \text{Bool}_{i'_1}, \text{Bool}_{i'_3} \rangle
\]

\[
\Delta^{\prec}(E_{31}, E_{21}, E_{11}) = \langle E'_{11}, E'_{11} \rangle \quad \Delta^{\prec}(E_{12}, E_{22}, E_{32}) = \langle E'_{12}, E'_{32} \rangle
\]

\[
\Delta^{\prec}(i_1, i_2, i_3) = \langle i'_1, i'_3 \rangle \quad \Delta^{\prec}(i_1, i_2, i_11) = \langle i'_1, i'_11 \rangle
\]

\[
\Delta^{\prec}(E_{11} \rightarrow_{\gamma_i} E_{12}, E_{21} \rightarrow_{\gamma_i'} E_{22}, E_{31} \rightarrow_{\gamma_i} E_{32}) = \langle E'_{11} \rightarrow_{\gamma_i} E'_{12}, E'_{31} \rightarrow_{\gamma_i'} E'_{32} \rangle
\]

\[
\Delta^{\prec}(\text{Ref}_1 \, E_1, \text{Ref}_2 \, E_2, \text{Ref}_3 \, E_3) = \langle \text{Ref}_{i'_1} \, E'_{1}, \text{Ref}_{i'_3} \, E'_{3} \rangle
\]

\[
\langle E_1, E_{21} \rangle \circ^{\prec} (E_{22}, E_3) = \Delta^{\prec}(E_1, E_{21} \cap E_{22}, E_3)
\]

Figure B.6: GSL\textsubscript{Ref}^{\prec}: Auxiliary functions for the dynamic semantics (Types)
\[ \text{Proposition 126} \ (\alpha_E \text{ is Sound}). \text{ If } \text{valid}(\mathcal{S}) \text{ then } \mathcal{S} \subseteq \gamma_E(\alpha_E(\mathcal{S})). \]

\[ \text{Proposition 127} \ (\alpha_E \text{ is Optimal}). \text{ If } \text{valid}(\mathcal{S}) \text{ and } \mathcal{S} \subseteq \gamma_E(E) \text{ then } \alpha_E(\mathcal{S}) \subseteq E. \]

With concretization of security type, we can now define security type precision.

**Definition 71** (Interval and Type Evidence Precision).

1. \( t_1 \) is less imprecise than \( t_2 \), notation \( t_1 \sqsubseteq t_2 \), if and only if \( \gamma_{\ell_1}(t_1) \subseteq \gamma_{\ell_2}(t_2) \); inductively:

   \[
   \frac{\ell_3 \preceq \ell_1 \quad \ell_2 \preceq \ell_4}{[\ell_1, \ell_2] \subseteq [\ell_3, \ell_4]}
   \]

2. \( E_1 \) is less imprecise than \( E_2 \), notation \( E_1 \sqsubseteq E_2 \), if and only if \( \gamma_E(E_1) \subseteq \gamma_E(E_2) \); inductively:

   \[
   \frac{t_1 \sqsubseteq t_2}{\text{Bool}_1 \sqsubseteq \text{ Bool}_2} \quad \frac{E_{11} \sqsubseteq E_{21}}{E_{11} \xrightarrow{t_1} E_{12} \sqsubseteq E_{21} \xrightarrow{t_2} E_{22}} \quad \frac{t_1 \sqsubseteq t_2}{\text{Ref}_{11} \sqsubseteq \text{ Ref}_{12}} \]

Figure B.7: GSL\(_{Ref}\): Inversion functions for evidence.
\[
\begin{align*}
\text{bounds}(?) &= [\bot, \top] \\
\text{bounds}(\ell) &= [\ell, \ell] \\
\text{bounds}(x_1 \searrow x_2) &= \text{bounds}(x_1) \searrow \text{bounds}(x_2) \\
\text{bounds}(x_1 \land x_2) &= \text{bounds}(x_1) \land \text{bounds}(x_2) \\
\text{bounds}(x_1 \cap x_2) &= \text{bounds}(x_1) \cap \text{bounds}(x_2) \\
\text{bounds}(F_1(x_i) \searrow F_2(x_i)) &= \text{bounds}(F_1(x_i)) \searrow \text{bounds}(F_2(x_i)) \\
\text{bounds}(F_1(x_i) \land F_2(x_i)) &= \text{bounds}(F_1(x_i)) \land \text{bounds}(F_2(x_i)) \\
\text{bounds}(F_1(x_i) \cap F_2(x_i)) &= \text{bounds}(F_1(x_i)) \cap \text{bounds}(F_2(x_i)) \\
\end{align*}
\]

\[
\begin{align*}
\text{bounds}(F_1(g_i)) &= [\ell_1, \ell_2] \\
\text{bounds}(F_2(g_j)) &= [\ell'_1, \ell'_2] \\
\end{align*}
\]

where \( F_1 : \text{GLABEL}^n \rightarrow \text{GLABEL} \) and \( F_2 : \text{GLABEL}^m \rightarrow \text{GLABEL} \).

\[
\begin{align*}
g(F(g_1, \ldots, g_n) \leq F_2(g_{n+1}, \ldots, g_{n+m})) &= ([\ell_1, \ell_2], [\ell_1', \ell_2']) \\
\end{align*}
\]

Figure B.8: GSL\textsuperscript{\textepsilon}_\text{Ref}: Initial evidence for gradual labels

### B.1.6 GSL\textsubscript{Ref}: Translation to GSL\textsuperscript{\textepsilon}_\text{Ref}

Figure B.8 presents the initial evidence function for consistent label ordering. The initial evidence function for consistent subtyping is presented in Figure B.9 using the following definition of operation pattern:

**Definition 72** (Operation pattern).

\[
P^T \in \text{GPattern}, P^t \in \text{LPattern} \\
P^T ::= \_ | P^T \circ P^T \text{ (pattern on types)} \\
op^T ::= \lor | \land | \cap \text{ (operations on types)} \\
P^t ::= \_ | P^t \circ P^t \text{ (pattern on labels)} \\
op^t ::= \lor | \land | \cap \text{ (operations on labels)}
\]
\[\text{lift}(\_ \_ \_ ) = - \]
\[\text{lift}(P_1^T \uplus P_2^T) = \text{lift}(P_1^T) \lor \text{lift}(P_2^T) \]
\[\text{lift}(P_1^T \land P_2^T) = \text{lift}(P_1^T) \land \text{lift}(P_2^T) \]
\[\text{lift}(P_1^T \land P_2^T) = \text{lift}(P_1^T) \land \text{lift}(P_2^T) \]

\[\text{invert}(\_ \_ \_ ) = - \]
\[\text{invert}(P_1^T \uplus P_2^T) = \text{invert}(P_1^T) \lor \text{invert}(P_2^T) \]
\[\text{invert}(P_1^T \land P_2^T) = \text{invert}(P_1^T) \lor \text{invert}(P_2^T) \]
\[\text{tomeet}(\_ \_ \_ ) = - \]
\[\text{tomeet}(P_1^T \uplus P_2^T) = \text{tomeet}(P_1^T) \lor \text{tomeet}(P_2^T) \]
\[\text{tomeet}(P_1^T \land P_2^T) = \text{tomeet}(P_1^T) \lor \text{tomeet}(P_2^T) \]

\[\mathcal{G}[\text{lift}(P(G_1)(\overline{t_1}) <: \text{lift}(P(G_2)(\overline{t_2}))] = \langle \iota_1, \iota_2 \rangle \]

\[\mathcal{G}[\text{lift}((\overline{g_1} \circ \overline{g_2})) <: \text{lift}(\overline{g_1} \circ \overline{g_2})] = \langle \iota_1, \iota_2 \rangle \]

\[\mathcal{G}[\text{invert}(G_1)(\overline{t_1}) <: \text{invert}(G_1)(\overline{t_1})] = \langle E_{11}, E_{12} \rangle \]
\[\mathcal{G}[\text{lift}(P(G_1)(\overline{t_1}) <: \text{lift}(P(G_2)(\overline{t_2}))] = \langle \iota_1, \iota_2 \rangle \]

\[\mathcal{G}[\text{tomeet}(G_1)(\overline{t_1}) <: \text{tomeet}(G_2)(\overline{t_2})] = \langle E_1, E_2 \rangle \]

\[\mathcal{G}[\text{tomeet}(G_1)(\overline{t_1}) <: \text{tomeet}(G_2)(\overline{t_2})] = \langle E_2, E_3 \rangle \]

\[\mathcal{G}[\text{Ref}_1(\overline{U_1}) <: \text{Ref}_2(\overline{U_2})] = \langle \text{Ref}_1, \text{Ref}_2 \rangle \]

where \( G_1 : \text{GLABEL}^n \rightarrow \text{GLABEL} \) and \( G_2 : \text{GLABEL}^m \rightarrow \text{GLABEL} \), and \( G_1(x_1, \ldots, x_n) = P_1^T(x_1, \ldots, x_n), G_2(x_1, \ldots, x_n) = P_2^T(x_1, \ldots, x_m) \).
B.2 Static Security Typing with References

In this section we present the proof of type preservation for SSL\textsubscript{Ref} in Sec. B.2.1 and the definitions and proof of noninterference for SSL\textsubscript{Ref} in Sec. B.2.2.

B.2.1 SSL\textsubscript{Ref}: Static type safety

In this section we present the proof of type safety for SSL\textsubscript{Ref}.

**Definition 73** (Well typeness of the store). A store $\mu$ is said to be well typed with respect to a typing context $\Gamma$ and a store typing $\Sigma$, written $\Gamma; \Sigma \vdash \mu$, if $\text{dom}(\mu) = \text{dom}(\Sigma)$ and $\forall o \in \text{dom}(\mu), \Gamma; \Sigma; \bot \vdash o : S$ and $S \triangleleft \Sigma(o)$.

**Lemma 128.** If $\Gamma; \Sigma; \ell_c \vdash t : S$ then $\forall \ell'_c \preceq \ell_c, \Gamma; \Sigma; \ell'_c \vdash t : S$.

**Proof.** By induction on the derivation of $\Gamma; \Sigma; \ell_c \vdash t : S$. Noticing that none of the inferred types of the type rules depend on $\ell_c$.

Case (Sx, Sb, Su, Sl). Trivial because neither the premises and the inferred type depend on the security effect.

Case ($S\oplus$). Then $t = b_{1\ell_1} \oplus b_{2\ell_2}$ and

\[
\begin{align*}
\text{(Sb)} & \quad \Gamma; \Sigma; \ell_c \vdash b_{1\ell_1} : \text{Bool}_{\ell_1} \\
\text{(Sb)} & \quad \Gamma; \Sigma; \ell_c \vdash b_{2\ell_2} : \text{Bool}_{\ell_2} \\
\text{(S\oplus)} & \quad \Gamma; \Sigma; \ell_c \vdash b_{1\ell_1} \oplus b_{2\ell_2} : \text{Bool}_{(\ell_1 \succ \ell_2)}
\end{align*}
\]

Suppose $\ell'_c$ such that $\ell'_c \preceq \ell_c$, then by induction hypotheses on the premises:

\[
\begin{align*}
\text{(Sb)} & \quad \Gamma; \Sigma; \ell'_c \vdash b_{1\ell_1} : \text{Bool}_{\ell_1} \\
\text{(Sb)} & \quad \Gamma; \Sigma; \ell'_c \vdash b_{2\ell_2} : \text{Bool}_{\ell_2} \\
\text{(S\oplus)} & \quad \Gamma; \Sigma; \ell'_c \vdash b_{1\ell_1} \oplus b_{2\ell_2} : \text{Bool}_{(\ell'_1 \succ \ell'_2)}
\end{align*}
\]

where $\ell'_1 = \ell_1$ and $\ell'_2 = \ell_2$ and the result holds.

Case (Sprot). Then $t = \text{prot}_\ell(t)$ and

\[
\begin{align*}
\text{(Sprot)} & \quad \Gamma; \Sigma; \ell_c \uppee \ell \vdash t : S \\
\quad & \quad \Gamma; \Sigma; \ell_c \vdash \text{prot}_\ell(t) : S \uppee \ell
\end{align*}
\]

Suppose $\ell'_c$ such that $\ell'_c \preceq \ell_c$. Considering that $\ell'_c \uppee \ell \preceq \ell_c \uppee \ell$, then by induction hypotheses on the premise:

\[
\begin{align*}
\text{(Sprot)} & \quad \Gamma; \Sigma; \ell'_c \uppee \ell \vdash t : S \\
\quad & \quad \Gamma; \Sigma; \ell'_c \vdash \text{prot}_\ell(t) : S \uppee \ell
\end{align*}
\]

and therefore the result holds.
Case (Sapp). Then \( t = t_1 \cdot t_2 \) and

\[
\begin{align*}
\text{(Sapp)} & \quad \frac{D_1}{\Gamma; \Sigma; \ell_c \vdash t_1 : S_1 \xrightarrow{e} S_{12}} \\
\frac{D_2}{\Gamma; \Sigma; \ell_c \vdash t_2 : S_2 \quad \ell_c \gamma \ell \preceq \ell''_c \quad S_2 < : S_{11}}{
\Gamma; \Sigma; \ell_c \vdash t_1 \cdot t_2 : S_{12} \gamma \ell}
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \preceq \ell_c \). Then by using induction hypotheses on the premises, considering \( S'_{11} \xrightarrow{e} e S'_{12} < : S_{11} \xrightarrow{e} e S_{12} \) and \( S'_2 < : S_2 \). As \( S_2 < : S_{11} \) and \( S_{11} < : S'_{11} \) then \( S'_2 < : S'_{11} \). Also, by definition of the join operator \( \ell_c \gamma \ell \preceq \ell_c \gamma \ell \preceq \ell''_c \preceq \ell''_c \), and then:

\[
\begin{align*}
\text{(Sapp)} & \quad \frac{D_1}{\Gamma; \Sigma; \ell'_c \vdash t_1 : S'_{11} \xrightarrow{e} e S'_{12}} \\
\frac{D_2}{\Gamma; \Sigma; \ell'_c \vdash t_2 : S'_2 \quad \ell'_c \gamma \ell' \preceq \ell''_c \quad S'_2 < : S'_{11}}{
\Gamma; \Sigma; \ell'_c \vdash t_1 \cdot t_2 : S'_{12} \gamma \ell'}
\end{align*}
\]

Where \( S'_{12} \gamma \ell' = S_2 \gamma \ell \) and the result holds.

Case (Sif-true). Then \( t = \text{if true}_\ell \text{ then } t_1 \text{ else } t_2 \) and

\[
\begin{align*}
\text{(Sif)} & \quad \frac{D_0}{\Gamma; \Sigma; \ell_c \vdash \text{true}_\ell : \text{Bool}_\ell} \\
\frac{D_1}{\Gamma; \Sigma; \ell_c \gamma \ell \vdash t_1 : S_1} \\
\frac{D_2}{\Gamma; \Sigma; \ell_c \gamma \ell \vdash t_2 : S_2}{
\Gamma; \Sigma; \ell_c \vdash \text{if true}_\ell \text{ then } t_1 \text{ else } t_2 : (S_1 \lor S_2) \gamma \ell}
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \preceq \ell_c \). As \( \ell'_c \gamma \ell \preceq \ell_c \gamma \ell \), by induction hypotheses in the premises:

\[
\begin{align*}
\text{(Sif)} & \quad \frac{D_0}{\Gamma; \Sigma; \ell'_c \vdash \text{true}_\ell : \text{Bool}_\ell} \\
\frac{D_1}{\Gamma; \Sigma; \ell'_c \gamma \ell \vdash t_1 : S'_1} \\
\frac{D_2}{\Gamma; \Sigma; \ell'_c \gamma \ell \vdash t_2 : S'_2}{
\Gamma; \Sigma; \ell'_c \vdash \text{if true}_\ell \text{ then } t_1 \text{ else } t_2 : (S'_1 \lor S'_2) \gamma \ell}
\end{align*}
\]

where \( S'_1 = S_1, S'_2 = S_2 \). Then \( (S'_1 \lor S'_2) \gamma \ell = (S_1 \lor S_2) \gamma \ell \) and therefore the result holds.

Case (Sif-false). Analogous to case (if-true).

Case (Sref). Then \( t = \text{ref}^S v \) and

\[
\begin{align*}
\text{(Sref)} & \quad \frac{\Gamma; \Sigma; \ell_c \vdash v : S' \quad S' < : S \quad \ell_c \preceq \text{label}(S)}{\Gamma; \Sigma; \ell_c \vdash \text{ref}^S v : \text{Ref}_\bot S}
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \preceq \ell_c \). By using induction hypotheses in the premise, considering \( \ell'_c \preceq \ell_c \preceq \text{label}(S) \):

\[
\begin{align*}
\text{(Sref)} & \quad \frac{\Gamma; \Sigma; \ell'_c \vdash v : S' \quad S' < : S \quad \ell'_c \preceq \text{label}(S)}{\Gamma; \Sigma; \ell'_c \vdash \text{ref}^S v : \text{Ref}_\bot S}
\end{align*}
\]

and the result holds.
Case (Sderef). Then \( t = \texttt{!}o : \ell \) and

\[
\begin{align*}
\text{(Sl)} & \quad o : S \in \Sigma \quad \ell \in \ell_c \\
\Gamma; \Sigma; \ell_c \vdash o : \texttt{Ref}_\ell S \\
\Gamma; \Sigma; \ell_c \vdash \texttt{!}o : \texttt{S} \gamma \ell
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \not\ll \ell_c \), then by using induction hypotheses in the premise:

\[
\begin{align*}
\text{(Sl)} & \quad o : S \in \Sigma \\
\Gamma; \Sigma; \ell'_c \vdash o : \texttt{Ref}_{\ell'} S \\
\Gamma; \Sigma; \ell'_c \vdash \texttt{!}o : S \gamma \ell'
\end{align*}
\]

where \( \ell' = \ell \) and the result holds.

Case (Sassgn). Then \( t = o : \ell := v \) and

\[
\begin{align*}
\text{(Sass)} & \quad o : S \in \Sigma \\
\Gamma; \Sigma; \ell_c \vdash o : \texttt{Ref}_\ell S \\
\Gamma; \Sigma; \ell_c \vdash v : S_2 \\
\frac{\ell \gamma \ell \not\ll \texttt{label}(S) \quad \ell \gamma \ell \not\ll \texttt{label}(S)}{\Gamma; \Sigma; \ell_c \vdash o : \texttt{Unit}_\bot}
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \not\ll \ell_c \). Considering that \( \ell'_c \gamma \ell \not\ll \ell_c \gamma \ell \not\ll \texttt{label}(S) \), and \( S'_2 \ll S_2 \ll S \), then:

\[
\begin{align*}
\text{(Sass)} & \quad o : S \in \Sigma \\
\Gamma; \Sigma; \ell'_c \vdash o : \texttt{Ref}_{\ell'} S \\
\Gamma; \Sigma; \ell'_c \vdash v : S'_2 \\
\frac{\ell' \gamma \ell \not\ll \texttt{label}(S) \quad \ell' \gamma \ell \not\ll \texttt{label}(S)}{\Gamma; \Sigma; \ell'_c \vdash o : \texttt{Unit}_\bot}
\end{align*}
\]

but

\[
\texttt{Unit}_\bot \ll \texttt{Unit}_\bot
\]

and therefore the result holds.

Case (S::). Then \( t = v :: S \) and

\[
\begin{align*}
\text{(S::)} & \quad \Gamma; \Sigma; \ell_c \vdash v : S_1 \\
\frac{S_1 \ll S \quad \text{Lemma 130}}{\Gamma; \Sigma; \ell_c \vdash v :: S : S}
\end{align*}
\]

Suppose \( \ell'_c \) such that \( \ell'_c \not\ll \ell_c \), then by Lemma 130

\[
\begin{align*}
\text{(S::)} & \quad \Gamma; \Sigma; \ell'_c \vdash v : S_1 \\
\frac{S_1 \ll S \quad \text{Lemma 130}}{\Gamma; \Sigma; \ell'_c \vdash v :: S : S}
\end{align*}
\]

and the result holds.

\[ \square \]

Lemma 129 (Substitution). If \( \Gamma, x : S_1; \Sigma; \ell_c \vdash t : S \) and \( \Gamma; \Sigma; \ell_c \vdash v : S'_1 \) such that \( S'_1 \ll S_1 \), then \( \Gamma; \Sigma; \ell_c \vdash [v/x]t : S' \) such that \( S' \ll S \).
Proof. By induction on the derivation of $\Gamma, x : S_1; \Sigma; \ell_c \vdash t : S$. \qed

Lemma 130. If $\Gamma; \Sigma; \ell_c \vdash v : S$ then $\forall \ell_c', \Gamma; \Sigma; \ell_c' \vdash v : S$.

Proof. By induction on the derivation of $\Gamma; \Sigma; \ell_c \vdash v : S$ observing that for values, there is no premise that depends on $\ell_c$. \qed

Proposition 131 (is well defined). If $\varnothing; \Sigma; \ell_c \vdash t : S$, $\varnothing; \Sigma \vdash \mu$ and $\forall \ell_r$, such that $\ell_r \preceq \ell_c$, $t \mid \mu \xrightarrow{\ell_r} t' \mid \mu'$ then, for some $\Sigma' \supseteq \Sigma$, $\varnothing; \Sigma'; \ell_c \vdash t' : S'$, where $S' <: S$ and $\varnothing; \Sigma' \vdash \mu'$.

Proof.

Case (S⊕). Then $t = b_{1\ell_1} \oplus b_{2\ell_2}$ and

$$
\begin{array}{c}
(Sb) \quad \varnothing; \Sigma; \ell_c \vdash b_{1\ell_1} : \text{Bool}_{\ell_1} \\
(Sb) \quad \varnothing; \Sigma; \ell_c \vdash b_{2\ell_2} : \text{Bool}_{\ell_2} \\
(S\oplus) \quad \varnothing; \Sigma; \ell_c \vdash b_{1\ell_1} \oplus b_{2\ell_2} : \text{Bool}_{(\ell_1 \gamma \ell_2)}
\end{array}
$$

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

$$
\begin{array}{c}
\ell_r \quad b_{1\ell_1} \oplus b_{2\ell_2} \mid \mu \\
= \quad (b_1 \mkern1mu \oplus \mkern1mu b_2)_{(\ell_1 \gamma \ell_2)} \mid \mu
\end{array}
$$

Then

$$
\begin{array}{c}
(S\oplus) \quad \ell_c \vdash (b_1 \mkern1mu \oplus \mkern1mu b_2)_{(\ell_1 \gamma \ell_2)} : \text{Bool}_{(\ell_1 \gamma \ell_2)}
\end{array}
$$

Case (Sprot). Then $t = \text{prot}_\ell(v)$ and

$$
\begin{array}{c}
(S\text{prot}) \quad \varnothing; \Sigma; \ell_c \gamma \ell \vdash v : S \\
\varnothing; \Sigma; \ell_c \vdash \text{prot}_\ell(v) : S \gamma \ell
\end{array}
$$

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

$$
\text{prot}_\ell(v) \mid \mu \xrightarrow{\ell_r} v \gamma \ell \mid \mu
$$

But by Lemma 128, $\varnothing; \Sigma; \ell_c \vdash v : S$.

$$
\begin{array}{c}
\varnothing; \Sigma; \ell_c \vdash v \gamma \ell : S \gamma \ell
\end{array}
$$

and the result holds.

Case (Sapp). Then $t = (\lambda^\ell x : S_{11}.t)_{\ell} v$ and

$$
\begin{array}{c}
(S\lambda) \quad \varnothing, x : S_{11}; \Sigma; \ell' \vdash t : S_{12} \\
\varnothing; \Sigma; \ell_c \vdash (\lambda^\ell x : S_{11}.t)_{\ell} : S_{11} \xrightarrow{\ell'} t_{S_{12}} \\
(S\text{app}) \quad \varnothing; \Sigma; \ell_c \vdash v : S_2 \\
\ell_c \gamma \ell \preceq \ell'_c \\
S_2 <: S_{11} \\
\varnothing; \Sigma; \ell_c \vdash (\lambda^\ell x : S_{11}.t)_{\ell} v : S_{12} \gamma \ell
\end{array}
$$

232
Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, and

$$(\lambda'x : S_{11}, t)_{\ell_r} v \mid \mu \xrightarrow{\ell_r} \text{prot}_{\ell_r}([v/x]t) \mid \mu$$

But as $\ell_c \parallel \ell \preceq \ell' c$ then by Lemma 128, $\emptyset; \Sigma; \ell_r \parallel t : S'_{12}$, where $S'_{12} <: S_{12}$.

By Lemma 228 and Lemma 130 $\emptyset; \Sigma; \ell_r \parallel [v/x] t : S''_{12}$, where $S''_{12} <: S'_{12} <: S_{12}$. Then

$$(Sprot) \frac{\emptyset; \Sigma; \ell_r \parallel [v/x] t : S''_{12}}{\emptyset; \Sigma; \ell_r \parallel \text{prot}_{\ell_r}([v/x] t) : S''_{12} \parallel \ell}$$

Where $S''_{12} \parallel \ell <: S_{12} \parallel \ell$ and the result holds.

Case $(\text{Sif-true})$. Then $t = \text{if true}_\ell$ then $t_1$ else $t_2$ and

$$(\text{Sif}) \frac{D_0}{\emptyset; \Sigma; \ell_r \parallel \text{true}_\ell : \text{Bool}_{\ell}} \quad \frac{D_1}{\emptyset; \Sigma; \ell_r \parallel t_1 : S_1} \quad \frac{D_2}{\emptyset; \Sigma; \ell_r \parallel t_2 : S_2}$$

Then

$$(\text{Sprot}) \frac{\emptyset; \Sigma; \ell_r \parallel t_1 : S_1}{\emptyset; \Sigma; \ell_r \parallel \text{prot}_{\ell_r}(t_1) : S_1 \parallel \ell}$$

and by definition of the join operator, $S_1 \parallel \ell <: (S_1 \parallel S_2) \parallel \ell$ and the result holds.

Case $(\text{Sif-false})$. Analogous to case (if-true).

Case $(\text{Sref})$. Then $t = \text{ref}^S v$ and

$$(\text{Sref}) \frac{\emptyset; \Sigma; \ell_r \parallel v : S' \quad S' <: S \quad \ell_r \preceq \text{label}(S)}{\emptyset; \Sigma; \ell_r \parallel \text{ref}^S v : \text{Ref}_\ell S}$$

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

$$\text{ref}^S v \mid \mu \xrightarrow{\ell_r} o_\perp \mid \mu[o \mapsto v \parallel \ell_r]$$

where $o \notin \text{dom}(\mu)$.

Let us take $\Sigma' = \Sigma, o : S$ and let us call $\mu' = \mu[o \mapsto v \parallel \ell_r]$. Then as $\text{dom}(\mu) = \text{dom}(\Sigma)$ then $\text{dom}(\mu') = \text{dom}(\Sigma')$. Also, as $\ell_r \preceq \ell_c \preceq \text{label}(S)$ then by Lemma 130 $\emptyset; \Sigma'; \perp \parallel v : S' \parallel \ell_r$ and $S' \parallel \ell_r <: \Sigma(o) = S$. Therefore $\emptyset; \Sigma' \parallel \mu'$.

Then

$$(\text{Sl}) \frac{\emptyset; \Sigma' \parallel o : S \in \Sigma'}{\emptyset; \Sigma' \parallel o \perp : \text{Ref}_\ell S}$$

and the result holds.
Case (Sderef). Then $t = !o_\ell$ and

\[
\frac{\ell \vdash o_\ell : S \in \Sigma}{\phi; \Sigma ; \ell_c \vdash o_\ell : \text{Ref}_\ell S}
\]

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

\[
!o_\ell | \mu \xrightarrow{\ell_r} v \gamma \ell | \mu \text{ where } \mu(o) = v
\]

Also $\phi; \Sigma \vdash \mu$ then $\phi; \Sigma ; \bot \vdash \mu(o) : S'$ and $S' \prec \prec S$. By Lemma 130 $\phi; \Sigma ; \ell_c \vdash v : S'$

But $S' \gamma \ell \prec S \gamma \ell$ and the result holds.

Case (Sassgn). Then $t = o_\ell := v$ and

\[
\frac{\ell \vdash o_\ell : S \in \Sigma}{\phi; \Sigma ; \ell_c \vdash o_\ell : \text{Ref}_\ell S}
\]

\[
\frac{\phi; \Sigma ; \ell_c \vdash v : S_2 \prec \prec S}{\phi; \Sigma ; \ell_c \vdash o_\ell := v : \text{Unit}_\bot}
\]

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

\[
o_\ell := v | \mu \xrightarrow{\ell_c} \text{unit}_\bot | \mu[o \mapsto v \gamma \ell_r \gamma \ell]
\]

Let us call $\mu' = \mu[o \mapsto v \gamma \ell_r \gamma \ell]$. Also $\phi; \Sigma \vdash \mu$ then $\text{dom}(\mu') = \text{dom}(\Sigma)$, and $\phi; \Sigma ; \ell_c \vdash v : S_2$ where $S_2 \prec \prec S$. Therefore $\phi; \Sigma ; \ell_c \vdash v \gamma \ell_r \gamma \ell \gamma \mu \gamma \ell \gamma \ell$. But $\ell_r \gamma \ell \preceq \ell_c \gamma \ell \preceq \text{label}(S)$, then $S_2 \gamma \ell_r \gamma \ell \prec \prec S$ and therefore $\phi; \Sigma \vdash \mu'$. Also

\[
\phi; \Sigma ; \ell_c \vdash \text{unit}_\bot : \text{Unit}_\bot
\]

but

\[
\text{Unit}_\bot \prec \prec \text{Unit}_\bot
\]

and therefore the result holds.

Case (S::). Then $t = v :: S$ and

\[
\phi; \Sigma ; \ell_c \vdash v :: S_1 \prec \prec S
\]

Suppose $\ell_r$ such that $\ell_r \preceq \ell_c$, then

\[
v :: S | \mu \xrightarrow{\ell_r} v \gamma \text{label}(S) | \mu
\]

But $S_1 \prec \prec S$ then $S_1 \gamma S = S$ and therefore $S_1 \gamma \text{label}(S) = S$. Therefore:

\[
\phi; \Sigma ; \ell_c \vdash v \gamma \text{label}(S) : S
\]

and the result holds.
Proposition 132 (Canonical forms). Consider a value \( v \) such that \( \varnothing; \Sigma; \ell_c \vdash v : S \). Then:

1. If \( S = \text{Bool}_\ell \) then \( v = b_\ell \) for some \( b \).
2. If \( S = \text{Unit}_\ell \) then \( v = \text{unit}_\ell \).
3. If \( S = S_1 \xrightarrow{\ell_c} S_2 \) then \( v = (\lambda x : S_1.t_2) \) for some \( t_2 \) and \( \ell_c' \).
4. If \( S = \text{Ref}_\ell S \) then \( v = o_\ell \) for some location \( o \).

Proof. By inspection of the type derivation rules.

Proposition 20 (Type Safety). If \( \varnothing; \Sigma; \ell_c \vdash t : S \) then either

- \( t \) is a value \( v \), or
- for any store \( \mu \) such that \( \Sigma \vdash \mu \) and any \( \ell_c' \preceq \ell_c \), we have \( t \mid \mu \xrightarrow{\ell_c'} t' \mid \mu' \) and \( \varnothing; \Sigma'; \ell_c \vdash t' : S' \) for some \( S' < : S \) and some \( \Sigma' \supseteq \Sigma \) such that \( \Sigma' \vdash \mu' \).

Proof. By induction on the structure of \( t \).

Case (\( \text{Sb, Su, S\lambda, S!} \)). \( t \) is a value.

Case (\( \text{Sprot} \)). Then \( t = \text{prot}_\ell(t) \) and

\[
\frac{\varnothing; \Sigma; \ell_c \vdash t_1 : S_1}{\varnothing; \Sigma; \ell_c \vdash \text{prot}_\ell(t_1) : S_1 \gamma \ell}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then by (R\( \rightarrow \)) and Canonical Forms (Lemma \( 230 \)). \( t \mid \mu \xrightarrow{\ell_c} t' \mid \mu \) and by Prop \( 229 \) \( \varnothing; \Sigma; \ell_c \vdash t' : S' \) where \( S' < : S \) and the result holds.

2. Suppose \( \ell_r \) such that \( \ell_r \preceq \ell_c \), then

\[
\frac{t_1 \mid \mu \xrightarrow{\ell_r} t_2 \mid \mu'}{\text{prot}_\ell(t_1) \mid \mu \xrightarrow{\ell_r} \text{prot}_\ell(t_2) \mid \mu'}
\]

As \( \ell_r \preceq \ell_c \) then \( \ell_r \gamma \ell \preceq \ell_c \gamma \ell \). Using induction hypotheses \( \varnothing; \Sigma'; \ell_c \gamma \ell \vdash t_2 : S'_1 \) where \( S'_1 < : S_1 \) and \( \varnothing; \Sigma' \vdash \mu' \). Therefore

\[
\frac{\varnothing; \Sigma; \ell_c \vdash t_2 : S'_1}{\varnothing; \Sigma; \ell_c \vdash \text{prot}_\ell(t_2) : S'_1 \gamma \ell}
\]

but \( S'_1 \gamma \ell < : S_1 \gamma \ell \) and the result holds.

Case (\( \text{S\oplus} \)). Then \( t = t_1 \oplus t_2 \) and

\[
\frac{\varnothing; \Sigma; \ell_c \vdash t_1 : \text{Bool}_{\ell_1} \quad \varnothing; \Sigma; \ell_c \vdash t_2 : \text{Bool}_{\ell_2}}{\varnothing; \Sigma; \ell_c \vdash t_1 \oplus t_2 : \text{Bool}_{(\ell_1 \gamma \ell_2)}}
\]
By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then by induction on \( t_2 \) one of the following holds:
   
   \( (a) \) \( t_2 \) is a value. Then by Canonical Forms (Lemma 230)
   
   \[
   \frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}{(R \rightarrow)}
   \]
   
   and by Prop 229 \( \phi; \Sigma; \ell_c \vdash t' : S' \), where \( S' <: S \), therefore the result holds.

   \( (b) \) \( t_2 \mid \mu \xrightarrow{\ell'_r} t'_2 \mid \mu' \) for all \( \ell'_r \) such that \( \ell'_r \not\equiv \ell_c \), in particular we pick \( \ell'_r = \ell_c \).
   
   Then by induction hypothesis, \( \phi; \Sigma'; \ell_c \vdash t_2 : \text{Bool}_{\ell_2} \), where \( \text{Bool}_{\ell_2} <: \text{Bool}_{\ell_2} \) and \( \phi; \Sigma' \vdash \mu' \).
   
   Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t_1 \oplus t'_2 \mid \mu' \) and:
   
   \[
   \frac{\phi; \Sigma; \ell_c \vdash t_1 : \text{Bool}_{\ell_1} \quad \phi; \Sigma; \ell_c \vdash t_2 : \text{Bool}_{\ell_2}}{(S \oplus)}
   \]
   
   but
   
   \[
   \frac{(\ell_2 \not\equiv \ell_2) \not\equiv (\ell_1 \not\equiv \ell_2)}{\text{Bool}_{(\ell_1 \not\equiv \ell_2)} <: \text{Bool}_{(\ell_1 \not\equiv \ell_2)}}
   \]
   
   and the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t'_1 \mid \mu' \) for all \( \ell_r \) such that \( \ell_r \not\equiv \ell_c \), in particular we pick \( \ell_r = \ell_c \). Then by induction hypotheses, \( \phi; \Sigma'; \ell_c \vdash t'_1 : \text{Bool}_{\ell_1} \) where \( \text{Bool}_{\ell_1} <: \text{Bool}_{\ell_1} \), and \( \phi; \Sigma \vdash \mu' \). Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t_1 \oplus t_2 \mid \mu' \) and:
   
   \[
   \frac{\phi; \Sigma; \ell_c \vdash t'_1 : \text{Bool}_{\ell_1} \quad \phi; \Sigma; \ell_c \vdash t_2 : \text{Bool}_{\ell_2}}{(S \oplus)}
   \]
   
   but
   
   \[
   \frac{(\ell'_1 \not\equiv \ell_2) \not\equiv (\ell_1 \not\equiv \ell_2)}{\text{Bool}_{(\ell'_1 \not\equiv \ell_2)} <: \text{Bool}_{(\ell'_1 \not\equiv \ell_2)}}
   \]
   
   and the result holds.

\textit{Case} (Sapp). Then \( t = t_1 \cdot t_2, S = S_{12} \not\equiv \ell \) and

\[
\phi; \Sigma; \ell_c \vdash t_1 : S_{11} \xrightarrow{\ell_c} t_1 S_{12} \quad \phi; \Sigma; \ell_c \vdash t_2 : S_{2} \quad S_2 \not\equiv : S_{11} \quad \ell_c \not\equiv \ell_c' 
\]

\[
\frac{S_{12} \not\equiv : S_{11}}{\phi; \Sigma; \ell_c \vdash t_1 \cdot t_2 : S_{12} \not\equiv \ell}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then by Canonical Forms (Lemma 230), and induction on \( t_2 \) one of the following holds:

236
(a) \( t_2 \) is a value. Then by Canonical Forms (Lemma 230)

\[
\frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}{(R\rightarrow)}
\]

and by Prop 229 \( \phi; \Sigma; \ell_c \vdash t' : S' \), where \( S' <: S \), therefore the result holds.

(b) \( t_2 \mid \mu \xrightarrow{\ell'_r} t'_2 \mid \mu' \) for all \( \ell'_r \) such that \( \ell'_r \ll \ell_c \), in particular we pick \( \ell'_r = \ell_r \). Then by induction hypothesis, \( \phi; \Sigma'; \ell_c \vdash t_2 : S'_2 \), where \( S'_2 <: S_2 \) and \( \phi; \Sigma' \vdash \mu' \).

Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t_1 t'_2 \mid \mu' \). But \( S'_2 <: S_2 <: S_{11} \) and then:

\[
\frac{\phi; \Sigma; \ell_c \vdash t_1 : S_{11} \xrightarrow{\ell_r} t_2 \phi; \Sigma; \ell_c \vdash t'_2 : S'_2}{(Sapp) \quad \ell' \check{} S_{11} \quad \ell' \check{} \ell_c}
\]

and the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t'_1 \mid \mu' \) for all \( \ell'_r \) such that \( \ell'_r \ll \ell_c \), in particular we pick \( \ell'_r = \ell_r \). Then by induction hypothesis, \( \phi; \Sigma'; \ell_c \vdash t'_1 : S'_{11} \xrightarrow{\ell'_r} t_{12} \) where \( S'_{11} \xrightarrow{\ell'_r} t_{12} <: S_{11} \xrightarrow{\ell_r} t_{12} \), and \( \phi; \Sigma' \vdash \mu' \). Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t'_1 t_2 \mid \mu' \). By definition of subtyping, \( S_2 <: S_{11} <: S'_{11} \), \( \ell' \ll \ell_c \) and \( \ell' \ll \ell \). Therefore \( \ell_c \check{} \ell' \ll \ell_c \check{} \ell' \ll \ell_c \). Then

\[
\frac{\phi; \Sigma; \ell_c \vdash t'_1 : S'_{11} \xrightarrow{\ell'_r} t_{12} \phi; \Sigma; \ell_c \vdash t_2 : S_2}{(Sapp) \quad \ell' \check{} S_{11} \quad \ell' \check{} \ell_c}
\]

but \( S'_{12} \check{} \ell' <: S_{12} \check{} \ell \) and the result holds.

Case (Sif). Then \( t = \text{if} \ t_0 \ \text{then} \ t_1 \ \text{else} \ t_2 \) and

\[
\frac{\phi; \Sigma; \ell_c \vdash t_0 : \text{bool}}{(Sif)} \quad \phi; \Sigma; \ell_c \gamma \ell \vdash t_1 : S_1 \quad \phi; \Sigma; \ell_c \gamma \ell \vdash t_2 : S_2
\]

By induction hypotheses, one of the following holds:

1. \( t_0 \) is a value. Then by Canonical Forms (Lemma 230)

\[
\frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}{(R\rightarrow)}
\]

and by Prop 229 \( \phi; \Sigma; \ell_c \vdash t' : S' \), where \( S' <: S \), therefore the result holds.

2. \( t_0 \mid \mu \xrightarrow{\ell'_r} t'_0 \mid \mu' \) for all \( \ell'_r \) such that \( \ell'_r \ll \ell_c \), in particular we pick \( \ell'_r = \ell_r \). Then by induction hypothesis, \( \phi; \Sigma; \ell_c \vdash t'_0 : \text{bool}_{\ell'_r} \), where \( \text{bool}_{\ell'_r} <: \text{bool}_{\ell} \) and \( \phi; \Sigma' \vdash \mu' \). Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} \text{if} \ t'_0 \ \text{then} \ t_1 \ \text{else} \ t_2 \mid \mu' \). As \( \ell_c \check{} \ell' \ll \ell_c \check{} \ell \), by Lemma 128 \( \phi; \Sigma; \ell_c \gamma \ell' \vdash t_1 : S'_1 \) and \( \phi; \Sigma; \ell_c \gamma \ell \vdash t_2 : S'_2 \), where \( S'_1 <: S_1 \) and \( S'_2 <: S_2 \). Therefore:

\[
\frac{\phi; \Sigma; \ell_c \vdash t'_0 : \text{bool}_{\ell'_r}}{(Sif)} \quad \phi; \Sigma; \ell_c \gamma \ell' \vdash t_1 : S'_1 \quad \phi; \Sigma; \ell_c \gamma \ell \vdash t_2 : S'_2
\]

but by definition of join and subtyping \((S'_1 \check{} S'_2) \gamma \ell' <: (S_1 \check{} S_2) \gamma \ell\) and the result holds.
Case (S::). Then \( t = t_1 :: S_2 \) and

\[
(S::) \quad \frac{\emptyset; \Sigma; \ell_c \vdash t_1 : S_1 \quad S_1 <: S_2}{\emptyset; \Sigma; \ell_c \vdash t_1 :: S_2 : S_2}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then

\[
(R\rightarrow) \quad \frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}
\]

and by Prop \(229\) \( \emptyset; \Sigma; \ell_c \vdash t_1 :: S_2, \) where \( S' <: S, \) therefore the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t_1' \mid \mu' \) for all \( \ell_r' \) such that \( \ell_r' \triangleleft \ell_c, \) in particular we pick \( \ell_r' = \ell_r. \) Then by induction hypothesis, \( \emptyset; \Sigma; \ell_c \vdash t_1' : S_1', \) where \( S_1' <: S_1 \) and \( \emptyset; \Sigma' \vdash \mu'. \) Then by (Sf), \( t_1 \mid \mu \xrightarrow{\ell_r} t_1' :: S_2 \mid \mu'. \) Also, \( S_1' <: S_1 <: S_2, \) and therefore:

\[
(S::) \quad \frac{\emptyset; \Sigma; \ell_c \vdash t_1' : S_1'}{\emptyset; \Sigma; \ell_c \vdash t_1' :: S_2 : S_2}
\]

and the result holds.

Case (Sref). Then \( t = \text{ref}^S \) \( t \) and

\[
(Sref) \quad \frac{\emptyset; \Sigma; \ell_c \vdash t_1 : S_1' \quad S_1' <: S_1 \quad \ell_c \triangleleft \text{label}(S_1)}{\emptyset; \Sigma; \ell_c \vdash \text{ref}^{S_1}; \ell_c \vdash t_1 : \text{Ref}_{\bot} S_1}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then

\[
(R\rightarrow) \quad \frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu'}{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu'}
\]

and by Prop \(229\) \( \emptyset; \Sigma; \ell_c \vdash t_1' :: S_2', \) where \( S' <: S, \) and \( \emptyset; \Sigma' \vdash \mu', \) therefore the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t_1' \mid \mu' \) for all \( \ell_r' \) such that \( \ell_r' \triangleleft \ell_c, \) in particular we pick \( \ell_r' = \ell_r. \) Then by induction hypothesis, \( \emptyset; \Sigma; \ell_c \vdash t_1' : S_1'', \) where \( S_1'' <: S_1' \) and \( \emptyset; \Sigma' \vdash \mu'. \) Then by (Sf), \( t_1 \mid \mu \xrightarrow{\ell_r} \text{ref}^{S_1} t_1' \mid \mu' \) and:

\[
(Sref) \quad \frac{\emptyset; \Sigma; \ell_c \vdash t_1' : S_1'' \quad S_1'' <: S_1 \quad \ell_c \triangleleft \text{label}(S_1)}{\emptyset; \Sigma; \ell_c \vdash \text{ref}^{S_1} t_1' : \text{Ref}_{\bot} S_1}
\]

and the result holds.

Case (Sderef). Then \( t = !t_1 \) and

\[
(Sderef) \quad \frac{\emptyset; \Sigma; \ell_c \vdash t_1 : \text{Ref}_l S_1}{\emptyset; \Sigma; \ell_c \vdash !t_1 : S_1 \neg \ell}
\]

By induction hypotheses, one of the following holds:
1. \( t_1 \) is a value. Then by Canonical Forms (Lemma 230)

\[
(R\rightarrow) \quad \frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu}
\]

and by Prop 229, \( \varnothing; \Sigma; \ell_c \vdash t' : S' \), where \( S' \precin S \), therefore the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t'_1 \mid \mu' \) for all \( \ell_r' \) such that \( \ell_r' \vartriangleleft \ell_c \), in particular we pick \( \ell_r' = \ell_r \). Then by induction hypothesis, \( \varnothing; \Sigma; \ell_c \vdash t'_1 : \text{Ref}_\ell S_1 \) where \( \text{Ref}_\ell S_1 \precin S_1 \) and \( \varnothing; \Sigma' \vdash \mu' \).

Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t' \mid \mu' \) and:

\[
(S\text{deref}) \quad \frac{\varnothing; \Sigma; \ell_c \vdash t'_1 : \text{Ref}_\ell S_1}{\varnothing; \Sigma; \ell_c \vdash \text{Unit} \mid \mu'}
\]

but \( S_1 \succ \ell' \succin S_1 \) and the result holds.

Case (Sasgn). Then \( t = t_1 := t_2 \) and

\[
(S\text{asgn}) \quad \frac{\varnothing; \Sigma; \ell_c \vdash t_1 : \text{Ref}_\ell S_1 \quad \varnothing; \Sigma; \ell_c \vdash t_2 : S_2 \quad S_2 \precin S_1 \quad \ell_c \succ \ell \prec \text{label}(S_1)}{\varnothing; \Sigma; \ell_c \vdash t_1 := t_2 : \text{Unit} \mid \mu'}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then by Canonical Forms (Lemma 230), and induction on \( t_2 \) one of the following holds:

   (a) \( t_2 \) is a value. Then by Canonical Forms (Lemma 230)

   \[
   \frac{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu'}{t \mid \mu \xrightarrow{\ell_r} t' \mid \mu'}
   \]

   and by Prop 229, \( \varnothing; \Sigma; \ell_c \vdash t' : S' \), where \( S' \precin S \) and \( \varnothing; \Sigma' \vdash \mu' \), therefore the result holds.

   (b) \( t_2 \mid \mu \xrightarrow{\ell_r'} t'_2 \mid \mu' \) for all \( \ell_r' \) such that \( \ell_r' \vartriangleleft \ell_c \), in particular we pick \( \ell_r' = \ell_r \). Then by induction hypothesis, \( \varnothing; \Sigma'; \ell_c \vdash t'_2 : S'_2 \) where \( S'_2 \precin S_2 \) and \( \varnothing; \Sigma' \vdash \mu' \).

   Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t_1 := t'_2 \mid \mu' \). As \( S'_2 \precin S_2 \precin S_1 \), then:

   \[
   (S\text{asgn}) \quad \frac{\varnothing; \Sigma; \ell_c \vdash t_1 : \text{Ref}_\ell S_1 \quad \varnothing; \Sigma; \ell_c \vdash t'_2 : S'_2 \quad S'_2 \precin S_1 \quad \ell_c \succ \ell \prec \text{label}(S_1)}{\varnothing; \Sigma; \ell_c \vdash t_1 := t'_2 : \text{Unit} \mid \mu'}
   \]

   and the result holds.

2. \( t_1 \mid \mu \xrightarrow{\ell_r} t'_1 \mid \mu' \) for all \( \ell_r' \) such that \( \ell_r' \vartriangleleft \ell_c \), in particular we pick \( \ell_r' = \ell_r \). Then by induction hypotheses, \( \varnothing; \Sigma' ; \ell_c \vdash t'_1 : \text{Ref}_\ell S_1 \) where \( \text{Ref}_\ell S_1 \precin S_1 \) and \( \varnothing; \Sigma' \vdash \mu' \).

   Then by (Sf), \( t \mid \mu \xrightarrow{\ell_r} t'_1 := t_2 \mid \mu' \). As \( \ell' \vartriangleleft \ell \) then \( \ell_c \succ \ell \succ \ell_c \succ \ell \succ \text{label}(S_1) \), and therefore:

   \[
   (S\text{asgn}) \quad \frac{\varnothing; \Sigma; \ell_c \vdash t'_1 : \text{Ref}_\ell S_1 \quad \varnothing; \Sigma; \ell_c \vdash t_2 : S_2 \quad S_2 \precin S_1 \quad \ell_c \succ \ell' \succ \text{label}(S_1)}{\varnothing; \Sigma; \ell_c \vdash t_1 := t_2 : \text{Unit} \mid \mu'}
   \]

   and the result holds.
In this section we present the proof of noninterference for SSL

\[ \theta \vdash (\ell_1, v_1, \mu_1) \approx^k_\text{ol} (\ell_2, v_2, \mu_2) : S \iff \ell_1 \approx_\text{ol} \ell_2 \land \theta \vdash \mu_1 \approx^k_\text{ol} \mu_2 \land \theta; \ell_1 \vdash v_1 : S'_1, S'_2 <; S, \]

\[ \land (\text{obs}_\text{ol}(\ell, S) \implies \theta \vdash \text{obs}_\text{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S) \]

\[ \theta \vdash \text{obs}_\text{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S \iff (\text{rval}(v_1) = \text{rval}(v_2)) \]

if \( S \in \{ \text{Bool}_y, \text{Unit}_y, \text{Ref}_y, S' \} \)

\[ \theta \vdash \text{obs}_\text{ol}^k(\ell_1, v_1, \mu_1, \ell_2, v_2, \mu_2) : S_1 \overset{\ell}{\longrightarrow} S_2 \iff \]

\[ \forall j \leq k, \forall \Sigma \subseteq \Sigma', \theta ; \ell \vdash (\ell_1, v'_1, \mu'_1) \approx^j_\text{ol} (\ell_2, v'_2, \mu'_2) : S_1, \]

\[ \theta ; \ell \vdash (\ell_1, v_1, v'_1, \mu'_1) \approx^j_\text{ol} (\ell_2, v_2, v'_2, \mu'_2) : C(S_2 \sim g) \]

\[ \theta \vdash (\ell_1, t_1, \mu_1) \approx^k_\text{ol} (\ell_2, t_2, \mu_2) : C(S) \iff \ell_1 \approx_\text{ol} \ell_2 \land \theta \vdash \mu_1 \approx^k_\text{ol} \mu_2 \land \theta; \ell_1 \vdash t_1 : S'_1, S'_2 <; S, \forall j < k \]

\[ (t_1 | \mu_1) \overset{t_1}{\longrightarrow} t'_1 | \mu'_1 \implies \theta ; \ell \vdash (\ell_1, \ell_1', \mu_1') \approx^{k-j}_\text{ol} (\ell_2, t'_2, \mu'_2) \land \]

\[ \text{irred}(t'_1) \implies \theta ; \ell \vdash (\ell_1, \ell_1', \mu_1') \approx^{k-j}_\text{ol} (\ell_2, t'_2, \mu'_2) : S) \]

\[ \theta \vdash \mu_1 \approx^k_\text{ol} \mu_2 \iff \theta ; \ell_1 \vdash \forall t_1, \ell_1 \approx_\text{ol} \ell_2, j < k, \forall o \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2) \]

\[ \theta ; \ell_1, \mu_1(o) \approx^j_\text{ol} (\ell_2, \mu_2(o), \mu_2) : \Sigma(o) \]

\[ \ell_1 \approx_\text{ol} \ell_2 \iff \text{obs}_\text{ol}(\ell_1) \lor \neg \text{obs}_\text{ol}(\ell_1) \]

\[ \mu_1 \rightarrow \mu_2 \iff \text{dom}(\mu_1) \subseteq \text{dom}(\mu_2) \]

\[ \text{obs}_\text{ol}(\ell, S) \iff \text{obs}_\text{ol}(\ell) \land \text{obs}_\text{ol}(\text{label}(S)) \]

\[ \text{obs}_\text{ol}(\ell) \iff \ell \approx_\text{ol} \]

Figure B.10: Security logical relations

\[ \square \]

### B.2.2 SSL\textsubscript{Ref}: Noninterference

In this section we present the proof of noninterference for SSL\textsubscript{Ref}. § \[ \text{B.2.2} \] present some auxiliary definitions and § \[ \text{B.2.2} \] present the proof of noninterference.

#### Definitions

To define the fundamental property of the step-indexed logical relations we first define how to relate substitutions:

**Definition 74.** Let \( \sigma \) be a substitution, \( \Gamma \) and \( \Sigma \) a type substitutions. We say that sub-
Proof.
By induction on the derivation of \( \Gamma; \Sigma; \ell \vdash t : S \) and \( \forall x \in \text{dom}(\Gamma), \forall \ell \in \Gamma; \Sigma; \ell \vdash \sigma(x) : S' \), where \( S' <: \Gamma(x) \).

**Definition 75** (Related substitutions). Tuples \( \langle \ell_1, \sigma_1, \mu_1 \rangle \) and \( \langle \ell_2, \sigma_2, \mu_2 \rangle \) are related on \( k \) steps, notation \( \Gamma; \Sigma; \ell \vdash \langle \ell_1, \sigma_1, \mu_1 \rangle \approx_k \langle \ell_2, \sigma_2, \mu_2 \rangle \), if \( \sigma_1 \models \Gamma; \Sigma, \Sigma \vdash \mu_1 \approx_k \mu_2 \) and

\[
\forall x \in \Gamma. \Sigma \vdash \langle \ell_1, \sigma_1(x), \mu_1 \rangle \approx_k \langle \ell_2, \sigma_2(x), \mu_2 \rangle : \Gamma(x)
\]

**Proof of noninterference**

**Lemma 133** (Substitution preserves typing). If \( \Gamma; \Sigma; \ell \vdash t : S \) and \( \sigma \models \Gamma; \Sigma \) then \( \Gamma; \Sigma; \ell \vdash \sigma(t) : S' \) and \( S' <: S \).

*Proof.* By induction on the derivation of \( \Gamma; \Sigma; \ell \vdash t \in S \).

**Lemma 134.** Consider stores \( \mu_1, \mu_2, \mu_1', \mu_2' \) such that \( \mu_i \rightarrow \mu_i' \), and substitutions \( \sigma_1 \) and \( \sigma_2 \), such that

\( \Gamma; \Sigma; \ell \vdash \langle \ell_1, \sigma_1, \mu_1 \rangle \approx_k \langle \ell_2, \sigma_2, \mu_2 \rangle \), then if \( \forall j \leq k \), if \( \Sigma \subseteq \Sigma' \), \( \Sigma' \vdash \mu_1' \approx_i \mu_2' \) then \( \Gamma; \Sigma' \vdash \langle \ell_1, \sigma_1, \mu_1' \rangle \approx_i \langle \ell_2, \sigma_2, \mu_2' \rangle \)

*Proof.* By definition of related computations and related stores. The key argument is that given that \( \mu_i \rightarrow \mu_i' \) then \( \mu_i' \) have at least the same locations of \( \mu_i \) and the values still are related as well given that they still have the same type.

**Lemma 135** (Substitution preserves typing). If \( \Gamma; \Sigma; \ell \vdash t : S \) then \( \forall \ell' \prec \ell, \Gamma; \Sigma; \ell' \vdash t : S \).

*Proof.* By induction on the derivation of \( \Gamma; \Sigma; \ell \vdash t \in S \).

**Lemma 136** (Downward Closed / Monotonicity). If

1. \( \Sigma \vdash \langle \ell_1, v_1, \mu_1 \rangle \approx_k \langle \ell_2, v_2, \mu_2 \rangle : S \) then
   \( \forall j \leq k, \Sigma \vdash \langle \ell_1, v_1, \mu_1 \rangle \approx_{ol}^j \langle \ell_2, v_2, \mu_2 \rangle : S \)

2. \( \Sigma \vdash \langle \ell_1, t_1, \mu_1 \rangle \approx_k \langle \ell_2, t_2, \mu_2 \rangle : C(S) \) then
   \( \forall j \leq k, \Sigma \vdash \langle \ell_1, t_1, \mu_1 \rangle \approx_{ol}^j \langle \ell_2, t_2, \mu_2 \rangle : C(S) \)

3. \( \Sigma \vdash \mu_1 \approx_{ol} \mu_2 \) then \( \forall j \leq k, \Sigma \vdash \mu_1 \approx_{ol}^j \mu_2 \)

*Proof.* By induction on type \( S \) and the definition of related stores.

**Lemma 137.** Consider simple values \( v_i : S_i \) and
\( \Sigma \vdash \langle \ell_1, v_1, \mu_1 \rangle \approx_k \langle \ell_2, v_2, \mu_2 \rangle : S \).

Then
\( \Sigma \vdash \langle \ell_1, (v_1 \gamma \ell), \mu_1 \rangle \approx_k \langle \ell_2, (v_2 \gamma \ell), \mu_2 \rangle : S \gamma \ell \)
Proof. By induction on type S. We proceed by definition of related values and observational-monotonicity of the join, considering that the label stamping can only make values non observable.

Lemma 138 (Reduction preserves relations). Consider $\Sigma; \ell_1 \vdash t_1 \in \text{TERM}_S$, $\mu_1 \in \text{STORE}$, $\Sigma \vdash \mu_1 \approx_{ol}^k \mu_2$. Consider $j < k$, posing $t_1 | \mu_1 \xrightarrow{\ell_1} j t'_1 | \mu'_1$, $\Sigma \subseteq \Sigma'$, $\Sigma' \vdash \mu'_1$ we have

$$\Sigma \vdash \langle \ell_1, t_1, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, t_2, \mu_2 \rangle : C(S)$$

if and only if $\Sigma' \vdash \langle \ell_1, t'_1, \mu'_1 \rangle \approx_{ol}^{k-j} \langle \ell_2, t'_2, \mu'_2 \rangle : C(S)$

Proof. Direct by definition of

$$\Sigma \vdash \langle \ell_1, t_1, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, t_2, \mu_2 \rangle : C(S)$$

and transitivity of $\xrightarrow{\ell} j$.

Lemma 139. Consider term $\Sigma; \ell \vdash t : S$, store $\mu$ and $j > 0$, such that $t | \mu \xrightarrow{\ell} j t' | \mu'$. Then $\mu \rightarrow \mu'$.

Proof. Trivial by induction on the derivation of $t$. The only rules that change the store are the ones for reference and assignment, neither of which remove locations.

Lemma 140. Suppose that $\Sigma \vdash \langle \ell_1 \gamma \ell'_1, t_1, \mu_1 \rangle \approx_{ol}^k \langle \ell_2 \gamma \ell'_2, t_2, \mu_2 \rangle : C(S)$, and that $\ell_1 \vdash \text{prot}_{\ell'_1}(t) : S \gamma \ell'_1$, $S \gamma \ell'_1 <: S \gamma \ell$ for $i \in \{1, 2\}$. If $\ell_1 \approx_{ol}^k \ell_2$, and $\ell'_1 \approx_{ol}^k \ell'_2$, then $\Sigma \vdash \langle \ell_1, \text{prot}_{\ell'_1}(t_1), \mu_1 \rangle \approx_{ol}^k \langle \ell_2, \text{prot}_{\ell'_2}(t_2), \mu_2 \rangle : C(S \gamma \ell)$

Proof. Consider $j < k$, we know by definition of related computations that

$$t_1 | \mu_1 \xrightarrow{\ell_1} j t'_1 | \mu'_1$$

then $\mu'_1 \approx_{ol}^j \mu'_2$, and by Lemma 139 $\mu_1 \rightarrow \mu'_1$. If $t'_1$ are reducible after $k - 1$ steps, then the result holds immediately by \(\text{(Rprot)}\). The interest case if $t'_1$ are irreducible after $j < k$ steps:

Suppose that after $j$ steps $t'_1 = v_1$, then $\Sigma' \vdash \langle \ell_1 \gamma \ell'_1, v_1, \mu_1 \rangle \approx_{ol}^{k-j} \langle \ell_2 \gamma \ell'_2, v_2, \mu'_2 \rangle : S$, for some $\Sigma'$ such that $\Sigma \subseteq \Sigma'$.

Therefore:

$$\text{prot}_{\ell'_1}(t_1) | \mu'_1 \xrightarrow{\ell_1} j \text{prot}_{\ell'_1}(v_1) | \mu'_1 \xrightarrow{\ell_1} 1 (v_1 \gamma \ell'_1) | \mu'_1$$

Let us suppose $\Sigma'; \ell_1 \vdash v_1 : S''_1$, where $S''_1 <: S'_1 <: S$. Then $\Sigma'; \ell_1 \vdash v_1 \gamma \ell'_1 : S''_1 \gamma \ell'_1$, and $S''_1 \gamma \ell'_1 <: S \gamma \ell$. If $\neg \text{obs}_{ol}(\ell_1 \gamma \ell'_1)$ by monotonicity of the join either $\neg \text{obs}_{ol}(\ell'_1)$ or $\neg \text{obs}_{ol}(\ell'_1)$. If $\neg \text{obs}_{ol}(\ell'_1)$ then $\neg \text{obs}_{ol}(S \gamma \ell'_1)$ and the result holds. If $\neg \text{obs}_{ol}(\ell_1)$ the result holds immediately. If $\text{obs}_{ol}(\ell'_1, S) \text{ then obs}_{ol}(\ell_1, S \gamma \ell'_1)$, then the result follows by Lemma 137, and by backward preservation of the relations (Lemma 138).
Lemma 141. Consider $\ell$, such that $\neg \text{obs}_\ell(\ell)$, then then $\forall k > 0$, such that, $\Sigma; \ell \vdash t : S$, $\Sigma \vdash \mu$
$t | \mu \xrightarrow{\ell} k t' | \mu'$, then $\forall \ell'$,

1. $\forall o \in \text{dom}(\mu') \setminus \text{dom}(\mu)$, $\neg \text{obs}_\ell(\ell', \mu'(o))$.

2. $\forall o \in \text{dom}(\mu') \cap \text{dom}(\mu) \wedge \mu'(o) \neq \mu(o)$, $\neg \text{obs}_\ell(\text{label}(\Sigma(o)))$.

Proof. We use induction on the derivation of $t$. The interest cases are the last step of reduction rules for references and assignments.

Case ($t = o_{\ell'} := v$). We are only updating the heap so we only have to prove (1) and (2).

Then $o_{\ell'} := v \xrightarrow{\ell} \text{unit}_\bot | \mu[o \mapsto (v \gamma (\ell \gamma \ell'))]$.

Next we have to prove that $\text{obs}_\ell(\text{label}(\Sigma(o)))$ is not defined. As $\Sigma; \ell \vdash t : S$, then we know that $\ell \gamma \ell'' \prec \text{label}(\Sigma(o))$, and as $\neg(\text{obs}_\ell(\ell))$ by monotonicity of the join the result holds.

Case ($t = \text{ref}^{s'} v$). We are extending the heap, so we need to only prove (1). Then

$\text{ref}^{s'} v | \mu \xrightarrow{\ell} o_{\bot} | \mu[o \mapsto (v \gamma \ell)]$

where $o \notin \text{dom}(\mu)$. We need to prove that $\text{obs}_\ell(\text{label}(v \gamma \ell))$ does not hold, which follows directly by monotonicity of the join.

Lemma 142. Consider $\ell$, such that $\text{obs}_\ell(\ell)$ does not hold, then then $\forall k > 0$, such that $\Sigma; \ell \vdash t_1 : S_1$, and that $t_1 | \mu_1 \xrightarrow{\ell} k t'_1 | \mu'_1$, then if $\Sigma ; \mu_1 \approx^k_\text{ol} \mu_2$, then $\Sigma' \vdash \mu'_1 \approx^k_\text{ol} \mu'_2$ for some $\Sigma'$ such that $\Sigma \subseteq \Sigma'$ and that $\Sigma'; \ell \vdash t'_1 : S'_1$, where $S'_1 \prec : S_1$.

Proof. By Lemma 141 we know three things:

1. $\forall o \in \text{dom}(\mu'_1) \setminus \text{dom}(\mu_1)$, $\text{obs}_\ell(\ell, \mu'_1(o))$ does not hold, i.e. new locations are not observable and therefore as $\Sigma'; \ell \vdash \mu'_1 : S'$ and $S \prec : \Sigma'(o)$, then $\neg \text{obs}_\ell(\text{label}(\Sigma(o)))$.

2. $\forall o \in \text{dom}(\mu'_1) \cap \text{dom}(\mu_1) \wedge \mu'_1(o) \neq \mu(o)$, $\neg \text{obs}_\ell(\text{label}(\Sigma(o)))$

i.e. for all updated references they have to be previously not observable, and by definition therefore related, and second they are still non observable after the update, and by definition those locations are still related under $\ell$ because $\Sigma(o) = \Sigma'(o)$.

Therefore $\Sigma' \vdash \mu'_1 \approx^k_\text{ol} \mu'_2$ and the result holds.

Lemma 143. Suppose that $\Sigma; \ell_i \vdash \text{prot}_i(t_1) : S' \gamma \ell'_i$, $S' \gamma \ell'_i \prec : S$ for $i \in \{1, 2\}$, where $\neg \text{obs}_\ell(\ell_i \gamma \ell'_i)$. Also consider two stores $\mu_i$ such that $\Sigma ; \mu_1 \approx^k_\text{ol} \mu_2$.

Then $\Sigma \vdash \langle \ell_1, \text{prot}_1(t_1), \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_2, \text{prot}_2(t_2), \mu_2 \rangle : C(S)$.
Proof. Suppose that after at least \( j \) more steps, where \( j < k \), both subterms reduce to a value :

\[
\begin{align*}
  t &\mid \mu_i &\xrightarrow{\ell_i \gamma \ell_i'} & j & v_i &\mid \mu_i' \\
  
\end{align*}
\]

Therefore:

\[
\begin{align*}
  \text{prot}_{\ell_i'}(t) &\mid \mu_i' \\
  \xrightarrow{\ell_i} & j & \text{prot}_{\ell_i'}(v_i) &\mid \mu_i' \\
  \xrightarrow{\ell_i} & 1 & (v_i \gamma \ell_i') &\mid \mu_i' \\
\end{align*}
\]

As the values can be radically different we have to make sure that both values are not observables. If \( \neg \text{obs}_{ol}(\ell_i) \) then the values are not observables because the security context is not observable. Let us assume that \( \text{obs}_{ol}(\ell_i) \) holds, but \( \neg \text{obs}_{ol}(\ell_i') \) not. Then by monotonicity of the join, \( \neg \text{obs}_{ol}(\text{label}(v_i) \gamma \ell_i') \) and the result follows.

Now we have to prove that the resulting stores are related, for some \( \Sigma' \) such that \( \Sigma \subseteq \Sigma' \). But by Lemma 142 the result follows immediately.

Next, we present the Noninterference proposition.

**Proposition 122 (Security Type Soundness).** If \( \Gamma; \Sigma; \ell_c \vdash t : S'_i \quad \implies \quad \forall S, S'_i < : S, \Gamma; \Sigma; \ell_c \vdash t : S \)

Proof. We proceed by proving a more general proposition instead:

If \( \Gamma; \Sigma; \ell_i \vdash t : S'_i, S'_i < : S \), then \( \forall \mu_i \in \text{STORE}, \Sigma \vdash \mu_i, \) and \( \forall k \geq 0, \forall \sigma_i \in \text{SUBST}, \Gamma; \Sigma \vdash \langle \ell_1, \sigma_1, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, \sigma_2, \mu_2 \rangle \), we have \( \Sigma \vdash \langle \ell_1, \sigma_1(t), \mu_1 \rangle \approx_{ol}^k \langle \ell_2, \sigma_2(t), \mu_2 \rangle : C(S) \).

By induction on the derivation of term \( t \). Let us take an arbitrary index \( k \geq 0 \).

**Case (x).** \( t = x \) and \( \Gamma(x) = S \). \( \Gamma; \Sigma \vdash \langle \ell_1, \sigma_1, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, \sigma_2, \mu_2 \rangle \) implies by definition that \( \Sigma \vdash \langle \ell_1, \sigma_1(x), \mu_1 \rangle \approx_{ol}^k \langle \ell_2, \sigma_2(x), \mu_2 \rangle : S \), and the result holds immediately.

**Case (b).** \( t = b_g \). By definition of substitution, \( \sigma_1(b_g) = \sigma_2(b_g) = b_g \). By definition, \( \Sigma \vdash \langle \ell_1, b_g, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, b_g, \mu_2 \rangle : \text{Bool}_g \) as required.

**Case (o).** \( t = o_{g_1} \) and \( \Sigma(o) = S \), where \( S = \text{Ref}_{g_1} S_1 \). By definition of substitution, \( \sigma_1(o_{g_1}) = \sigma_2(o_{g_1}) = o_{g_1} \). We know that \( \Sigma; \ell_i \vdash o_{g_1} : \text{Ref}_{g_1} S_1 \). By definition of related stores, \( \Sigma \vdash \langle \ell_1, o_{g_1}, \mu_1 \rangle \approx_{ol}^k \langle \ell_2, o_{g_1}, \mu_2 \rangle : \text{Ref}_{g_1} S_1 \) as required, and the result holds.
Case ($\lambda$). $t = (\lambda^{e'}x : S'_1.t_1)_\ell'$. Then $S'_1 = S'_1 \xrightarrow{\ell''} S''_1$, and $S = S_1 \xrightarrow{\ell'} S_2$, where $S' <: S$.

By definition of substitution, assuming $x \not\in \text{dom}(\sigma_1)$, and Lemma 133:

$$\Gamma; \Sigma; \ell_1 \vdash \sigma_1(t) = \Gamma; \Sigma; \ell_1 \vdash (\lambda^{e''}x : S_1.\sigma_1(t_1))_\ell' : S'_1 \xrightarrow{\ell''} S''_1$$

where $S''_1 <: S'_1$. Consider $j \leq k$, $\mu'_1, \mu'_2$ such that $\mu_i \rightarrow \mu'_1$, and $\Sigma \subseteq \Sigma'$ $\Sigma' \vdash \mu'_1 \approx_{ol}^j \mu'_2$, and assume two values $v_1$ and $v_2$ such that $\Sigma' \vdash (\ell_1, v_1, \mu'_1) \approx_{ol}^j (\ell_2, v_2, \mu'_2) : S_1$.

We need to show that:

$$\Sigma' \vdash (\ell_1, (\lambda^{e''}x : S_1.\sigma_1(t_1))_\ell' v_1, \mu'_1) \approx_{ol}^j (\ell_2, (\lambda^{e''}x : S_1.\sigma_2(t_1))_\ell' v_2, \mu'_2) : C(S_2)$$

Then:

$$(\lambda^{e''}x : S_1.\sigma_1(t_1))_\ell' v_1 \mid \mu'_1$$

$$\xrightarrow{\ell_1} \text{prot}_{\ell'}([v_1/x]\sigma_1(t_1)) \mid \mu'_1$$

$$\xrightarrow{\ell_1}^* \text{prot}_{\ell'}([v_1/x]\sigma_1(t_1)) \mid \mu'_1$$

We then extend the substitutions to map $x$ to the arguments:

$$\sigma'_i = \sigma_1\{x \mapsto v_1\}$$

We know that $\Sigma' \vdash (\ell_1, v_1, \mu'_1) \approx_{ol}^j (\ell_2, v_2, \mu'_2) : S_1$. So as $\mu_i \rightarrow \mu'_1$ then by Lemma 134 $\Gamma, x : S_1; \Sigma' \vdash (\ell_1, \sigma'_1, \mu'_1) \approx_{ol}^j (\ell_2, \sigma'_2, \mu'_2)$.

By Lemma 133 $\Gamma; \Sigma' ; \ell''_c \vdash \sigma'_1(t_1) : S''_1$ where $S''_1 <: S'''_1 <: S_2$. We know that $\ell_1 \gamma \ell' < \ell''_c$, therefore by Lemma 128 $\Gamma; \Sigma'; \ell_1 \gamma \ell \vdash \sigma'_1(t_1) : S''_1$. Then by induction hypothesis and Lemma 136

$$\Sigma' \vdash (\ell_1 \gamma \ell', \sigma'_1(t_1), \mu'_1) \approx_{ol}^j-1 (\ell_2 \gamma \ell', \sigma'_2(t_1), \mu'_2) : C(S_2),$$

Finally, by Lemma 140

$$\Sigma' \vdash (\ell_1, \text{prot}_{\ell'}(\sigma'_1(t_1)), \mu'_1) \approx_{ol}^j (\ell_2, \text{prot}_{\ell'}(\sigma'_2(t_1)), \mu'_2) : C(S_2)$$

and finally the result holds by backward preservation of the relations (Lemma 138).

---

Case (!). $t = !t'$, where $\Sigma; \ell_1 \vdash t' : \text{Ref}_{\ell''} S_1$, where $S_1 \gamma \ell''_1 < : S = S_1 \gamma \ell$.

By definition of substitution:

$$\sigma_1(t) = !\sigma_1(t')$$

We have to show that

$$\Sigma \vdash (\ell_1, !\sigma_1(t'), \mu_1) \approx_{ol}^k (\ell_2, !\sigma_1(t'), \mu_2) : C(S)$$

By Lemma 133

$$\Sigma; \ell_1 \vdash !\sigma_1(t') : S_1 \gamma \ell''_1$$
where \( \ell_1'' \preceq \ell_1' \preceq \ell \). By induction hypotheses on the subterm:

\[
\Sigma \vdash \langle \ell_1, \sigma_1(t'), \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_2, \sigma_2(t'), \mu_2 \rangle : C(\text{Ref}_\ell S_1)
\]

Consider \( j < k \), then by definition of related computations

\[
\sigma_i(t') \mid \mu_i \xrightarrow{\ell_i} j t'_i \mid \mu'_i \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu'_i \approx_{\text{ol}}^{j-i} \mu\text{ and Lemma 133}.
\]

If \( \mu_i \approx_{\text{ol}}^{j-i} \mu' \) and the result holds.

If after at most \( j \) steps \( t'_i \) is irreducible it means that for some \( j' \leq j \), \( !\sigma_i(t) \mid \mu_i \xrightarrow{\ell_i} j' !v_i \mid \mu'_i \).

If \( j' = j \) then we use the same same argument for reducible terms and the result holds.

Let us consider now \( j' < j \). Then \( \Sigma' \vdash \langle \ell_1, v_1, \mu'_1 \rangle \approx_{\text{ol}}^{k-j'} \langle \ell_2, v_2, \mu'_2 \rangle : \text{Ref}_\ell S_1 \). By Lemma 230 each \( v_i \) is a location \( o_{\ell_i'} \), such that \( \Sigma'(o_{\ell_i'}) = \text{Ref}_\ell S_1 \) and \( \ell'_i \preceq \ell \). Then:

\[
\sigma_i(t) \mid \mu \xrightarrow{\ell_i} j' + !o_{\ell_i'} \mid \mu'_i
\]

\[
\xrightarrow{\ell_i} 1 \text{ prot}_{\ell_i}(v'_i) \mid \mu'_i
\]

with \( \ell'_i \preceq \ell''_i \), \( v'_i = \mu'_i(o_{\ell_i'}) \). As \( \Sigma' \vdash \langle \ell_1, v_1, \mu'_1 \rangle \approx_{\text{ol}}^{k-j'} \langle \ell_2, v_2, \mu'_2 \rangle : \text{Ref}_\ell S_1 \), then by By monotonicity of the join either both \( \text{obs}_{\text{ol}}(\ell'_i) \) or \( \neg \text{obs}_{\text{ol}}(\ell'_i) \). Finally as \( \Sigma' \vdash \langle \ell_1, v'_1, \mu'_1 \rangle \approx_{\text{ol}}^{k-j'} \langle \ell_2, v'_2, \mu'_2 \rangle : S_1 \), by Lemma 222

\[
\Sigma' \vdash \langle \ell_1, \text{prot}_{\ell_1}(v'_1), \mu'_1 \rangle
\]

\[
\approx^j_{\text{ol}} \langle \ell_2, \text{prot}_{\ell_2}(v'_2), \mu'_2 \rangle : C(S \varnothing \ell)
\]

and finally the result holds by backward preservation of the relations (Lemma 138).

---

**Case (:=).** \( t = t_1 := t_2 \). Then \( S = \text{Unit}_\bot \).

By definition of substitution:

\[
\sigma_i(t) = \sigma_i(t_1) := \sigma_i(t_2)
\]

and Lemma 133

\[
\Sigma; \ell_i \vdash \sigma_i(t_1) := \sigma_i(t_2) : \text{Unit}_\bot
\]

We have to show that

\[
\Sigma \vdash \langle \ell_1, \sigma_1(t_1) := \sigma_1(t_2), \mu_1 \rangle
\]

\[
\approx^k_{\text{ol}} \langle \ell_2, \sigma_2(t_1) := \sigma_2(t_2), \mu_2 \rangle : C(S)
\]

By induction hypotheses

\[
\Sigma \vdash \langle \ell_1, \sigma_1(t_1), \mu_1 \rangle \approx^k_{\text{ol}} \langle \ell_2, \sigma_2(t_1), \mu_2 \rangle : C(S_1)
\]
Suppose \( j_1 < k \), and that \( \sigma_i(t_1) \) are irreducible after \( j_1 \) steps (otherwise, similar to case 1, the result holds immediately). Then by definition of related computations:

\[
\sigma_i(t_1) \mid \mu_i \xrightarrow{\ell_1} j_1 v_1 \mid \mu_i' \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu_1' \approx_{\text{ol}}^k j_1 \mu_2' \wedge \Sigma' \vdash \langle \ell_1, v_1, \mu_1' \rangle \approx_{\text{ol}}^{k-j_1} \langle \ell_2, v_2, \mu_2' \rangle : \text{Ref}_\ell S_1
\]

By Lemma \[139\] \( \mu_i \rightarrow \mu_i' \), and \( \mu_1' \approx_{\text{ol}}^{k-j_1} \mu_2' \) then by Lemma \[204\] \( \Sigma' \vdash \langle \ell_1, \sigma_1, \mu_1' \rangle \approx_{\text{ol}}^{k-j_1} \langle \ell_2, \sigma_2, \mu_2' \rangle \). By induction hypotheses:

\[
\Sigma' \vdash \langle \ell_1, \sigma_1(t_2), \mu_1' \rangle \approx_{\text{ol}}^k \langle \ell_2, \sigma_2(t_2), \mu_2' \rangle : C(S_2)
\]

Again, consider \( j_2 = k - j_1 \), if after \( j_2 \) steps \( \sigma_i(t_2) \) is reducible or is a value, the result holds immediately. The interest case if after \( j_2' < j_2 \) steps \( \sigma_i(t_{S_2}) \) reduces to values \( v_1' \):

\[
\sigma_i(t_{S_2}) \mid \mu_i' \xrightarrow{\ell_1} j_1 v_1' \mid \mu_i'' \implies \\
\Sigma' \subseteq \Sigma'', \Sigma'' \vdash \mu_i'' \approx_{\text{ol}}^{k-j_1-j_2} \mu_2'' \wedge \Sigma'' \vdash \langle \ell_1, v_1', \mu_i'' \rangle \approx_{\text{ol}}^{k-j_1-j_2} \langle \ell_2, v_2', \mu_2'' \rangle : S_2
\]

Then

\[
\sigma_i(t_S) \mid \mu_i \xrightarrow{\ell_1 \ell_2} j_1 + j_2 v_1 := v_1' \mid \mu_i'' \wedge \Sigma'' \vdash \mu_i'' \approx_{\text{ol}}^{k-j_1-j_2} \mu_2''
\]

As both values \( v_1 \) are related at some reference type, then by canonical forms (Lemma \[230\]) they both must be locations \( o_{\ell'} \) for some \( S'_1 <: S_1 \). We consider when the values are observable and the locations are identical (otherwise the result is trivial):

\[
\begin{align*}
\mu_i'' & = \mu_2'' \\
\xrightarrow{\ell_1 \ell_2} & = \text{Unit} \mid \mu_i''
\end{align*}
\]

Where \( \mu_i'' = \mu_2''[o \mapsto (v_1' \gamma (\ell_1 \gamma \ell_1'))] \). As \( \Sigma'' \vdash \langle \ell_1, v_1', \mu_i'' \rangle \approx_{\text{ol}}^{k-j_1-j_2} \langle \ell_2, v_2', \mu_2'' \rangle : S_2 \), and as \( \ell_1 \gamma \ell_1' \ll \text{label}(S_1) \), where \( \ell_1' \ll \ell \), and \( \text{label}(v_1') \ll \text{label}(S_1') \), then \( \Sigma''; \ell_1 \vdash v_1' \gamma (\ell_1 \gamma \ell_1') : S' \) and \( S' <: S_1 \). Then by monotonicity of the join Lemma \[137\]

\[
\Sigma'' \vdash \langle \ell_1, (v_1' \gamma (\ell_1 \gamma \ell_1')), \mu_1'' \rangle \\
\approx_{\text{ol}}^{k-j_1-j_2} \langle \ell_2, (v_2' \gamma (\ell_2 \gamma \ell_2')), \mu_1'' \rangle
\]

But if \( \neg \text{obs}_{ol}(\ell_1) \) then by monotonicity of the join \( \neg \text{obs}_{ol}(v_1' \gamma (\ell_1 \gamma \ell_1')) \). Therefore, \( \forall \ell_1'' \) such that \( \ell_1'' \approx_{\text{ol}}^k \ell_2'' \)

\[
\Sigma'' \vdash \langle \ell_1', (v_1' \gamma (\ell_1 \gamma \ell_1')), \mu_1'' \rangle \\
\approx_{\text{ol}}^{k-j_1-j_2} \langle \ell_2', (v_2' \gamma (\ell_2 \gamma \ell_2')), \mu_1'' \rangle
\]

As every values are related at type \text{Unit}, we only have to prove that \( \Sigma'' \vdash \mu_i'' \approx_{\text{ol}}^{k-j_1-j_2-3} \mu_i'' \), but using monotonicity (Lemma \[209\]), it is trivial to prove that because either both both stores update the same location \( o \) to values that are related, therefore the result holds.

---

Case (ref.). \( t = \text{ref}_{S_1} t_{S_1'} \). Then \( S = \text{Ref}_\perp S_1 \).

By definition of substitution:

\[
\sigma_i(t) = \text{ref}_{S_1} \sigma_i(t')
\]
and Lemma $\text{Lemma 133}$

$$\ell_1 \vdash \text{ref}^{S_1} \sigma_1(t') : \text{Ref}_S S_1$$

We have to show that

$$\Sigma \vdash \langle \ell_1, \text{ref}^{S_1} \sigma_1(t'), \mu_1 \rangle \approx^{k_1}_{\text{ol}} \langle \ell_2, \text{ref}^{S_2} \sigma_2(t'), \mu_2 \rangle : C(S_1)$$

As $\Sigma; \ell_1 \vdash \sigma_1(t') : S'_1$ where $S'_1 <: S_1$, by induction hypotheses:

$$\Sigma \vdash \langle \ell_1, \sigma_1(t'), \mu \rangle \approx^{k_1}_{\text{ol}} \langle \ell_2, \sigma_2(t'), \mu \rangle : C(S_1)$$

Consider $j < k$, by definition of related computations

$$\sigma_1(t') | \mu_1 \xrightarrow{\ell_1} j \sigma_1(t') | \mu'_1 \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu'_1 \approx^{k-j}_{\text{ol}} \mu_2$$

If terms $t'_1$ are reducible after $j = k - 1$ steps, then

$$\text{ref}^{S_1} \sigma_1(t') | \mu_1 \xrightarrow{\ell_1} j \text{ref}^{S_1} t'_1 | \mu'_1$$

If after at most $j$ steps $t'_1$ is irreducible, it means that for some $j' \leq j$ ref$^{S_1} \sigma_1(t') | \mu_1 \xrightarrow{\ell_1} j' \text{ref}^{S_1} v_1 | \mu'_1$. If $j' = j$ then we use the same same argument for reducible terms and the result holds.

Let us consider now $j' < j$. Then:

$$\sigma_1(t) | \mu \xrightarrow{\ell_1} j' + 1 \text{ref}^{S_1} v_1 | \mu'_1$$

with $\mu''_1 = \mu'_1[\sigma \mapsto (v_1 \gamma \ell_1)]$. Also, as $\Sigma' \vdash \langle \ell_1, v_1, \mu'_1 \rangle \approx^{k-j'}_{\text{ol}} \langle \ell_2, v_2, \mu'_2 \rangle : S_1$, then $\Sigma'' \vdash \langle \ell_1, v_1, \mu''_1 \rangle \approx^{k-j'}_{\text{ol}} \langle \ell_2, v_2, \mu''_2 \rangle : S_1$, with $\Sigma'' = \Sigma, \mu : S_1$. And as $\text{label}(v_1) \gamma \ell_1 \leq \text{label}(S), then by Lemma $\text{Lemma 137}$ $\Sigma'' \vdash \langle \ell_1, v_1, \mu'' \rangle \approx^{k-j'}_{\text{ol}} \langle \ell_2, v_2, \gamma \ell_2, \mu''_2 \rangle : S_1$.

If $\neg \text{obs}_{\text{ol}}(\ell_1)$ then by monotonicity of the join $\neg \text{obs}_{\text{ol}}(\text{label}(v_1 \gamma \ell_1))$ and $\neg \text{obs}_{\text{ol}}(\text{label}(\Sigma''(\sigma)))$. Therefore, $\forall \ell''_1$ such that $\ell''_1 \approx^{k} \ell'_2 \Sigma'' \vdash \langle \ell''_1, v_1 \gamma \ell_1, \mu'_1 \rangle \approx^{k-j'}_{\text{ol}} \langle \ell'_2, v_2 \gamma \ell_2, \mu'_2 \rangle : S_1$. By definition of related stores $\Sigma'' - \mu'' \approx^{k-j'}_{\text{ol}} \mu''_2$. Then by Monotonicity of the relation (Lemma $\text{Lemma 209}$) $\Sigma'' - \mu'' \approx^{k-j'-2}_{\text{ol}} \mu''_2$ and the result holds.

Case $(\oplus)$. $t = t_1 \oplus t_2$

By definition of substitution:

$$\sigma_1(t) = \sigma_1(t_1) \oplus \sigma_1(t_2)$$

and Lemma $\text{Lemma 133}$

$$\Sigma; \ell_1 \vdash \sigma_1(t_1) \oplus \sigma_1(t_2) : S''$$

with $S'' : S'_1 < S$. We use a similar argument to case $:= for reducible terms. The interest case is when we suppose some $j_1$ and $j_2$ such that $j_1 + j_2 < k - 3$ where:

$$\sigma_1(t_1) | \mu_1 \xrightarrow{j_1} j_1 v_{11} | \mu'_1 \implies \Sigma \subseteq \Sigma', \Sigma' \vdash \mu'_1 \approx^{k-j_1}_{\text{ol}} \mu_2' \wedge \Sigma' \vdash \langle \ell_1, v_{11}, \mu'_1 \rangle \approx^{k-j_1}_{\text{ol}} \langle \ell_2, v_{21}, \mu'_2 \rangle : S_1$$
By definition of substitution:

\[ \sigma_1(t_2) \mid \mu_1' \xrightarrow{\ell_i} j_2 v_{i2} \mid \mu_1'' \implies \Sigma' \subseteq \Sigma'', \Sigma'' \vdash \mu_1'' \approx_{ol}^{k-j_1-j_2} \mu_2'' \land \Sigma'' \vdash \langle \ell_1, v_{i1}, \mu_1'' \rangle \approx_{ol}^{k-j_1-j_2} \langle \ell_2, v_{i2}, \mu_2'' \rangle : S_2 \]

By Lemma 230, each \( v_{ij} \) is a boolean \( (b_{ij})_{\ell_{ij}} \) then:

\[
\begin{align*}
&\sigma_1(t) \mid \mu_i'' \\
&\xrightarrow{\ell_1 + j_1 + j_2 + 2} (b_{i1})_{\ell_1} \oplus (b_{i2})_{\ell_2} \mid \mu_i'' \\
&\xrightarrow{1} (b_i)_{\ell_i} \mid \mu_i''
\end{align*}
\]

with \( b_i = b_{i1} \oplus b_{i2} \), \( \ell_i = \ell_{i1} \land \ell_{i2} \), and \( \ell_i' \leq \text{label}(S''_i) \leq \text{label}(S) \). It remains to show that:

\[ \Sigma'' \vdash \langle \ell_1, (b_1)_{\ell_i'}, \mu_1'' \rangle \approx_{ol}^{k-j_1-j_2-3} \langle \ell_2, (b_2)_{\ell_i'}, \mu_2'' \rangle : S \]

If \( \neg \text{obs}_{ol} (\ell_i) \), then the result is trivial because the resulting booleans are also related as they are not observable.

If \( \text{obs}_{ol} (\ell_i) \), and \( \neg \text{obs}_{ol} (\ell_{i1}') \) or \( \neg \text{obs}_{ol} (\ell_{i2}') \), then by monotonicity of the join, \( \neg \text{obs}_{ol} (\ell_i') \) and the result holds. If \( \text{obs}_{ol} (\ell_{ij}) \) then \( \text{obs}_{ol} (\ell_i') \) and therefore \( b_{i1} = b_{21} \) and \( b_{i2} = b_{22} \), so \( b_i = b_2 \), and the result holds.

\[\text{Case} \text{ (app). } t = t_1 t_2, \text{ with } \Sigma; \ell_i \vdash t_1 : S_{i1} \xrightarrow{\ell_i} \ell_i' S_{i2}, \text{ and } \Sigma; \ell_i \vdash t_2 : S''_{i2}. \text{ Also } S_{i1} \xrightarrow{\ell_i} \ell_i' S_{i2} \lessdot: S_1 \xrightarrow{\ell_i} \ell_i S_2, \text{ and } S = S_2.\]

By definition of substitution:

\[ \sigma_1(t) = \sigma_1(t_1) \sigma_1(t_2) \]

and Lemma 133

\[ \Sigma; \ell_i \vdash \sigma_1(t_1) \sigma_1(t_2) : S''_{i2} \]

with \( S''_{i2} \lessdot: S_{i2} \lessdot: S_2 \). We use a similar argument to case := for reducible terms. The interest case is when we suppose some \( j_1 \) and \( j_2 \) such that \( j_1 + j_2 < k \) where by induction hypotheses and the definition of related computations:

\[ \sigma_1(t_1) \mid \mu_1 \xrightarrow{\ell_1} j_1 v_{11} \mid \mu_1' \implies \Sigma' \subseteq \Sigma', \Sigma' \vdash \mu_1' \approx_{ol}^{k-j_1} \mu_2' \land \Sigma' \vdash \langle \ell_1, v_{11}, \mu_1' \rangle \approx_{ol}^{k-j_1} \langle \ell_2, v_{21}, \mu_2' \rangle : S_1 \]

\[ \sigma_1(t_2) \mid \mu_1' \xrightarrow{\ell_2} j_2 v_{i2} \mid \mu_1'' \implies \Sigma' \subseteq \Sigma'', \Sigma'' \vdash \mu_1'' \approx_{ol}^{k-j_1-j_2} \mu_2'' \land \Sigma'' \vdash \langle \ell_1, v_{i1}, \mu_1'' \rangle \approx_{ol}^{k-j_1-j_2} \langle \ell_2, v_{i2}, \mu_2'' \rangle : S_2 \]

Then

\[ \sigma_1(t) \mid \mu_i \xrightarrow{\ell_i} j_1 + j_2 v_{i1} v_{i2} \mid \mu_i'' \]

If \( \text{obs}_{ol} (\ell_i, v_{i1}) \) then, by definition of \( \approx_{ol} \) at values of function type, we have:

\[ \Sigma' \vdash \langle \ell_1, (v_{11} v_{i12}), \mu_1'' \rangle \approx_{ol}^{k-j_1-j_2} \langle \ell_2, (v_{21} v_{i22}), \mu_2'' \rangle : C(S_2 \uplus \ell) \]

249
Finally, by backward preservation of the relations (Lemma 138) the result holds.

If \( \neg \text{obs}_o(\ell_1, v_{11}) \), and we assume by canonical forms that \( v_{11} = (\lambda^\ell x. t_1)_\ell'' \) then, either \( \neg \text{obs}_o(\ell_1) \) or \( \neg \text{obs}_o(\ell_1'') \) and

\[
\begin{align*}
(v_{11} v_{12}) | \mu_1'' & = (\langle \lambda^\ell x. t_1 \rangle_{\ell''} v_{12}) | \mu_1'' \\
\ell_1 \to 1 & \text{ prot}_{\ell''}(t_1') | \mu_1''
\end{align*}
\]

If either \( \neg \text{obs}_o(\ell_1) \) or \( \neg \text{obs}_o(\ell_1'') \) then by Lemma 143,

\[
\approx_{ol}^{k-j_1-j_2} \langle \ell_1, \text{prot}_{\ell''}(t_1'), \mu_1'' \rangle \approx_{ol}^{k-j_1} t_2, \text{prot}_{\ell''}(t_2'), \mu_2'' : C(S_2 \gamma \ell) \]

Finally, by backward preservation of the relations (Lemma 138) the result holds.

——

Case (if). \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \), with \( \Sigma; \ell_1 \vdash t_1 : S_1, \Sigma; \ell_1' \vdash t_2 : S_2, \Sigma; \ell_1'' \vdash t_3 : S_3, \ell_1' = \ell_1 \gamma \text{label}(S_1) \), and \( S' = S_2 \lor S_3 <: S \)

By definition of substitution:

\[ \sigma_i(t) = \text{if } \sigma_i(t_1) \text{ then } \sigma_i(t_2) \text{ else } \sigma_i(t_3) \]

We use a similar argument to case := for reducible terms. The interest case is when we suppose some \( j_1 \) and \( j_2 \) such that \( j_1 + j_2 < k \) where by induction hypotheses and related computations we have that:

\[
\sigma_i(t_1) | \mu_1 \ell_1 \to^{j_1} v_{11} | \mu_1' \implies \Sigma' \subseteq \Sigma, \Sigma' \vdash \mu_1' \approx_{ol}^{k-j_1-j_2} \mu_2' \land \Sigma' \vdash \langle \ell_1, v_{11}, \mu_1' \rangle \approx_{ol}^{k-j_1} \langle \ell_2, v_{21}, \mu_2' \rangle : S_1
\]

By Lemma 230, each \( v_{11} \) is a boolean \( (b_{11})_{\ell_{11}} \), such that \( \Sigma'; \ell_1 \vdash (b_{11})_{\ell_{11}} : \text{Bool}_{\ell_{11}} \text{ and } \text{Bool}_{\ell_{11}} <: S_1 \), implies \( S_1 = \text{Bool}_{\ell_{11}} \). Then:

\[ \sigma_i(t) | \mu_1 \ell_1 \to^{j_1+1} \text{if } (b_{11})_{\ell_{11}} \text{ then } \sigma_i(t_2) \text{ else } \sigma_i(t_3) | \mu_1' \]

Let us consider \( \neg \text{obs}_o(\ell_1, (b_{11})_{\ell_{11}}) \). Let us assume the worst case scenario and that both execution reduce via different branches of the conditional.

Then

\[
\sigma_1(t) | \mu_1 \ell_1 \to^{j_1+2} \text{prot}_{\ell_{11}}(\sigma_1(t_2)) | \mu_1'
\]

\[
\sigma_2(t) | \mu_2 \ell_1 \to^{j_1+2} \text{prot}_{\ell_{11}}(\sigma_2(t_3)) | \mu_2'
\]

But because \( \neg \text{obs}_o(\ell_{11}, (b_{11})_{\ell_{11}}) \), then either \( \neg \text{obs}_o(\ell_1) \) or \( \neg \text{obs}_o(\ell_{11}) \) and therefore, \( \neg \text{obs}_o(\ell_1 \gamma \ell_{11}) \). Then by Lemma 143

\[
\Sigma' \vdash \langle \ell_1, \text{prot}_{\ell_{11}}(\sigma_1(t_2)), \mu_1' \rangle \approx_{ol}^k \langle \ell_2, \text{prot}_{\ell_{21}}(\sigma_2(t_3)), \mu_2' \rangle
\]

and the result holds by backward preservation of the relations (Lemma 138).
Now let us consider if $\text{obs}_o(\ell_1, (b_{11})_{\ell_1})$ holds. Then by definition of $\approx_o$ on boolean values, $b_{11} = b_{21}$. Because $b_{11} = b_{21}$, both $\sigma_1(t)$ and $\sigma_2(t)$ step into the same branch of the conditional. Let us assume the condition is true (the other case is similar):

Then by induction hypothesis $\Sigma' \vdash \langle \ell_1 \gamma \ell_{11}, \sigma_1(t_2), \mu'_1 \rangle \approx_o^k \langle \ell_2 \gamma \ell_{21}, \sigma_2(t_2), \mu'_2 \rangle : S_2$, and by Lemma 140

$$\Sigma' \vdash \langle \ell_1, \text{prot}_{\ell_{11}}(\sigma_1(t_2)), \mu'_1 \rangle \approx_o^k \langle \ell_2, \text{prot}_{\ell_{21}}(\sigma_2(t_2)), \mu'_2 \rangle : S$$

and the result holds by backward preservation of the relations (Lemma 138).

Case ($\text{prot}()$). Direct by using Lemma 140.

\[\Box\]

### B.3 Gradualizing the Static Semantics

In § B.3.1 we show the proof of optimality and soundness of the abstraction. In § B.3.2 we present the proof for the Static Gradual Guarantee.

#### B.3.1 From Gradual Labels to Gradual Types

**Proposition 144 (α is Sound).** If $\hat{\ell} \neq \emptyset$ then $\hat{\ell} \subseteq \gamma(\alpha(\hat{\ell}))$.

*Proof.** By case analysis on the structure of $\hat{\ell}$. If $\hat{\ell} = \{ \ell \}$ then $\gamma(\alpha(\{ \ell \})) = \gamma(\ell) = \{ \ell \} = \hat{\ell}$, otherwise $\gamma(\alpha(\hat{\ell})) = \gamma(\text{LABEL}) \supseteq \hat{\ell}$. \[\Box\]

**Proposition 145 (α is Optimal).** If $\hat{\ell} \subseteq \gamma(g)$ then $\alpha(\hat{\ell}) \subseteq g$.

*Proof.** By case analysis on the structure of $g$. If $g = \ell$, $\gamma(g) = \{ \ell \}$; $\hat{\ell} \subseteq \{ \ell \}$, $\hat{\ell} \neq \emptyset$ implies $\alpha(\hat{\ell}) = \alpha(\{ \ell \}) = \ell \subseteq g$ (if $\hat{\ell} = \emptyset$, $\alpha(\hat{\ell})$ is undefined). If $g = \text{?}$, $g' \subseteq g$ for all $g'$. \[\Box\]

**Proposition 26 (α is Sound and Optimal).** If $\hat{\ell} \neq \emptyset$ then,

(i) $\hat{\ell} \subseteq \gamma(\alpha(\hat{\ell}))$.

(ii) If $\hat{\ell} \subseteq \gamma(g)$ then $\alpha(\hat{\ell}) \subseteq g$.

*Proof.** Trivial using Prop 144 and 145. \[\Box\]

**Proposition 146 (α_S is Sound).** If $\hat{S}$ valid, then $\hat{S} \subseteq \gamma_S(\alpha_S(\hat{S}))$.

251
Proof. By well-founded induction on \( \mathcal{S} \) according to the ordering relation \( \mathcal{S} \sqsubseteq \mathcal{S} \) defined as follows:

\[
\begin{align*}
\overrightarrow{\text{dom}}(\mathcal{S}) & \sqsubseteq \mathcal{S} \\
\overrightarrow{\text{cod}}(j\mathcal{S}) & \sqsubseteq \mathcal{S}
\end{align*}
\]

Where \( \overrightarrow{\text{dom}}, \overrightarrow{\text{cod}} : \mathcal{P}(\text{GType}) \rightarrow \mathcal{P}(\text{GType}) \) are the collecting liftings of the domain and codomain functions \( \text{dom}, \text{cod} \) respectively, e.g.,

\[
\overrightarrow{\text{dom}}(\mathcal{S}) = \{ \text{dom}(S) \mid S \in \mathcal{S} \}.
\]

We then consider cases on \( \mathcal{S} \) according to the definition of \( \alpha_S \).

Case (\( \{ \text{Bool}_i \} \)).

\[
\gamma_S(\alpha_S(\{ \text{Bool}_i \})) = \gamma_S(\text{Bool}_{\alpha(\{ \overrightarrow{t}_i \})})
\]

\[
\subseteq \{ \text{Bool}_\ell \mid \ell \in \gamma(\alpha(\{ \overrightarrow{t}_i \})) \}
\]

by soundness of \( \alpha \).

Case (\( \{ S_{i1} \xrightarrow{\ell_i} S_{i2} \} \)).

\[
\gamma_S(\alpha_S(\{ S_{i1} \xrightarrow{\ell_i} S_{i2} \})) = \gamma_S(S_{i1} \xrightarrow{\ell_i} S_{i2})
\]

\[
\gamma_S(S_{i1} \xrightarrow{\ell_i} S_{i2}) = \{ S \mid S \in \gamma_S(S_{i2}) \}
\]

by induction hypothesis on \( \{ S_{i1} \} \) and \( \{ S_{i2} \} \), and soundness of \( \alpha \).

Case (\( \{ \text{Ref}_i S \} \)).

\[
\gamma_S(\alpha_S(\{ \text{Ref}_i S \})) = \gamma_S(S_{i1} \xrightarrow{\ell_i} S_{i2})
\]

\[
\gamma_S(S_{i1} \xrightarrow{\ell_i} S_{i2}) = \{ \text{Ref}_\ell S \mid \ell \in \gamma(\alpha(\{ \overrightarrow{t}_i \}), S \in \gamma_S(\{ S_{i1} \})) \}
\]

by induction hypothesis on \( \{ S_{i1} \} \) and soundness of \( \alpha \).

\( \Box \)

**Proposition 147 (\( \alpha_S \) is Optimal).** If \( \mathcal{S} \) valid and \( \mathcal{S} \sqsubseteq \gamma_S(U) \) then \( \alpha_S(\mathcal{S}) \sqsubseteq U \).

Proof. By induction on the structure of \( U \).

Case (\( \text{Bool}_g \)). \( \gamma_S(\text{Bool}_g) = \{ \text{Bool}_\ell \mid \ell \in \gamma(g) \} \)

So \( \mathcal{S} = \{ \text{Bool}_\ell \mid \ell \in \gamma(g) \} \) for some \( \ell \subseteq \gamma(g) \). By optimality of \( \alpha \), \( \alpha(\ell) \subseteq g \), so \( \alpha_S(\{ \text{Bool}_\ell \mid \ell \in \gamma(g) \}) = \text{Bool}_{\alpha(\ell)} \sqsubseteq \text{Bool}_g \).

252
Case \((U_1 \rightarrow_g U_2)\). \(\gamma_S(U_1 \xrightarrow{g_c} U_2) = \gamma_S(U_1) \gamma(g_c) \gamma_S(U_2)\).
So \(\hat{S} = \{S_i \xrightarrow{\ell_i} \gamma_S(U_i), \ell_i \in \gamma(g)\} \subseteq \gamma_S(U_1), \gamma_S(U_2)\). By induction hypothesis, \(\alpha_S(\{S_i \xrightarrow{\ell_i} \gamma_S(U_i)\}) \subseteq U_1\) and \(\alpha_S(\{S_i \xrightarrow{\ell_i} \gamma_S(U_i)\}) \subseteq U_2\), and by optimality of \(\alpha\), \(\alpha(\{\ell_i\}) \subseteq g_c\) and \(\alpha(\{\gamma\}) \subseteq g\). Hence \(\alpha_S(\{S_i \xrightarrow{\ell_i} \gamma_S(U_i)\}) = \alpha_S(\{S_i \xrightarrow{\ell_i} \gamma_S(U_i)\}) \subseteq U_1 \xrightarrow{g_c} U_2\).

**Proposition 123** (\(\alpha_S\) is Sound and Optimal). Assuming \(\hat{S} \) valid:
(i) \(\hat{S} \subseteq \gamma_S(\alpha_S(\hat{S}))\)
(ii) If \(\hat{S} \subseteq \gamma_S(U)\) then \(\alpha_S(\hat{S}) \subseteq U\).

**Proof.** Trivial using Prop 146 and 147.

### B.3.2 Static Criteria for Gradual Typing

In this section we present the proof of Static Gradual Guarantee for \(\text{GSL}_{\text{Ref}}\).

**Proposition 22** (Static conservative extension). Let \(\vdash_S\) denote \(\text{SSL}_{\text{Ref}}\)'s type system. Then for any static language term \(t \in \text{TERM}, \emptyset; \Sigma; \ell_c \vdash_S t : S\) if and only if \(\emptyset; \Sigma; \ell_c \vdash t : S\).

**Proof.** By induction over the typing derivations. The proof is trivial because static types are given singleton meanings via concretization.

**Definition 76** (Term precision).

\[
\begin{align*}
\text{(Px)} & \quad x \in x \\
\text{(Pb)} & \quad g \subseteq g' \\
\text{(Pu)} & \quad g \subseteq g', \quad \text{unit}_g \subseteq \text{unit}_{g'} \\
\text{(P\lambda)} & \quad \frac{t \subseteq t', \ g_c \subseteq g', \ g \subseteq g'}{(\lambda^c : x : U) \xrightarrow{g} \in (\lambda^c : x : U') \xrightarrow{g'}}, \quad \text{with} \ x \notin \text{REF} \\
\text{(P\oplus)} & \quad \frac{t_1 \subseteq t_1', \ t_2 \subseteq t_2'}{t_1 \oplus t_2 \subseteq t_1' \oplus t_2'} \\
\text{(P\if)} & \quad \frac{t \subseteq t, \ t_1 \subseteq t_1', \ t_2 \subseteq t_2', \ \text{if } t \text{ then } t_1 \text{ else } t_2 \subseteq \text{if } t' \text{ then } t_1 \text{ else } t_2}
\end{align*}
\]
Definition 77 (Type environment precision).

\[
\Gamma \subseteq \Gamma', \ U \subseteq U' \\
\frac{\Gamma', x : U \subseteq \Gamma', x : U'}{\Gamma, x : U \subseteq \Gamma', x : U'}
\]

Lemma 148. If \( \Gamma; \emptyset; g_c \vdash t : U \) and \( \Gamma \subseteq \Gamma' \), then \( \Gamma'; \emptyset; g_c \vdash t : U' \) for some \( U \subseteq U' \).

Proof. Simple induction on typing derivations. \(\square\)

Lemma 149. If \( U_1 \preceq U_2 \) and \( U_1 \subseteq U'_1 \) and \( U_2 \subseteq U'_2 \) then \( U'_1 \preceq U'_2 \).

Proof. By definition of \( \preceq \), there exists \( (S_1, S_2) \in \gamma^2(U_1, U_2) \) such that \( S_1 \preceq S_2 \). \( U_1 \subseteq U'_1 \) and \( U_2 \subseteq U'_2 \) mean that \( \gamma(U_1) \subseteq \gamma(U'_1) \) and \( \gamma(U_2) \subseteq \gamma(U'_2) \), therefore \( (S_1, S_2) \in \gamma^2(U'_1, U'_2) \). \(\square\)

Lemma 150. If \( g_1 \gamma g_2 \preceq g_3 \), \( g_1 \subseteq g'_1 \), \( g_2 \subseteq g'_2 \) and \( g_3 \subseteq g'_3 \), then \( g'_1 \gamma g'_2 \preceq g'_3 \).

Proof. By definition of the consistent judgment, there exists \( (\ell_1, \ell_2, \ell_3) \in \gamma^3(g_1, g_2, g_3) \) such that \( \ell_1 \gamma \ell_2 \preceq \ell_3 \). \( g_1 \subseteq g'_1 \), \( g_2 \subseteq g'_2 \) and \( g_3 \subseteq g'_3 \) mean that \( \gamma(g_1) \subseteq \gamma(g'_1) \), \( \gamma(g_2) \subseteq \gamma(g'_2) \) and \( \gamma(g_3) \subseteq \gamma(g'_3) \) respectively. Therefore \( (\ell_1, \ell_2, \ell_3) \in \gamma^3(g_1, g_2, g_3) \). \(\square\)

Lemma 151. If \( g_1 \preceq g_2 \), \( g_1 \subseteq g'_1 \) and \( g_2 \subseteq g'_2 \), then \( g'_1 \preceq g'_2 \).

Proof. Using almost identical argument of Lemma 150. \(\square\)

Proposition 23 (Static gradual guarantee). Suppose \( g_{c1} \subseteq g_{c2} \) and \( t_1 \subseteq t_2 \).
If \( \emptyset; \emptyset; g_{c1} \vdash t_1 : U_1 \) then \( \emptyset; \emptyset; g_{c2} \vdash t_2 : U_2 \) where \( U_1 \subseteq U_2 \).

Proof. We prove the property on opens terms instead of closed terms: If \( \Gamma; \emptyset; g_{c1} \vdash t_1 : U_1 \), \( g_{c1} \subseteq g_{c2} \) and \( t_1 \subseteq t_2 \) then \( \Gamma; \emptyset; g_{c2} \vdash t_2 : U_2 \) and \( U_1 \subseteq U_2 \).

The proof proceed by induction on the typing derivation.

Case (\( Ux, Ub, Uu \)). Trivial by definition of \( \subseteq \) using \((P x), (P b), (P u)\) respectively.

Case (\( U\lambda \)). Then \( t_1 = (\lambda^{g_c} x : U_1.t) \) and \( U_1 = U'_1 \xrightarrow{g_c} U''_1 \). By \( (U\lambda) \) we know that:

\[
(U\lambda) \frac{\Gamma; x : U_1; \emptyset; g_c \vdash t : U_2'}{\Gamma; \emptyset; g_{c1} \vdash (\lambda^{g_c} x : U'_1.t) : U''_1}
\]

Consider \( g_{c2} \) such that \( g_{c1} \subseteq g_{c2} \) and \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = (\lambda^{g_c} x : U''_1.t') \) and therefore

\[
(U\lambda) \frac{g'_c \subseteq g''_c \quad U_1' \subseteq U''_1 \quad g \subseteq g'}{\lambda^{g_c} x : U'_1.t) \subseteq (\lambda^{g'_c} x : U''_1.t')_g'}
\]

Using induction hypotheses on the premise of \( [D.1] \), \( \Gamma; x : U'_1; \emptyset; g_{c2} \vdash t' : U''_2 \) with \( U_2' \subseteq U''_2 \).

By Lemma 267 \( \Gamma; \emptyset; g_{c2} \vdash t' : U''_2 \) where \( U_2' \subseteq U''_2 \). Then we can use rule \( (U\lambda) \) to derive:

\[
(U\lambda) \frac{\Gamma; x : U''_1; \emptyset; g_{c2} \vdash t' : U''_2}{\Gamma; \emptyset; g_{c1} \vdash (\lambda^{g''_c} x : U'_1.t')_g' \xrightarrow{g''_c} g' U''_2}
\]

254
Using the premise of [B.2] and the definition of type precision we can infer that
\[ U' \xrightarrow{g} U_2' \subseteq U_1'' \xrightarrow{g} U_2'' \]
and the result holds.

**Case (Uo).** This case can not happen because initial programs do not contain locations.

**Case (Uprot).** Then \( t_1 = \text{prot}_g(t) \) and \( U_1 = U \parallel g \). By (Uprot) we know that:

\[
\begin{align*}
(U_{\text{prot}}) & \quad \frac{\Gamma; \varnothing; g_{c1} \parallel g \vdash t : U}{\Gamma; \varnothing; g_{c1} \vdash \text{prot}_g(t) : U \parallel g} \\
\end{align*}
\]

(B.3)

Consider \( g_{c2} \) such that \( g_{c1} \parallel g_{c2} \) and \( t_2 \) such that \( t_1 \parallel t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = \text{prot}_g'(t') \) and therefore

\[
\begin{align*}
(P_{\text{prot}}) & \quad \frac{t \subseteq t' \quad g \subseteq g'}{\text{prot}_g(t) \subseteq \text{prot}_g'(t')} \\
\end{align*}
\]

(B.4)

By definition of join on consistent labels, \( g_{c1} \parallel g \parallel g_{c2} \parallel g' \). Using induction hypotheses on the premises of [B.3] we can use rule (Uprot) to derive:

\[
\begin{align*}
(U_{\text{prot}}) & \quad \frac{\Gamma; \varnothing; g_{c2} \parallel g' \vdash t' : U'}{\Gamma; \varnothing; g_{c2} \vdash \text{prot}_{g'}(t') : U' \parallel g'} \\
\end{align*}
\]

For some \( U' \), where \( U \subseteq U' \). Using the premise of [B.4] and the definition of join we can infer that

\[ U \parallel g \subseteq U' \parallel g' \]

and the result holds.

**Case (U⊕).** Then \( t_1 = t_1' \oplus t_2' \) and \( U_1 = \text{Bool}_{(g_1 \parallel g_2)} \). By (U⊕) we know that:

\[
\begin{align*}
(U_{\oplus}) & \quad \frac{\Gamma; \varnothing; g_{c1} \vdash t_1' : \text{Bool}_{g_1}}{\Gamma; \varnothing; g_{c1} \vdash t_1 \oplus t_2' : \text{Bool}_{(g_1 \parallel g_2)}} \\
& \quad \frac{\Gamma; \varnothing; g_{c2} \vdash t_2' : \text{Bool}_{g_2}}{\Gamma; \varnothing; g_{c2} \vdash t_1' \oplus t_2' : \text{Bool}_{(g_1 \parallel g_2)}} \\
\end{align*}
\]

(B.5)

Consider \( g_{c2} \) such that \( g_{c1} \parallel g_{c2} \) and \( t_2 \) such that \( t_1 \parallel t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t_2'' \oplus t_2'' \) and therefore

\[
\begin{align*}
(P_{\oplus}) & \quad \frac{t_1' \subseteq t_1'' \quad t_2' \subseteq t_2''}{t_1' \oplus t_2' \subseteq t_1'' \oplus t_2''} \\
\end{align*}
\]

(B.6)

Using induction hypotheses on the premises of [B.5] we can use rule (U⊕) to derive:

\[
\begin{align*}
(U_{\oplus}) & \quad \frac{\Gamma; \varnothing; g_{c2} \vdash t_1'' : \text{Bool}_{g_1'}}{\Gamma; \varnothing; g_{c2} \vdash t_1'' \oplus t_2' : \text{Bool}_{(g_1' \parallel g_2)}} \\
& \quad \frac{\Gamma; \varnothing; g_{c2} \vdash t_2'' : \text{Bool}_{g_2'}}{\Gamma; \varnothing; g_{c2} \vdash t_1' \oplus t_2' : \text{Bool}_{(g_1' \parallel g_2')}} \\
\end{align*}
\]

Where \( g_1' \subseteq g_1'' \) and \( g_2' \subseteq g_2'' \). Using the premise of [B.6] and the definition of type precision we can infer that

\[
\begin{align*}
(\text{Bool}_{(g_1' \parallel g_2')}) \subseteq (\text{Bool}_{(g_1'' \parallel g_2'')}) \\
\end{align*}
\]

and the result holds.
Case (Uapp). Then \( t_1 = t'_1 \) \( t_2 \) and \( U_1 = U_{12} \) \( g \). By (Uapp) we know that:

\[
\begin{align*}
\Gamma; \omega; g_c \vdash t'_1 : U_{11} \quad &\quad \Gamma; \omega; g_c \vdash t'_2 : U'_{12} \\
U'_2 \subseteq U_{11} \quad &\quad g \, \gamma \, g_c \leq g'_c
\end{align*}
\]  

By (Uapp)\( \vdash \)

\[
\Gamma; \omega; g_c \vdash t'_1 \, t'_2 : U_{12} \, \gamma \, g
\]

Consider \( g_{c2} \) such that \( g_{c1} \subseteq g_{c2} \) and \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 \, t''_2 \) and therefore

\[
\begin{align*}
\Gamma; \omega; g_c \vdash t''_1 \, t''_2 : U_{12} \, \gamma \, g' \\
t''_1 \subseteq t''_2
\end{align*}
\]  

Using induction hypotheses on the premises of [D.11] \( \Gamma; \omega; g_{c2} \vdash t''_1 : U'_{11} \, g''_{U_{12}} \) and \( \Gamma; \omega; g_{c2} \vdash t''_2 : U''_2 \), where \( U''_2 \subseteq U''_1 \, U_{11} \, g''_{U_{12}} \subseteq U'_{11} \, g''_{U_{12}} \). By Lemma 268, \( U''_2 \subseteq U'_{11} \). By definition of precision of types, \( g'_c \subseteq g''_c \) and \( g \subseteq g' \), therefore by Lemma 150, \( g' \, \gamma \, g_{c2} \leq g''_c \). Then we can use rule (Uapp) to derive:

\[
\begin{align*}
\Gamma; \omega; g_c \vdash t''_1 : U'_{11} \quad &\quad \Gamma; \omega; g_c \vdash t''_2 : U''_2 \\
U''_2 \subseteq U_{11} \quad &\quad g' \, \gamma \, g_c \leq g''_c
\end{align*}
\]  

Using the definition of type precision we can infer that \( U_{12} \, \gamma \, g \subseteq U'_{12} \, \gamma \, g' \)

and the result holds.

Case (Uif). Then \( t_1 = if \, t \, then \, t'_1 \, else \, t_2 \) and \( U_1 = (U'_1 \, \tilde{\wedge} \, U'_2) \, \gamma \, g \). By (Uif) we know that:

\[
\begin{align*}
\Gamma; \omega; g_c \vdash t : \text{Bool}_g \\
\Gamma; \omega; g_c \, g \, g' + t'_1 : U'_{1} \\
\Gamma; \omega; g_c \, g + t_2 : U'_{2}
\end{align*}
\]  

By (Uif)\( \vdash \)

\[
\Gamma; \omega; g_c \vdash if \, t \, then \, t'_1 \, else \, t_2 : (U'_1 \, \tilde{\wedge} \, U'_2) \, \gamma \, g
\]

Consider \( g_{c2} \) such that \( g_{c1} \subseteq g_{c2} \) and \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = if \, t' \, then \, t''_1 \, else \, t''_2 \) and therefore

\[
\begin{align*}
t \subseteq t' \\
t_1 \subseteq t'_1 \\
t_2 \subseteq t''_2
\end{align*}
\]  

Consider any \( \ell' \) such that \( \ell \subseteq \ell' \). As \( g_{c1} \, \gamma \, g \subseteq g_{c2} \, \gamma \, g' \) then we can use induction hypotheses on the premises of [C.7] and derive:

\[
\begin{align*}
\Gamma; \omega; g_c \vdash t' : \text{Bool}_{g'} \\
\Gamma; \omega; g_{c2} \, g' + t'_1 : U''_{1} \\
\Gamma; \omega; g_{c2} \, g' + t''_2 : U''_{2}
\end{align*}
\]  

By (Uif)\( \vdash \)

\[
\Gamma; \omega; g_c \vdash if \, t' \, then \, t'_1 \, else \, t''_2 : (U''_{1} \, \tilde{\wedge} \, U''_{2}) \, \gamma \, g'
\]

Where \( U'_1 \subseteq U''_{1} \) and \( U'_2 \subseteq U''_{2} \). Using the definition of type precision we can infer that

\[
(U'_1 \, \tilde{\wedge} \, U'_2) \, \gamma \, g \subseteq (U''_{1} \, \tilde{\wedge} \, U''_{2}) \, \gamma \, g'
\]

and the result holds.
Case \((U::)\). Then \(t_1 = t :: U_1\). By \((U::)\) we know that:

\[
\frac{\Gamma; \varnothing; g_{c_1} \vdash t : U'_1 \quad U'_1 \sqsubseteq U_1}{\Gamma; \varnothing; g_{c_1} \vdash t :: U'_1 : U_1}
\] (B.11)

Consider \(g_{c_2}\) such that \(g_{c_1} \sqsubseteq g_{c_2}\) and \(t_2\) such that \(t_1 \sqsubseteq t_2\). By definition of term precision \(t_2\) must have the form \(t_2 = t' :: U_2\) and therefore

\[
\frac{t \sqsubseteq t' \quad U_1 \subseteq U_2}{t :: U_1 \sqsubseteq t' :: U_2}
\] (B.12)

Using induction hypotheses on the premises of \([B.7]\): \(\Gamma; \varnothing; g_{c_1} \vdash t': U'_2\) where \(U'_1 \sqsubseteq U'_2\). We can use rule \((U::)\) and Lemma 268 to derive:

\[
\frac{\Gamma; \varnothing; g_{c_2} \vdash t' : U'_2 \quad U'_2 \sqsubseteq U_2}{\Gamma; \varnothing; g_{c_2} \vdash t' :: U_2 : U_2}
\]

Where \(U_1 \sqsubseteq U_2\) and the result holds.

Case \((U\text{ref})\). Then \(t_1 = \text{ref}^{U} t\) and \(U_1 = \text{Ref}_{g_1} U\). By \((U\text{ref})\) we know that:

\[
\frac{\Gamma; \varnothing; g_{c_1} \vdash t : U'_1 \quad U'_1 \sqsubseteq U \quad g_{c_1} \sqsubseteq \text{label}(U)}{\Gamma; \varnothing; g_{c_1} \vdash \text{ref}^{U} t : \text{Ref}_{\perp} U}
\] (B.13)

Consider \(g_2\) such that \(g_{c_1} \sqsubseteq g_{c_2}\) and \(t_2\) such that \(t_1 \sqsubseteq t_2\). By definition of term precision \(t_2\) must have the form \(t_2 = t' \quad \text{ref}^{U'} t'\) and therefore

\[
\frac{t \sqsubseteq t' \quad U \sqsubseteq U'}{\text{ref}^{U} t \sqsubseteq \text{ref}^{U'} t'}
\] (B.14)

Using induction hypotheses on the premises of \([B.13]\) we can use rule \((U\text{ref})\) and Lemma 268 and \([151]\) to derive:

\[
\frac{\Gamma; \varnothing; g_{c_2} \vdash t' : U''_1 \quad U''_1 \sqsubseteq U' \quad g_{c_2} \sqsubseteq \text{label}(U')}{\Gamma; \varnothing; g_{c_2} \vdash \text{ref}^{U'} t' : \text{Ref}_{\perp} U'}
\]

Where \(U \sqsubseteq U'\) and \(U'_1 \sqsubseteq U''_1\). Using the the definition of type precision we can infer that

\[
\frac{\text{Ref}_{\perp} U \sqsubseteq \text{Ref}_{\perp} U'}{}
\]

and the result holds.

Case \((U\text{deref})\). Then \(t_1 = !t\) and \(U_1 = U \triangledown g\). By \((U\text{deref})\) we know that:

\[
\frac{\Gamma; \varnothing; g_{c_1} \vdash t : \text{Ref}_{g} U}{\Gamma; \varnothing; g_{c_1} \vdash !t : U \triangledown g}
\] (B.15)

Consider \(g_2\) such that \(g_{c_1} \sqsubseteq g_{c_2}\) and \(t_2\) such that \(t_1 \sqsubseteq t_2\). By definition of term precision \(t_2\) must have the form \(t_2 = !t'\) and therefore

\[
\frac{t \sqsubseteq t' \quad \text{!t} \sqsubseteq \text{!t}'}{\text{!t} \sqsubseteq \text{!t}'}
\] (B.16)
Using induction hypotheses on the premises of \textbf{B.15}, we can use rule \((U\text{deref})\) to derive:

\[
(U\text{deref}) \quad \Gamma; \emptyset; g; c_2 \vdash t' : \text{Ref}_{g'} U' \\
\quad \Gamma; \emptyset; g; c_2 \vdash !t' : U' \Upsilon g'
\]

Where \(g \sqsubseteq g'\) and \(U \sqsubseteq U'\). Using the premise of \textbf{B.16} and the definition of type precision we can infer that

\[
U \Upsilon g \sqsubseteq U' \Upsilon g'
\]

and the result holds.

\textit{Case \((U\text{asgn})\).} Then \(t_1 = t_1'; = t_2'\) and \(U_1 = \text{Unit}_\bot\). By \((U\text{asgn})\) we know that:

\[
(U\text{asgn}) \quad \Gamma; \emptyset; g; c_1 \vdash t_1' : \text{Ref}_{g} U'_1 \\
\quad \Gamma; \emptyset; g; c_1 \vdash t_2' : U'_2 \\
\quad U'_2 \preceq U'_1 \\
\quad g \Upsilon g; c_1 \preceq \text{label}(U'_1) \\
\quad \Gamma; \emptyset; g; c_2 \vdash t'_1 := t'_2 : \text{Unit}_\bot
\]

Consider \(g; c_2\) such that \(g; c_1 \sqsubseteq g; c_2\) and \(t_2\) such that \(t_1 \sqsubseteq t_2\). By definition of term precision \(t_2\) must have the form \(t_2 = t''_1 := t''_2\) and therefore

\[
(P\text{asgn}) \quad t'_1 \sqsubseteq t''_1 \\
\quad t'_2 \sqsubseteq t''_2 \\
\quad t'_1 := t'_2 \sqsubseteq t''_1 := t''_2
\]

Using induction hypotheses on the premises of \textbf{B.17}, \(\Gamma; \emptyset; g; c_2 \vdash t_1' : \text{Ref}_{g'} U'_1\) and \(\Gamma; \emptyset; g; c_2 \vdash t_2' : U'_2\), where \(\text{Ref}_{g} U'_1 \sqsubseteq \text{Ref}_{g'} U'_2\) and \(U'_2 \sqsubseteq U'_2\). By definition of precision on types and Lemma \textbf{268}, \(U'_2 \preceq U'_2\). Also, as, \(g \sqsubseteq g'\) and \(U_1 \sqsubseteq U'_1\), by Lemma \textbf{150}, \(g' \Upsilon g; c_2 \preceq \text{label}(U'_1)\). Then we can use rule \((U\text{asgn})\) to derive:

\[
(U\text{asgn}) \quad \Gamma; \emptyset; g; c_2 \vdash t'_1 : \text{Ref}_{g'} U''_1 \\
\quad \Gamma; \emptyset; g; c_2 \vdash t'_2 : U''_2 \\
\quad U''_2 \preceq U'_1 \\
\quad g' \Upsilon g; c_2 \preceq \text{label}(U'_1) \\
\quad \Gamma; \emptyset; g; c_2 \vdash t''_1 := t''_2 : \text{Unit}_\bot
\]

Using the definition of type precision we can infer that

\[
\text{Unit}_\bot \sqsubseteq \text{Unit}_\bot
\]

and the result holds.
B.4 Gradualizing the Dynamic Semantics

In this section we present the formalization of the evidences for GSL_{Ref}. §B.4.1 presents the structure of evidence and the abstraction and concretization functions. In §B.4.2, we show how to calculate the initial evidence. In particular we give definition for the initial evidence of consistent judgments for labels and types. In §B.4.2, we present how to evolve evidence. We define the consistent transitivity operator, the meet operator and join of evidences. In §B.4.4, we present the algorithmic definitions of initial evidence and consistent transitivity. Finally, in §B.4.5, we present some of the proofs of the propositions for evidence presented.

B.4.1 Precise Evidence for Consistent Security Judgments

Definition 78 (Interval). An interval is a bounded unknown label $[\ell_1, \ell_2]$ where $\ell_1$ is the upper bound and $\ell_2$ is the lower bound.

$\forall \ell \in \text{Label}^2$

$\ell := [\ell, \ell]$ (interval)

Definition 79 (Interval Concretization). Let $\gamma_i : \text{Label}^2 \rightarrow \mathcal{P}(\text{LABEL})$ be defined as follows:

$\gamma_i([\ell_1, \ell_2]) = \{\ell \mid \ell \in \text{LABEL}, \ell_1 \preceq \ell \preceq \ell_2\}$

We can only concretize valid intervals:

Definition 80 (Valid Gradual Label).

$\frac{\ell_1 \preceq \ell_2}{\text{valid}([\ell_1, \ell_2])}$

Definition 81 (Label Evidence Concretization). Let $\gamma_{\epsilon_i} : \text{LABEL}^4 \rightarrow \mathcal{P}(\text{LABEL}^2)$ be defined as follows:

$\gamma_{\epsilon_i}([\ell_1, \ell_2]) = \{[\ell_1, \ell_2] \mid \ell_1 \in \gamma_i(\ell_1), \ell_2 \in \gamma_i(\ell_2)\}$

Definition 82 (Interval Abstraction). Let $\alpha_i : \mathcal{P}(\text{LABEL}) \rightarrow \text{Label}^2$ be defined as follows:

$\alpha_i(\emptyset)$ is undefined

$\alpha_i(\{\ell\}) = [\prec \ell, \prec \ell]$ otherwise

Definition 83 (Label Evidence Abstraction). Let $\alpha_{\epsilon_i} : \mathcal{P}(\text{LABEL}^2) \rightarrow \text{LABEL}^4$ be defined as follows:

$\alpha_{\epsilon_i}(\emptyset)$ is undefined

$\alpha_{\epsilon_i}(\{[\ell_1, \ell_2]\}) = \langle\alpha_i(\{\ell_1\}), \alpha_i(\{\ell_2\})\rangle$ otherwise

Definition 84 (Type Evidence). An evidence type is a gradual type labeled with an interval:

$E \in \text{GETYPE}, \quad \ell \in \text{Label}^2$

$E ::= \text{Bool}, \ | \ E \xrightarrow{i} E | \text{Ref}_i E | \text{Unit} \quad \text{(evidence types)}$
Definition 85 (Type Evidence Concretization). Let $\gamma_E : \text{GETYPE} \rightarrow \mathcal{P}(\text{TYPE})$ be defined as follows:

\[
\gamma_E(\text{Bool}_i) = \{ \text{Bool}_\ell \mid \ell \in \gamma_i(i) \}
\]
\[
\gamma_E(E_1 \rightarrow E_2) = \gamma_E(E_1)^{\rightarrow} \gamma(E_2)
\]
\[
\gamma_E(\text{Ref}_i E) = \{ \text{Ref}_\ell S \mid \ell \in \gamma_i(i), S \in \gamma_E(E) \}
\]

where $\rightarrow$ is the set of all possible combinations of function types, using each member of the sets obtained by the $\gamma_E$ and $\gamma_i$ functions.

Definition 86 (Evidence Concretization). Let $\gamma_{\varepsilon_i} : \text{GETYPE}^2 \rightarrow \mathcal{P}(\text{TYPE}^2)$ be defined as follows:

\[
\gamma_{\varepsilon_i}(\langle E_1, E_2 \rangle) = \{ \langle S_1, S_2 \rangle \mid S_1 \in \gamma_E(E_1), S_2 \in \gamma_E(E_2) \}
\]

Definition 87 (Type Evidence Abstraction). Let the abstraction function $\alpha_E : \mathcal{P}(\text{TYPE}) \rightarrow \text{GETYPE}$ be defined as:

\[
\alpha_E(\{ \text{Bool}_i \}) = \text{Bool}_{\alpha_i((i))}
\]
\[
\alpha_E(\{ \text{Ref}_i S_i \}) = \text{Ref}_{\alpha_i((i))} \alpha_E(\{ S_i \})
\]

where $\alpha_E(\hat{S})$ is undefined otherwise.

Definition 88 (Evidence Abstraction). Let $\alpha_\varepsilon : \mathcal{P}(\text{TYPE}^2) \rightarrow \text{GETYPE}^2$ be defined as follows:

\[
\alpha_\varepsilon(\emptyset) \text{ is undefined}
\]
\[
\alpha_\varepsilon(\{ \langle S_{1i}, S_{2i} \rangle \}) = \langle \alpha_E(\{ S_{1i} \}), \alpha_E(\{ S_{2i} \}) \rangle \text{ otherwise}
\]

We can only abstract valid sets of security types, i.e. in which elements only defer by security labels.

Definition 89 (Valid Type Sets).

\[
\begin{array}{ccc}
\text{valid}(\{ \text{Bool}_i \}) & \text{valid}(\{ S_{1i} \}) & \text{valid}(\{ S_{2i} \}) \\
\text{valid}(\{ S_{1i} \rightarrow S_{12} \}) & \text{valid}(\{ \text{Ref}_i S_i \}) & \text{valid}(\{ \text{Unit}_i \})
\end{array}
\]

Proposition 152 ($\alpha_i$ is Sound). If $\hat{\ell}$ is not empty, then $\hat{\ell} \subseteq \gamma_i(\alpha_i(\hat{\ell}))$.

Proposition 153 ($\alpha_i$ is Optimal). If $\hat{\ell}$ is not empty, and $\hat{\ell} \subseteq \gamma_i(i)$ then $\alpha_i(\hat{\ell}) \subseteq i$.

Proposition 154 ($\alpha_E$ is Sound). If $\text{valid}(\hat{S})$ then $\hat{S} \subseteq \gamma_E(\alpha_E(\hat{S}))$.  

260
**Proposition 155** ($\alpha_E$ is Optimal). If $\text{valid}(\hat{S})$ and $\hat{S} \subseteq \gamma_E(E)$ then $\alpha_E(\hat{S}) \subseteq E$.

With concretization of security type, we can now define security type precision.

**Definition 90** (Interval and Type Evidence Precision).

1. $i_1$ is less imprecise than $i_2$, notation $i_1 \sqsubseteq i_2$, if and only if $\gamma_{\varepsilon_\ell}(i_1) \subseteq \gamma_{\varepsilon_\ell}(i_2)$; inductively:

\[
[\ell_1, \ell_2] \subseteq [\ell_3, \ell_4] \\
\frac{\ell_3 \ll \ell_1}{\ell_2 \ll \ell_4}
\]

2. $E_1$ is less imprecise than $E_2$, notation $E_1 \sqsubseteq E_2$, if and only if $\gamma_E(E_1) \subseteq \gamma_E(E_2)$; inductively:

\[
\begin{array}{c}
i_1 \sqsubseteq i_2 \\
\text{Bool}_1 \sqsubseteq \text{Bool}_2 \\
\end{array} \\
\begin{array}{c}
E_{11} \sqsubseteq E_{21} \\
i_1 \sqsubseteq i_2 \\
E_{12} \sqsubseteq E_{22} \\
i_1' \sqsubseteq i_2' \\
E_{11} \not \rightarrow_i \ E_{12} \not \rightarrow_i E_{21} \not \rightarrow_i E_{22} \not \rightarrow_i E_{22}
\end{array}
\]

\[
\begin{array}{c}
i_1 \sqsubseteq i_2 \\
\text{Ref}_{i_1} E_1 \sqsubseteq \text{Ref}_{i_2} E_2
\end{array}
\]

**B.4.2 Initial evidence**

With the definition of concretization and abstraction we can now define the initial evidence of label ordering and subtyping:

**Definition 91** (Initial Evidence of label ordering). Let $F_1 : \text{LABEL}^n \rightarrow \text{LABEL}$ and $F_2 : \text{LABEL}^m \rightarrow \text{LABEL}$ be functions over labels. The initial evidence of the judgment $F_1(\overline{g_i}) \ll F_2(\overline{g_j})$, notation $\mathfrak{J}[F_1(\overline{g_i}) \ll F_2(\overline{g_j})]$, is defined as follows:

\[
\mathfrak{J}[F_1(g_1, ...g_n) \ll F_2(g_{n+1}, ...g_{n+m})] = \alpha_{\varepsilon_\ell}(\{\langle F_1(\overline{g_i}), F_2(\overline{g_j})\rangle | \langle \overline{g_i}\rangle \in \gamma^n(\overline{g_{[1/n]}}, \langle \overline{g_j}\rangle \in \gamma^n(\overline{g_{[n+1/m]}}, F_1(\overline{g_i}) \ll F_2(\overline{g_j}))\})
\]

Suppose $F_1 = F_{i_1}$

**Definition 92** (Initial Evidence of subtyping). Let $F_1 : \text{TYPE}^n \rightarrow \text{TYPE}$ and $F_2 : \text{TYPE}^m \rightarrow \text{TYPE}$ be functions over types. The initial evidence of the judgment $F_1(\overline{U_i}) \ll F_2(\overline{U_j})$, notation $\mathfrak{J}[F_1(\overline{U_i}) \ll F_2(\overline{U_j})]$, is defined as follows:

\[
\mathfrak{J}[F_1(U_1, ...U_n) \ll F_2(U_{n+1}, ...U_{n+m})] = \alpha_{\varepsilon_\ell}(\{\langle F_1(\overline{S_i}), F_2(\overline{S_j})\rangle | \langle \overline{S_i}\rangle \in \gamma_S^n(\overline{U_{[1/n]}}, \langle \overline{S_j}\rangle \in \gamma_S^m(\overline{U_{[n+1/m]}}, F_1(\overline{S_i}) \ll F_2(\overline{S_j}))\})
\]

**Proposition 156.** [Elaboration preserves typing] Consider $\Gamma; \Sigma; g_c \vdash t : U$ then if $\Gamma; \Sigma; g_c \vdash t \leadsto t' : U$, and $\varepsilon = \mathfrak{J}[\gamma_\varepsilon(\ell_c)]$, then $\Gamma; \Sigma; \varepsilon g_c \vdash t' : U$.

**Proof.** Straightforward induction on judgment $\Gamma; \Sigma; g_c \vdash t : U$.

\[\square\]
B.4.3 Evolving evidence: Consistent Transitivity

Now that we know how to extract initial evidence from consistent judgments, we need a way to combine evidences to use during program evaluation, i.e. we need to find a way to \textit{evolve} evidence. We define \textit{consistent transitivity} for label ordering and subtyping, $\circ^{<}$ and $\circ^{\leq}$ respectively, to combine evidences as follows:

\begin{definition}[Consistent transitivity for label ordering] Let function $\circ^{<} : \text{INTERVAL}^{2} \times \text{INTERVAL}^{2} \rightarrow \text{LABEL}^{2}$ be defined as:
\begin{align*}
\langle t_{11}, t_{12} \rangle \circ^{<} \langle t_{21}, t_{22} \rangle &= \alpha_{\varepsilon}((\ell_{11}, \ell_{22}) \in \gamma_{\varepsilon}(\langle t_{11}, t_{22} \rangle) \mid \exists \ell \in \gamma_{i}(t_{12}) \cap \gamma_{i}(t_{21}). \ell_{11} \not\approx \ell \land \ell \not\ll \ell_{22})
\end{align*}
\end{definition}

\begin{proposition}
Suppose $\varepsilon_{1} \vdash F_{1}(\overline{g_{i}}) \not\ll F_{2}(\overline{g_{j}})$ and $\varepsilon_{2} \vdash F_{2}(\overline{g_{j}}) \ll F_{3}(\overline{g_{k}})$.
If $\varepsilon_{1} \circ^{<} \varepsilon_{2}$ is defined, then $\varepsilon_{1} \circ^{<} \varepsilon_{2} \vdash F_{1}(\overline{g_{i}}) \not\ll F_{3}(\overline{g_{k}})$
\end{proposition}

\begin{proposition}
$\gamma_{i}(t_{1} \cap t_{2}) = \gamma_{i}(t_{1}) \cap \gamma_{i}(t_{2})$.
where $t \cap t' = \alpha(\gamma(t) \cap \gamma(t'))$.
\end{proposition}

\begin{proposition}
$\langle t_{1}, t_{21} \rangle \circ^{<} \langle t_{22}, t_{3} \rangle = \Delta^{<}(t_{1}, t_{21} \cap t_{22}, t_{3})$
where
\begin{align*}
\Delta^{<}(t_{1}, t_{2}, t_{3}) &= \alpha_{\varepsilon}((\ell_{1}, \ell_{3}) \in \varepsilon(\langle t_{1}, t_{3} \rangle) \mid \exists \ell_{2} \in \gamma_{i}(t_{2}). \ell_{1} \not\ll \ell_{2} \land \ell_{2} \not\ll \ell_{3})
\end{align*}
\end{proposition}

\begin{definition}[Consistent transitivity for subtyping]
We deduce evidence for consistent transitivity for subtyping:
\begin{align*}
\langle E_{11}, E_{12} \rangle \vdash F_{1}(\overline{U_{i}}) \ll: F_{2}(\overline{U_{j}}) \quad \langle E_{21}, E_{22} \rangle \vdash F_{2}(\overline{U_{j}}) \ll: F_{3}(\overline{U_{k}})
\end{align*}
where $\circ^{<} : \text{ETYPE}^{2} \times \text{ETYPE}^{2} \rightarrow \text{ETYPE}^{2}$ is defined as:
\begin{align*}
\langle E_{11}, E_{12} \rangle \circ^{<} \langle E_{21}, E_{22} \rangle &= \\
&\alpha_{\varepsilon}((S_{11}, S_{22}) \in \varepsilon(\langle E_{11}, E_{22} \rangle) \mid \exists S \in \gamma_{E}(E_{12}) \cap \gamma_{E}(E_{21}). S_{11} \ll: S \land S \ll: S_{22})
\end{align*}
\end{definition}

\begin{proposition}
$\gamma_{E}(E_{1} \cap E_{2}) = \gamma_{E}(E_{1}) \cap \gamma_{E}(E_{2})$.
\end{proposition}

Then following AGT,

\begin{proposition}
$\langle E_{1}, E_{21} \rangle \circ^{<} \langle E_{22}, E_{3} \rangle = \Delta^{<}(E_{1}, E_{21} \cap E_{22}, E_{3})$
where
\begin{align*}
\Delta^{<}(E_{1}, E_{2}, E_{3}) &= \alpha_{\varepsilon}((S_{1}, S_{3}) \in \varepsilon(\langle E_{1}, E_{3} \rangle) \mid \exists S_{2} \in \gamma_{i}(E_{2}). S_{1} \ll: S_{2} \land S_{2} \ll: S_{3})
\end{align*}
\end{proposition}

\begin{definition}[Intervals join]
$[\ell_{1}, \ell_{2}] \bar{\gamma} [\ell_{3}, \ell_{4}] = [\ell_{1} \gamma \ell_{3}, \ell_{2} \gamma \ell_{4}]$
\end{definition}
Definition 96 (Evidence label join).

\[ \langle t_1, t_2 \rangle \mathcal{R} \langle t_3, t_4 \rangle = \langle t_1 \mathcal{R} t_3, t_2 \mathcal{R} t_4 \rangle \]

Definition 97.

\[ \text{Bool}_{t_1} \mathcal{R} t_2 = \text{Bool}_{(t_1 \mathcal{R} t_2)} \]

\[ E_1 \xrightarrow{t_2} t_1, E_2 \mathcal{R} t_3 = E_1 \xrightarrow{t_2} (t_1 \mathcal{R} t_3), E_2 \]

\[ \text{Ref}_{t_1} E \mathcal{R} t_2 = \text{Ref}_{(t_1 \mathcal{R} t_2)} E \]

Definition 98.

\[ \langle E_1, E_2 \rangle \mathcal{R} \langle t_1, t_2 \rangle = \langle E_1 \mathcal{R} t_1, E_2 \mathcal{R} t_2 \rangle \]

Proposition 161. If \( \varepsilon_S \vdash U_1 \lesssim U_2 \) and \( \varepsilon_I \vdash g_1 \lesssim g_2 \) then \( \varepsilon_S \mathcal{R} \varepsilon_I \vdash U_1 \mathcal{R} g_1 \lesssim U_2 \mathcal{R} g_2 \)

B.4.4 Algorithmic definitions

This section gives algorithmic definitions of consistent transitivity and initial evidence for label ordering and subtyping.

Label Evidences

Definition 99 (Intervals join).

\[ [\ell_1, \ell_2] \mathcal{R} [\ell_3, \ell_4] = [\ell_1 \mathcal{R} \ell_3, \ell_2 \mathcal{R} \ell_4] \]

Definition 100 (Intervals meet).

\[ [\ell_1, \ell_2] \mathcal{H} [\ell_3, \ell_4] = [\ell_1 \mathcal{H} \ell_3, \ell_2 \mathcal{H} \ell_4] \]

Definition 101. Let \( F_1 : \text{GLABEL}^n \to \text{GLABEL} \) and \( F_2 : \text{GLABEL}^m \to \text{GLABEL} \). The initial evidence for consistent judgment \( F_1(\overline{\pi_1}) \lesssim F_2(\overline{\pi_2}) \) is defined as follows:

\[
\begin{align*}
\text{bounds}(?) &= [\bot, \top] \\
\text{bounds}(&\ell) = [\ell, \ell] \\
\text{bounds}(x_1 \mathcal{R} x_2) &= \text{bounds}(x_1) \mathcal{R} \text{bounds}(x_2) \\
\text{bounds}(x_1 \mathcal{H} x_2) &= \text{bounds}(x_1) \mathcal{H} \text{bounds}(x_2) \\
\text{bounds}(x_1 \mathcal{H} x_2) &= \text{bounds}(x_1) \mathcal{H} \text{bounds}(x_2) \\
\text{bounds}(F_1(\overline{\pi_1}) \mathcal{R} F_2(\overline{\pi_2})) &= \text{bounds}(F_1(\overline{\pi_1})) \mathcal{R} \text{bounds}(F_2(\overline{\pi_2})) \\
\text{bounds}(F_1(\overline{\pi_1}) \mathcal{H} F_2(\overline{\pi_2})) &= \text{bounds}(F_1(\overline{\pi_1})) \mathcal{H} \text{bounds}(F_2(\overline{\pi_2})) \\
\text{bounds}(F_1(\overline{\pi_1}) \mathcal{H} F_2(\overline{\pi_2})) &= \text{bounds}(F_1(\overline{\pi_1})) \mathcal{H} \text{bounds}(F_2(\overline{\pi_2})) \\
\end{align*}
\]

\[
\begin{align*}
\text{bounds}(F_1(\overline{\pi_1})) &= [\ell_1, \ell_2] \\
\text{bounds}(F_2(\overline{\pi_2})) &= [\ell_1', \ell_2'] \\
\end{align*}
\]

\[ j(F_1(g_1, \ldots g_n) \lesssim F_2(g_{n+1}, \ldots g_{n+m})) = \langle [\ell_1, \ell_2], [\ell_1', \ell_2'] \rangle \]
where \( F_1: \text{GLabel}^n \to \text{GLabel} \) and \( F_2: \text{GLabel}^m \to \text{GLabel} \).

\[
g^\cup(F(g_1, \ldots, g_n)) = g(F(g_1, \ldots, g_n) \preceq F(g_1, \ldots, g_n))
\]

The algorithmic definition of meet:

\[
{[\ell_1 \land \ell_2]} \cap {[\ell_3 \land \ell_4]} = {[\ell_1 \lor \ell_3 \land \ell_2 \land \ell_4]} \quad \text{if } \text{valid}([\ell_1 \lor \ell_3 \land \ell_2 \land \ell_4])
\]

\(\_ \cap \_\) undefined otherwise

We calculate the algorithmic definition of \( \triangle \preceq \):

\[
\triangle \preceq ([\ell_1 \land \ell_2], [\ell_3 \land \ell_4], [\ell_5 \land \ell_6]) = \langle [\ell_1 \land \ell_2 \land \ell_4 \land \ell_6], [\ell_1 \lor \ell_3 \lor \ell_5 \land \ell_6] \rangle
\]

Type Evidences

We define a function \( \text{liftP}() \) to transform functions over types into functions over labels. Also we define function \( \text{invert}() \) to invert the operator on types, used in the domain and latent effect of function types. Finally we define function \( \text{tomeet}() \) to transform type operators into meets, given the invariant property of references.

We start defining a pattern of operations:

**Definition 102 (Operation pattern).**

\[
\begin{align*}
P^T & \in \text{GPattern}, P^t \in \text{LPattern} \\
P^T & ::= \_ | P^T \text{op}^T P^T \quad \text{(pattern on types)} \\
op^T & ::= \lor | \land | \cap \quad \text{(operations on types)} \\
P^t & ::= \_ | P^t \text{op}^t P^t \quad \text{(pattern on labels)} \\
op^t & ::= \land | \lor | \cap \quad \text{(operations on labels)}
\end{align*}
\]

\[
\begin{align*}
\text{liftP}(\_) & = \_ \\
\text{liftP}(P^T \lor P^T_2) & = \text{liftP}(P^T_1) \lor \text{liftP}(P^T_2) \\
\text{liftP}(P^T_1 \land P^T_2) & = \text{liftP}(P^T_1) \land \text{liftP}(P^T_2) \\
\text{liftP}(P^T_1 \cap P^T_2) & = \text{liftP}(P^T_1) \cap \text{liftP}(P^T_2) \\
\text{invert}(\_) & = \_ \\
\text{invert}(P^T \lor P^T_2) & = \text{invert}(P^T_1) \lor \text{invert}(P^T_2) \\
\text{invert}(P^T_1 \land P^T_2) & = \text{invert}(P^T_1) \land \text{invert}(P^T_2) \\
\text{invert}(P^T_1 \cap P^T_2) & = \text{invert}(P^T_1) \cap \text{invert}(P^T_2) \\
\text{tomeet}(\_) & = \_ \\
\text{tomeet}(P^T \lor P^T_2) & = \text{tomeet}(P^T_1) \lor \text{tomeet}(P^T_2) \\
\text{tomeet}(P^T_1 \land P^T_2) & = \text{tomeet}(P^T_1) \land \text{tomeet}(P^T_2) \\
\text{tomeet}(P^T_1 \cap P^T_2) & = \text{tomeet}(P^T_1) \cap \text{tomeet}(P^T_2)
\end{align*}
\]
We use case-based analysis to calculate the algorithmic rules for the initial evidence of consistent subtyping on gradual security types:

\[
\begin{align*}
g\left[\text{lift}P(G_1)(\overline{t_i}) <: \text{lift}P(G_2)(\overline{t_j})\right] & = \langle \iota_1, \iota_2 \rangle \\
g\left[G_1(\text{Bool}_{\iota_1}) \ll G_2(\text{Bool}_{\iota_2})\right] & = \langle \text{Bool}_{\iota_1}, \text{Bool}_{\iota_2} \rangle \\
g\left[\text{invert}(G_2)(\overline{U_{j1}}) <: \text{invert}(G_1)(\overline{U_{i1}})\right] & = \langle E'_{i1}, E'_{j1} \rangle \\
g\left[G_1(\overline{U_{i1}}) \ll G_2(\overline{U_{j1}})\right] & = \langle E_{i1}, E_{j1} \rangle \\
g\left[\text{lift}P(G_1)(\overline{t_{i1}}) <: \text{lift}P(G_2)(\overline{t_{j1}})\right] & = \langle \iota_{i1}, \iota_{j1} \rangle \\
g\left[\text{tomeet}(G_1)(\overline{U_i}) <: \text{tomeet}(G_2)(\overline{U_j})\right] & = \langle E_1, E_2 \rangle \\
g\left[\text{tomeet}(G_2)(\overline{U_j}) <: \text{tomeet}(G_1)(\overline{U_i})\right] & = \langle E'_1, E'_2 \rangle \\
g\left[G_1(\text{Ref}_{\iota_1} \overline{U_i}) <: G_2(\text{Ref}_{\iota_2} \overline{U_j})\right] & = \langle \text{Ref}_{\iota_1} \text{E}_1 \cap \text{E}'_1, \text{Ref}_{\iota_2} \text{E}_2 \cap \text{E}'_2 \rangle
\end{align*}
\]

where \( G_1 : \text{GLABEL}^n \rightarrow \text{GLABEL} \) and \( G_2 : \text{GLABEL}^m \rightarrow \text{GLABEL} \), and \( G_1(x_1, ..., x_n) = P^T_1(x_1, ..., x_n), G_2(x_1, ..., x_n) = P^T_2(x_1, ..., x_m) \).

\[
g^\triangledown(F(\overline{U_1}, ..., \overline{U_n})) = g\left[F(\overline{U_1}, ..., \overline{U_n}) <: F(\overline{U_1}, ..., \overline{U_n})\right]
\]

We calculate a recursive meet operator for gradual types:

\[
\text{Bool}_{\iota_1} \cap \text{Bool}_{\iota_2} = \text{Bool}_{\iota_1 \cap \iota_2}
\]

\[
(E_{11} \xrightarrow{\iota_1} E_{12}) \cap (E_{21} \xrightarrow{\iota_2} E_{22}) = (E_{11} \cap E_{21}) \xrightarrow{\iota_1 \cap \iota_2} (E_{12} \cap E_{22})
\]

\[
\text{Ref}_{\iota_1} \text{E}_1 \cap \text{Ref}_{\iota_2} \text{E}_2 = \text{Ref}_{\iota_1 \cap \iota_2} \text{E}_1 \cap \text{E}_2
\]

\[
U \cap U' \text{ undefined otherwise}
\]

We calculate a recursive definition for \( \triangle \ll \) by case analysis on the structure of the second argument,
\[ \Delta^{	riangleleft}(t_1, t_2, t_3) = \langle t'_1, t'_3 \rangle \]
\[ \Delta^\triangleleft(\text{Bool}_{i_1}, \text{Bool}_{i_2}, \text{Bool}_{i_3}) = \langle \text{Bool}_{t'_1}, \text{Bool}_{t'_3} \rangle \]

\[ \Delta^\triangleleft(E_{31}, E_{21}, E_{11}) = \langle E'_{31}, E'_{11} \rangle \]
\[ \Delta^\triangleleft(E_{12}, E_{22}, E_{32}) = \langle E'_{12}, E'_{32} \rangle \]
\[ \Delta^\triangleleft(t_1, t_2, t_3) = \langle t'_1, t'_3 \rangle \]
\[ \Delta^\triangleleft(t_{13}, t_{12}, t_{11}) = \langle t'_{13}, t'_{11} \rangle \]

\[ \Delta^\triangleleft(E_{31} \rightarrow_{t_1} E_{12}, E_{21} \rightarrow_{t_2} E_{32}, E_{31} \rightarrow_{t_3} E_{32}) \]
\[ = \langle E'_{12}, E'_{31}, E'_{13}, E'_{32} \rangle \]
\[ \Delta^\triangleleft(t_1, t_2, t_3) = \langle t'_1, t'_3 \rangle \]
\[ E'_3 = E_1 \sqcap E_2 \quad E'_3 = E_2 \sqcap E_3 \]
\[ \Delta^\triangleleft(\text{Ref}_{i_1}, E_1, \text{Ref}_{i_2}, E_2, \text{Ref}_{i_3}, E_3) = \langle \text{Ref}_{t'_1}, E'_1, \text{Ref}_{t'_3}, E'_3 \rangle \]

**Evidence inversion functions**

The evidence inversion functions are defined as follows

\[ ilbl(\langle \text{Bool}_{i_1}, \text{Bool}_{i_2} \rangle) = \langle t_1, t_2 \rangle \]
\[ ilbl(\langle \text{Unit}_{i_1}, \text{Unit}_{i_2} \rangle) = \langle t_1, t_2 \rangle \]
\[ ilbl(\langle \text{Ref}_{i_1}, U_{i_1}, \text{Ref}_{i_2}, U_{i_2} \rangle) = \langle t_1, t_2 \rangle \]
\[ ilbl(\langle E_{1} \rightarrow_{i_1} E_{2}, E'_{1} \rightarrow_{i'_1} E'_{2} \rangle) = \langle t_1, t'_1 \rangle \]

\[ iref(\langle \text{Ref}_{i_1}, E_{1}, \text{Ref}_{i_2}, E_{2} \rangle) = \langle E_{1}, E_{2} \rangle \]
\[ iref(\langle E_{1}, E_{2} \rangle) = \text{undefined otherwise} \]

\[ idom(\langle E_{1} \rightarrow_{i_1} E_{2}, E'_{1} \rightarrow_{i'_1} E'_{2} \rangle) = \langle E'_{1}, E_{1} \rangle \]
\[ idom(\langle E_{1}, E_{2} \rangle) = \text{undefined otherwise} \]

\[ icod(\langle E_{1} \rightarrow_{i_1} E_{2}, E'_{1} \rightarrow_{i'_1} E'_{2} \rangle) = \langle E_{2}, E'_{2} \rangle \]
\[ icod(\langle E_{1}, E_{2} \rangle) = \text{undefined otherwise} \]

**B.4.5 Proofs**

**Proposition 152** (\( \alpha_i \) is Sound). If \( \hat{\ell} \) is not empty, then \( \hat{\ell} \subseteq \gamma_i(\alpha_i(\hat{\ell})) \).
Proof. Suppose \( \hat{\ell} = \{ \ell_i \} \). By definition of \( \alpha_{\varepsilon_{\ell}} \), \( \alpha_{\varepsilon_{\ell}}(\{ \ell_i \}) = [\lambda \ell_i, \gamma_{\ell_i}] \). Therefore
\[
\gamma_i(\alpha_{\varepsilon_{\ell}}(\{ \ell_i \})) = \{ \ell \mid \ell \in \text{LABEL}, \lambda \ell_i \triangleleft \ell \triangleleft \gamma_{\ell_i} \}
\]
And it is easy to see that if \( \ell \in \{ \ell_i \} \), then \( \ell \in \gamma_i(\alpha_{\varepsilon_{\ell}}(\{ \ell_i \})) \), and therefore the result holds. \( \square \)

**Proposition 153** (\( \alpha_i \) is Optimal). If \( \hat{\ell} \) is not empty, and \( \hat{\ell} \subsetneq \gamma_i(i) \) then \( \alpha_i(\hat{\ell}) \subset i \).

Proof. By case analysis on the structure of \( i \). If \( i = [\ell_1, \ell_2] \), \( \gamma_{\varepsilon_i}(i) = \{ \ell \mid \ell \in \text{LABEL}, \ell_1 \triangleleft \ell \triangleleft \ell_2 \} \); \( \hat{\ell} \subset \{ \ell \mid \ell \in \text{LABEL}, \ell_1 \triangleleft \ell \triangleleft \ell_2 \} \); \( \hat{\ell} \neq \emptyset \) implies \( \alpha_{\varepsilon_i}(\hat{\ell}) = [\ell_3, \ell_4] \), where \( \ell_1 \triangleleft \ell_3 \) and \( \ell_4 \triangleleft \ell_2 \), therefore \( [\ell_3, \ell_4] \subset i \) (if \( \ell = \emptyset \), \( \alpha_{\varepsilon_i}(\hat{\ell}) \) is undefined). \( \square \)

**Proposition 154** (\( \alpha_E \) is Sound). If \( \text{valid}(\hat{S}) \) then \( \hat{S} \subseteq \gamma_E(\alpha_E(\hat{S})) \).

Proof. By well-founded induction on \( \hat{S} \). Similar to Prop 146. \( \square \)

**Proposition 155** (\( \alpha_E \) is Optimal). If \( \text{valid}(\hat{S}) \) and \( \hat{S} \subseteq \gamma_E(E) \) then \( \alpha_E(\hat{S}) \subseteq E \).

Proof. By induction on the structure of \( U \). Similar to Prop 147. \( \square \)

**Proposition 157**. \( \gamma_i(t_1 \cap t_2) = \gamma_i(t_1) \cap \gamma_i(t_2) \).

Proof. \[
\gamma_i(t_1 \cap t_2) = \gamma_i(\alpha_i(\gamma_i(t_1) \cap \gamma_i(t_2))) \subseteq \gamma_i(t_1) \cap \gamma_i(t_2) \quad \text{(soundness of} \ \alpha_i) \]
Let \( \ell \in \gamma_i(t_1) \cap \gamma_i(t_2) \). We now that \( \gamma_i(t_1 \cap t_2) \) is defined. Suppose \( t_1 = [\ell_1, \ell_2] \) and \( t_2 = [\ell_3, \ell_4] \). Therefore \( t_1 \cap t_2 = [\ell_1 \land \ell_3, \ell_2 \land \ell_4] \).

But \( \gamma_i(t_1) \cap \gamma_i(t_2) = \{ \ell \mid \ell \in \text{LABEL}, \ell_1 \triangleleft \ell \triangleleft \ell_2 \} \cap \{ \ell \mid \ell \in \text{LABEL}, \ell_3 \triangleleft \ell \triangleleft \ell_4 \} \).
Which is equivalent to \( \{ \ell \mid \ell \in \text{LABEL}, \ell_1 \triangleleft \ell \triangleleft \ell_2 \land \ell_3 \triangleleft \ell \triangleleft \ell_4 \} \), equivalent to \( \{ \ell \mid \ell \in \text{LABEL}, \ell_1 \land \ell_3 \triangleleft \ell \triangleleft \ell_2 \land \ell_4 \} \). Which is by definition \( \gamma_i([\ell_1 \land \ell_3, \ell_2 \land \ell_4]) \), and the result holds. \( \square \)

**Proposition 158**. \( \langle t_1, t_2 \rangle \circ^\triangleleft \langle t_2, t_3 \rangle = \Delta^\triangleleft(t_1, t_2 \cap t_2, t_3) \)

Proof. Follows directly from the definition of consistent transitivity and Prop 157. \( \square \)

**Proposition 159**. \( \gamma_E(E_1 \cap E_2) = \gamma_E(E_1) \cap \gamma_E(E_2) \).

Proof. By induction on evidence types \( \varepsilon_1 \) and \( \varepsilon_2 \) and Prop 157. \( \square \)

**Proposition 160**.
\[
\langle E_1, E_2 \rangle \circ^\triangleleft \langle E_2, E_3 \rangle = \Delta^\triangleleft(E_1, E_2 \cap E_2, E_3)
\]
where
\[
\Delta^\triangleleft(E_1, E_2, E_3) = \alpha_\varepsilon(\{ \langle S_1, S_3 \rangle \in \gamma_\varepsilon(E_1, E_3) \mid \exists S_2 \in \gamma_i(E_2). S_1 \triangleleft S_2 \triangleleft S_2 \triangleleft S_3 \})
\]

267
Proof. Follows directly from the definition of consistent transitivity and Prop 159.

Proposition 161. If $\varepsilon_S \vdash U_1 \lesssim U_2$ and $\varepsilon_I \vdash g_1 \lesssim g_2$ then $\varepsilon_S \varepsilon_I \vdash U_1 \varepsilon_I g_1 \lesssim U_2 \varepsilon_I g_2$

Proof. By induction on types $U_1$ and $U_2$, using the definition of $\lesssim$ and Proposition 28.

Proposition 162. $[\ell_1, \ell_2] \gamma [\ell_3, \ell_4] = [\ell_1 \gamma \ell_3, \ell_2 \gamma \ell_4]$

Proof. Follows directly by definition of $\gamma$ and $\gamma$.

Proposition 163.\[\langle t_1, t_2 \rangle \lesssim \langle t'_1, t'_2 \rangle = \langle t_1 \gamma t'_1, t_2 \gamma t'_2 \rangle\]

Proof. Follows directly from the definition of consistent join monotonicity and Prop 162.

Proposition 164.\[[\ell_1, \ell_2] \cap [\ell_3, \ell_4] = [\ell_1 \gamma \ell_3, \ell_2 \wedge \ell_4] \quad \text{if} \quad \ell_1 \gamma \ell_3 \equiv \ell_2 \wedge \ell_4\]

Proof. By definition of meet:
\[[\ell_1, \ell_2] \cap [\ell_3, \ell_4] = \alpha_6(\{l' \mid l' \in \gamma([\ell_1, \ell_2]) \cap \gamma([\ell_3, \ell_4])\})\]

But by definition of intersection on intervals, $\gamma([\ell_1, \ell_2]) \cap \gamma([\ell_3, \ell_4]) = \gamma([\ell_1 \gamma \ell_3, \ell_2 \wedge \ell_4])$ if $\ell_1 \gamma \ell_3 \equiv \ell_2 \wedge \ell_4$ (otherwise the intersection is empty), and the result follows by definition of $\alpha_6$.

Proposition 165.\[
\frac{\ell_1 \lesssim \ell_4 \quad \ell_3 \lesssim \ell_6 \quad \ell_1 \lesssim \ell_6}{\triangleleft([\ell_1, \ell_2], [\ell_3, \ell_4], [\ell_5, \ell_6]) = ([\ell_1, \ell_2 \wedge \ell_4 \wedge \ell_5], [\ell_1 \gamma \ell_3 \gamma \ell_5, \ell_6])}\]

Proof. By definition:
\[
\triangleleft([\ell_1, \ell_2], [\ell_3, \ell_4], [\ell_5, \ell_6]) = \alpha_6(\{\langle l'_{11}, l'_{3} \rangle \in \gamma_6([\ell_1, \ell_2], [\ell_5, \ell_6]) \mid \exists l'_2 \in \gamma_6([\ell_3, \ell_4], l'_{1} \lesssim l'_2 \lesssim l'_{3}\})\}
\]

It is easy to see that $\alpha_6(\{l'_{11}\}) = [\ell_1, \ell'_1_{12}]$, for some $\ell'_1_{12}$. We know that $\ell'_1_{12} \lesssim \ell_2$, $\ell'_1_{12} \lesssim \ell_4$ and $\ell'_1_{12} \lesssim \ell_6$, i.e. $\ell'_1_{12} \lesssim \ell_2 \wedge \ell_4 \wedge \ell_6$. But $\ell_2 \wedge \ell_4 \wedge \ell_6 \lesssim \ell_4 \lesssim \ell_6$ therefore
\[
\langle l_2 \wedge l_4 \wedge l_6, l_6 \rangle \in \{\langle l'_{11}, l'_{3} \rangle \in \gamma_6([\ell_1, \ell_2], [\ell_5, \ell_6]) \mid \exists l'_2 \in \gamma_6([\ell_3, \ell_4], l'_{1} \lesssim l'_2 \lesssim l'_{3}\})
\]

and by definition of $\alpha_6$, $\ell_2 \wedge l_4 \wedge \ell_6 \lesssim \ell'_1_{12}$, then $\alpha_6(\{l'_{11}\}) = [\ell_1, \ell_2 \wedge \ell_4 \wedge \ell_6]$. Similar argument is used to prove that $\alpha_6(\{l'_{3i}\}) = [\ell_1 \gamma \ell_3 \gamma \ell_5, \ell_6]$. Similar argument is used to prove that $\alpha_6(\{l'_{3i}\}) = [\ell_1 \gamma \ell_3 \gamma \ell_5, \ell_6]$.

Lemma 166. Let $l_1 \in \text{LABEL}$, then $(l_1 \wedge l_2) \gamma (l_3 \wedge l_4) \lesssim (l_1 \gamma l_3) \wedge (l_2 \gamma l_4)$.
Proposition 29. Suppose $\varepsilon_1 \vdash F_1(\overline{\mathfrak{g}}) \preceq F_2(\overline{\mathfrak{g}})$ and $\varepsilon_2 \vdash F_2(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})$.

If $\varepsilon_1 \circ \varepsilon_2$ is defined, then $\varepsilon_1 \circ \varepsilon_2 \vdash F_1(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})$.

Proof. Suppose $\varepsilon_1 = \langle t_{11}, t_{12} \rangle$ and $\varepsilon_2 = \langle t_{21}, t_{22} \rangle$. Then by definition of initial evidence:

$$\langle t_{11}, t_{12} \rangle = \langle [\ell_1, \ell_2, [\ell_3, \ell_4]] \supseteq g[F_1(\overline{\mathfrak{g}}) \preceq F_2(\overline{\mathfrak{g}})] \rangle = \langle t'_{11}, t'_{12} \rangle$$

and

$$\langle t_{21}, t_{22} \rangle = \langle [\ell_5, \ell_6, [\ell_7, \ell_8]] \supseteq g[F_2(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})] \rangle = \langle t'_{21}, t'_{22} \rangle$$

Suppose that $g[F_1(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})] = \langle t'_1, t'_3 \rangle$. We have to prove that $\langle t_{11}, t_{12} \rangle \circ \varepsilon_2 \langle t_{21}, t_{22} \rangle \subseteq \langle t'_1, t'_3 \rangle$.

If $\text{bounds}(F_1(\overline{\mathfrak{g}})) = [\ell'_1, \ell'_2]$, $\text{bounds}(F_2(\overline{\mathfrak{g}})) = [\ell'_3, \ell'_4]$, and $\text{bounds}(F_3(\overline{\mathfrak{g}})) = [\ell'_5, \ell'_6]$ we know that $g[F_1(\overline{\mathfrak{g}}) \preceq F_2(\overline{\mathfrak{g}})] = \langle [\ell'_1, \ell'_2 \sqcap \ell'_4], [\ell'_1 \sqcap \ell'_3, \ell'_4] \rangle$. Therefore $\ell'_1 \preceq \ell_1$, $\ell'_2 \preceq \ell_2$, $\ell'_3 \preceq \ell'_4$. Using the same argument, $g[F_2(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})] = \langle [\ell'_3, \ell'_4 \sqcap \ell'_6], [\ell'_3 \sqcap \ell'_5, \ell'_6] \rangle$. Therefore $\ell'_3 \preceq \ell_5$, $\ell'_6 \preceq \ell'_4$, $\ell'_3 \sqcap \ell'_5 \preceq \ell_7$ and $\ell'_8 \preceq \ell'_6$.

But $g[F_1(\overline{\mathfrak{g}}) \preceq F_3(\overline{\mathfrak{g}})] = \langle [\ell'_1, \ell'_2 \sqcap \ell'_6], [\ell'_1 \sqcap \ell'_5, \ell'_6] \rangle$ and

$$\langle t_{11}, t_{12} \rangle \circ \varepsilon_2 \langle t_{21}, t_{22} \rangle = \bigtriangleup^\varepsilon(\langle t_{11}, t_{12} \cap t_{21}, t_{22} \rangle) =$$

$$\bigtriangleup^\varepsilon([\ell_1, \ell_2], [\ell_3 \sqcap \ell_5, \ell_4 \sqcap \ell_6], [\ell_7, \ell_8])$$

$$= \langle [\ell_1, \ell_2 \sqcap \ell_4 \sqcap \ell_6 \sqcap \ell_8], [\ell_1 \sqcap \ell_3 \sqcap \ell_5 \sqcap \ell_7 \sqcap \ell_8] \rangle$$

we need to prove that

$$\langle [\ell_1, \ell_2 \sqcap \ell_4 \sqcap \ell_6 \sqcap \ell_8], [\ell_1 \sqcap \ell_3 \sqcap \ell_5 \sqcap \ell_7 \sqcap \ell_8] \rangle \sqsubseteq$$

$$\langle [\ell'_1, \ell'_2 \sqcap \ell'_6], [\ell'_1 \sqcap \ell'_5, \ell'_6] \rangle$$

But we know that $\ell'_1 \preceq \ell_1$. Also that $\ell'_2 \preceq \ell'_3 \sqcap \ell'_4$ and therefore $\ell_2 \preceq \ell'_2$. The same for $\ell'_6 \preceq \ell'_6$ and therefore $\ell_2 \sqcap \ell_4 \sqcap \ell_6 \sqcap \ell_8 \preceq \ell'_2 \sqcap \ell'_6$, i.e. $[\ell_1, \ell_2 \sqcap \ell_4 \sqcap \ell_6 \sqcap \ell_8] \sqsubseteq [\ell'_1, \ell'_2 \sqcap \ell'_6]$. The argument is applied for the second components and the result holds.
**Proposition 28.** Suppose \( \varepsilon_1 \vdash F_{11}(\overline{g_1}) \trianglelefteq F_{12}(\overline{g_1}) \) and \( \varepsilon_2 \vdash F_{21}(\overline{g_1}) \trianglelefteq F_{22}(\overline{g_1}) \). Then \( \varepsilon_1 \equiv \varepsilon_2 \vdash F_{11}(\overline{g_1}) \vee F_{21}(\overline{g_1}) \trianglelefteq F_{12}(\overline{g_1}) \vee F_{22}(\overline{g_1}) \)

**Proof.** By definition of initial evidence noticing that \( \varepsilon_1 \equiv \varepsilon_2 \) can be more precise than the initial evidence of judgment

Suppose \( \varepsilon_1 = \langle [\ell_1, \ell_2], [\ell_3, \ell_4] \rangle \), and \( \varepsilon_2 = \langle [\ell_5, \ell_6], [\ell_7, \ell_8] \rangle \), then \( \varepsilon_1 \equiv \varepsilon_2 = \langle [\ell_1 \vee \ell_5, \ell_2 \vee \ell_6], [\ell_3 \vee \ell_6, \ell_4 \vee \ell_8] \rangle \).

If \( \text{bounds}(F_{11}(\overline{g_1})) = [\ell'_{111}, \ell'_{112}], \text{bounds}(F_{12}(\overline{g_1})) = [\ell'_{121}, \ell'_{122}], \text{bounds}(F_{21}(\overline{g_1})) = [\ell'_{211}, \ell'_{212}] \) and \( \text{bounds}(F_{22}(\overline{g_1})) = [\ell'_{221}, \ell'_{222}] \).

We know that \( \mathcal{J}[F_{11}(\overline{g_1}) \trianglelefteq F_{12}(\overline{g_1})] = \langle [\ell'_{111}, \ell'_{112} \wedge \ell'_{122}], [\ell'_{111} \vee \ell'_{121}, \ell'_{122}] \rangle \). Therefore \( \ell'_{111} \triangleleft \ell_1, \ell_2 \triangleleft \ell'_{112} \wedge \ell'_{122}, \ell'_{111} \gamma \ell_{121} \triangleleft \ell_3 \) and \( \ell_4 \triangleleft \ell'_{122} \). Using the same argument, \( \mathcal{J}[F_{21}(\overline{g_1}) \trianglelefteq F_{22}(\overline{g_1})] = \langle [\ell'_{211}, \ell'_{212} \wedge \ell'_{222}], [\ell'_{211} \gamma \ell'_{221}, \ell'_{222}] \rangle \). Therefore \( \ell'_{211} \triangleleft \ell_5, \ell_6 \triangleleft \ell'_{212} \wedge \ell'_{222}, \ell'_{211} \gamma \ell'_{221} \triangleleft \ell_7 \) and \( \ell_8 \triangleleft \ell'_{222} \).

But the \( \mathcal{J}[F'_1(\overline{g_1}) \trianglelefteq F'_2(\overline{g_1})] = \langle [\ell'_1, \ell'_2 \wedge \ell'_4], [\ell'_1 \vee \ell'_3, \ell'_4] \rangle \) where

\[ \text{bounds}(F'_1(\overline{g_1})) = \text{bounds}(F_{11}(\overline{g_1})) \vee \text{bounds}(F_{21}(\overline{g_1})) = [\ell'_{111}, \ell'_{112}] \vee [\ell'_{211}, \ell'_{212}] = [\ell'_{111} \vee \ell'_{211}, \ell'_{112} \vee \ell'_{212}] \]

and

\[ \text{bounds}(F'_2(\overline{g_1})) = \text{bounds}(F_{12}(\overline{g_1})) \vee \text{bounds}(F_{22}(\overline{g_1})) = [\ell'_{121}, \ell'_{122}] \vee [\ell'_{221}, \ell'_{222}] = [\ell'_{121} \vee \ell'_{221}, \ell'_{122} \vee \ell'_{222}] \]

We need to prove that \( [\ell_1 \vee \ell_5, \ell_2 \vee \ell_6] \trianglerighteq [\ell'_{111} \vee \ell'_{211}, \ell'_{112} \vee \ell'_{212}] \), i.e. \( \ell'_{111} \vee \ell'_{211} \trianglerighteq \ell_1 \vee \ell_5 \) and \( \ell_2 \vee \ell_6 \trianglerighteq \ell'_{112} \vee \ell'_{212} \). But \( \ell'_{111} \trianglerighteq \ell_1 \) and \( \ell'_{211} \trianglerighteq \ell_5 \), therefore \( \ell'_{111} \vee \ell'_{211} \trianglerighteq \ell_1 \vee \ell_5 \).

Similarly, as \( \ell_2 \trianglerighteq \ell'_{112} \wedge \ell'_{212} \) and \( \ell_6 \trianglerighteq \ell'_{212} \wedge \ell'_{222} \), then \( \ell_2 \vee \ell_6 \trianglerighteq \ell'_{112} \vee \ell'_{212} \). Therefore \( [\ell_1 \vee \ell_5, \ell_2 \vee \ell_6] \trianglerighteq [\ell'_{111} \wedge \ell'_{211}, \ell'_{112} \wedge \ell'_{212}] \).

Using analogous argument, we also know that \( [\ell_3 \vee \ell_6, \ell_4 \vee \ell_8] \trianglerighteq [\ell'_{121} \vee \ell'_{221}, \ell'_{122} \vee \ell'_{222}] \). Therefore \( \varepsilon_1 \equiv \varepsilon_2 \trianglerighteq \mathcal{J}[F'_1(\overline{g_1}) \trianglelefteq F'_2(\overline{g_1})] \), and the result holds.

\[ \square \]

**Lemma 167.** Let \( S_1, S_2 \in \text{TYPE} \). Then

1. If \( (S_1 \triangleright S_2) \) is defined then \( S_1 \trianglerighteq (S_1 \triangleright S_2) \).

2. If \( (S_1 \wedge S_2) \) is defined then \( (S_1 \wedge S_2) \trianglerighteq S_1 \).

**Proof.** We start by proving (1) assuming that \( (S_1 \vee S_2) \) is defined. We proceed by case analysis on \( S_1 \).

270
Case (⌈Bool⌉). If \( S_1 = \text{Bool}_{t_1} \) then as \( (S_1 \triangledown S_2) \) is defined then \( S_2 \) must have the form \( \text{Bool}_{(t_1 \gamma t_2)} \) for some \( t_2 \). Therefore \( (S_1 \triangledown S_2) = \text{Bool}_{(t_1 \gamma t_2)} \). But by definition of \( \preceq \), \( t_1 \preceq (t_1 \gamma t_2) \) and therefore we use \( (<:\text{Bool}) \) to conclude that \( \text{Bool}_{t_1} <: \text{Bool}_{(t_1 \gamma t_2)} \), i.e. \( S_1 <: (S_1 \triangledown S_2) \).

Case \( (S \to_{t_{\ell}} S) \). If \( S_1 = S_{11} \to_{t_1} S_{12} \) then as \( (S_1 \triangledown S_2) \) is defined then \( S_2 \) must have the form \( S_{21} \to_{t_2} S_{22} \) for some \( S_{21}, S_{22} \) and \( t_2 \).

We also know that \( (S_1 \triangledown S_2) = (S_{11} \triangledown S_{21}) \to_{(t_1 \gamma t_2)} (S_{12} \triangledown S_{22}) \). By definition of \( \preceq \), \( t_1 \preceq (t_1 \gamma t_2) \).

Also, as \( (S_1 \triangledown S_2) \) is defined then \( (S_{11} \triangledown S_{21}) \) is defined. Using the induction hypothesis of (2) on \( S_{11} \), \( (S_{11} \triangledown S_{21}) <: S_{11} \). Also, using the induction hypothesis of (1) on \( S_{12} \) we also know that \( S_{12} <: (S_{12} \triangledown S_{22}) \). Then by \( (<:\text{---}) \) we can conclude that \( S_{11} \to_{t_1} S_{12} <: (S_{11} \triangledown S_{21}) \to_{(t_1 \gamma t_2)} (S_{12} \triangledown S_{22}) \), i.e. \( S_1 <: (S_1 \triangledown S_2) \).

The proof of (2) is similar to (1) but using the argument that \( (t_1 \wedge t_2) \preceq t_1 \).

**Lemma 168.** Let \( S \in \text{TYPE} \) and \( \ell \in \text{LABEL} \). Then \( S <: S \cdot \ell \).

**Proof.** Straightforward case analysis on type \( S \) using the fact that \( \ell \preceq (\ell' \gamma \ell) \) for any \( \ell' \).

**Lemma 169.** Let \( S_1, S_2 \in \text{TYPE} \) such that \( S_1 <: S_2 \), and let \( \ell_1, \ell_2 \in \text{LABEL} \) such that \( \ell_1 \preceq \ell_2 \). Then \( S_1 \cdot \ell_1 <: S_2 \cdot \ell_2 \).

**Proof.** Straightforward case analysis on type \( S \) using the definition of label stamping on types.

### B.5 GSL\( _{\text{Ref}}^{\varepsilon} \): Dynamic properties

Notice that for convenience, the proofs and properties are defined over intrinsic terms \[^{[44]}\] instead of terms of the internal language. They are actually the same as terms of the internal language, but keeping all static annotations explicitly. First we introduce the static semantics of intrinsic terms in Sec. [B.5.1] Their dynamic semantics in Sec. [B.5.2] The relation between intrinsic and evidence-augmented terms in Sec. [B.5.3] Then the proof of type safety is presented Sec. [B.5.4] the proof of dynamic gradual guarantee for GSL\( _{\text{Ref}}^{\varepsilon} \) without the specific check in rule \( (\varepsilon 7) \) in § [B.5.5] and the proof of noninterference in Sec. [B.5.6]

#### B.5.1 Intrinsic Terms: Static Semantics

Following Garcia et al. \[^{[44]}\], we develop *intrinsically typed* terms \[^{[25]}\]: a term notation for gradual type derivations. These terms serve as our internal language for dynamic semantics: they play the same role that cast calculi play in typical presentations of gradual typing \[^{[109]}\]. Intrinsically-typed terms \( t^U \) comprise a family \( \text{TERM}_U \) of type-indexed sets, such that ill-typed terms do not exist. They are built up from disjoint families \( x^U \in \text{VAR}_U \) and \( o^U \in \text{LOC}_U \) of intrinsically typed variables and locations respectively. Unless required, we omit the type exponent on intrinsic terms, writing \( \bar{t} \in \text{TERM}_U \).
\[ \epsilon \in \text{Evidence}, \ et \in \text{EvTerm}, \ ev \in \text{EvValue}, \ v \in \text{Value}, \ u \in \text{SimpleValue}, g \in \text{EvTerm}, \ f \in \text{TmFrame} \]

\[
\begin{align*}
    u & ::= x^U | b_g | (\lambda^g.x^U.t)_g | o_g^U | \text{unit}_g \\
    v & ::= u \mid \epsilon u :: U \\
    f & ::= h[\epsilon] \\
    \mu & ::= \bullet \mid \mu, o^U \mapsto v \\
    p & ::= x^U \mid o^U \\
    q & ::= p \mid \epsilon p :: U \\
    h & ::= \Box \oplus^g et \mid ev \oplus^g \Box \mid U^\mu \Box \mid ev \oplus^g U^\mu \Box \mid U :: U \mid if^\mu \Box \text{then } et \text{ else } et \\
\end{align*}
\]

Figure B.11: GSL Ref: Syntax of the Intrinsic Term Language

To each typing rule corresponds an intrinsic term formation rule that captures all the information needed to ensure that an intrinsic term is isomorphic to a typing derivation. Because intrinsic variables and locations reflect their typings, intrinsic terms do not need explicit type environments \( \Gamma \) or store environments \( \Sigma \); however, the typing judgment depends on a security effect \( g_c \), which intrinsic terms must account for.

Additionally, because intrinsic terms represent typing derivations of programs as they reduce, they must account for the possibility that runtime values have more precise types than those used in the original typing derivation. For instance, the term in function position of an application can be a subtype of the function type used to type-check the program originally. The formation rule of the application intrinsic term must permit this extra subtyping leeway, justified by evidence. The same holds for the security information. Therefore, an intrinsic term has the general form \( \phi \triangleright t \), where the context information \( \phi \triangleq (\epsilon.g_c, g_c) \) contains the static program counter label \( g_c \) used to type-check the source term, as well as the runtime program counter label \( g_c \), along with the evidence \( \epsilon \vdash g_c \sim g_c \)\(^1\). For simplicity we define accessors \( \phi.\epsilon.g_c \triangleq g_c \), \( \phi.g_c \triangleq g_c \), and \( \phi.\epsilon \triangleq \epsilon \).

Figure B.11 presents the syntax of intrinsic terms. Fig. B.12 presents the intrinsic terms formation rules for GSLRef. In rule (Iprot), labels \( g \) and \( g' \) represent the static and dynamic information of the label used to increase the program counter label in the subterm, respectively.

Evidence \( \epsilon_1 \) justifies that the type of the subterm is a consistent subtype of \( U \), the static type of the subterm. \( \phi' \) represents the context information associated to the subterm \( \hat{t} \): \( \phi'.g_c \) (resp. \( \phi'.g_c \)) is the program counter label used to typecheck (resp. evaluate) \( \hat{t} \).

In the intrinsic term formation rule for applications (Iapp), \( U_1 \) is the runtime type of the function term. We annotate the initial static type information with @. The evidence \( \epsilon_2 \) for the label ordering premise is also annotated, since it is needed to reconstruct the derivation. The intrinsic term of a conditional, described in Rule (Iif)\(^2\), carries the static information of

---

\(^1\)We use color to make distinctions when is needed: green is for effects and static information; orange is for the runtime information of the security effect.

\(^2\)Evidence inversion functions \( (\text{idom}, \text{idc}, \text{idref}, \text{idbl} \text{ and } \text{idlat}) \) manifest the evidence for the inversion principles on consistent subtyping judgments; e.g. starting from the evidence that \( U_1 \lesssim U_2 \), \( \text{ilbl} \) produces the evidence of the judgment \( \text{label}(U_1) \lesssim \text{label}(U_2) \).
Figure B.12: GSL\textsubscript{Ref}: Gradual Intrinsic Terms
the label of the conditional term \( g \). The context information \( \phi' \) used for both branches is obtained by joining the term context \( \phi \) point-wise with the evidence and labels associated with the consistent subtyping judgment of the conditional. Evidences \( \varepsilon_2 \) and \( \varepsilon_3 \) justify that the type of each branch is a consistent subtype of the join of both types. Finally, rule (Iassign) is built similarly to the application rule (Iapp).

**B.5.2 Intrinsic Terms: Dynamic Semantics**

Next we present the full definition of the intrinsic reduction rules in Figure B.13 and the full definition of notions of intrinsic reduction in Figures B.14 and B.15.

Because the security context information of a term is maintained at each step, we also adopt the lightweight notation \( \tilde{t}_1 | \mu_1 \xrightarrow{\phi} \tilde{t}_2 | \mu_2 \), to denote the reduction of the intrinsic term \( \phi \downarrow \tilde{t}_1 \in \text{TERM}_U \) in store \( \mu_1 \) to the intrinsic term \( \phi \downarrow \tilde{t}_2 \in \text{TERM}_U \) in store \( \mu_2 \). We note \( C[U] \) the combination of a term \( \tilde{t} \in \text{TERM}_U \) (without context) and a store \( \mu \). Function applications reduce to to an error if consistent transitivity fails to justify \( U_2 <: U_1 \). Conditionals similarly reduce to a new \( \text{prot} \) term, which is constructed using the static and dynamic information of the conditional term. Assignments may reduce to an ascribed unit value. Similarly to references, the stored value is ascribed the statically determined type \( U \). Therefore consistent transitivity may fail to justify that the actual type of the stored value is a subtype of \( U \). As the value is stamped with actual labels, the term may also reduce to an error if consistent transitivity cannot support the judgment \( \phi \downarrow \tilde{t} \ll U \).
Notions of Reduction

\[ \phi \rightarrow : C[U] \times (C[U] \cup \{\text{error}\}) \]

\[ \epsilon_1 (b_1)_{g_1} \oplus^g \epsilon_2 (b_2)_{g_2} \mid \mu \xrightarrow{\phi} (\epsilon_1 \overline{\gamma} \epsilon_2) (b_1 [\oplus] b_2)_{(g_1 \overline{\gamma} g_2)} : \text{Bool}_g \mid \mu \]

\[ \text{pro}^{g, U}_{\epsilon_2 \rightarrow g} \phi' (\epsilon_1 u) \mid \mu \xrightarrow{\phi} (\epsilon_1 \overline{\gamma} \epsilon_2) (u \overline{\gamma} g') :: U \overline{\gamma} g \mid \mu \]

\[ \epsilon_1 (\lambda^g x U_{11}. \iota_J^*)_{g_2} @_{\epsilon_1} U_{12}^U \epsilon_2 u \mid \mu \xrightarrow{\phi} \left\{ \begin{array}{l} \text{pro}^{g_1, U_2}_{\epsilon_2 \rightarrow g_2} \phi' (\text{icod}(\epsilon_1) ([(\epsilon u :: U_{11}) / x U_{11}] t^*)) \mid \mu \\
\text{error} \quad \text{if } \epsilon \text{ or } \epsilon' \text{ are not defined}
\end{array} \right. \]

where \[ \epsilon = \epsilon_2 \circ^{<} \text{idom}(\epsilon_1) \], \[ \epsilon' = (\phi, \epsilon \overline{\gamma} \iota \text{bl}(\epsilon_1)) \circ^g \epsilon_3 \circ ^{<} \text{ilat}(\epsilon_1) \]
and \[ \phi' = (\epsilon', \phi, g_c \overline{\gamma} g_2, g_2') \]

if \[ \epsilon \text{true}_{g_1} \] then \[ \epsilon_2 U_{12} :: \epsilon_3 U_{13} \mid \mu \xrightarrow{\phi} \text{pro}^{g, U}_{\epsilon_2 \rightarrow g_1} \phi' (\epsilon_2 U_{12}) \mid \mu \]
where \[ \phi' = (\phi, \epsilon \overline{\gamma} \iota \text{bl}(\epsilon_1), \phi, g_c \overline{\gamma} g_1, \phi, g_c \overline{\gamma} g) \] and \[ U = (U_2 \overline{\gamma} U_3) \]

if \[ \epsilon \text{false}_{g_1} \] then \[ \epsilon_2 U_{12} :: \epsilon_3 U_{13} \mid \mu \xrightarrow{\phi} \text{pro}^{g, U}_{\epsilon_2 \rightarrow g_1} \phi' (\epsilon_3 U_{13}) \mid \mu \]
where \[ \phi' = (\phi, \epsilon \overline{\gamma} \iota \text{bl}(\epsilon_1), \phi, g_c \overline{\gamma} g_1, \phi, g_c \overline{\gamma} g) \] and \[ U = (U_2 \overline{\gamma} U_3) \]

\[ \text{ref}_{U} \epsilon u \mid \mu \xrightarrow{\phi} \left\{ \begin{array}{l} o^U \mid \mu [o^U \mapsto \epsilon'(u \overline{\gamma} \phi, g_c) :: U] \quad \text{where } o^U \not\in \text{dom}(\mu) \\
\text{error} \quad \text{if } (\phi, \epsilon \circ ^{<} \epsilon_\ell) \text{ is not defined}
\end{array} \right. \]

where \[ \epsilon' = \epsilon \overline{\gamma} (\phi, \epsilon \circ ^{<} \epsilon_\ell) \]

\[ !\text{Ref}_g U \epsilon o^U' \mid \mu \xrightarrow{\phi} \text{pro}^{g, U}_{\epsilon \rightarrow g} \phi' (\text{iref}(\epsilon) v) \]
where \[ \mu(o^U) = v \] and \[ \phi' = (\phi, \epsilon \overline{\gamma} \iota \text{bl}(\epsilon), \phi, g_c \overline{\gamma} g', \phi, g_c \overline{\gamma} g) \]

\[ \epsilon_1 o^U := \epsilon_3 \epsilon_2 u \mid \mu \xrightarrow{\phi} \left\{ \begin{array}{l} \text{unit} \mid \mu [o^U \mapsto \epsilon'(u \overline{\gamma} (\phi, g_c \overline{\gamma} g)) :: U] \\
\text{error} \quad \text{if } \epsilon' \text{ is not defined, or}
\end{array} \right. \]

where \[ \epsilon' = (\epsilon_2 \circ^{<} \text{iref}(\epsilon_1)) \overline{\gamma} ((\phi, \epsilon \overline{\gamma} \iota \text{bl}(\epsilon_1)) \circ^{<} \epsilon_3 \circ ^{<} \text{ilbl}(\text{iref}(\epsilon_1)) \]
and \[ \mu(o^U) = \epsilon u' :: U \]

Figure B.14: GSLRef: Intrinsic Notions of Reduction part 1

B.5.3 Relating Intrinsic and Evidence-augmented Terms

In this section we present the translation rules from GSLRef terms to intrinsic terms in Figures B.16 and B.17. Also this section presents the erasure function in in Figure B.18—highlighting the syntactics differences between terms in gray—along properties that relates evidence-augmented terms and intrinsic terms.

In particular we identify four important properties. First, that given a source language the erasure of the translation to intrinsic term is equal to the translation of the source term to an evidence-augmented term:

Proposition 170. If \[ \Gamma; \Sigma; g \vdash t \rightsquigarrow \tilde{t} : U \] and \[ \Gamma; \Sigma; g \vdash t' \rightsquigarrow U \], then \[ |\tilde{t}| = t' \].
Proof. By induction on the type derivation of $t$. □

Second, given a reducible intrinsic term $\bar{t}$, if it reduces to an error, then it erasure also reduces to an error; or, if reduces to an intrinsic term $\bar{t}'$, then the erasure of $\bar{t}'$ also reduces to the erasure of $\bar{t}'$:

**Proposition 171.** Consider $\phi = \varepsilon g_c, \phi \triangleright \bar{t} \in \text{TER}_U$, and $\phi; \Sigma; \varepsilon g_c \vdash t : U$, such that $\Sigma \models \mu_2$. Then if $\bar{t} = t$ and $\mu_1 = \mu'_1$ then either

- $\bar{t} | \mu_1 \xrightarrow{\phi} \bar{t}' | \mu_2 \Rightarrow |\bar{t}| |\mu_2| |\varepsilon g_c| |\bar{t}'| |\mu'_2|$, or

- $\bar{t} | \mu_1 \xrightarrow{\phi} \text{error} \Rightarrow |\bar{t}| |\mu_2|\text{error}$

**Proof.** By induction on the type derivation of $\bar{t}$.

*Case (I::)*. Then $\bar{t} = \varepsilon_1 \bar{t}' :: U$ and by $(E::)$, $t = \varepsilon_1 t'$ for some $t'$ such that $\bar{t}' = t'$. Suppose that $\varepsilon_1 \vdash U' \subseteq U$. By inspection on the type derivations, $\phi \triangleright \bar{t}' \in \text{TER}_U$, and $\phi; \Sigma; \varepsilon g_c \vdash t' : U'$.

Let us suppose that $\bar{t}' | \mu_1 \xrightarrow{\phi} \bar{t}'' | \mu_2$, then by induction hypothesis $t' | \mu_2 \xrightarrow{\varepsilon g_c} t'' | \mu'_2$ and $\bar{t}'' = t''$ and $\mu'_1 = \mu_2$. Then $\varepsilon_1 \bar{t}'' :: U | \mu_1 \xrightarrow{\phi} \varepsilon_1 \bar{t}'' :: U | \mu'_1$ and $\varepsilon_1 t' | \mu_2 \xrightarrow{\varepsilon g_c} \varepsilon_1 t'' | \mu'_2$. But as $\mu'_1 = \mu'_2$, and by $(E::)$ $\varepsilon_1 \bar{t}'' :: U = \varepsilon_1 t''$, the result holds.

Let us suppose now that $\bar{t}' = \varepsilon_2 u :: U'$. Then as $\bar{t}' = t'$, $t' = \varepsilon_2 u'$, for some $u'$ such that $u = u'$. If $\varepsilon_2 \circ \varepsilon_1$ is not defined the result holds immediately. Suppose $\varepsilon_2 \circ \varepsilon_1 = \varepsilon'$, then $\varepsilon_1 (\varepsilon_2 u :: U') :: U | \mu_1 \xrightarrow{\phi} \varepsilon' u :: U | \mu_1$ and $\varepsilon_1 (\varepsilon_2 u') | \mu_2 \xrightarrow{\varepsilon g_c} \varepsilon' u' | \mu_2$. But as $\mu_1 = \mu_2$, and by $(E::)$ $\varepsilon' u :: U = \varepsilon' u'$, the result holds.

If $\bar{t}' = u$, then as $\bar{t}' = t'$, $t' = \varepsilon_2 u'$, for some $u'$ such that $u = u'$, and the result holds immediately.

The other cases proceed analogous. □

Fourth, if an intrinsic term type checks, then its erasure also type checks to the same type.

**Proposition 172.** Consider $\phi \triangleright \bar{t} \in \text{TER}_U$ then, for $\Gamma \models \bar{t}$ and $\Sigma \models \bar{t}$, $\Gamma; \Sigma; |\phi| \vdash |\bar{t}| : U$. 

276
Lemma 174. Consider \( \Gamma; \Sigma; g_c \vdash t \rightsquigarrow \bar{t} : U \)

\[
\begin{align*}
(Tx) & \quad \Gamma(x) = U \\
(Tu) & \quad \Gamma; \Sigma; g_c \vdash \text{unit}_g \rightsquigarrow \text{unit}_g : \text{Unit}_g \\
(T\lambda) & \quad \Gamma; \Sigma; g' \vdash t \rightsquigarrow \bar{t} : U_2 \\
(T\oplus) & \quad \Gamma; \Sigma; g_c \vdash t_1 \oplus t_2 \rightsquigarrow \bar{t}_1 \oplus \bar{t}_2 : \text{Bool}_{g_1} \oplus \text{Bool}_{g_2}
\end{align*}
\]

Proof. By induction on the type derivation of \( \bar{t} \).

Finally, if an evidence-augmented term type checks, then there must exists some intrinsic term that have the same type and that it erasure is the original evidence-augmented term.

Proposition 173. Consider \( \Gamma; \Sigma; \varepsilon g_c \vdash t : U \). Then \( \exists \bar{t}, \exists \phi \) such that \( |\bar{t}| = t \) and \( |\phi| = \varepsilon g_c \) and \( \phi \triangleright \bar{t} \in \text{TERM}_U \)

Proof. By induction on the type derivation of \( t \).

Case \( (\varepsilon t') \). Then \( t = \varepsilon t' \), for some \( \varepsilon', t' \). But we know that \( \Gamma; \Sigma; \varepsilon g_c \vdash \varepsilon t' : U \) and suppose \( \varepsilon' \vdash U' \subseteq U \) and \( \varepsilon \vdash g_c \subseteq g_c' \). Then by choosing \( \phi = (\varepsilon, g_c) g_c' \) and induction hypothesis on \( t' \), \( \exists \bar{t} \) such that \( \phi \triangleright \bar{t} \in \text{TERM}_{U'} \).

The other cases proceed analogous.

Lemma 174. Consider \( \phi \triangleright \bar{t}_1 \in \text{TERM}_U \). If \( \bar{t}_1 \sqsubseteq \bar{t}_2 \) then \( |\bar{t}_1| \sqsubseteq |\bar{t}_2| \).

Figure B.16: GSL_{Ref}: translation to GSL_{Ref} intrinsic terms part 1
Figure B.17: GSL_\text{Ref}^\downarrow: translation to GSL_\text{Ref} intrinsic terms part 2

Figure B.18: GSL_\text{Ref}^\downarrow: Equivalence between intrinsic terms and evidence-augmented terms
Proof. By induction on the type derivation of $\bar{t}_1$ and the definition of $\|$. \qed

**Lemma 175.** Consider $\phi \triangleright \bar{t}_1 \in \text{TERM}_U$. If $|\bar{t}_1| \subseteq t_2$, then $\exists \bar{t}_2$, such that $\bar{t}_1 \subseteq \bar{t}_2$ and that $|\bar{t}_2| = t_2$.

**Proof.** By induction on $\bar{t}_1$ and the definition of $\|$.

**Case (1::).** Then $\bar{t}_1 = \varepsilon \bar{t}_1' :: U$, and $|\bar{t}_1| = \varepsilon |\bar{t}_1'|$. By definition of $\subseteq$, $t_2$ has the form $\varepsilon_2 \bar{t}_2'$, where $\varepsilon_2 \subseteq \varepsilon$ and $|\bar{t}_1'| \subseteq |\bar{t}_2'|$. By induction hypothesis, $\exists \bar{t}_2'$ such that $\bar{t}_1' \subseteq \bar{t}_2'$ and that $|\bar{t}_2'| = t_2'$. By definition of evidence, we can build the term $\varepsilon_2 \bar{t}_2' :: ?$, but we know that $\varepsilon_1 \bar{t}_1' :: U \sqsubseteq \varepsilon_2 \bar{t}_2' :: ?$ and that $|\varepsilon_2 \bar{t}_2' :: ?| = \varepsilon_2 |\bar{t}_2'| = \varepsilon_2 t_2$ and the result holds.

The other cases proceed analogous. \qed

### B.5.4 Type Safety

In this section we present the proof of type safety for GSLRef.

We define what it means for a store to be well typed with respect to a term. Informally, all free locations of a term and of the contents of the store must be defined in the domain of that store. Also, the store must preserve types between intrinsic locations and underlying values.

**Definition 103** ($\mu$ is well typed). A store $\mu$ is said to be well typed with respect to an intrinsic term $t^U$, written $t^U \vdash \mu$, if

1. $\text{freeLocs}(t^U) \subseteq \text{dom}(\mu)$, and
2. $\forall v \in \text{cod}(\mu), v \vdash \mu$ and
3. $\forall o^U \in \text{dom}(\mu), \forall \phi$, then $\phi \triangleright \mu(o^U) \in \text{TERM}_U$.

**Lemma 176.** Suppose $\phi \triangleright t^U \in \text{TERM}_U$, then $\forall g'_r, \forall \varepsilon'_r$, such that $g'_r \ll \phi.g_c$ and $\varepsilon'_r \vdash g'_r \ll \phi.g_c$, $\phi' = (\varepsilon'_r, g'_r, \phi.g_c)$ then $\phi' \triangleright t^U \in \text{TERM}_U$.

**Proof.** By induction on the derivation of $\phi \triangleright t^U \in \text{TERM}_U$. Noticing that no typing derivation depends on $\varepsilon'_r, g'_r$, save for the judgement $\varepsilon'_r \vdash g'_r \ll g_c$ which is premise of this lemma. \qed

**Lemma 177.** Suppose $\phi \triangleright v \in \text{TERM}_U$, then $\forall \phi'$, then $\phi' \triangleright v \in \text{TERM}_U$.

**Proof.** By induction on the derivation of $\phi' \triangleright v$ observing that for values, there is no premise that depends on the security effect. \qed

**Lemma 178** (Canonical forms). Consider a value $v \in \text{TERM}_U$. Then either $v = u$, or $v = \varepsilon u :: U$ with $u \in \text{TERM}_U'$ and $\varepsilon \vdash U' \gg U$. Furthermore:
1. If \( U = \text{Bool}_g \) then either \( v = b_g \) or \( v = \varepsilon b_{g'} \) :: \( \text{Bool}_{g'} \) with \( b_{g'} \in \text{TERM}_{\text{Bool}_{g'}} \) and \( \varepsilon \vdash \text{Bool}_{g'} \preceq \text{Bool}_g \).

2. If \( U = U_1 \overset{g}{\rightarrow}_g U_2 \) then either \( v = (\lambda^{g_c}x^{U_1}.t^{U_2})_g \) with \( t^{U_2} \in \text{TERM}_{U_2} \) or \( v = \varepsilon(\lambda^{g_c}x^{U_1}.t^{U_2})_g' :: U_1 \overset{g}{\rightarrow}_g U_2 \) with \( t^{U_2} \in \text{TERM}_{U_2} \) and \( \varepsilon \vdash U_1 \overset{g}{\rightarrow}_g U_2' \preceq U_1 \overset{g}{\rightarrow}_g U_2 \).

3. If \( U = \text{Ref}_g U_1 \) then either \( v = o_{g'}^{U_1} \) or \( v = \varepsilon o_{g'}^{U_1} :: \text{Ref}_g U_1 \) with \( o_{g'}^{U_1} \in \text{Ref}_{g'} U_1' \) and \( \varepsilon \vdash \text{Ref}_{g'} U_1' \preceq \text{Ref}_g U_1 \).

**Proof.** By direct inspection of the formation rules of gradual intrinsic terms (Figure B.12).

**Lemma 179** (Substitution). If \( \phi \triangleright t^U \in \text{TERM}_U \) and \( \phi \triangleright v \in \text{TERM}_{U_1} \), then \( \phi \triangleright [v/x^{U_1}]t^U \in \text{TERM}_U \).

**Proof.** By induction on the derivation of \( \phi \triangleright t^U \).

**Proposition 180** (\( \rightarrow \) is well defined). If \( t^U \mid \mu \rightarrow r \) and \( t^U \vdash \mu \), then \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) and if \( r = t^{U'} \mid \mu' \in \text{CONFIG}_U \), then also \( t^{U'} \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

**Proof.** By induction on the structure of a derivation of \( t^G \mid \mu \rightarrow r \), considering the last rule used in the derivation.

**Case** (I+). Then \( t^U = b_1\ell_1 \oplus g_2\ell_2 \). By construction we can suppose that \( g = g_1' \tilde{\gamma} g_2' \), then

\[
\begin{array}{c}
\phi.\varepsilon \triangleright \phi.\varepsilon \preceq g_c \preceq \phi.g_c \\
\phi \triangleright b_1\ell_1 \in \text{Bool}_{g_1} \quad \varepsilon_1 \vdash \text{Bool}_{g_1} \preceq \text{Bool}_{g_1'} \\
\phi \triangleright b_2\ell_2 \in \text{Bool}_{g_2} \quad \varepsilon_2 \vdash \text{Bool}_{g_2} \preceq \text{Bool}_{g_2'} \\
(I+) \quad \phi \triangleright \varepsilon_1 b_1\ell_1 \oplus^g \varepsilon_2 b_2\ell_2 \in \text{TERM}_{\text{Bool}_g}
\end{array}
\]

Therefore

\[
\phi \triangleright (\varepsilon_1 b_1)_g \oplus^g_2 (b_2)_g \mid \mu
\]

Then

\[
(I+) \quad \phi.\varepsilon \vdash \phi.g_c \preceq \phi.g_c
\]

and the result holds.

**Case** (Iprot). Then \( t^U = \phi \triangleright \text{prot}_{g'}^{gU} \phi'(\varepsilon u) \) and

\[
\begin{array}{c}
\phi.\varepsilon \vdash \phi.g_c \preceq \phi.g_c \quad \varepsilon' \vdash g_r \quad g' \preceq g_c' \\
\phi' \vdash u \in \text{TERM}_{U''} \\
(I\text{prot}) \quad \varepsilon \vdash U'' \preceq U \quad \varepsilon \vdash g' \preceq g \\
\phi \triangleright \text{prot}_{g'}^{gU} \phi'(\varepsilon u) \in \text{TERM}_{U'\tilde{\gamma} g}
\end{array}
\]

Therefore

\[
\text{prot}_{g'}^{gU} \phi'(\varepsilon u) \mid \mu \overset{\phi}{\rightarrow} (\varepsilon \tilde{\gamma} \varepsilon \ell)(u \tilde{\gamma} g') :: U \tilde{\gamma} g \mid \mu
\]
But by Lemma 177, \( \phi \triangleright u \in \text{TERM}_{U \cdot \gamma} \). Therefore by definition of join, \( \phi \triangleright (u \mapsto g') \in \text{TERM}_{U \cdot \gamma \cdot g'} \). Then using Lemma 28

\[
\phi \triangleright (u \mapsto g') \in \text{TERM}_{U \cdot \gamma \cdot g'}
\]

and the result holds.

**Case (lapp).** Then \( t^U = \varepsilon_1(\lambda^{g''}_{\varepsilon} x^{U_{11}}. t^{U_{12}})_{g_1} \@_{\varepsilon_1}^{g''_{\varepsilon}} \delta_{\varepsilon}^{U_{11}} u \) and \( U = U_2 \mapsto g \).

\[
\phi \triangleright t^{U_{12}} \in \text{TERM}_{U \cdot \varepsilon_{11}}
\]

\[
\phi \triangleright (\lambda^{g''}_{\varepsilon} x^{U_{11}}. t^{U_{12}})_{g_1} \in \text{TERM}_{U_{11}, \varepsilon_{11} \triangleright \delta_{\varepsilon}^{U_{11}}}
\]

\[
\phi \triangleright u \in \text{TERM}_{U_2}
\]

\[
\varepsilon_2 \triangleright U_2 \leq U_1
\]

\[
\varepsilon_1 \triangleright U_{11} \overset{\delta_{\varepsilon}^{U_{11}}}{\rightarrow} U_{12} \leq U_1 \overset{g'_2}{\longrightarrow} U_2
\]

\[
\varepsilon_1 \triangleright \text{g}_c \gamma \gamma \leq g'_c
\]

\[
\phi \triangleright \phi . g_c \sim \phi . g_c
\]

If \( \varepsilon'_1 = (\varepsilon_2 \circ \kappa : \text{idom}(\varepsilon_1)) \) or \( \varepsilon'_1 = (\phi . g_c \mapsto g'_c) \circ \varepsilon_1 \circ \kappa : \text{ilat}(\varepsilon_1) \) are not defined, then \( t^U \mid \mu \not\propto \phi \cdot U \cdot \gamma \cdot g \), and then the result hold immediately. Suppose that consistent transitivity does hold, then if \( \phi' = (\phi . g_c \mapsto g_1 \mapsto g') \)

\[
\varepsilon_1(\lambda^{g''}_{\varepsilon} x^{U_{11}}. t^{U_{12}})_{g_1} \@_{\varepsilon_1}^{g''_{\varepsilon}} \delta_{\varepsilon}^{U_{11}} u \mid \mu \not\propto \text{prot}_{\text{idom}(\varepsilon_1) \cdot g_1 \cdot g'_2}(\text{icod}(\varepsilon_1)([\varepsilon' \cdot U_{11}/x^{U_{11}}] t^{U_{12}})) \mid \mu
\]

As \( \varepsilon_2 \triangleright U_2 \leq U_1 \) and by inversion lemma \( \text{idom}(\varepsilon_1) \triangleright U_1 \leq U_{11} \), then \( \varepsilon'_1 \triangleright U_2 \leq U_1 \). Therefore \( \phi \triangleright \varepsilon'_1 \cdot U_11 \in \text{TERM}_{U \cdot \varepsilon_{11}} \), and by Lemma 179, \( \phi \triangleright \varepsilon'_1 \cdot U_{11}/x^{U_{11}}] t^{U_{12}} \in \text{TERM}_{U \cdot \varepsilon_{12}} \).

We know that \( \varepsilon_1 \triangleright g_c \gamma \gamma \leq g'_c \). By inversion on the label of types, \( \text{ilbl}(\varepsilon_1) \triangleright g_1 \leq g \). Also by monotonicity of the join, \( \phi . g_c \mapsto \text{ilbl}(\varepsilon_1) \triangleright \phi . g_c \mapsto \text{g}_c \gamma \gamma \leq g_c \gamma \gamma \). Then, by inversion on the latent effect of function types, \( \text{ilat}(\varepsilon_1) \triangleright g'_c \leq g_c \). Therefore combining evidences, as \( \phi . g_c = (\phi . g_c \mapsto \text{ilbl}(\varepsilon_1)) \circ \kappa \cdot \varepsilon_1 \circ \kappa : \text{ilat}(\varepsilon_1) \), we may justify the runtime judgment \( \phi . g_c \rightleftarrows \phi . g_c \).

Let us call \( t^{U_{12}} = \varepsilon_1(\lambda^{g''}_{\varepsilon} x^{U_{11}}. t^{U_{12}})_{g_1} \@_{\varepsilon_1}^{g''_{\varepsilon}} \delta_{\varepsilon}^{U_{11}} u \). By Lemma 176, \( \phi' \triangleright t^{U_{12}} \in \text{TERM}_{U \cdot \varepsilon_{12}} \). Then

\[
\phi' \triangleright \text{prot}_{\text{idom}(\varepsilon_1) \cdot g_1 \cdot g'_2}(\text{icod}(\varepsilon_1)([\varepsilon' \cdot U_{11}/x^{U_{11}}] t^{U_{12}})) \in \text{TERM}_{U \cdot \varepsilon_{12}}
\]

and the result holds.
Case (if-true). Then \( t^U = \text{if}^g \varepsilon_1 b_{g_1} \) then \( \varepsilon_2 t^{U_2} \) else \( \varepsilon_3 t^{U_3} \), \( U = (U_2 \vee U_3) \bar{g} g \) and

\[
\begin{align*}
\phi \triangleright b_{g_1} & \in \text{TERM}_{\text{Bool}_{g_1}} & \varepsilon_1 & \vdash \text{Bool}_{g_1} \subseteq \text{Bool}_{g} \\
\phi' = \langle \phi, \varepsilon \bar{g} \text{ilbl}(\varepsilon_1)(\phi, g_c \bar{g} g_1), \phi, g_c \bar{g} g \rangle & & \phi \varepsilon \vdash \phi, g_c \bar{g} \phi, g_c \\
\phi' \triangleright t^{U_2} & \in \text{TERM}_{U_2} & \varepsilon_2 & \vdash U_2 \subseteq (U_2 \vee U_3) \\
\phi' \triangleright t^{U_3} & \in \text{TERM}_{U_3} & \varepsilon_3 & \vdash U_3 \subseteq (U_2 \vee U_3)
\end{align*}
\]

Therefore

\[
\text{if}^g \varepsilon_1 b_{g_1} \text{ then } \varepsilon_2 t^{U_2} \text{ else } \varepsilon_3 t^{U_3} \mid \mu \xrightarrow{\phi} \text{prot}^{g_\text{ilbl}(\varepsilon_1)g_1}_{g_1} \phi' (\varepsilon_2 t^{U_2}) \mid \mu
\]

But

\[
\phi, \varepsilon \vdash \phi, g_c \bar{g} \phi, g_c \\
\phi' \triangleright t^{U_2} \in \text{TERM}_{U_2}
\]

(Iprot)

\[
\varepsilon_2 \vdash U_2 \subseteq (U_2 \vee U_3) \text{ ilbl}(\varepsilon_1) \vdash g_1 \bar{g} g
\]

\[
\phi \triangleright \text{prot}^{g_\text{ilbl}(\varepsilon_1)g_1}_{g_1} \phi' (\varepsilon_2 t^{U_2}) \in \text{TERM}_{(U_2 \vee U_3) \bar{g} g}
\]

and the result holds.

Case (if-false). Analogous to case (if-true).

Case (If). Then \( t^U = \text{ref}^U \varepsilon u \) and

\[
\begin{align*}
\phi, \varepsilon & \vdash \phi, g_c \bar{g} \phi, g_c & \phi \triangleright u & \in \text{TERM}_{U''} \\
\varepsilon & \vdash U'' \subseteq U' & \varepsilon_{\ell} & \vdash g_c \bar{g} \text{label}(U')
\end{align*}
\]

(Iref)

\[
\phi \triangleright \text{ref}^U \varepsilon u \in \text{TERM}_{\text{Ref}_{\ell} U'}
\]

If \( \varepsilon' = \varepsilon \bar{g} (\phi, \varepsilon \circ \varepsilon_{\ell}) \) is not defined, then \( t^{U''} \mid \mu \xrightarrow{\phi} \text{error} \), and then the result hold immediately. Suppose that consistent transitivity does hold, then

\[
\text{ref}^U \varepsilon u \mid \mu \xrightarrow{\phi} o^{U''}_{\perp} \mid \mu \mid o^{U''} \mapsto \varepsilon'(u \bar{g} \phi, g_c) :: U'
\]

where \( o^{U''} \notin \operatorname{dom}(\mu) \).

We know that \( \varepsilon_{\ell} \vdash g_c \bar{g} \text{label}(U') \), therefore \( \phi, \varepsilon \circ \varepsilon_{\ell} \vdash \phi, g_c \bar{g} \text{label}(U') \). We also know that \( \varepsilon \vdash U'' \subseteq U' \). Therefore combining both evidences we can justify that \( \varepsilon \bar{g} (\phi, \varepsilon \circ \varepsilon_{\ell} \vdash U_2 \vee g \phi, g_c \ll \cdot U' \). But

\[
\phi, \varepsilon \vdash \phi, g_c \bar{g} \phi, g_c \\
o^{U''} \in \text{TERM}_{\text{Ref}_{\ell} U'}
\]

Let us call \( \mu' = \mu [o^{U''} \mapsto \varepsilon'(u \bar{g} \phi, g_c) :: U'] \). It is easy to see that \( \text{freeLocs}(o^{U''}) = o^{U''} \) and \( \operatorname{dom}(\mu') = \operatorname{dom}(\mu) \cup o^{U''} \), then \( \text{freeLocs}(o^{U''}) \subseteq \operatorname{dom}(\mu') \). Given that \( t^{U''} \vdash \mu \) then \( \text{freeLocs}(u) \subseteq \operatorname{dom}(\mu) \), and therefore \( \forall u \in \operatorname{cod}(\mu') = \operatorname{cod}(\mu) \cup (\varepsilon'(u \bar{g} \phi, g_c) :: U') \), \( \text{freeLocs}(U') \subseteq \operatorname{dom}(\mu') \). Finally as \( t^{U''} \vdash \mu \) and \( \mu'(o^{U''}) = \varepsilon'(u \bar{g} \phi, g_c) :: U' \in \text{TERM}_{U''} \) then we can conclude that \( t^{U''} \vdash \mu' \) and \( \operatorname{dom}(\mu) \subseteq \operatorname{dom}(\mu') \), and the result holds.
Case (Ideref). Then $t^U = !^\text{Ref}_g U' ::_{g_1}^g \varepsilon_2 u$ and

$$
\begin{align*}
\phi \triangleright o_{g_1}^{U''} &\in \text{TERM}_{\text{Ref}_g'} U'' \\
\varepsilon &\vdash \text{Ref}_g' U'' \lessapprox \text{Ref}_g U' \\
\phi.\varepsilon &\vdash \phi. g_c \lessapprox \phi. g_c
\end{align*}
$$

(Ideref)

Then for $\phi' = ((\phi. \varepsilon \bar{\gamma} \text{ilbl}(\varepsilon))(\phi. g_c \bar{\gamma} g'), \phi. g_c \bar{\gamma} g)$$

$$
!^\text{Ref}_g U' ::_{g_1}^g \varepsilon_2 u \xrightarrow{\phi} \text{prot}_{\text{ilbl}(\varepsilon)g'} \phi'(\text{iref}(\varepsilon)v) | \mu
$$

where $\mu(o_{g_1}^{U''}) = v$. As the store is well typed, therefore $\phi \triangleright v \in \text{TERM}_{U''}$. By Lemma 177 $\phi' \triangleright v \in \text{TERM}_{U''}$. By inversion lemma on references, $\text{ilbl}(\varepsilon) \vdash g' \lessapprox g$ and $\text{iref}(\varepsilon) \vdash U'' \lessapprox U'$

$$
\begin{align*}
\phi.\varepsilon &\vdash \phi. g_c \lessapprox \phi. g_c \\
\phi' \triangleright v &\in \text{TERM}_{U''} \\
\text{iref}(\varepsilon) &\vdash U'' \lessapprox U' \\
\text{ilbl}(\varepsilon) &\vdash g' \lessapprox g
\end{align*}
$$

(Iprot)

and the result holds.

Case (Iassgn). Then $t^U = \varepsilon_1 o_{g_1}^{U'} ::_{\varepsilon_2} \varepsilon_2 u$ and

$$
\begin{align*}
\varepsilon_1 &\vdash \text{Ref}_g' U'_1 \lessapprox \text{Ref}_g U_1 \\
\varepsilon_2 &\vdash U_2 \lessapprox U_1 \\
\phi.\varepsilon &\vdash \phi. g_c \lessapprox \phi. g_c \\
\varepsilon_2 &\vdash \phi. g_c \lessapprox \text{label}(U_1)
\end{align*}
$$

(Iassgn)

If $\varepsilon' = (\varepsilon_2 \circ^\text{iref}(\varepsilon_1)) \bar{\gamma} (((\phi. \varepsilon \bar{\gamma} \text{ilbl}(\varepsilon_1)) \circ^\varepsilon \varepsilon_2 \circ^\text{ilbl}(\text{iref}(\varepsilon_1)))$ is not defined, then $t^{U'} | \mu \xrightarrow{\phi} \text{error}$, and then the result hold immediately. Suppose that consistent transitivity does hold, then

$$
\varepsilon_1 o_{g_1}^{U'} ::_{\varepsilon_2} \varepsilon_2 u \xrightarrow{\phi} \text{unit}_1 | \mu(o^U \mapsto \varepsilon'(u \bar{\gamma} (\phi. g_c \bar{\gamma} g)) :: U'_1)
$$

We know that $\varepsilon_2 \vdash \phi. g_c \gamma \g \lessapprox \text{label}(U_1)$. Then by inversion on reference evidence types and inversion in the label of types, $\text{ilbl}(\text{iref}(\varepsilon_1)) \vdash \text{label}(U_1) \lessapprox \text{label}(U'_1)$. But $\text{ilbl}(\varepsilon_1) \vdash g' \lessapprox g$, using monotonicity of the join, $\phi.\varepsilon \bar{\gamma} \text{ilbl}(\varepsilon_1) \vdash \phi. g_c \gamma g' \lessapprox \phi. g_c \gamma g$. Therefore $((\phi. \varepsilon \bar{\gamma} \text{ilbl}(\varepsilon_1)) \circ^\varepsilon \varepsilon_2 \circ^\text{iref}(\varepsilon_1)) \vdash U_2 \lessapprox U'_1$. Combining both evidences, $\varepsilon' = (\varepsilon_2 \circ^\text{iref}(\varepsilon_1)) \bar{\gamma} (((\phi. \varepsilon \bar{\gamma} \text{ilbl}(\varepsilon_1)) \circ^\varepsilon \varepsilon_2 \circ^\text{ilbl}(\text{iref}(\varepsilon_1)))$, and by Proposition 28 we can then justify that $\varepsilon' \vdash U_2 \gamma (\phi. g_c \gamma g) \lessapprox U'_1$ and therefore justify the ascription in the heap.

Let us call $\mu' = \mu[o^U \mapsto \varepsilon'(u \bar{\gamma} (\phi. g_c \bar{\gamma} g)) :: U'_1]$. As $\text{freeLocs}(\text{unit}_1) = \emptyset$ then $\text{freeLocs}(\text{unit}_1) \subseteq \mu'$.

As $t^{U'} | \mu$ then $\text{freeLocs}(u) \in \text{dom}(\mu)$, and as $\text{dom}(\mu) = \text{dom}(\mu')$ then it is trivial to see that $\forall v' \in \text{cod}(\mu'), \text{freeLocs}(v') \subseteq \text{dom}(\mu')$, and the result holds.
Proposition 181 (→→ is well defined). If \( t^U \mid \mu \xrightarrow{\phi} r \) and \( t^U \vdash \mu \), then \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) and if \( r = t^U \mid \mu' \in \text{CONFIG}_U \) then also \( t^U \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

Proof. By induction on the structure of a derivation of \( t^U \mid \mu \xrightarrow{\phi} r \).

Case (R→→). \( t^U \mid \mu \xrightarrow{\phi} r \). By well-definedness of \( \xrightarrow{\cdot} \) (Prop 280), \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \), and if \( r = t^U \mid \mu' \in \text{CONFIG}_U \) then also \( t^U \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

Case (Rprot). \( t^U = \text{prot}_{\varepsilon g'} \phi'(\varepsilon t^U_{\mu''}) \) and

\[
\begin{align*}
\phi.\varepsilon & \vdash \phi.\varepsilon' \simeq g' \phi' \simeq g' \\
\phi' \triangleright t^U_{\mu''} & \in \text{TERM}_{U''} \\
\varepsilon & \vdash U'' \simeq U'' \\
\varepsilon & \vdash g' \simeq g
\end{align*}
\]

(iProt)

\[
\phi \triangleright \text{prot}_{\varepsilon g'} \phi'(\varepsilon t^U_{\mu''}) \in \text{TERM}_{U'' \varepsilon g}
\]

Using induction hypothesis on the premise of (Rprot()), then

\[
\begin{align*}
t^U_{\mu''} \mid \mu \xrightarrow{\phi' \phi} t^U_{\mu'} & \mid \mu' \\
\text{prot}_{\varepsilon g'} \phi'(\varepsilon t^U_{\mu''}) & \mid \mu \xrightarrow{\phi} \text{prot}_{\varepsilon g'} \phi'(\varepsilon t^U_{\mu''}) \mid \mu'
\end{align*}
\]

where \( \phi' \triangleright t^U_{\mu''} \in \text{TERM}_{U''} \), \( t^U_{\mu''} \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \). Therefore

\[
\begin{align*}
\phi.\varepsilon & \vdash \phi.\varepsilon' \simeq g' \phi' \simeq g' \\
\phi' \triangleright t^U_{\mu''} & \in \text{TERM}_{U''} \\
\varepsilon & \vdash U'' \simeq U'' \\
\varepsilon & \vdash g' \simeq g
\end{align*}
\]

(iProt)

\[
\phi \triangleright \text{prot}_{\varepsilon g'} \phi'(\varepsilon t^U_{\mu''}) \in \text{TERM}_{U'' \varepsilon g}
\]

and the result holds.

Case (Rf). \( t^U = f[t^U] \), \( \phi \triangleright f[t^U] \in \text{TERM}_U \), \( t^U_{\mu''} \mid \mu \xrightarrow{\phi} t^U_{\mu'}, \) and consider \( F : \text{TERM}_U \rightarrow \text{TERM}_U \), where \( F(\phi \triangleright t^U) = \phi \triangleright f[t^U] \). By induction hypothesis, \( \phi \triangleright t^U_{\mu''} \in \text{TERM}_U \), so \( F(\phi \triangleright t^U_{\mu''}) = \phi \triangleright f[t^U_{\mu''}] \in \text{TERM}_U \).

By induction hypothesis we also know that \( t^U_{\mu''} \vdash \mu' \).

If \( \text{freeLocs}(t^U_{\mu''}) \subseteq \mu' \), \( \text{freeLocs}(f[t^U]) \subseteq \mu \), and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \), then it is easy to see that \( \text{freeLocs}(f[t^U_{\mu}]) \subseteq \mu' \), and therefore conclude that \( f[t^U_{\mu} \vdash \mu' \).

Case (Rferr, Rherr, Rprot()ferr, Rprot()herr). \( r = \text{error} \).

Case (Rh). \( t^U = h[et], \phi \triangleright h[t^U] \in \text{TERM}_U \), and consider \( G : \text{EVTerm} \times \text{GLabel} \times \text{GLabel} \times \text{EVTerm} \rightarrow \text{TERM}_U \), \( G(\phi, et) = \phi \triangleright h[et] \) and \( et \rightarrow c \ et' \). Then there exists \( U_e, U_x \) such that \( et = \varepsilon \varepsilon t^{U_e} \) and \( \varepsilon \varepsilon \vdash U_e \subseteq U_x \). Also, \( t_e = \varepsilon \varepsilon v :: U_e \), with \( v \in \text{TERM}_{U_v} \) and \( \varepsilon \varepsilon \vdash U_e \subseteq U_v \).

We know that \( \varepsilon \varepsilon \triangleright v \circ \varepsilon \varepsilon \) is defined, and \( et = \varepsilon \varepsilon t^{U_e} \rightarrow_c \varepsilon \varepsilon v = et' \). By definition of \( \circ \leq \) we have \( \varepsilon \varepsilon \vdash U_v \subseteq U_x \), so \( G(\phi, et) = \phi \triangleright h[et] \in \text{TERM}_U \).

As \( \text{freeLocs}(et) = \text{freeLocs}(et') \) and \( \mu' = \mu \) then it is easy to conclude that \( h[et] \vdash \mu \).

Case (Rprot()h). Similar case to (Rh) case, using \( P : \text{EVTerm} \rightarrow \text{TERM}_U \), \( P(et) = \phi \triangleright \text{prot}_{\varepsilon g'} \phi'(et) \).

284
Now we can establish type safety: programs do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

**Proposition 182** (Type Safety). If \( \phi \vdash t^U \in \text{TERM}_U \) then either \( t^U \) is a value \( v \); \( t^U \vdash_{\phi} v \); or if \( t^U \vdash \mu \) then \( t^U \vdash_{\phi} t^U \vdash_{\mu'} \) for some term \( \phi \vdash t^U \in \text{TERM}_U \) and some \( \mu' \) such that \( t^U \vdash \mu' \) and \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

**Proof.** By induction on the structure of \( \phi \vdash t^U \).

**Case (Iu,Il, Ib, Ix, Iλ).** \( t^U \) is a value.

**Case (Iprot).** \( t^U = \text{prot}_{\phi,g}^{\phi'}(\varepsilon t^{U'}) \), and

\[
\phi.\varepsilon \vdash \phi.g_c \overset{\triangleright}{\sim} \phi.g_c \quad \varepsilon_r \vdash g_r \overset{\triangleright}{\sim} g_r' \\
\phi' \vdash t^{U'} \in \text{TERM}_{U'} \quad \varepsilon \vdash U' \overset{\triangleright}{\prec} U \quad \varepsilon_r \vdash g' \overset{\triangleright}{\sim} g
\]

By induction hypothesis on \( t^{U'} \), one of the following holds:

1. \( t^{U'} \) is a simple value, then by \( (R_{ightarrow}) \), \( t^U \vdash_{\phi} v \vdash_{\mu} \), and by Prop 282 \( \phi \vdash v \in \text{TERM}_U \) and the result holds.

2. \( t^{U'} \) is an ascribed value \( v \), then, \( \varepsilon t^{U'} \rightarrow_c et' \) for some \( et' \in \text{EVTERM} \cup \{ \text{error} \} \). Hence \( t^U \vdash \mu \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either \( (Rg) \), or \( (Rerrg) \).

3. \( t^{U'} \vdash_{\mu} r_1 \) for some \( r_1 \in \text{TERM}_{U_1} \cup \{ \text{error} \} \). Hence \( t^U \vdash \mu \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either \( (Rprot) \), or \( (Rprotferr) \).

**Case (I::).** \( t^U = \varepsilon_1 t^{U_1} :: U_2 \), and

\[
\phi \vdash t^{U_1} \in \text{TERM}_{U_1} \\
(\varepsilon_1 \vdash U_1 \overset{\triangleright}{\leq} U_2 \quad \phi.\varepsilon \vdash \phi.g_c \overset{\triangleright}{\sim} \phi.g_c) \\
\phi \vdash \varepsilon_1 t^{U_1} :: U_2 \in \text{TERM}_{U_2}
\]

By induction hypothesis on \( t^{U_1} \), one of the following holds:

1. \( t^{U_1} \) is a value, in which case \( t^U \) is also a value.

2. \( t^{U_1} \vdash_{\mu} r_1 \) for some \( r_1 \in \text{TERM}_{U_1} \cup \{ \text{error} \} \). Hence \( t^U \vdash \mu \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either \( (Rf) \), or \( (Rferr) \).
Case (Iif). \( t^U = \text{if}^g \varepsilon_1 t^{U_1} \text{then} \varepsilon_2 t^{U_2} \text{else} \varepsilon_3 t^{U_3} \) and

\[
\begin{align*}
\phi \triangleright t^{U_1} & \in \text{TERM}_{U_1} \quad \varepsilon_1 \vdash U_1 \preceq \text{Bool}_g \\
\phi' & = (\phi, \varepsilon \triangleright \text{ilbl}(\varepsilon_1))(\phi, g_c \triangleright \text{label}(U_1)), g_c \triangleright g \\
\phi' \triangleright t^{U_2} & \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \preceq (U_2 \triangleright U_3) \\
\phi \triangleright t^{U_3} & \in \text{TERM}_{U_3} \quad \varepsilon_3 \vdash U_3 \preceq (U_2 \triangleright U_3)
\end{align*}
\]

By induction hypothesis on \( t^{U_1} \), one of the following holds:

1. \( t^{U_1} \) is a value \( u \), then by (R\(\rightarrow\)), \( t^U \mid \mu \mapsto \phi \) r and \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \)

2. \( t^{U_1} \) is an ascribed value \( v \), then, \( \varepsilon_1 t^{U_1} \longrightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \).
   Hence \( t^U \mid \mu \mapsto \phi \) r for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \) and either (Rg), or (Rgerr).

3. \( t^{U_1} \mid \mu \mapsto \phi \) r_1 for some \( r_1 \in \text{TERM}_{U_1} \cup \{ \text{error} \} \). Hence \( t^U \mid \mu \longrightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \) and either (Rf), or (Rferr).

Case (Iapp). \( t^U = \varepsilon_1 t^{U_1} \oplus \varepsilon_2 t^{U_2} \)

\[
\begin{align*}
\phi \triangleright t^{U_1} & \in \text{TERM}_{U_1} \quad \varepsilon_1 \vdash U_1 \preceq U_{11} \overset{g_c}{\rightarrow}_g U_{12} \\
\phi \triangleright t^{U_2} & \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \preceq U_{11} \\
\varepsilon_2 \vdash g_c \triangleright g & \preceq g'_c \\
\phi, \varepsilon & \vdash \phi, g_c \triangleright \phi, g_c
\end{align*}
\]

By induction hypothesis on \( t^{U_1} \), one of the following holds:

1. \( t^{U_1} \) is a value \( (\lambda x^{U_1}, t^{U_2})_{g'} \) (by canonical forms Lemma\(\text{274} \)), posing \( U_1 = U_{11} \overset{g'}{\rightarrow}_g U_{12} \).
   Then by induction hypothesis on \( t^{U_2} \), one of the following holds:
   
   (a) \( t^{U_2} \) is a value \( u \), then by (R\(\rightarrow\)), \( t^U \mid \mu \mapsto \phi \) r and \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \)
   
   (b) \( t^{U_2} \) is an ascribed value \( v \), then, \( \varepsilon_2 t^{U_2} \longrightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \).
   Hence \( t^U \mid \mu \mapsto \phi \) r for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \) and either (Rg), or (Rgerr).
   
   (c) \( t^{U_2} \mid \mu \mapsto \phi \) r_2 for some \( r_2 \in \text{CONFIG}_{U_2} \cup \{ \text{error} \} \). Hence \( t^U \mid \mu \mapsto \phi \) r for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \) and either (Rf), or (Rferr). Also by Prop\(\text{282} \) if \( r = t^U \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

2. \( t^{U_1} \) is an ascribed value \( v \), then, \( \varepsilon_1 t^{U_1} \longrightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \).
   Hence \( t^U \mid \mu \mapsto \phi \) r for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop\(\text{282} \) and either (Rg), or (Rgerr).

286
3. \( t^U \mid \mu \to r_1 \) for some \( r_1 \in \text{CONFIG}_U \cup \{ \text{error} \} \). Hence \( t^U \mid \mu \to r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either (Rf), or (Rferr). Also by Prop 282 if \( r = t^U \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

**Case (Ideref).** Similar case to (Iapp)

**Case (Iref).** \( t^U = \text{ref}^U_{\epsilon_t} \in t^{U''} \) and

\[
\begin{align*}
\phi \cdot \epsilon & \vdash \phi \cdot g_c \not\leq \phi \cdot g_c \\
\phi \cdot t^{U''} & \in \text{TERM}_{U''} \\
\epsilon & \vdash U'' \not\leq U' \\
& \text{by Prop 282, if } \epsilon \vdash U'' \not\leq U'
\end{align*}
\]

By induction hypothesis on \( t^{U''} \), one of the following holds:

1. \( t^{U''} \) is a value \( v \), then by (R\to), \( t^{U''} \mid \mu \to r \) and \( r \in \text{CONFIG}_{U'} \) by Prop 282. Also by Prop 282 if \( r = t^U \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

2. \( t^{U''} \) is an ascribed value \( v \), then, \( \epsilon t^{U''} \to \epsilon' \) for some \( \epsilon' \in \text{EvTERM} \cup \{ \text{error} \} \). Hence \( t^U \mid \mu \to r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either (Rg), or (Rgerr).

3. \( t^{U''} \mid \mu \to r_1 \) for some \( r_1 \in \text{CONFIG}_{U''} \cup \{ \text{error} \} \). Hence \( t^U \mid \mu \to r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either (Rf), or (Rferr). Also by Prop 282 if \( r = t^U \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

**Case (Iassign).** \( t^U = \epsilon_1 t^{U''}_{\epsilon_1} \in t^{U''} \) and

\[
\begin{align*}
\epsilon_1 & \vdash \text{Ref}_{g'} U_1' \not\leq \text{Ref}_g U_1 \\
\epsilon_2 & \vdash U_2 \not\leq U_1 \\
\phi & \vdash t^{U''} \in \text{TERM}_{\text{Ref}_{g'} U_1'} \\
\phi & \vdash t^{U_2} \in \text{TERM}_{U_2} \\
\phi & \cdot \epsilon_1 \cdot g_c \not\leq \phi \cdot g_c \\
\phi & \cdot t^{U_2} \in \text{TERM}_{U_2} \\
\phi & \cdot \epsilon_1 t^{U''}_{\epsilon_1} \not\leq \phi \cdot g_c \not\leq \phi \cdot \epsilon_1 t^{U_2} \\
\phi & \cdot \epsilon_2 t^{U_2} \in \text{TERM}_{\text{UNIT}_{U_2}} \\
\phi & \cdot \epsilon_1 t^{U''}_{\epsilon_1} \not\leq \phi \cdot \epsilon_2 t^{U_2} \in \text{TERM}_{\text{UNIT}_{U_2}}
\end{align*}
\]

287
By induction hypothesis on \( t^{U_1} \), one of the following holds:

1. \( t^{U_1} \) is a value \( U^{U_1} \) (by canonical forms Lemma \( 274 \)), where \( U^{U_1} = \text{Ref}_{\ell_1} U^{U_1} \). Then by induction hypothesis on \( t^{U_2} \), one of the following holds:

   (a) \( t^{U_2} \) is a value \( u \), then by \( (\text{Ref} \to) \), \( t^{U} \mid \mu \xrightarrow{\phi} r \) and \( r \in \text{CONFIG}_U \cup \{\text{error}\} \) by Prop \( 282 \). Also by Prop \( 282 \) if \( r = t^{U} \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

   (b) \( t^{U_2} \) is an ascribed value \( v \), then, \( \varepsilon_2 t^{U_2} \rightarrow_c \ell et' \) for some \( et' \in \text{EvTerm} \cup \{\text{error}\} \). Hence \( t^{U} \mid \mu \xrightarrow{\phi} r \) for some \( r \in \text{CONFIG}_U \cup \{\text{error}\} \) by Prop \( 282 \) and either \( (\text{Rg}) \), or \( (\text{Rgerr}) \). Also by Prop \( 282 \) if \( r = t^{U} \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

2. \( t^{U_1} \) is an ascribed value \( v \), then, \( \varepsilon_1 t^{U_1} \rightarrow_c \ell et' \) for some \( et' \in \text{EvTerm} \cup \{\text{error}\} \). Hence \( t^{U} \mid \mu \xrightarrow{\phi} r \) for some \( r \in \text{CONFIG}_U \cup \{\text{error}\} \) by Prop \( 282 \) and either \( (\text{Rg}) \), or \( (\text{Rgerr}) \).

3. \( t^{U_1} \mid \mu \xrightarrow{\phi} r_1 \) for some \( r_1 \in \text{CONFIG}_U \cup \{\text{error}\} \). Hence \( t^{U} \mid \mu \xrightarrow{\phi} r \) for some \( r \in \text{CONFIG}_U \cup \{\text{error}\} \) by Prop \( 282 \) and either \( (\text{Rg}) \), or \( (\text{Rgerr}) \). Also by Prop \( 282 \) if \( r = t^{U} \mid \mu' \in \text{TERM}_U \) then \( \text{dom}(\mu) \subseteq \text{dom}(\mu') \).

\[ \square \]

**Proposition 183 (Static terms do not fail).** Let us define \( \text{StaticTerm} \) the set of evidence augmented terms with full static annotations. Then consider \( t_s \in \text{StaticTerm} \), \( \phi = \langle \varepsilon, \ell \rangle \), and \( \mu_s \), such that \( \varepsilon = \mathcal{E}\ell' \lesssim \ell \), \( \phi \triangleright t_s \in \text{Term}_s \), and that \( \forall v_s \in \text{cod}(\mu_s) \), \( v_s \in \text{StaticTerm} \). Then either \( t_s \) is a value, or

\[ t_s \mid \mu_s \xrightarrow{\phi} t'_s \mid \mu'_s \]

**Proof.** We know that if you follow AGT to derive the dynamic semantics of a gradual language, then by construction the resulting language satisfy the dynamic conservative extension property. As we follow AGT to derive the dynamic semantics, we get this property by construction, save for the assignment elimination reduction rule. In this rule we add an extra check of the form \( \phi \varepsilon \lesssim \varepsilon \). So if we prove that the extra check is always satisfied, then the result holds.

Let us consider a \( t'_i \) fully static like so:

\[
\begin{align*}
\varepsilon_1 &\vdash \text{Ref}_{\ell_1} S_1' \lesssim \text{Ref}_{\ell_1} S_1' \\
\varepsilon_2 &\vdash S_2 \lesssim S_1 \\
\phi \triangleright \ell' &\lesssim \ell \\
\varepsilon_2 &\vdash \ell_1 \lesssim \ell_2 \\
\varepsilon_2 &\vdash \ell \lesssim \text{label}(S_1) \\
\phi \triangleright \varepsilon_1 &\ell S_1' \lesssim S_1 \\
\phi \triangleright \varepsilon_2 &\ell S_1' \lesssim S_1 \\
\phi \triangleright u &\in \text{Term}_{U_2} \\
\varepsilon_2 &\vdash \ell_2 \lesssim \text{label}(S_1) \\
\phi \triangleright \varepsilon_1 &\ell S_1' \lesssim S_1 \\
\phi \triangleright \varepsilon_2 &\ell S_1' \lesssim S_1 \\
\phi \triangleright u &\in \text{Term}_{U_2} \\
\end{align*}
\]

288
By inspection of the reduction rules we have to prove that $\phi \varepsilon \preceq_i ilbl(\varepsilon)$. $\phi \varepsilon \preceq_i ilbl(\varepsilon)$. We know by definition of interior between two static labels that $\varepsilon = \mathcal{G}[\mathcal{L}_1, \mathcal{L}_0] = \langle [\mathcal{L}_1, \mathcal{L}_0], [\mathcal{L}_0, \mathcal{L}_1] \rangle$. Also, if $\mu_\mathcal{L}(o_{\mathcal{L}_1}) = \varepsilon u' :: S_1'$, as everything is static, $ilbl(\varepsilon)$ have to have the form $\langle [\ell_u, \ell_u], \text{label}(S_1'), \text{label}(S_1') \rangle$, for some $\ell_u$. Then we have to prove that $\ell_c \preceq \text{label}(S_1')$, but notice that as everything is static, $\varepsilon \ell \preceq \ell \preceq \text{label}(S_1')$ is equivalent to $\varepsilon \ell \vdash \ell \preceq \text{label}(S_1)$, therefore we know that $\ell_c \preceq \text{label}(S_1)$ and the result holds.

\[ \square \]

### B.5.5 Dynamic Gradual Guarantee

In this section we present the proof the Dynamic Gradual Guarantee for GSLRef without the specific check in rule (r7).

**Definition 104** (Intrinsic term precision). Let $\Omega \in \mathcal{P}(\mathcal{VAR}_* \times \mathcal{VAR}_*) \cup \mathcal{P}(\mathcal{LOC}_* \times \mathcal{LOC}_*)$ be defined as $\Omega := \{ \pi^{U_1} \subseteq \pi^{U_2}, \rho^{U_1} \subseteq \rho^{U_2} \}$. We define an ordering relation $(\cdot \vdash \cdot \subseteq \cdot) \in (\mathcal{P}(\mathcal{VAR}_* \times \mathcal{VAR}_*) \cup \mathcal{P}(\mathcal{LOC}_* \times \mathcal{LOC}_*)) \times \mathcal{TERM}_* \times \mathcal{TERM}_*$ shown in Figure C.10.

**Definition 105** (Well Formedness of $\Omega$). We say that $\Omega$ is well formed iff $\forall \{ U_1 \subseteq U_2 \} \in \Omega. U_{11} \subseteq U_{12}$

Before proving the gradual guarantee, we first establish some auxiliary properties of precision. For the following propositions, we assume Well Formedness of $\Omega$ (Definition 115).

**Proposition 184.** If $\Omega \vdash t^{U_1} \subseteq t^{U_2}$ for some $\Omega \in \mathcal{P}(\mathcal{VAR}_* \times \mathcal{VAR}_*) \cup \mathcal{P}(\mathcal{LOC}_* \times \mathcal{LOC}_*)$, then $U_1 \subseteq U_2$.

**Proof.** Straightforward induction on $\Omega \vdash t^{U_1} \subseteq t^{U_2}$, since the corresponding precision on types is systematically a premise (either directly or transitively).

**Proposition 185.** Let $g_1, g_2 \in \text{EvFrame}$ such that $\phi_1 \triangleright g_1[\varepsilon_{11} t^{U_1}] \in \text{TERM}_{U_1}$, $\phi_2 \triangleright g_2[\varepsilon_{21} t^{U_2}] \in \text{TERM}_{U_2}$, with $U_1' \subseteq U_2'$. Then if $g_1[\varepsilon_{11} t^{U_1}] \subseteq g_2[\varepsilon_{21} t^{U_2}]$, $\varepsilon_{12} \subseteq \varepsilon_{22}$ and $t^{U_1} \subseteq t^{U_2}$, then $g_1[\varepsilon_{12} t^{U_1}] \subseteq g_2[\varepsilon_{22} t^{U_2}]$

**Proof.** We proceed by case analysis on $g_i$.

**Case** ($\square \triangleright \mathcal{U}_i \ et$). Then for $i \in \{1, 2\}$, $g_i$ must have the form $\square \triangleright \mathcal{U}_i \varepsilon_i t^{U_i}$ for some $U_i''$, $\varepsilon_i$ and $t^{U_i}$. As $g_1[\varepsilon_{11} t^{U_1}] \subseteq g_2[\varepsilon_{21} t^{U_2}]$ then by $\sqsubseteq \text{APP} \ v e \subseteq v e', v e' \subseteq v e_i, U_1'' \subseteq U_2''$ and $t^{U_1} \subseteq t^{U_2}$.

As $\varepsilon_{12} \subseteq \varepsilon_{22}$ and $t^{U_1} \subseteq t^{U_2}$, then by $\sqsubseteq \text{APP} \ v e_{12} t^{U_1} \triangleright \mathcal{U}_i \varepsilon_i t^{U_i} \subseteq v e_{22} t^{U_2} \triangleright \mathcal{U}_i \varepsilon_i t^{U_i}$, and the result holds.

**Case** ($\square \oplus \varepsilon \varepsilon, ev \oplus \mathcal{U}_i \mathcal{O}, ev \oplus \mathcal{U}_i \mathcal{O}, \square ::: U, U_i \mathcal{O}, \square :: U_i \mathcal{O}$, $\square :: \varepsilon_i \ et$, $ev := \varepsilon_i \ et$, if $\square$ then $\ et$ else $\ et$). Straightforward using similar argument to the previous case.
Figure B.19: Intrinsic term precision
Proposition 186. Let \( g_1, g_2 \in \text{EvFrame} \) such that \( \phi_1 \triangleright g_1[\varepsilon_1 t_{U_1}] \in \text{TERM}_{U_1}, \phi_2 \triangleright g_2[\varepsilon_2 t_{U_2}] \in \text{TERM}_{U_2} \), with \( U_1' \subseteq U_2' \). Then if \( g_1[\varepsilon_1 t_{U_1}] \subseteq g_2[\varepsilon_2 t_{U_2}] \) then \( t_{U_1} \subseteq t_{U_2} \) and \( \varepsilon_1 \subseteq \varepsilon_2 \).

Proof. We proceed by case analysis on \( g_1 \).

Case \((\Box \oplus^U \text{et})\). Then there must exist some \( \varepsilon_{\ell_1}, U_1, \varepsilon_1 \) and \( t'_{U_1} \) such that \( g[\varepsilon_1 t_{U_1}] = \varepsilon_1 t'_{U_1} \ominus^U \varepsilon_{\ell_1} \) and \( g[\varepsilon_2 t_{U_2}] = \varepsilon_2 t'_{U_2} \ominus^U \varepsilon_{\ell_2} \). Then by the hypothesis and the premises of \((\Box_{APP})\), \( t_{U_1} \subseteq t_{U_2} \) and \( \varepsilon_1 \subseteq \varepsilon_2 \), and the result holds immediately.

Case \((\Box \oplus^g \text{et}, ev \ominus^g \Box, ev \ominus^\ell U, \Box : U, U \Box, \Box : \varepsilon_{\ell_1} \text{ et}, ev : = \varepsilon_{\ell_1} \), if \( \Box \) then \( \text{et else et} \)). Straightforward using similar argument to the previous case.

\[\square\]

Proposition 187. Let \( f_1, f_2 \in \text{EvFrame} \) such that \( \phi_1 \triangleright f_1[t_{U_1}] \in \text{TERM}_{U_1}, \phi_2 \triangleright f_2[t_{U_2}] \in \text{TERM}_{U_2} \), with \( U_1' \subseteq U_2' \). Then if \( f_1[t_{U_1}] \subseteq f_2[t_{U_2}] \) and \( t_{U_1} \subseteq t_{U_2} \), then \( f_1[t_{U_1}] \subseteq f_2[t_{U_2}] \).

Proof. Suppose \( f_1[t_{U_1}] = g_1[\varepsilon_1 t_{U_1}] \). We know that \( \phi_1 \triangleright g_1[\varepsilon_1 t_{U_1}] \in \text{TERM}_{U_1}, \phi_2 \triangleright g_2[\varepsilon_2 t_{U_2}] \in \text{TERM}_{U_2} \) and \( U_1' \subseteq U_2' \). Therefore if \( g_1[\varepsilon_1 t_{U_1}] \subseteq g_1[\varepsilon_1 t_{U_2}] \), by Prop \[249\] \( \varepsilon_1 \subseteq \varepsilon_2 \). Finally by Prop \[48\] we conclude that \( g_1[\varepsilon_1 t_{U_1}] \subseteq g_1[\varepsilon_1 t_{U_2}] \).

\[\square\]

Proposition 188. Let \( f_1, f_2 \in \text{EvFrame} \) such that \( \phi_1 \triangleright f_1[t_{U_1}] \in \text{TERM}_{U_1}, \phi_2 \triangleright f_2[t_{U_2}] \in \text{TERM}_{U_2} \), with \( U_1' \subseteq U_2' \). Then if \( f_1[t_{U_1}] \subseteq f_2[t_{U_2}] \) then \( f_1[t_{U_1}] \subseteq f_2[t_{U_2}] \).

Proof. Suppose \( f_1[t_{U_1}] = g_1[\varepsilon_1 t_{U_1}] \). We know that \( \phi_1 \triangleright g_1[\varepsilon_1 t_{U_1}] \in \text{TERM}_{U_1}, \phi_2 \triangleright g_2[\varepsilon_2 t_{U_2}] \in \text{TERM}_{U_2} \) and \( U_1' \subseteq U_2' \). Therefore if \( g_1[\varepsilon_1 t_{U_1}] \subseteq g_1[\varepsilon_2 t_{U_2}] \), then using Prop \[249\] we conclude that \( t_{U_1} \subseteq t_{U_2} \).

\[\square\]

Proposition 189 (Substitution preserves precision). If \( \Omega \cup \{x^{U_3} \subseteq x^{U_4}\} \vdash t_{U_1} \subseteq t_{U_2} \) and \( \Omega \vdash t_{U_3} \subseteq t_{U_4} \), then \( \Omega \vdash [t_{U_1}/x^{U_3}]t_{U_1} \subseteq [t_{U_4}/x^{U_3}]t_{U_2} \).

Proof. By induction on the derivation of \( t_{U_1} \subseteq t_{U_2} \), and case analysis of the last rule used in the derivation. All cases follow either trivially (no premises) or by the induction hypotheses.

\[\square\]

Proposition 190 (Monotone precision for \( \circ^{<} \)). If \( \varepsilon_1 \subseteq \varepsilon_2 \) and \( \varepsilon_3 \subseteq \varepsilon_4 \) then \( \varepsilon_1 \circ^{<} \varepsilon_3 \subseteq \varepsilon_2 \circ^{<} \varepsilon_4 \).

Proof. By definition of consistent transitivity for \( < \); and the definition of precision.

\[\square\]

Proposition 191 (Monotone precision for \( \circ^{\leq} \)). If \( \varepsilon_1 \subseteq \varepsilon_2 \) and \( \varepsilon_3 \subseteq \varepsilon_4 \) then \( \varepsilon_1 \circ^{\leq} \varepsilon_3 \subseteq \varepsilon_2 \circ^{\leq} \varepsilon_4 \).
Proof. By definition of consistent transitivity for \( \preceq \) and the definition of precision.

Proposition 192 (Monotone precision for join). If \( \varepsilon_1 \subseteq \varepsilon_2 \) and \( \varepsilon_3 \subseteq \varepsilon_4 \) then \( \varepsilon_1 \sqsubseteq \varepsilon_3 \subseteq \varepsilon_2 \sqsupseteq \varepsilon_4 \).

Proof. By definition of join and the definition of precision.

Proposition 193. If \( \text{Ref} \ U_1 \subseteq \text{Ref} \ U_2 \) then \( U_1 \subseteq U_2 \).

Proof. By definition of precision we know that 
\[ \{ \text{Ref} \ T \mid T \in \gamma(U_1) \} \subseteq \{ \text{Ref} \ T \mid T \in \gamma(U_2) \} \] 
This relation is true only if \( \gamma(U_1) \subseteq \gamma(U_2) \) which is equivalent to \( U_1 \subseteq U_2 \).

Proposition 194. If \( U_{11} \subseteq U_{12} \) and \( U_{21} \subseteq U_{22} \) then \( U_{11} \sqcup U_{21} \subseteq U_{12} \sqcup U_{22} \).

Proof. By induction on the type derivation of the types and consistent join.

Lemma 195. If \( \varepsilon_1 \vdash \text{Ref}_{g_1} \ U_{11} \sqsubseteq \text{Ref}_{g_2} \ U_{12} \) and \( \varepsilon_2 \vdash \text{Ref}_{g_1} \ U_{21} \sqsubseteq \text{Ref}_{g_2} \ U_{22} \), and \( \varepsilon_1 \sqsubseteq \varepsilon_2 \), then \( \text{iref}(\varepsilon_1) \subseteq \text{iref}(\varepsilon_2) \).

Proof. By definition of precision and \( \text{iref} \).

Proposition 196 (Dynamic guarantee for \( \rightarrow \)). Suppose \( \Omega \vdash t^{U_1}_{1} \sqsubseteq t^{U_2}_{1} \), \( \phi_1 \subseteq \phi_2 \), and \( \Omega \vdash \mu_1 \subseteq \mu_2 \). If \( t^{U_1}_{1} \mid \mu_1 \xrightarrow{\phi_1} t^{U_1}_{2} \mid \mu_1' \) then \( t^{U_2}_{1} \mid \mu_2 \xrightarrow{\phi_2} t^{U_2}_{2} \mid \mu_2' \) where \( \Omega' \vdash t^{U'_1}_{2} \sqsubseteq t^{U'_2}_{2} \) and \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \), for some \( \Omega' \geq \Omega \).

Proof. By induction on the structure of \( t^{U_1}_{1} \sqsubseteq t^{U_2}_{1} \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

Case (\( \rightarrow \)). We know that \( t^{U_1}_{1} = (\varepsilon_{11}(b_1)_{g_1} \oplus \varepsilon_{12}(b_2)_{g_2}) \) then by (\( \sqsubseteq \)) \( t^{U_2}_{1} = (\varepsilon_{21}(b_1)_{g_21} \oplus \varepsilon_{22}(b_2)_{g_22}) \) for some \( \varepsilon_{21} \sqsubseteq \varepsilon_{22} \), \( g_{21} \sqsubseteq g_{22} \), \( g_{11} \subseteq g_{12} \). If \( t^{U_1}_{1} \mid \mu_1 \xrightarrow{\phi_1} b_3 \mid \mu_1 \) where \( b_3 = (\varepsilon_{11} \sqsupseteq \varepsilon_{12})(b_1 \bigoplus b_2)_{(g_{21})_{(g_{21} \sqsupseteq g_{22})}} \). By Lemma 192 (\( \varepsilon_{11} \sqsupseteq \varepsilon_{12} \sqsubseteq (\varepsilon_{21} \sqsupseteq \varepsilon_{22}) \)). Also \( (g_{11} \sqsupseteq g_{21}) \sqsubseteq (g_{12} \sqsupseteq g_{22}) \). Then \( t^{U_2}_{1} \mid \mu_2 \xrightarrow{\phi_2} b_3' \mid \mu_2 \) where \( b_3' = (\varepsilon_{21} \sqsupseteq \varepsilon_{22})(b_1 \bigoplus b_2)_{(g_{21} \sqsupseteq g_{22})} \). By Lemma 192 (\( \varepsilon_{21} \sqsupseteq \varepsilon_{22} \sqsubseteq (\varepsilon_{21} \sqsupseteq \varepsilon_{22}) \)). Also \( (g_{21} \sqsupseteq g_{22}) \sqsubseteq (g_{21} \sqsupseteq g_{22}) \).

Therefore \( t^{U_1}_{2} \sqsubseteq t^{U_2}_{2} \). As \( \Omega' = \Omega, \mu'_1 = \mu_1 \) and \( \mu_2 = \mu'_2 \) then \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \).
Case ($\rightarrow$prot). We know that $t_1^{U_1} = \text{prot}_{\varepsilon_1}^{\mu_1} \phi'_1(\varepsilon_1 u_1)$, then by (\textsf{\textit{app}}) $t_1^{U_2} = \text{prot}_{\varepsilon_2}^{\mu_2} \phi'_2(\varepsilon_2 u_2)$, and therefore

\[
t_1^{U_1} \mid \mu_1 \xrightarrow{\phi_1'} (\varepsilon_1 \tilde{\gamma}(\varepsilon_1)) (u_1 \tilde{\gamma} g'_1) : U_1 \tilde{\gamma} g_1 \mid \mu_1.
\]

Therefore, $t_1^{U_2} \mid \mu_2 \xrightarrow{\phi_2'} (\varepsilon_2 \tilde{\gamma}(\varepsilon_2)) (u_2 \tilde{\gamma} g'_2) : U_2 \tilde{\gamma} g_2 \mid \mu_2$. By Lemma 192, $(\varepsilon_1 \bar{\varepsilon}(\varepsilon_1)) \subseteq (\varepsilon_2 \bar{\varepsilon}(\varepsilon_2))$, and as join is monotone $U_1 \tilde{\gamma} g_1 \subseteq U_2 \tilde{\gamma} g_2$ and $(u_1 \tilde{\gamma} g'_1) \subseteq (u_2 \tilde{\gamma} g'_2)$. Therefore by \textsf{\textit{inc}}\hspace{1pt}, $(\varepsilon_1 \bar{\varepsilon}(\varepsilon_1)) (u_1 \tilde{\gamma} g'_1) : U_1 \tilde{\gamma} g_1 \subseteq (\varepsilon_2 \bar{\varepsilon}(\varepsilon_2)) (u_2 \tilde{\gamma} g'_2) : U_2 \tilde{\gamma} g_2$.

As $\Omega' = \Omega$, $\mu'_1 = \mu_1$ and $\mu_2 = \mu'_2$ then $\Omega' \vdash \mu'_1 \subseteq \mu'_2$.

Case ($\rightarrow$inc). We know that

\[
t_1^{U_1} = \varepsilon_1(\lambda x^U_{x_1}. t^{U_1}_{x_1}) \mid \mu_1 \mid U_1 \subseteq \varepsilon_1 u_1 \quad \text{then by (\textsf{\textit{inc}})} \quad t_1^{U_2} \text{ must have the form}
\]

\[
t_1^{U_2} = \varepsilon_2(\lambda x^U_{x_2}. t^{U_2}_{x_2}) g'_2 \mid \mu_2 \mid U_2 \subseteq \varepsilon_2 u_2 \quad \text{for some } \varepsilon_1, \varepsilon_2, U_1, U_2 \quad \text{and } \mu_1, \mu_2 \text{ as well, then by Prop 253 } \varepsilon_1 \subseteq \varepsilon_2.
\]

We also know by (\textsf{\textit{app}}) and (\textsf{\textit{inc}}) that $\Omega \cup \{x^U_{x_1} \subseteq \varepsilon_1 \} \vdash t^{U_2} \subseteq t^{U_2}$. By Substitution preserves precision (Prop 252) $t_1 \subseteq t_2$, therefore $\text{inc}(\varepsilon_1) t_1 \subseteq t_2 \subseteq \text{inc}(\varepsilon_2) t_1$. Also, as $\phi_1 \subseteq \phi_2$, $\text{inc}(\varepsilon_1) \subseteq \text{inc}(\varepsilon_2)$, then by (\textsf{\textit{inc}}) $\varepsilon_1 \subseteq \varepsilon_2$. Also, as $\phi_1 \subseteq \phi_2$, by monotonicity of the join $\text{join} \tilde{\gamma} \phi_1 . g \subseteq \text{join} \tilde{\gamma} \phi_2 . g$, and as $\phi_1 . g \subseteq \phi_2 . g$ also by monotonicity of the join $\phi_1 \tilde{\gamma} \phi_1 . g \subseteq \phi_2 \tilde{\gamma} \phi_2 . g$.

Case ($\rightarrow$if-true). $t_1^{U_1} = \text{inc}(\varepsilon_1) \text{true}_{\varepsilon_1}$, then else $\varepsilon_1 \theta^{U_{22} \varepsilon_{13} t^{U_{13}}}$ then by (\textsf{\textit{if}}) $t_1^{U_2}$ has the form $t_1^{U_2} = \text{inc}(\varepsilon_2) \text{true}_{\varepsilon_2}$, then else $\varepsilon_2 \theta^{U_{22} \varepsilon_{23} t^{U_{23}}}$ for some $\varepsilon_2, \varepsilon_{22}, t^{U_{22}}, t^{U_{23}}$, and $U^{U_{22}}, t^{U_{23}}$. Then

\[
t_1^{U_1} \mid \mu_1 \xrightarrow{\phi_1'} \text{inc}(\varepsilon_1 \theta^{U_{12}}) \phi_1' \mid \mu_1 \text{ and }
\]

\[
t_1^{U_2} \mid \mu_2 \xrightarrow{\phi_2'} \text{inc}(\varepsilon_2 \theta^{U_{23}}) \phi_2' \mid \mu_2.
\]

Where $\phi'_1 = (\text{join} \tilde{\gamma} \phi_1 . g \subseteq (\text{join} \tilde{\gamma} \phi_1 . g \subseteq \text{join} \tilde{\gamma} \phi_1 . g)$. Using the fact that $t_1^{U_1} \subseteq t_1^{U_2}$ we know that $\varepsilon_{12} \subseteq \varepsilon_{22}, t^{U_{12}} \subseteq t^{U_{22}}, g'_1 \subseteq g'_2$, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. As $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 247 we know that $U_{12} \subseteq U_{22}$ and $U_{13} \subseteq U_{23}$.

Also, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 254 $U_{12} \tilde{\gamma} U_{13} \subseteq (U_{22} \tilde{\gamma} U_{23})$. Also, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 192 $U_{12} \tilde{\gamma} U_{13} \subseteq (U_{22} \tilde{\gamma} U_{23})$. Also, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 247 we know that $U_{12} \subseteq U_{22}$ and $U_{13} \subseteq U_{23}$.

Therefore by Prop 254 $U_{12} \tilde{\gamma} U_{13} \subseteq (U_{22} \tilde{\gamma} U_{23})$. Also, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 192 $U_{12} \tilde{\gamma} U_{13} \subseteq (U_{22} \tilde{\gamma} U_{23})$. Also, as $\phi_1 . g \subseteq \phi_2 . g$ and $g_1 \subseteq g_2$, and as join is monotone, $\phi_1 . g \subseteq \phi_2 . g$ and $g'_1 \subseteq g'_2$. By Prop 247 we know that $U_{12} \subseteq U_{22}$ and $U_{13} \subseteq U_{23}$.
Case (→-if-false). Same as case →-if-true, using the fact that $\varepsilon_{13} \sqsubseteq \varepsilon_{23}$ and $t^{u_{13}} \sqsubseteq t^{u_{23}}$.

Case (→-ref). We know that $t'_{1u} = \text{ref}^{u''}_{\varepsilon_{11}} \varepsilon_{1u1}$, then by ($\sqsubseteq_{\text{ref}}$) $t'_{2u} = \text{ref}^{u''}_{\varepsilon_{22}} \varepsilon_{2u2}$, and therefore

\[
\begin{align*}
U''_1 & \sqsubseteq U''_2 & \varepsilon_{11} & \sqsubseteq \varepsilon_{12} & g_{11} & \sqsubseteq g_{12} & \Omega \vdash u_1 \sqsubseteq u_2 \\
\Omega \vdash \text{ref}^{u''}_{\varepsilon_{11}} \varepsilon_{11} u_1 & \sqsubseteq \text{ref}^{u''}_{\varepsilon_{22}} \varepsilon_{22} u_2
\end{align*}
\]

for some $\varepsilon_{1}, u_{2}, U''_{1}$ and $\text{ref}_{2}$, where $u_{1} \in \text{TERM}_{U'_{1}}$ and $u_{2} \in \text{TERM}_{U'_{2}}$. If

Then $t'_{1u} \mid \mu_1 \xrightarrow{\phi_1} \phi_1^{U''_{1}} \mid \mu_1[t^{u''_{1}} \mapsto v'_{1}]$, for some $U''_{1} \not\sqsubseteq \mu_1$ and where $v'_{1} = \varepsilon'_{1}(u_1 \tilde{\gamma} g_{11}) :: U''_{1}$, $\varepsilon'_{1} = \varepsilon_{1} \sqrt{\gamma} (\phi_{1} \circ \varepsilon \varepsilon_{1})$. Therefore, $t'_{2u} \mid \mu_2 \xrightarrow{\phi_2} \phi_2^{U''_{2}} \mid \mu_2[t^{u''_{2}} \mapsto v'_{2}]$, for some $U''_{2} \not\sqsubseteq \mu_2$ and where $v'_{2} = \varepsilon'_{2}(u_2 \tilde{\gamma} g_{12}) :: U''_{2}$, $\varepsilon'_{2} = \varepsilon_{2} \sqrt{\gamma} (\phi_{2} \cdot \varepsilon \circ \varepsilon_{2})$. By Lemma 192 and 253, $\varepsilon'_{1} \sqsubseteq \varepsilon'_{2}$, Also as $\phi_{1} \circ \varepsilon \varepsilon_{1} \subseteq \phi_{2} \circ \varepsilon \varepsilon_{2}$ and $U_1 \subseteq U_2$, then by definition of $\text{ref}$, $\varepsilon'_{1} \subseteq \varepsilon'_{2}$. Then using $\Omega' = \Omega \cup \{ U''_{1} \subseteq U''_{2} \}$ and that $\perp \subseteq \perp$, by ($\sqsubseteq_{\text{ref}}$) we can see that $\Omega' \vdash t^{u_{11}} \sqsubseteq t^{u_{21}}$. As $g_{11} \sqsubseteq g_{12}$, by monotonicity of the join, $u_1 \tilde{\gamma} g_{11} \sqsubseteq u_2 \tilde{\gamma} g_{12}$. Therefore using $\sqsubseteq_{\text{ref}}$, $\Omega' \vdash v'_{1} \sqsubseteq v'_{2}$. Also because $\Omega \sqsubseteq \Omega'$, then by the fact that $\Omega \vdash \mu_1 \sqsubseteq \mu_2$, it is easy to see that $\Omega \cup \{ t^{u_{11}} \sqsubseteq t^{u_{21}} \} \vdash \mu_1[t^{u''_{1}} \mapsto v'_{1}] \sqsubseteq \mu_2[t^{u''_{2}} \mapsto v'_{2}]$, i.e. $\Omega' \vdash \mu'_1 \sqsubseteq \mu'_2$.

Case (→-deref). We know that $t'_{1u} = t^{\text{ref}_{1}} \varepsilon_{1} t^{u''_{1}} g_{11}^{'}, t'_{2u} = t^{\text{ref}_{2}} \varepsilon_{2} t^{u''_{2}} g_{12}^{'}, \text{ and so } \Omega \vdash \mu_1 \sqsubseteq \mu_2$, using ($\sqsubseteq_{\mu}$) then $\Omega \vdash \mu_1(t^{u''_{1}}) \sqsubseteq \mu_2(t^{u''_{2}})$.

Then $\text{ref}_{1} t^{u''_{1}} \sqsubseteq t^{\text{ref}_{1}} \varepsilon_{1} t^{u''_{1}} g_{11}^{'}, \mu_1 \xrightarrow{\phi_1} \text{ref}_{1}(\text{ref}(\varepsilon_{1}) \mu_1(\text{ref}(\varepsilon_{2}))) \text{ where } \varepsilon_{1} = \varepsilon_{1} \text{ is } \text{dilbl}(\varepsilon_{1})$. Therefore $\text{ref}_{2} t^{u''_{2}} \sqsubseteq t^{\text{ref}_{2}} \varepsilon_{2} t^{u''_{2}} g_{12}^{'}, \mu_2 \xrightarrow{\phi_2} \text{ref}_{2}(\text{ref}(\varepsilon_{2}) \mu_2(\text{ref}(\varepsilon_{2}))) \text{ where } \varepsilon_{2} = \varepsilon_{2} \text{ is } \varepsilon_{2} \text{ is } \varepsilon_{2} \text{.}$

Where $\text{ref}_{1} = \langle (\phi_{1} \circ \varepsilon \sqrt{\gamma} \varepsilon_{1}), (\phi_{1} \cdot \varepsilon \varepsilon_{1}) \rangle, (\phi_{1} \cdot \varepsilon \sqrt{\gamma} \varepsilon_{1}) \rangle$. By monotonicity of the join $\phi_{1} \cdot \varepsilon \sqrt{\gamma} \varepsilon_{1} \sqsubseteq \phi_{2} \cdot \varepsilon \sqrt{\gamma} \varepsilon_{2}$, $\phi_{1} \cdot \varepsilon \sqrt{\gamma} \varepsilon_{1} \sqsubseteq \phi_{2} \cdot \varepsilon \square(\varepsilon_{2})$ and $(\phi_{1} \circ \varepsilon \varepsilon_{1}) \sqsubseteq (\phi_{2} \circ \varepsilon \varepsilon_{2})$. As $\varepsilon_{1} \sqsubseteq \varepsilon_{2}$, then by Lemma 195 $\text{ref}(\varepsilon_{1}) \sqsubseteq \text{ref}(\varepsilon_{2})$. Then Using ($\sqsubseteq_{\text{dilbl}}$) we can conclude that $\Omega \vdash v'_{1} \sqsubseteq v'_{2}$. As $\Omega' = \Omega$, $\mu_1 = \mu'_1$ and $\mu_2 = \mu'_2$, then also $\Omega' \vdash \mu'_1 \sqsubseteq \mu'_2$.

Case (→-assign). We know that $t'_{1u} = t^{u_{11}} g_{11}^{'}, t^{u_{21}} g_{21}^{'}, t^{u_{22}} g_{22}^{'}, t^{u_{22}} g_{22}^{''}, \text{ and so } \Omega \vdash \mu_1 \sqsubseteq \mu_2$, using ($\sqsubseteq_{\mu}$) then $\Omega \vdash \mu_1(t^{u_{11}}) \sqsubseteq \mu_2(t^{u_{21}})$.

Then $t^{u_{11}} \mid \mu_1 \xrightarrow{\phi_1} \text{unit}_{1} \mid \mu_1[t^{u_{11}} \mapsto v_{1}]$, where $v_{1} = \varepsilon_{1}(u_1 \tilde{\gamma} (g_{11} \tilde{\gamma} g_{11})) :: U_{11}$, and $\varepsilon_{1} = \varepsilon_{12} \circ \text{ref}(\varepsilon_{11}) \sqrt{\gamma} (((\phi_{1} \circ \varepsilon \sqrt{\gamma} \varepsilon_{1})) \varepsilon_{12} \circ \varepsilon_{11} \text{ is } \text{dilbl}(\varepsilon_{11}))$. Similarly, then $t^{u_{21}} \mid \mu_2 \xrightarrow{\phi_2} \text{unit}_{2} \mid \mu_2[t^{u_{21}} \mapsto v_{2}]$, where $v_{2} = \varepsilon_{2}(u_2 \tilde{\gamma} (g_{21} \tilde{\gamma} g_{21})) :: U_{21}$, and $\varepsilon_{2} = \varepsilon_{22} \circ \text{ref}(\varepsilon_{21}) \sqrt{\gamma} (((\phi_{2} \cdot \varepsilon \varepsilon_{2}) \circ \varepsilon_{22} \circ \varepsilon_{21} \text{ is } \text{dilbl}(\varepsilon_{21}))$. We need to prove that $\mu'_1 = \mu_1[t^{u_{11}} \mapsto v_{1}] \sqsubseteq \mu'_2 = \mu_2[t^{u_{21}} \mapsto v_{2}]$. Because $\Omega \vdash \mu_1 \sqsubseteq \mu_2$ then $\Omega \vdash t^{u_{11}} \sqsubseteq t^{u_{21}}$ by ($\sqsubseteq_{\mu}$). By well formedness of $\Omega$ we also know that $U_{11} \sqsubseteq U_{21}$. Therefore, by Lemmas 253, 191 and 192, $\varepsilon_{1}' \sqsubseteq \varepsilon_{2}'$. Then using $\sqsubseteq_{\square}$, $v_{1} \sqsubseteq v_{2}$, following that $\Omega' = \Omega \vdash \mu'_1 \sqsubseteq \mu'_2$.

\[\Box\]

**Proposition 197** (Dynamic guarantee). Suppose $t^{u_{1}} \sqsubseteq t^{u_{2}}, \phi_{1} \sqsubseteq \phi_{2}$, and $\mu_1 \sqsubseteq \mu_2$. If $t^{u_{1}} \mid \mu_1 \xrightarrow{\phi_1} t^{u_{2}} \mid \mu_1$ then $t^{u_{1}} \mid \mu_2 \xrightarrow{\phi_2} t^{u_{2}} \mid \mu_2$ where $t^{u_{1}} \sqsubseteq t^{u_{2}}$ and $\mu_1 \sqsubseteq \mu_2$.

**Proof.** We prove the following property instead: Suppose $\Omega \vdash t_{1u} \sqsubseteq t_{2u}, \phi_{1} \sqsubseteq \phi_{2}$, and
\( \Omega \vdash \mu_1 \subseteq \mu_2 \). If \( t_1^{U_1} \mid \mu_1 \xrightarrow{\phi_1} t_2^{U_1} \mid \mu'_1 \) then \( t_1^{U_2} \mid \mu_2 \xrightarrow{\phi_2} t_2^{U_2} \mid \mu'_2 \) where \( \Omega' \vdash t_2^{U_1} \subseteq t_2^{U_2} \) and \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \), for some \( \Omega' \supseteq \Omega \).

By induction on the structure of a derivation of \( t_1^{U_1} \subseteq t_2^{U_2} \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case** (R \( \rightarrow \)). \( \Omega \vdash t_1^{U_1} \subseteq t_1^{U_2}, \Omega \vdash \mu_1 \subseteq \mu_2 \) and
\( t_1^{U_1} \mid \mu_1 \xrightarrow{\phi_1} t_2^{U_1} \mid \mu'_1 \). By dynamic guarantee of \( \rightarrow \) (Prop 255), \( t_1^{U_2} \mid \mu_2 \xrightarrow{\phi_2} t_2^{U_2} \mid \mu'_2 \) where \( \Omega' \vdash t_2^{U_1} \subseteq t_2^{U_2} \), \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \) for some \( \Omega' \supseteq \Omega \). And the result holds immediately.

**Case** (Rf). \( t_1^{U_1} = f_1[t_1^{U_1}], t_2^{U_1} = f_2[t_2^{U_1}] \). We know that \( \Omega \vdash f_1[t_1^{U_1}] \subseteq f_2[t_2^{U_1}] \). By using Prop 247, \( U_1' \subseteq U_2' \). By Prop 251 we also know that \( \Omega \vdash t_1^{U_1'} \subseteq t_1^{U_2'} \). By induction hypothesis, \( t_1^{U_1'} \mid \mu_1 \xrightarrow{\phi_1} t_2^{U_1'} \mid \mu'_1, t_1^{U_2'} \mid \mu_2 \xrightarrow{\phi_2} t_2^{U_2'} \mid \mu'_2 \), \( \Omega' \vdash t_2^{U_1'} \subseteq t_2^{U_2'} \) and \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \) for some \( \Omega' \supseteq \Omega \).
Then by Prop 250 then \( \Omega' \vdash f_1[t_2^{U_1'}] \subseteq f_2[t_2^{U_2'}] \) and the result holds.

**Case** (Rprot). Then \( t_1^{U_1} = \text{prot}_{\varepsilon_1}^{g_1,t_{1}^{U_1}}(\varepsilon_1 t_1^{U_1}) \) and \( t_2^{U_2} = \text{prot}_{\varepsilon_2}^{g_2,t_{2}^{U_2}}(\varepsilon_2 t_2^{U_2}) \)

As \( t_1^{U_1} \subseteq t_2^{U_2} \) then by (\( \subseteq_{\text{prot}(t)} \)), \( t_1^{U_1'} \subseteq t_2^{U_1'}, \phi_1 \subseteq \phi_2, \varepsilon_1 \subseteq \varepsilon_2, g_1 \subseteq g_2, g'_1 \subseteq g'_2, \) and \( \varepsilon_1 \subseteq \varepsilon_2 \). By (Rprot), \( t_1^{U_1'} \mid \mu \xrightarrow{\phi'_1} t_2^{U_1'} \mid \mu' \) and by induction hypothesis, \( t_2^{U_1'} \subseteq t_2^{U_2'} \) and \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \) for some \( \Omega' \supseteq \Omega \).

But then by (\( \subseteq_{\text{prot}(t)} \)), \( \Omega' \vdash \text{prot}_{\varepsilon_1}^{g_1,t_{1}^{U_1}}(\varepsilon_1 t_{1}^{U_1}) \subseteq \text{prot}_{\varepsilon_2}^{g_2,t_{2}^{U_2}}(\varepsilon_2 t_{2}^{U_2}) \) and the result holds.

**Case** (Rg). \( t_1^{U_1} = g_1[et_1], t_1^{U_2} = g_2[et_2], \) where \( \Omega \vdash g_1[et_1] \subseteq g_2[et_2] \). Also \( et_1 \rightarrow_c et' \) and \( et_2 \rightarrow_c et'_2 \).
Then there exists \( U_1, \varepsilon_{11}, \varepsilon_{12} \) and \( v_1 \) such that \( et_1 = \varepsilon_{11}(\varepsilon_{12}v_1 :: U_1) \). Also there exists \( U_2, \varepsilon_{21}, \varepsilon_{22} \) and \( v_2 \) such that \( et_2 = \varepsilon_{21}(\varepsilon_{22}v_2 :: U_2) \). By Prop 249 \( \varepsilon_{11} \subseteq \varepsilon_{21} \), and by (\( \subseteq \)) \( \varepsilon_{12} \subseteq \varepsilon_{22} \), \( v_1 \subseteq v_2 \) and \( U_1 \subseteq U_2 \). Then as \( et_1 \rightarrow_c (\varepsilon_{12} \circ \varepsilon_{11})v_1 \) and \( et_2 \rightarrow_c (\varepsilon_{22} \circ \varepsilon_{21})v_2 \) then, by Prop 253 we know that \( \varepsilon_{12} \circ \varepsilon_{11} \subseteq \varepsilon_{22} \circ \varepsilon_{21} \). Then using this information, and the fact that \( v_1 \subseteq v_2 \), by Prop 248 it follows that \( \Omega \vdash g_1[et'_1] \subseteq g_1[et'_2] \). As \( \Omega' = \Omega, \mu'_1 = \mu_1 \) and \( \mu_2 = \mu'_2 \) then \( \Omega' \vdash \mu'_1 \subseteq \mu'_2 \).

**Case** (Rprogt). Analogous to (Rprot) case but using \( \rightarrow_c \) instead.
B.5.6 Noninterference

In this section we present the proof of noninterference for GSL_{Ref}. We use a logical relation that is more general than the one presented in § 4.5. The main difference (beside using intrinsic terms), is that the logical relation is no longer indexed by a static security effect. As $\phi$ embeds the static security effect information, we generalize the logical relation to also relate two different static security effects as well. § B.5.6 present some auxiliary definitions. § B.5.6 presents the proof of Noninterference (Prop 227), which implies Security Type Soundness (Prop 25) presented in § 4.5.

Definitions

We introduce a function $uval$, which strips away ascriptions from a simple value:

$$uval : \text{GType} \rightarrow \text{SimpleValue}$$

$$uval(u) = u$$

$$uval(\varepsilon u :: U) = u.$$ 

In order to compare the observable results of program, we introduce the $rval(v)$ operator, which strips away any checking-related information like labels or evidence-carrying ascriptions:

$$rval : \text{Value} \rightarrow \text{RawValue}$$

$$rval(b_g) = b$$

$$rval(\varepsilon b_g :: U) = b$$

$$rval(\text{unit}_g) = \text{unit}$$

$$rval(\varepsilon \text{unit}_g :: U) = \text{unit}$$

$$rval(\varepsilon o_g^U) = o$$

$$rval(\varepsilon o_g^U') :: U = o$$

$$rval((\lambda x^{U_1}.t^{U_2})_g) = (\lambda x^{U_1}.t^{U_2})$$

$$rval(\varepsilon (\lambda x^{U_1}.t^{U_2})_g :: U) = (\lambda x^{U_1}.t^{U_2})$$

**Definition 106** (Gradual security logical relations). For an arbitrary element $ol$ of the security lattice, the $ol$-level gradual security relations are step-indexed and type-indexed binary relations on tuples of security effect, closed terms and stores defined inductively as presented in Figure B.20. The notation $\langle \phi_1, v_1, \mu_1 \rangle \approx_{ol}^k \langle \phi_2, v_2, \mu_2 \rangle : U$ indicates that the tuple of security effect $\phi_1$, value $v_1$ and store $\mu_1$ is related to the tuple of security effect $\phi_2$, value $v_2$ and store $\mu_2$ at type $U$ for $k$ steps when observed at the security level $ol$. Similarly, the notation $\langle \phi_{\approx_{ol}^k}, t_{\approx_{ol}^k}, \mu_{\approx_{ol}^k} \rangle \approx_{ol}^k (\phi_1, t, \mu) C(U)$ indicates that the tuple of security effect $\phi_1$, term $t_1$ and store $\mu_1$, and the tuple of security effect $\phi_2$, term $t_2$ and store $\mu_2$ are related computations for $k$ steps, that produce related values and related stores at type $U$ when observed at the security level $ol$. Notation $\mu_1 \approx_{ol}^k \mu_1$ relates stores $\mu_1$ and $\mu_2$ for $k$ steps when observed at security level $ol$. Finally, notation $\phi_1 \approx_{ol} \phi_2$, relates security effect $\phi_1$ and $\phi_2$ for any number of steps at security level $ol$. 

296
\[
\langle \phi_1, v_1, \mu_1 \rangle \asymp_{\text{ol}}^k \langle \phi_2, v_2, \mu_2 \rangle : U \iff \phi_1 \asymp_{\text{ol}} \phi_2 \land \mu_1 \asymp_{\text{ol}}^k \mu_2 \land \phi_1 \triangleright v_1 \in \text{TERM}_U \land \\
(\text{obs}_{\text{ol}}(\phi_1 \triangleright v_1) \lor \neg \text{obs}_{\text{ol}}(\phi_1 \triangleright v_1)) \land \\
(\text{obs}_{\text{ol}}(\phi_1 \triangleright v_1) \implies \text{obsRel}_{k,\text{ol}}^U(\phi_1, v_1, \mu_1, \phi_2, v_2, \mu_2))
\]

\[
\text{obsRel}_{k,\text{ol}}^U(\phi_1, v_1, \mu_1, \phi_2, v_2, \mu_2) \iff (\text{rval}(v_1) = \text{rval}(v_2)) \quad \text{if } U \in \{\text{Bool}, \text{Unit}_g, \text{Ref}_g U'\}
\]

\[
\text{obsRel}_{k,\text{ol}}^U(\phi_1, v_1, \mu_1, \phi_2, v_2, \mu_2) \iff \forall j < k. U' = U'' \overset{\varepsilon_1'}{\longrightarrow}_{g_1} U_2 \subseteq U', \text{ and } \varepsilon_2' \vdash U'' \subseteq \varepsilon''_1 \vdash g'_2 \gamma g'_2 \leq g''_2, \text{ we have:}
\forall v', \mu', \langle \phi_1, v_1, \mu'_1 \rangle \asymp_{\text{ol}}^j \langle \phi_2, v_2', \mu'_2 \rangle : U'_1, \text{dom}(\mu_1) \subseteq \text{dom}(\mu'_1), \\
\langle \phi_1, (\varepsilon_1' v_1 \overset{\text{ol}}{\circ} U' \varepsilon_2' v_1'), \mu'_1 \rangle \asymp_{\text{ol}}^j \langle \phi_2, (\varepsilon_1' v_2 \overset{\text{ol}}{\circ} U'' \varepsilon_2' v'_2), \mu'_2 \rangle : C(U'' \gamma g'_2)
\]

\[
\langle \phi_1, t_1, \mu_1 \rangle \asymp_{\text{ol}}^k \langle \phi_2, t_2, \mu_2 \rangle : C(U) \iff \phi_1 \asymp_{\text{ol}} \phi_2 \land \mu_1 \asymp_{\text{ol}}^k \mu_2 \land \forall \phi'_1, \text{ s.t. } \phi_1 \leq \phi'_1 \text{ and } \phi'_1 \triangleright t_1 \in \text{TERM}_U \text{ we have } \forall j < k
\]

\[
(t_1 | \mu_1 \overset{\phi'_1}{\longrightarrow} t'_1 | \mu'_1 \implies \mu'_1 \asymp_{\text{ol}}^{k-j} \mu_2') \land \\
(\text{irred}(t'_1) \implies \langle \phi_1, t'_1, \mu'_1 \rangle \asymp_{\text{ol}}^{k-j} \langle \phi_2, t'_2, \mu'_2 \rangle : U)
\]

\[
\mu_1 \asymp_{\text{ol}}^k \mu_2 \iff \forall \phi_1, \phi_2, j \leq k, \forall U \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)
\langle \phi_1 \triangleright \mu_1(\phi_U), \mu_1 \rangle \asymp_{\text{ol}}^j \langle \phi_2 \triangleright \mu_2(\phi_U), \mu_2 \rangle : U
\]

\[
\phi_1 \asymp_{\text{ol}} \phi_2 \iff \text{obs}_{\text{ol}}(\phi_1 \varepsilon \phi_1 \cdot g_c) \lor \neg \text{obs}_{\text{ol}}(\phi_1 \varepsilon \phi_1 \cdot g_c)
\]

\[
\phi_1 \leq_{\text{ol}} \phi_2 \iff \text{obs}_{\text{ol}}(\phi_2 \varepsilon \phi_2 \cdot g_c) \Rightarrow \text{obs}_{\text{ol}}(\phi_1 \varepsilon \phi_1 \cdot g_c)
\]

\[
\mu_1 \triangleright \mu_2 \iff \text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)
\]

\[
\text{obs}_{\text{ol}}(\phi \triangleright v) \iff \phi \triangleright v \in \text{TERM}_U \land \text{obs}_{\text{ol}}(\phi) \land \text{label}(U) \simeq \text{ol} \land \\
\left((v = \varepsilon u : U) \iff \text{obs}_{\text{ol}}(\text{lab}(\varepsilon) \text{label}(U))\right)
\]

\[
\text{obs}_{\text{ol}}(\phi) \iff \text{obs}_{\text{ol}}(\phi \varepsilon \phi \cdot g_c)
\]

\[
\text{obs}_{\text{ol}}(\varepsilon g) = \iff \varepsilon \circ \varepsilon \text{ is defined, where } \varepsilon' = \mathcal{G}_{\varepsilon'}(g, \text{ol})
\]

Figure B.20: Gradual security logical relations
We say that a value is *observable* at level $\ell$ if, given a security effect $\phi$, the value is typeable, the security effect is observable, and the label of the value is sublabel of $\ell$. Also, as value $v$ can be a casted value, we need to analyze if its underlying evidence justifies that the security level of the bare value is also subsumed by the observer security level. We do this by demanding that the underlying evidence and label is also observable. We say that a security effect is observable if its underlying evidence and static label is also observable. We say that an evidence and label are observable, if any value with that underlying evidence and static label, can be used used as argument of a function that expects a value with security level $\ell$. If the consistent transitivity check of the reduction of the application does not hold, then it is not plausible that the security level of the value is subsumed by $\ell$, and therefore is not observable. For instance, consider $\ell = L$, evidence $\varepsilon = \langle [H, \top], [\bot, \top] \rangle$ and static label $g = \_$. We can construct any value such as $v = \varepsilon \text{true} :: \text{Bool}_g$. The level of the value and the bare value are sublabel of $\ell$. But the evidence describes that at some point during reduction, the security level of the bare value was required to be at least as high as $H$. Therefore, $v$ is not observable at level $L$ (considering $L \preceq H$), because as $\delta_{\preceq}(?, \ell) = \langle [\bot, L], [L, L] \rangle$, the consistent transitivity operation $\langle [H, \top], [\bot, \top] \rangle \circ:\langle [\bot, L], [L, L] \rangle$ does not hold.

Two stores are related at $k$ steps if each value in the heap of the locations they have in common, are related at $j < k$ steps for any related security effects. We say that store $\mu_2$ is the evolution of store $\mu_1$, annotated $\mu_1 \rightarrow \mu_2$ if the domain of $\mu_1$ is a subset of $\mu_2$.

Two tuples of security effects, values and stores are related for $k$ steps at type $\text{Bool}_g$ if the security effects are related, the stores are related for $k$ steps, the values can be typed as $\text{Bool}_g$ using the security effects as context (any security effect will do, given that the typing of values do not depend on the security effect). Additionally, both security effect and values must both be either observable or not observable. If the security effect and values are observable then the raw values are the same. Two tuples are observables at type $\text{Unit}_g$ and $\text{Ref}_g U$ analogous to booleans.

Pairs are related at function types similarly to booleans. The difference is that functions can not be compared as booleans. Two functions are related if, given two related values and stores for $j \leq k$ steps at the argument type, the application of those function to the related values are also related for $j$ steps at at the return type.

Two tuples of terms and stores are related computations for $k$ steps at type $U$, first, if the security effects are related, and the stores are related for $k$ steps. Second the terms must be typed as $U$ using a observationally higher security effect. Third, if for any $j < k$ both terms can be reduced for at least $j$ steps, then the resulting stores are related for the remaining $k-j$ steps. Finally, if after at least $j$ steps the resulting terms are irreducible, then the resulting terms are also related values for the remaining $k-j$ steps at type $U$. Notice that the logical relation also relates programs that do not terminate as long as after $k$ steps the new stores are also related.

To define the fundamental property of the step-indexed logical relations we first define how to relate substitutions:

**Definition 107.** Let $\sigma$ be a substitution and $\Gamma$ a type substitution. We say that substitution $\sigma$ satisfy environment $\Gamma$, written $\sigma \models \Gamma$, if and only if $\text{dom}(\sigma) = \Gamma$.  

298
Definition 108 (Related substitutions). Tuples \( \langle \phi_1, \sigma_1, \mu_1 \rangle \) and \( \langle \phi_2, \sigma_2, \mu_2 \rangle \) are related on \( k \) steps under \( \Gamma \), notation \( \Gamma \vdash \langle \phi_1, \sigma_1, \mu_1 \rangle \approx^k_{ol} \langle \phi_2, \sigma_2, \mu_2 \rangle \), if \( \sigma_1 \models \Gamma \), \( \mu_1 \approx^k_{ol} \mu_2 \) and
\[
\forall x^U \in \Gamma. \langle \phi_1, \sigma_1(x^U), \mu_1 \rangle \approx^k_{ol} \langle \phi_2, \sigma_2(x^U), \mu_2 \rangle : U
\]

Proof of noninterference

Lemma 198 (Noninterference for booleans). Suppose \( k > 0 \), and
\begin{itemize}
  \item an open term \( \phi \triangleright t^U \in \text{TERM}_{\text{bool}} \) where \( \text{FV}(t) = \{ x^{U_1} \} \) with \( \text{label}(U_1) \nleq ol \)
  \item two compatible valid stores \( t^U \vdash \mu_i, \mu_1 \approx^k_{ol} \mu_2 \)
\end{itemize}
Then for any \( j < k \), \( v_1, v_2 \in \text{TERM}_{U_1} \), if both
\begin{itemize}
  \item \( t^U[v_1/x^{U_1}] | \mu_1 \xrightarrow{\phi} j v'_1 | \mu'_1 \)
  \item \( t^U[v_2/x^{U_1}] | \mu_2 \xrightarrow{\phi} j v'_2 | \mu'_2 \)
\end{itemize}
we have that \( \text{rval}(v'_1) = \text{rval}(v'_2) \), and \( \mu'_1 \approx^k_{ol} \mu'_2 \).

Proof. The result follows as a special case of Proposition \( \textbf{227} \) below. \( \square \)

In this theorem, we treat \( t^U \) as a program that takes \( x^{U_1} \) as its input. Furthermore, the security level \( g' = \text{label}(U_1) \) of the input is not subsumed by the security level \( ol \) of the observer. As such, noninterference dictates that changing non-observable input must not change the observable value of the output (i.e., change \texttt{true} to \texttt{false} or vice-versa). However, this theorem is technically \textit{termination-insensitive} in that it is vacuously true if a change of inputs changes a program that terminates with a value into one that either terminates with an \texttt{error}, or does not terminate at all. If a program does not terminate after any number of steps, then at least the stores are related at observation level \( ol \).

Note that we compare equality of raw values at first-order type. Restricting attention to first-order types (i.e., \texttt{Bool}) is common when investigating observational equivalence of typed languages. We strip away security information because a person or client who uses the program ultimately observes only the raw value that the program produces.

Also, gradual security dynamically traps some information leaks, so a change in equivalent inputs may cause a program that previously yielded a value or diverged to now produce an \texttt{error}. This change in behavior falls under the notion of \textit{termination-insensitive}, since yielding an error is simply a third form of termination behavior (in addition to producing a value and diverging).

Finally, we use notation \( t^S | \mu \xrightarrow{\phi} k t'^S | \mu' \) to describe that configuration \( t^S | \mu \) reduces, in at most \( k \) steps, to configuration \( t'^S | \mu' \).
Lemma 199. Consider $\varepsilon_1 \vdash g \lessapprox g'$. If $\forall \varepsilon_2$ such that $\varepsilon_2 \vdash g' \lessapprox ol$, $\varepsilon_1 \circ \varepsilon_2 \vdash g \lessapprox ol$ is not defined. Then if $\varepsilon_3 \vdash g' \lessapprox g''$, then $\forall \varepsilon_4$ such that $\varepsilon_4 \vdash g'' \lessapprox ol$, then $(\varepsilon_1 \circ \varepsilon_3) \circ \varepsilon_4 \vdash g \lessapprox ol$ is not defined.

Proof. Applying associativity: $(\varepsilon_1 \circ \varepsilon_3) \circ \varepsilon_4 = (\varepsilon_1 \circ (\varepsilon_3 \circ \varepsilon_4))$, but $(\varepsilon_3 \circ \varepsilon_4) \vdash g' \lessapprox g_o$, and we know that $\varepsilon_1 \circ \varepsilon_2$ is not defined $\forall \varepsilon_2$ such that $\varepsilon_2 \vdash g' \lessapprox ol$. Therefore $(\varepsilon_1 \circ \varepsilon_3) \circ \varepsilon_4 \vdash g \lessapprox ol$ is not defined and the result holds.

Lemma 200. Consider $\varepsilon_1 \vdash g \lessapprox g'$. If $\forall \varepsilon_2$ such that $\varepsilon_2 \vdash g' \lessapprox ol$, $\varepsilon_1 \circ \varepsilon_2 \vdash g \lessapprox ol$ is not defined. Also $\varepsilon_0 \vdash g_1 \lessapprox g_2$, if $\varepsilon_3 \vdash g_2 \gamma g' \lessapprox ol$, then $(\varepsilon_0 \gamma \varepsilon_1) \circ \varepsilon_3 \vdash g_1 \gamma g \lessapprox ol$ is not defined.

Proof. Let us prove that if $(\varepsilon_0 \gamma \varepsilon_1) \circ \varepsilon_3 \vdash g_1 \gamma g \lessapprox ol$ is defined, then $\varepsilon_1 \circ \varepsilon_2$ is defined.

As join is monotone $\exists \varepsilon'_0$ such that $\varepsilon'_0 \vdash g' \lessapprox g_2 \gamma g'$.

Suppose $\varepsilon_1 = ([\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}])$, $\varepsilon_0 = ([\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}])$, $\varepsilon'_0 = ([\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}])$, and $\varepsilon_3 = ([\ell_{71}, \ell_{72}], [\ell_{81}, \ell_{82}])$.

As $\varepsilon_0 \gamma \varepsilon_1 = ([\ell_{11} \gamma \ell_{31}, \ell_{12} \gamma \ell_{32}], [\ell_{21} \gamma \ell_{41}, \ell_{22} \gamma \ell_{42}])$ is defined, then $\ell_{11} \gamma \ell_{31} \lessapprox \ell_{12} \gamma \ell_{32}$ and $\ell_{21} \gamma \ell_{41} \lessapprox \ell_{22} \gamma \ell_{42}$. Also as

$$(\varepsilon_0 \gamma \varepsilon_1) \circ \varepsilon_3 = ([\ell_{11} \gamma \ell_{31}, (\ell_{12} \gamma \ell_{32}) \wedge ((\ell_{22} \gamma \ell_{42}) \wedge \ell_{72}) \wedge \ell_{82}],
\ell_{11} \gamma \ell_{31} \gamma \ell_{41} \gamma \ell_{41} \gamma \ell_{72} \gamma \ell_{81} \gamma \ell_{82})$$

is defined then $\ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox (\ell_{22} \gamma \ell_{42}) \wedge \ell_{72}$, $\ell_{11} \gamma \ell_{31} \lessapprox (\ell_{22} \gamma \ell_{42}) \wedge \ell_{72}$, $\ell_{11} \gamma \ell_{31} \lessapprox \ell_{82}$, and $\ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox \ell_{82}$.

If we choose $\varepsilon'_0$ as the interior of the judgment, then we do not get new information, therefore $[\ell_{21}, \ell_{22}] \subseteq [\ell_{51}, \ell_{52}]$, i.e. $\ell_{51} \lessapprox \ell_{21}$ and $\ell_{22} \lessapprox \ell_{52}$. Using the same argument $\ell_{61} \lessapprox \ell_{71}$ and $\ell_{72} \lessapprox \ell_{62}$.

Then

$$\varepsilon'_0 \circ \varepsilon_3 = \Delta \circ ([\ell_{51}, \ell_{52}, [\ell_{61}, \ell_{62}] \cap [\ell_{71}, \ell_{72}, [\ell_{81}, \ell_{82}])
\Delta \circ ([\ell_{51}, \ell_{52}, \ell_{61} \gamma \ell_{71}, \ell_{62} \wedge \ell_{72}, [\ell_{81}, \ell_{82}])
\Delta \circ ([\ell_{51}, \ell_{52}, \ell_{71}, \ell_{72}, [\ell_{81}, \ell_{82}])$$

which is defined if $\ell_{51} \lessapprox \ell_{72}$, $\ell_{71} \lessapprox \ell_{82}$ and $\ell_{51} \lessapprox \ell_{82}$. But $\ell_{51} \lessapprox \ell_{21} \lessapprox \ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox (\ell_{22} \gamma \ell_{42}) \wedge \ell_{72}$, $\ell_{51} \lessapprox \ell_{21} \lessapprox \ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox \ell_{82}$ and $\ell_{71} \lessapprox \ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox \ell_{82}$.

Therefore

$$\varepsilon'_0 \circ \varepsilon_3 = ([\ell_{51}, \ell_{52} \wedge \ell_{72} \wedge \ell_{82}], [\ell_{51} \gamma \ell_{71} \gamma \ell_{81} \gamma \ell_{82})$$

Using the same method, $\varepsilon_1 \circ \varepsilon_0$ is defined if $\ell_{21} \gamma \ell_{51} \lessapprox \ell_{22} \wedge (\ell_{52} \wedge \ell_{72} \wedge \ell_{82})$, $\ell_{11} \lessapprox \ell_{22} \wedge (\ell_{52} \wedge \ell_{72} \wedge \ell_{82})$, and $\ell_{11} \lessapprox \ell_{82}$.

But by definition of $\varepsilon_1 \circ \varepsilon_3$, also $\ell_{21} \lessapprox \ell_{22} \lessapprox \ell_{52}$, $\ell_{21} \lessapprox \ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox (\ell_{22} \gamma \ell_{42}) \wedge \ell_{72} \lessapprox \ell_{72}$, $\ell_{21} \lessapprox \ell_{21} \gamma \ell_{41} \gamma \ell_{71} \lessapprox \ell_{82}$, and $\ell_{51} \lessapprox \ell_{71} \lessapprox \ell_{72}$, therefore $\ell_{21} \gamma \ell_{51} \lessapprox \ell_{22} \wedge (\ell_{52} \wedge \ell_{72} \wedge \ell_{82})$.
Also $\ell_{11} \preceq \ell_{22} \preceq \ell_{52}$, $\ell_{11} \preceq \ell_{11} \lor \ell_{31} \preceq (\ell_{22} \lor \ell_{42}) \lor \ell_{72} \preceq \ell_{72}$, and $\ell_{11} \preceq \ell_{11} \lor \ell_{31} \preceq \ell_{82}$, therefore $\ell_{11} \preceq \ell_{22} \lor (\ell_{52} \lor \ell_{72} \lor \ell_{82})$, and $\ell_{11} \preceq \ell_{82}$.

Then as $\varepsilon_1 \circ \preceq (\varepsilon'_0 \circ \preceq \varepsilon_3)$ is defined then if we choose $\varepsilon_2 = (\varepsilon'_0 \circ \preceq \varepsilon_3) \vdash g' \preceq ol$, the result holds.

\begin{proof}
Suppose $\varepsilon_1 = \langle [\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}] \rangle$, $\varepsilon_2 = \langle [\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \rangle$, and $\varepsilon_3 = \langle [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}] \rangle$

Then

$$\begin{align*}
(\varepsilon_1 \circ \preceq \varepsilon_2) \circ \preceq \varepsilon_3 & = \Delta^{\preceq}(\ell_{11}, \ell_{12}, [\ell_{21}, \ell_{22}] \cap [\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}]) \circ \preceq \varepsilon_3 \\
& = \Delta^{\preceq}(\ell_{11}, \ell_{12}, [\ell_{21} \lor \ell_{31}, \ell_{22} \lor \ell_{32}], [\ell_{41}, \ell_{42}]) \circ \preceq \varepsilon_3 \\
& = \langle [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}] \rangle \circ \preceq (\ell_{41} \lor \ell_{31} \lor \ell_{42} \lor \ell_{32}) \\
& = \Delta^{\preceq}(\ell_{51}, \ell_{52}, [\ell_{61}, \ell_{62}]), [\ell_{41} \lor \ell_{31} \lor \ell_{42} \lor \ell_{32}], [\ell_{61}, \ell_{62}]) \\
& = \Delta^{\preceq}(\ell_{51}, \ell_{52}, [\ell_{61}, \ell_{62}]) \circ \preceq \langle [\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}] \rangle
\end{align*}$$

where $\ell'_{21} = \ell_{12} \lor (\ell_{22} \lor \ell_{32}) \lor \ell_{42} \lor \ell_{52}$ and $\ell'_{61} = \ell_{11} \lor (\ell_{21} \lor \ell_{31}) \lor \ell_{41} \lor \ell_{51} \lor \ell_{61}$. But

$$\begin{align*}
\varepsilon_1 \circ \preceq (\varepsilon_2 \circ \preceq \varepsilon_3) & = \varepsilon_1 \circ \preceq \Delta^{\preceq}([\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \cap [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}]) \\
& = \varepsilon_1 \circ \preceq \Delta^{\preceq}([\ell_{31}, \ell_{32}], [\ell_{41} \lor \ell_{51}, \ell_{42} \lor \ell_{52}], [\ell_{61}, \ell_{62}]) \\
& = \langle [\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \cap [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}] \rangle \\
& = \Delta^{\preceq}([\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \cap [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}]) \\
& = \Delta^{\preceq}([\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}]) \circ \preceq \langle [\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \cap [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}] \rangle \\
& = \Delta^{\preceq}([\ell_{11}, \ell_{12}], [\ell_{21}, \ell_{22}]) \circ \preceq \langle [\ell_{31}, \ell_{32}], [\ell_{41}, \ell_{42}] \cap [\ell_{51}, \ell_{52}], [\ell_{61}, \ell_{62}] \rangle
\end{align*}$$

where $\ell'_{21} = \ell_{12} \lor (\ell_{22} \lor \ell_{32}) \lor \ell_{42} \lor \ell_{52}$ and $\ell'_{61} = \ell_{11} \lor (\ell_{21} \lor \ell_{31}) \lor \ell_{41} \lor \ell_{51} \lor \ell_{61}$, and the result holds.
\end{proof}

\begin{lemma}
Consider $\varepsilon_1, \varepsilon_2$, and $\varepsilon_3$ such that $\varepsilon_1 \vdash g_1 \preceq g_2$, $\varepsilon_2 \vdash g_2 \preceq g_3$ and $\varepsilon_3 \vdash g_3 \preceq g_4$. If $\varepsilon_1 \vdash (\varepsilon_2 \circ \preceq \varepsilon_3)$ is defined, then $(\varepsilon_1 \vdash (\varepsilon_2 \circ \preceq \varepsilon_3))$ is defined
\end{lemma}

\begin{proof}
By definition of join and consistent transitivity, using the property that the join operator is monotone.
\end{proof}

\begin{lemma}
If $\overline{\varepsilon}_1$, such that $\varepsilon_1 \vdash g_1 \preceq g_2$, then $\overline{\varepsilon}_2$, such that $\varepsilon_2 \vdash g_3 \preceq g_2$.
\end{lemma}
Proof. By definition of join and consistent transitivity, using the property that the join operator is monotone.

Lemma 204. Consider stores $\mu_1, \mu_2, \mu_1', \mu_2'$ such that $\mu_i \rightarrow \mu_i'$, and substitutions $\sigma_1$ and $\sigma_2$, such that

$$\Gamma \vdash \langle \phi_1, \sigma_1, \mu_1 \rangle \leq_{ol}^k \langle \phi_2, \sigma_2, \mu_2 \rangle$$

then if $\forall j \leq k$, if $\mu_1' \leq_{ol}^j \mu_2'$ then $\Gamma \vdash \langle \phi_1, \sigma_1, \mu_1' \rangle \leq_{ol}^j \langle \phi_2, \sigma_2, \mu_2' \rangle$

Proof. By definition of related computations and related stores. The key argument is that given that $\mu_i \rightarrow \mu_i'$ then $\mu_i'$ have at least the same locations of $\mu_i$ and the values still are related as well given that they still have the same type.

Lemma 205 (Substitution preserves typing). If $\phi \triangleright t^U \in \text{TERM}_U$ and $\sigma \models \text{FV}(t^U)$ then $\phi \triangleright \sigma(t^U) \in \text{TERM}_U$.

Proof. By induction on the derivation of $\phi \triangleright t^U \in \text{TERM}_U$

Lemma 206 (Reduction preserves relations). Consider $\phi_i \leq_{ol} \phi_i'$, $\phi_i' \triangleright t_i \in \text{TERM}_U$, $\mu_i \in \text{STORE}$, $t_i \vdash \mu_i$, and $\mu_1 \leq_{ol}^k \mu_2$. Consider $j < k$, posing $t_i \models \mu_i \triangleright \phi_i' \triangleright h_i^j \triangleright \mu_i'$, we have

$$\langle \phi_1, t_1, \mu_1 \rangle \leq_{ol}^k \langle \phi_2, t_2, \mu_2 \rangle : C(U)$$

if and only if $\langle \phi_1, t_1', \mu_1' \rangle \leq_{ol}^{k-j} \langle \phi_2, t_2', \mu_2' \rangle : C(U)$

Proof. Direct by definition of

$$\langle \phi_1, t_1, \mu_1 \rangle \leq_{ol}^k \langle \phi_2, t_2, \mu_2 \rangle : C(U)$$

and transitivity of $\rightarrow_{\phi}^{'}$.

Lemma 207 (Ascription preserves relation). Suppose $\varepsilon \vdash U' \lesssim U$.

1. If $\langle \phi_1, v, \mu \rangle 1 \leq_{ol}^k \langle \phi_2, v, \mu \rangle 2 : U'$ then

$$\langle \phi_1, \varepsilon v_1 :: U, \mu_1 \rangle \leq_{ol}^{k+1} \langle \phi_2, \varepsilon v_2 :: U, \mu_2 \rangle : C(U).$$

2. If $\langle \phi_1, t, \mu \rangle 1 \leq_{ol}^k \langle \phi_2, t, \mu \rangle 2 : C(U')$ then

$$\langle \phi_1, \varepsilon t_1 :: U, \mu_1 \rangle \leq_{ol}^k \langle \phi_2, \varepsilon t_2 :: U, \mu_2 \rangle : C(U).$$

Proof. Following Zdancewic [131], the proof proceeds by induction on the judgment $\varepsilon \vdash U' \lesssim U$. The difference here is that consistent subtyping is justified by evidence, and that the terms have to be ascribed to exploit subtyping. In particular, case 1 above establishes a computation-level relation because each ascribed term ($\varepsilon v_1 :: U$) may not be a value: each value $v_i$ is either a bare value $u_i$ or a casted value $\varepsilon i u_i :: U_i$, with $\varepsilon_i \vdash S_i \lesssim U$. In the latter case, $(\varepsilon (\varepsilon i u_i :: U_i) :: U)$ either steps to error (in which case the relation is vacuously established), or steps to $\varepsilon' i u_i :: U$, which is a value. Next if both values were originally observables, then whatever the label of $U$ both values are going to be related. If both values were originally not observables, then by Lemma 207 both values are going to be still non observables.

302
Lemma 208. If $\langle \phi_1, v_1, \mu_1 \rangle \approx_{ol}^k \langle \phi_2, v_2, \mu_2 \rangle : U$ and, $\phi_i \triangleright uval(v_i) \in \text{TERM}_{\ell_i}$ where $U_i \subseteq U$, then $\forall U', U' \subseteq U', \varepsilon_1 \vdash U', \varepsilon_1 uval(v_1) :: U', \langle \phi_1, \varepsilon_1 uval(v_1) :: U', \mu_1 \rangle \approx_{ol}^k \langle \phi_2, \varepsilon_2 uval(v_2) :: U', \mu_1 \rangle : U'$.

Proof. Consider $U'$ and $\varepsilon$, such that $\varepsilon \vdash U \subseteq U'$. By Lemma 207, $\langle \phi_1, \varepsilon v_1 :: U', \mu_1 \rangle \approx_{ol}^{k+1} \langle \phi_2, \varepsilon v_2 :: U', \mu_2 \rangle : C(U)$. Next we consider the case were evidence combination do not fails. In case of a failure the lemma vacuously holds. Then as $\phi_i' \triangleright \varepsilon v_1 :: U' \in \text{TERM}_{U'}$, $\varepsilon v_1 :: U' \mid \mu_i \triangleright \phi_i uval(v_i) :: U' \mid \mu_i$ and the result follows using Lemma 206 and observational monotonicity of the transitivity (Lemma 214).

Lemma 209 (Downward Closed / Monotonicity). If

1. $\langle \phi_1, v_1, \mu_1 \rangle \approx_{ol}^k \langle \phi_2, v_2, \mu_2 \rangle : U$ then
   $\forall j \leq k, \langle \phi_1, v_1, \mu_1 \rangle \approx_{ol}^j \langle \phi_2, v_2, \mu_2 \rangle : U$

2. $\langle \phi_1, t_{i_1}'^U, \mu_1 \rangle \approx_{ol}^k \langle \phi_2, t_{i_2}'^U, \mu_2 \rangle : C(U)$ then
   $\forall j \leq k, \langle \phi_1, t_{i_1}'^U, \mu_1 \rangle \approx_{ol}^j \langle \phi_2, t_{i_2}'^U, \mu_2 \rangle : C(U)$

3. $\mu_1 \approx_{ol}^k \mu_2$ then $\forall j \leq k, \mu_1 \approx_{ol}^j \mu_2$

Proof. By induction on type $U$ and the definition of related stores.

Lemma 210. Consider $\varepsilon_1 \vdash g_1' \lesssim g_1$ and $\varepsilon_2 \vdash g_2' \lesssim g_2$. Then $(\neg \text{obs}_{ol}(\varepsilon_1 g_1) \land \varepsilon_1 \mid \leq \mid \varepsilon_2) \Rightarrow \neg \text{obs}_{ol}(\varepsilon_2 g_2)$.

Proof. Suppose $\varepsilon_1 = \langle \ell_{11}, \ell_{12}, \ell_{13}, \ell_{14} \rangle$ and $\varepsilon_2 = \langle \ell_{21}, \ell_{22}, \ell_{23}, \ell_{24} \rangle$.
Also consider $\varepsilon_1' = \varphi_{\lesssim}(g_1, ol) = \langle \ell_{11}', \ell_{12}' \rangle$ and $\varepsilon_2' = \varphi_{\lesssim}(g_2, ol) = \langle \ell_{21}', \ell_{22}' \rangle$.

If $\varepsilon_1 \circlearrowleft \varepsilon_1' = \triangle_{\lesssim}(\ell_{11}, \ell_{12}, \ell_{13}, \ell_{14}) \lesssim (\ell_{11}', \ell_{12}')$, $\ell_{13} \lesssim \ell_{14}$ is not defined then

1. $\ell_{13} \not\lesssim \ell_{14}$, $\ell_{11}$
2. $\ell_{11} \not\lesssim \ell_{14}$ or $\ell_{12}$
3. $\ell_{13} \not\lesssim ol$
4. $\ell_{11} \not\lesssim ol$

By construction we know that $\ell_{11} \not\lesssim \ell_{14}$. By $\varepsilon_1 \mid \leq \mid \varepsilon_2$ we know that $\ell_{13} \not\lesssim \ell_{23}$.

If $g_1 = \ell$, then $\ell_{11}' = \ell_{12}' = \ell_{13}' = \ell_{14}' = \ell$, therefore $\ell$ is not defined. If $\ell \not\lesssim \ell_{23}$, then $\ell_{23} \not\lesssim \ell_{21}'$ and the result holds immediately. If $\ell \not\lesssim \ell_{23}$, by construction of evidence we know that it must be the case that $\ell_{11} \not\lesssim \ell_{13}$, then either

1. $\ell \not\lesssim \ell_{23}$ (which is impossible),
2. $\ell_{11} \preceq \ell \land \ell$ (which is a contradiction by construction of evidence), or

3. $\ell \land \ell \preceq \omega \ell$ (which contradicts $\ell \preceq \omega \ell$) or

4. $\ell_{11} \preceq \omega \ell$

so the only possibility is that $\ell_{11} \preceq \omega \ell$, but we know that $\ell_{11} \preceq \ell_{13}$, i.e. $\ell_{11} \preceq \ell$ and that $\ell \preceq \omega \ell$, then by transitivity $\ell_{11} \preceq \omega \ell$ which is a contradiction so $\ell \preceq \omega \ell$ case cannot happen.

If $g_1 = \top$, then $[\ell_{11}', \ell_{12}'] = [\bot, \omega \ell]$.

If (1) holds, i.e. $\ell_{13} \preceq \ell_{14} \land \omega \ell$, by construction we know that $\ell_{13} \preceq \ell_{14}$, therefore it must be the case that $\ell_{13} \preceq \omega \ell$, but $\ell_{13} \preceq \ell_{23}$ and the result holds because (3) does not hold for $\varepsilon_2$.

If (2) holds, i.e. $\ell_{11} \preceq \ell_{14} \land \omega \ell$, by construction we know that $\ell_{11} \preceq \ell_{14}$, therefore it must be the case that $\ell_{11} \preceq \omega \ell$. We also know by construction that $\ell_{11} \preceq \ell_{13}$, then $\ell_{13} \preceq \omega \ell$. As $\ell_{13} \preceq \ell_{23}$, then $\ell_{23} \preceq \omega \ell$, and therefore (3) does not hold for $\varepsilon_2$, i.e. $\ell_{23} \land \ell_{21}' \preceq \omega \ell$. If (3) holds, i.e. $\ell_{13} \preceq \omega \ell$, but $\ell_{13} \preceq \ell_{23}$ and the result holds because (3) does not hold for $\varepsilon_2$.

If (4) holds, i.e. $\ell_{11} \preceq \omega \ell$, as $\ell_{11} \preceq \ell_{13} \preceq \ell_{23}$ then $\ell_{23} \preceq \omega \ell$, and therefore (3) does not hold for $\varepsilon_2$, i.e. $\ell_{23} \land \ell_{21}' \preceq \omega \ell$. □

**Lemma 211.** Consider $\varepsilon_1 \vdash g_1', g_2, \varepsilon_2 \vdash g_2', g_2 \preceq g_2$ and $\varepsilon_3 = \varepsilon_1 \neg \varepsilon_2$ such that $\varepsilon_3 \vdash g_1' \land g_2' \preceq g_1 \land g_2$. Then $\text{obs}(\varepsilon_1 g_1) \land \text{obs}(\varepsilon_2 g_2) \Rightarrow \text{obs}(\varepsilon_3(g_1 \neg g_2))$.

Proof. Suppose $\varepsilon_1 = \langle [\ell_{11}, \ell_{12}'), [\ell_{13}, \ell_{14}] \rangle$ and $\varepsilon_2 = \langle [\ell_{21}, \ell_{22}], [\ell_{23}, \ell_{24}] \rangle$.

Then $\varepsilon_1 \neg \varepsilon_2 = \varepsilon_3 = \langle [\ell_{11} \land \ell_{21}, \ell_{12} \land \ell_{22}], [\ell_{13} \land \ell_{23}, \ell_{14} \land \ell_{24}] \rangle$. Also consider $\varepsilon_1' = g_1(\ell_{11}, \omega \ell) = \langle [\ell_{11}'', \ell_{12}'], [\omega \ell, \omega \ell] \rangle$, $\varepsilon_2' = g_2(\ell_{21}, \omega \ell) = \langle [\ell_{21}', \ell_{22}], [\omega \ell, \omega \ell] \rangle$, and $\varepsilon_3' = g_2(\ell_{21} \neg \varepsilon_3, \omega \ell) = \langle [\ell_{31}', \ell_{32}'], [\omega \ell, \omega \ell] \rangle$.

If $g_1 = \ell_1$ and $g_2 = \ell_2$, then $\ell_{32}' = \ell_1 \land \ell_2$, $\ell_{22}' = \ell_2$ and $\ell_{12}' = \ell_1$. Also $\ell_{31}' = \ell_1 \land \ell_2$, $\ell_{21}' = \ell_2$ and $\ell_{11}' = \ell_1$.

If $g_1 = ?$ or $g_2 = \ell_2$ (the other case is analogous) then $\ell_{32}' = \omega \ell$ and, $\ell_{12}' = \omega \ell$ and $\ell_{22}' = \ell_2$ such that $\ell_2 \preceq \omega \ell$. Also $\ell_{11}' = \bot$, $\ell_{21}' = \ell_2$, but $\ell_{31}' = \bot$. Therefore $\ell_{32}' = \ell_1 \land \ell_{22}$ and $\ell_{31}' \preceq \ell_{11} \land \ell_{21}$.

We know that

1. $\ell_{13} \land \ell_{11}' \preceq \ell_{14} \land \ell_{12}'$,
2. $\ell_{11} \preceq \ell_{14} \land \ell_{12}'$, or
3. $\ell_{13} \land \ell_{11}' \preceq \omega \ell$ or
4. $\ell_{11} \preceq \omega \ell$.
5. $\ell_{23} \land \ell_{21}' \preceq \ell_{24} \land \ell_{22}'$,
6. $\ell_{21} \preceq \ell_{24} \land \ell_{22}'$, or
7. $\ell_{23} \land \ell_{21}' \preceq \omega \ell$ or
8. $\ell_21 \preceq ol$.

We have to prove

10. $(\ell_{13} \land \ell_{23}) \land \ell_{31} \preceq (\ell_{14} \land \ell_{24}) \land \ell_{32}$;
11. $(\ell_{11} \land \ell_{21}) \preceq (\ell_{14} \land \ell_{24}) \land \ell_{32}$, or
12. $(\ell_{13} \land \ell_{23}) \land \ell_{31} \preceq ol$ or
13. $(\ell_{11} \land \ell_{21}) \preceq ol$.

(13) follows directly by (4) and (8).
(12) follows from (3) and (7) and monotonicity of the join.
By definition of evidence and interior and $\ell_{32} \preceq ol$ and $\ell_{31} \preceq \ell_{32}$. Therefore, from (1) $\ell_{13} \preceq \ell_{14}$, from (5) $\ell_{23} \preceq \ell_{24}$ and therefore $\ell_{13} \land \ell_{23} \preceq \ell_{14} \land \ell_{24}$. Also as $\ell_{13} \preceq \ell_{12}$ and $\ell_{23} \preceq \ell_{12}$, then $\ell_{13} \land \ell_{23} \preceq \ell_{12} \land \ell_{22} = \ell_{32}$. By similar argument $\ell_{31} \preceq (\ell_{14} \land \ell_{24})$, and also $\ell_{11} \land \ell_{21} \preceq \ell_{32}$. But then $\ell_{31} \preceq \ell_{11} \land \ell_{21} \preceq \ell_{32}$ and (10) holds.

**Lemma 212.** Consider $\varepsilon_1 \vdash g_1 \preceq g_2$, $\varepsilon_2 \vdash g_2 \preceq g_3$, and $\varepsilon_3 = \varepsilon_1 \circ^\land \varepsilon_2$ such that $\varepsilon_3 \vdash g_1 \preceq g_3$. Then $\text{obs}_\text{ol}(\varepsilon_3(g_3)) \Rightarrow (\text{obs}_\text{ol}(\varepsilon_1 g_2) \land \text{obs}_\text{ol}(\varepsilon_2 g_3))$.

**Proof.** Suppose $\varepsilon_1 = \langle [\ell_1, \ell_2], [\ell_3, \ell_4] \rangle$, $\varepsilon_2 = \langle [\ell_5, \ell_6], [\ell_7, \ell_8] \rangle$.
$\varepsilon_1 \circ^\land \varepsilon_2 = \Delta^\land([\ell_1, \ell_2], [\ell_3 \land \ell_5 \land \ell_4 \land \ell_6], [\ell_7, \ell_8]) = \langle [\ell_1, \ell_2 \land \ell_4 \land \ell_6 \land \ell_8], [\ell_1 \land \ell_3 \land \ell_5 \land \ell_7 \land \ell_8] \rangle$

Notice that as $\ell_3 \preceq \ell_1 \land \ell_3 \land \ell_5 \land \ell_7$ then $\varepsilon_1 \preceq \varepsilon_3$, and as $\ell_7 \preceq \ell_1 \land \ell_3 \land \ell_5 \land \ell_7$ then $\varepsilon_2 \preceq \varepsilon_3$. What we have to prove is equivalent to prove that

$$(\neg \text{obs}_\text{ol}(\varepsilon_1 g_2) \lor \neg \text{obs}_\text{ol}(\varepsilon_2 g_3)) \Rightarrow \neg \text{obs}_\text{ol}(\varepsilon_3(g_3))$$

If $\neg \text{obs}_\text{ol}(\varepsilon_1 g_2)$ and as $\varepsilon_1 \preceq \varepsilon_3$, then by Lemma 210 $\neg \text{obs}_\text{ol}(\varepsilon_3(g_3))$ and the result holds. Similarly, if $\neg \text{obs}_\text{ol}(\varepsilon_2 g_3)$ and as $\varepsilon_2 \preceq \varepsilon_3$, then by Lemma 210 $\neg \text{obs}_\text{ol}(\varepsilon_3(g_3))$ and the result holds.

**Lemma 213.** Consider $\varepsilon_1 \vdash g_1 \preceq g_2$, $\varepsilon_2 \vdash g_2 \preceq g_3$, and $\varepsilon_3 = \varepsilon_1 \circ^\land \varepsilon_2$ such that $\varepsilon_3 \vdash g_1 \preceq g_3$. Then $(\text{obs}_\text{ol}(\varepsilon_1 g_2) \land \text{obs}_\text{ol}(\varepsilon_2 g_3)) \Rightarrow \text{obs}_\text{ol}(\varepsilon_3(g_3))$.

**Proof.** Suppose $\varepsilon_1 = \langle [\ell_1, \ell_2], [\ell_3, \ell_4] \rangle$, $\varepsilon_2 = \langle [\ell_5, \ell_6], [\ell_7, \ell_8] \rangle$.
$\varepsilon_1 \circ^\land \varepsilon_2 = \Delta^\land([\ell_1, \ell_2], [\ell_3 \land \ell_5 \land \ell_4 \land \ell_6], [\ell_7, \ell_8]) = \langle [\ell_1, \ell_2 \land \ell_4 \land \ell_6 \land \ell_8], [\ell_1 \land \ell_3 \land \ell_5 \land \ell_7 \land \ell_8] \rangle$

By definition of the transitivity operator, $\ell_1 \preceq \ell_8$, $\ell_1 \preceq \ell_4 \land \ell_6$, and $\ell_3 \land \ell_5 \preceq \ell_8$.

Let us consider $\varepsilon'_1 = \mathcal{G}_\lor(g_2, ol) = \langle [\ell'_1, \ell'_2], [ol, ol] \rangle$, $\varepsilon'_2 = \mathcal{G}_\lor(g_3, ol) = \langle [\ell'_5, \ell'_6], [ol, ol] \rangle$. We know that
1. \( \ell_3 \gamma \ell_1' \preceq \ell_4 \land \ell_2' \),
2. \( \ell_1 \preceq \ell_4 \land \ell_2' \), or
3. \( \ell_3 \gamma \ell_1' \preceq ol \) or
4. \( \ell_1 \preceq ol \).
5. \( \ell_7 \gamma \ell_5' \preceq \ell_8 \land \ell_6' \),
6. \( \ell_5 \preceq \ell_8 \land \ell_6' \), or
7. \( \ell_7 \gamma \ell_5' \preceq ol \) or
8. \( \ell_5 \preceq ol \).

We have to prove
9. \( \ell_1 \equiv \ell_1' \) or \( \ell_1' \equiv \ell_1 \).

Notice that if \( g_3 = ? \) then \( \ell_6' = ol \) and therefore by (4) \( \ell_1 \preceq \ell_6' \), and by (3), \( \ell_3 \preceq \ell_6' \). Also \( \ell_5' = \perp \) and therefore \( \ell_5' \preceq \ell_7 \preceq \ell_8 \). If \( g_3 = \ell \), then \( \ell_5' = \ell_6' = \ell \) and \( \ell_7 = \ell_8 = \ell \), but we know that \( \ell_1 \preceq \ell_8 \), and therefore \( \ell_1 \preceq \ell_6' \) and \( \ell_5' \preceq \ell_8 \). Also as \( \ell_3 \preceq \ell_8 \) then \( \ell_3 \preceq \ell_6' \).

We also know that \( \ell_3 \gamma \ell_5 \preceq \ell_8 \) and by definition of intervals \( \ell_7 \preceq \ell_8 \). We know that \( \ell_1 \preceq \ell_6' \). By (5) \( \ell_7 \gamma \ell_5' \preceq \ell_6' \). By (6) \( \ell_5 \preceq \ell_6' \). Also \( \ell_3 \preceq \ell_6' \) and (10) follows.

We know that \( \ell_1 \preceq \ell_8 \) and that \( \ell_1 \preceq \ell_6' \) therefore (11) holds. By (4), (3), (7), (8) and because \( \ell_5' \preceq ol \) by definition of interior, (12) holds. Finally (13) holds by (4).

\[ \square \]

**Lemma 214.** Consider \( \varepsilon_1 \vdash \varepsilon_1 \gamma_2 \gamma_3 \), and \( \varepsilon_3 \equiv \varepsilon_3' \). Then \( \neg \text{obs}_ol(\varepsilon_1 \gamma_2) \lor \neg \text{obs}_ol(\varepsilon_2 \gamma_3) \leftrightarrow \neg \text{obs}_ol(\varepsilon_3 \gamma_3) \).

**Proof.** Direct by Lemmas 212 and 213.

**Lemma 215.** Consider \( \varepsilon_1 \) and \( \varepsilon_1' = \varepsilon_2 \gamma (\varepsilon_1 \circ \varepsilon_3) \), for some \( \varepsilon_2 \) and \( \varepsilon_3 \). Then \( \varepsilon_1 \models \varepsilon_1' \)

**Proof.** Suppose \( \varepsilon_2 = ([\ell_1, \ell_2], [\ell_3, \ell_4]) \), \( \varepsilon_1 = ([\ell_5, \ell_2], [\ell_7, \ell_8]) \), and \( \varepsilon_3 = ([\ell_9, \ell_{10}, [\ell_{11}, \ell_{12}]). \)

\( \varepsilon_1 \circ \varepsilon_3 = \varepsilon_1 \gamma \varepsilon_3 = \varepsilon_1 \gamma ([\ell_5, \ell_6], [\ell_7 \gamma \ell_9, \ell_{10}], [\ell_{11}, \ell_{12}]) = ([\ell_5, \ell_6 \land \ell_8 \land \ell_{10} \land \ell_{12}], [\ell_5 \gamma \ell_7 \gamma \ell_9 \gamma \ell_{11} \land \ell_{12}]) \)

306
\[ \varepsilon_2 \sim (\varepsilon_1 \circ \varepsilon_3) = \langle [\ell_1 \gamma \ell_5, \ell_2 \gamma (\ell_6 \land \ell_8 \lor \ell_{10} \land \ell_{12})], [\ell_3 \gamma \ell_5 \gamma \ell_7 \gamma \ell_9 \gamma \ell_{11}, \ell_4 \gamma \ell_{12}] \rangle. \]

But \( \ell_7 \not\leq \ell_3 \gamma \ell_5 \gamma \ell_7 \gamma \ell_9 \gamma \ell_{11} \) and therefore, \( \varepsilon_1 \mid \leq \varepsilon'_1 \).

**Lemma 216.** Consider \( \varepsilon_1 \vdash g'_1 \sim g_1 \) and \( \varepsilon'_1 = \varepsilon_2 \sim (\varepsilon_1 \circ \varepsilon_3) \) such that \( \varepsilon_1 \vdash g'_2 \sim g_2 \).

Then \( \neg \text{obs}(\varepsilon_1 g_1) \rightarrow \neg \text{obs}(\varepsilon'_1 g_2) \).

**Proof.** By Lemma 215 and Lemma 210 the result holds immediately.

**Lemma 217.** Consider \( \varepsilon_1 \vdash g'_1 \sim g_1 \), \( \varepsilon_2 \vdash g'_2 \sim g_2 \), and \( \varepsilon_3 = \varepsilon_1 \sim \varepsilon_2 \) such that \( \varepsilon_3 \vdash g'_1 \gamma g'_2 \not\leq g_1 \gamma g_2 \). Then \( \varepsilon_1 \mid \leq \varepsilon_3 \).

**Proof.** Suppose \( \varepsilon_1 = \langle [\ell_1, \ell_2], [\ell_3, \ell_4] \rangle, \varepsilon_2 = \langle [\ell_5, \ell_6], [\ell_7, \ell_8] \rangle \), then \( \varepsilon_3 = \langle [\ell_1 \gamma \ell_5, \ell_2 \gamma \ell_6], [\ell_3 \gamma \ell_7, \ell_4 \gamma \ell_8] \rangle \).

As \( \ell_3 \not\leq \ell_3 \gamma \ell_7 \gamma \ell_1 \gamma \ell_2 \) therefore, \( \varepsilon_1 \mid \leq \varepsilon_3 \) and the result holds.

**Lemma 218.** Consider \( \varepsilon_1 \vdash g'_1 \sim g_1 \), \( \varepsilon_2 \vdash g'_2 \sim g_2 \), and \( \varepsilon_3 = \varepsilon_1 \sim \varepsilon_2 \) such that \( \varepsilon_3 \vdash g'_1 \gamma g'_2 \not\leq g_1 \gamma g_2 \). Then \( \neg \text{obs}(\varepsilon_1 g_1) \lor \neg \text{obs}(\varepsilon_2 g_2) \iff \neg \text{obs}(\varepsilon_3 (g_1 \gamma g_2)) \).

**Proof.** First we prove the \( \rightarrow \) direction. By Lemma 217, \( \varepsilon_1 \mid \leq \varepsilon_3 \). Suppose \( \text{obs}(\varepsilon_1 g_1) \) does not hold (the other case is analogous). Then by Lemma 210 the result holds immediately. Then for the \( \iff \) we use Lemma 211 and the result holds immediately.

**Lemma 219.** Consider \( \phi' \triangleright t^U \in \text{TERM}_U \), and \( \mu \), such that \( t^U \vdash \mu \) and \( \neg \text{obs}(\phi') \), and \( \forall k > 0 \), such that \( t^U \mid \mu \overset{\phi'}{\rightarrow} k t^U \mid \mu' \), then \( \forall \phi' \),

1. \( \forall o'' \in \text{dom}(\mu') \setminus \text{dom}(\mu) \), \( \neg \text{obs}(\phi \triangleright \mu'(o'')) \).

2. \( \forall o'' \in \text{dom}(\mu') \cap \text{dom}(\mu) \land \mu'(o'') \neq \mu(o'') \),
   
   (a) \( \neg \text{obs}(\phi \triangleright \mu(o'')) \), and

   (b) \( \neg \text{obs}(\phi \triangleright \mu'(o'')) \).

**Proof.** We use induction on the derivation of \( t^U \). The interest cases are the last step of reduction rules for references and assignments.

**Case** \( t = \varepsilon_1 o'^U_g := \varepsilon_2 u \). We are only updating the heap so we only have to prove (a) and (b). Then

\[ \varepsilon_1 o'^U_g := \varepsilon_2 u \overset{\phi'}{\rightarrow} \text{unit} \mid \mu | o'^U \overset{\varepsilon'}{\rightarrow} \varepsilon'(u \sim (\phi' \gamma g \gamma g')) :: U' \]

where \( \varepsilon' = (\varepsilon_2 \circ \varepsilon_1) \gamma (\phi' \gamma g \gamma g') \). We have to prove that (b) \( \neg \text{obs}(\varepsilon'_1 \gamma \text{label}(U')) \). As \( \neg \text{obs}(\phi') \), by Lemma 218, \( \neg \text{obs}(\phi' \gamma \text{label}(U')) \).
$g$). Then by Lemma 216, $(\text{obs}_{ol}(\varepsilon \text{label}(U'))$. Next we have to prove that $(a) \text{obs}_{ol}(\phi \triangleright \mu(o^U'))$ is not defined. Consider that $\mu(o^U') = \varepsilon U'$. We know that $\text{obs}_{ol}(\phi \varepsilon \phi', g_c)$ is not defined, and that $\phi' \varepsilon [\leq] \varepsilon$, therefore by Lemma 210 $\text{obs}_{ol}(\varepsilon U')$ is not defined, concluding that $\text{obs}_{ol}(\phi \triangleright \mu(o^U'))$ is not defined as well and the result holds.

Case ($t = \text{ref}^U_{\varepsilon s u}$). We are extending the heap, so we need to only prove $(1)$. Then

$$\text{ref}^U_{\varepsilon s u} | \mu \implies o^U_1 | \mu [o^U' \mapsto \varepsilon'(u \tilde{\gamma} \phi', g_c) :: U']$$

where $o^U' \notin \text{dom}(\mu), \varepsilon' = \varepsilon s \tilde{\gamma} (\phi', g_c, o \varepsilon \varepsilon)$. We need to prove that $\text{obs}_{ol}(\phi \triangleright \varepsilon'(u \tilde{\gamma} \phi', g_c) :: U')$ does not hold. In order to do so, we will show that $\text{obs}_{ol}(\text{ilbl}(\varepsilon') \text{label}(U'))$ does not hold, which follows directly by Lemma 216.

Lemma 220. Consider $\phi'$, such that $\text{obs}_{ol}(\phi' \varepsilon \phi', g_c)$ does not hold, then then $\forall k > 0$, such that $t^U_i | \mu_i \implies k t^U_i | \mu'_i$, then if $\mu_1 \approx^k_{ol} \mu_2$, then $\mu'_1 \approx^k_{ol} \mu'_2$

Proof. By Lemma 219 we know three things:

1. $\forall o'' \in \text{dom}(\mu'_i) \setminus \text{dom}(\mu_i), \text{obs}_{ol}(\phi \triangleright \mu'_i(o''))$ does not hold, i.e. new locations are not observable.
2. $\forall o'' \in \text{dom}(\mu'_i) \cap \text{dom}(\mu_i) \land \mu'_i(o'') \neq \mu(o'''),$
   (a) $\text{obs}_{ol}(\phi \triangleright \mu'_i(o'''))$ does not hold, and
   (b) $\text{obs}_{ol}(\phi \triangleright \mu'_i(o'''))$ does not hold.

   i.e. for all updated references they have to be previously not observable, and by definition therefore related, and second they are still non observable after the update, and by definition those locations are still related under $\phi$.

Therefore $\mu'_1 \approx^k_{ol} \mu'_2$ and the result holds.

Lemma 221. Consider simple values $u_i \in \text{TERM}_{t_i}$ and

$\langle \phi_1, \varepsilon_1 u_1 :: U, \mu_1 \rangle \approx^k_{ol} \langle \phi_2, \varepsilon_2 u_2 :: U, \mu_2 \rangle : U$.

If $\varepsilon_1 \vdash g \lesssim U$, then

$\langle \phi_1, (\varepsilon'_1 \tilde{\gamma} \varepsilon_1)(u_1 \tilde{\gamma} g) :: U, \mu_1 \rangle \approx^k_{ol} \langle \phi_2, (\varepsilon'_2 \tilde{\gamma} \varepsilon_2)(u_2 \tilde{\gamma} g) :: U, \mu_2 \rangle : U$

Proof. By definition of related values and Lemma 218 (observational-monotonicity of the join), considering that the label stamping can make the new values non observable and that join of evidences does not introduce imprecision.

Lemma 222. Suppose that $\phi_i \leq_{ol} \phi'_i, \phi'_i \triangleright \text{prot}^U_{\varepsilon_i^U g_i} \phi''_{\varepsilon_i^U g_i} \in \text{TERM}_{U \tilde{\gamma} g}$, for $i \in \{1, 2\}$, where $\text{obs}_{ol}(\phi'_i \varepsilon \phi'_i, g_c)$ does not hold, and either $\text{obs}_{ol}(\phi'_i \varepsilon \phi'_i, g_c)$ or $\text{obs}_{ol}(\varepsilon_i^U g)$ does not hold. Also
consider two stores $\mu_1$ such that $\mu_1 \approx_{\text{ol}}^k \mu_2$.

Then $\langle \phi_1, \text{prot}_{\epsilon_1}^U \phi_1' (\epsilon_1 t_{U_1}) s, \mu_1 \rangle \approx_{\text{ol}}^k \langle \phi_2, \text{prot}_{\epsilon_2}^U \phi_2' (\epsilon_2 t_{U_2}), \mu_2 \rangle$

Proof. Suppose that after at least $j$ more steps, where $j < k$, both subterms reduce to a value (let us assume no cast errors are produced, otherwise the lemma vacuously holds):

$$t_{U_1} | \mu_1 \overset{\phi''}{\longrightarrow} j \quad \epsilon_1' v_1 | \mu'_1$$

Therefore:

$$\text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1 t_{U_1}) | \mu'_1$$

$$\overset{\phi'_1}{\longrightarrow} j \quad \text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1 u_1) | \mu'_1$$

$$\overset{\phi'_1}{\longrightarrow} 1 \ (\epsilon_1' \rightarrow \epsilon_1') (u_1 \rightarrow g_1') :: U \rightarrow g \ | \mu'_1$$

As the values can be radically different we have to make sure that both values are not observables. If $\text{obs}_{\text{ol}}(\phi_1, \epsilon_1, g_1)$ does not hold then the values are not observables because the security context is not observable. Let us assume that $\text{obs}_{\text{ol}}(\epsilon_1, g_1)$ holds, but $\text{obs}_{\text{ol}}(\epsilon_1' g_1')$ not. Then by Lemma 218 $\text{obs}_{\text{ol}}((\epsilon_1', \epsilon_1') (\text{label}(U) \rightarrow g))$ does not hold, and therefore $\text{obs}_{\text{ol}}(\phi_1 \rightarrow (\epsilon_1', \epsilon_1') (u_1 \rightarrow g_1') :: U \rightarrow g)$ does not hold.

Now we have to prove that the resulting stores are related. But by Lemma 220 the result immediately.

$\Box$

Lemma 223. Suppose that $\phi_1 \leq_{\text{ol}} \phi_1', \phi_1 \leq_{\text{ol}} \phi_1'', \langle \phi_1, t_1, \mu_1 \rangle \approx_{\text{ol}}^k \langle \phi_2, t_2, \mu_2 \rangle : C(U')$, and that $\phi_1' \rightarrow \text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1 t_{U_1}) \in \text{TERM}_{\rightarrow \epsilon_2} \gamma$, for $i \in \{1, 2\}$. If $\phi_1 \approx_{\text{ol}}^k \phi_2$, and both $\text{obs}_{\text{ol}}(\epsilon_1, g_1)$ hold or does not hold,
then $\langle \phi_1, \text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1 t_{U_1}), \mu_1 \rangle \approx_{\text{ol}}^k \langle \phi_2, \text{prot}_{\epsilon_2}^U \phi_2'' (\epsilon_2 t_{U_2}), \mu_2 \rangle : C(U \rightarrow g)$

Proof. In case that combining evidence may fail, then the Lemma vacuously holds. Let us assume that combining evidence always success. Consider $j < k$, we know by definition of related computations that

$$t_{U_1}'' | \mu_1 \overset{\phi''}{\longrightarrow} j t_{U_1}'' | \mu'_1$$

then $\mu'_1 \leq_{\text{ol}}^{\sim}^j \mu'_2$, and by Lemma 224 $\mu_1 \rightarrow \mu'_1$. If $t_{U_1}''$ are reducible after $k - 1$ steps, then the result holds immediately by (Rprot()). The interest case if $t_{U_1}''$ are irreducible after $j < k$ steps:

Suppose that after $j$ steps $t_{U_1}'' = v_1$, then $\langle \phi_1, v_1, \mu'_1 \rangle \approx_{\text{ol}}^{k-j} \langle \phi_2, v_2, \mu'_2 \rangle : U'$. Therefore:

$$\text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1 t_{U_1}'') | \mu'_1$$

$$\overset{\phi'_1}{\longrightarrow} j \quad \text{prot}_{\epsilon_1}^U \phi_1'' (\epsilon_1' u_1) | \mu'_1$$

$$\overset{\phi'_1}{\longrightarrow} 1 \ (\epsilon_1' \rightarrow \epsilon_1') (u_1 \rightarrow g_1') :: U \rightarrow g | \mu'_1$$

309
If both $\text{obs}_{\text{ol}}(\phi \triangleright v_i)$ do not hold, then by Lemma 226 $\text{obs}_{\text{ol}}(\varepsilon''u_i :: U')$ also does not hold. Finally by Lemma 218 $\text{obs}_{\text{ol}}((\varepsilon'' \tilde{\gamma} \varepsilon_i')(\text{label}(U) \tilde{\gamma} g))$ does not hold and therefore the values are related.

Let us consider that $\text{obs}_{\text{ol}}(\phi \triangleright v_i)$ holds and that $\text{obs}_{\text{ol}}(\phi \triangleright \varepsilon''u_i :: U')$ holds (otherwise we follow by the previous argument). If both $\text{obs}_{\text{ol}}(\phi_i \varepsilon \phi_i.g_c)$ do not hold, then the values are not observables because the security contexts are not observable.

Let us assume that both $\text{obs}_{\text{ol}}(\phi_i \varepsilon \phi_i.g_c)$ hold, but $\text{obs}_{\text{ol}}(\varepsilon_i g)$ not. Then by Lemma 218 $\text{obs}_{\text{ol}}((\varepsilon'' \tilde{\gamma} \varepsilon_i')(\text{label}(U) \tilde{\gamma} g))$ do not hold, and therefore $\text{obs}_{\text{ol}}(\phi_i \triangleright (\varepsilon'' \tilde{\gamma} \varepsilon_i')(u_i \tilde{\gamma} g') :: U \tilde{\gamma} g)$ do not hold.

If $\text{obs}_{\text{ol}}(\phi_i \varepsilon \phi_i.g_c)$ and $\text{obs}_{\text{ol}}((\varepsilon'' \tilde{\gamma} \varepsilon_i')(\text{label}(U) \tilde{\gamma} g))$ hold, then the result follows by Lemma 221 and by backward preservation of the relations (Lemma 206).

\begin{lemma}
Consider term $\phi \triangleright t^U \in \text{TERM}_U$, store $\mu$ and $j > 0$, such that $\mu \triangleright t^U \triangleright \phi \triangleright t'^U | \mu'$. Then $\mu \rightarrow \mu'$.
\end{lemma}

\begin{proof}
Trivial by induction on the derivation of $t'^U$. The only rules that change the store are the ones for reference and assignment, neither of which remove locations.
\end{proof}

\begin{lemma}
If $\phi \leq_{\text{ol}} \phi'$ and $\phi' \leq_{\text{ol}} \phi''$, then $\phi \leq_{\text{ol}} \phi''$.
\end{lemma}

\begin{proof}
Trivial because if $\phi$ is not observable, then $\phi'$ is not observable as well by definition of $\leq_{\text{ol}}$, and therefore $\phi''$ must also be not observable.
\end{proof}

\begin{lemma}
Consider $\phi_i \triangleright v \in \text{TERM}_U$, and $\varepsilon :: U \subseteq U'$. Suppose $\varepsilon v :: U' \rightarrow i \varepsilon' u :: U'$. If 
\begin{align*}
\neg \text{obs}_{\text{ol}}(\phi_i \triangleright v) \lor \neg \text{obs}_{\text{ol}}(\varepsilon U') \iff \neg \text{obs}_{\text{ol}}(\phi_i \triangleright \varepsilon' u :: U').
\end{align*}
\end{lemma}

\begin{proof}
Direct by Lemma 214.
\end{proof}

Next, we present the Noninterference proposition, which naturally implies the Security Type Soundness proposition (Prop 25) presented in § 4.5.

\begin{proposition}[Noninterference]
If $\phi_i \triangleright \tilde{i} \in \text{TERM}_U$, $\mu_i \in \text{STORE}$, $\tilde{i} \vdash \mu_i$, $\Gamma = \text{FV}(\tilde{i})$, and $\forall k \geq 0, \phi_1 \leq_{\text{ol}} \phi_i', \Gamma \vdash \langle \phi_1, \sigma_1, \mu_1 \rangle \approx_{k} \langle \phi_2, \sigma_2, \mu_2 \rangle$, then 
\begin{align*}
\langle \phi_1, \sigma_1(\tilde{i}), \mu_1, \approx_{\text{ol}} \rangle [\alpha]^{k} \langle \phi_2, \sigma_2(\tilde{i}), \mu_2, :: \rangle C(U).
\end{align*}
\end{proposition}

\begin{proof}
By induction on the derivation of term $\tilde{i} \in \text{TERM}_U$. Let us take an arbitrary index $k \geq 0$.

\textbf{Case (x).} $\tilde{i} = x^U$ so $\Gamma = \{x^U\}$. $\Gamma \vdash \langle \phi_1, \sigma_1, \mu_1 \rangle \approx_{k} \langle \phi_2, \sigma_2, \mu_2 \rangle$ implies by definition that $\langle \phi_1, \sigma_1(x^U), \mu_1 \rangle \approx_{k} \langle \phi_2, \sigma_2(x^U), \mu_2 \rangle : U$, and the result holds immediately.

310
Case (b). \( \hat{t} = b \). By definition of substitution, \( \sigma_1(b) = \sigma_2(b) = b \). By definition, \( \langle \phi_1, b, \mu_1 \rangle \approx_{ol} \langle \phi_2, b, \mu_2 \rangle : \text{Bool}_g \) as required.

Case (o). \( \hat{t} = \phi_{g_1} \) where \( U = \text{Ref}_g U_1 \). By definition of substitution, \( \sigma_1(\phi_{g_1}) = \sigma_2(\phi_{g_1}) = \phi_{g_1} \). We know that \( \phi_1 \triangleright \phi_{g_1} \in \text{TERM}_{\text{Ref}_g} U_1 \). By definition of related stores, \( \langle \phi_1, \phi_{g_1}, \mu_1 \rangle \approx_k \langle \phi_2, \phi_{g_1}, \mu_2 \rangle : \text{Ref}_g U_1 \) as required, and the result holds.

Case (\( \lambda \)). \( t^U = (\lambda^{g_1} x U_1 t U_2)_g \). Then \( U = U_1 \overset{\sigma_1}{\rightarrow}_g U_2 \).

By definition of substitution, assuming \( x U_1 \notin \text{dom}(\sigma_1) \), and Lemma 205

\[ \phi_1 \triangleright \sigma_1(t^U) = \phi_1 \triangleright (\lambda^{g_1} x U_1 \cdot \sigma_1(t U_2))_g \in \text{TERM}_U \]

Consider \( j \leq k, \mu_1, \mu_2 \) such that \( \mu_i \rightarrow \mu_i' \) and \( \mu_1 \approx_{ol} \mu_2 \), and assume two values \( v_1 \) and \( v_2 \) such that \( \langle \phi_1, v_1, \mu_1 \rangle \approx_{ol} \langle \phi_2, v_2, \mu_2 \rangle : U_1 \). Consider \( U' = U_1^{g_1} \overset{\sigma', \mu'}{\rightarrow}_g U_2^{g_2} \) and \( \varepsilon_1, \varepsilon_2, \varepsilon_\ell \), such that \( \varepsilon_1 \vdash U_1 \overset{\sigma_1}{\rightarrow}_g U_2 \subseteq U' \), that \( \varepsilon_2 \vdash U_1 \subseteq U''_1 \), and that \( \varepsilon_\ell \vdash \phi_1 \triangleright \sigma' \rightarrow \gamma g'' \).

For simplicity, let us annotate \( U_2' = U_2'' \overset{\gamma}{\rightarrow} g'' \). We need to show that:

\[ \langle \phi_1, \varepsilon_1, (\lambda^{g_1} x U_1 \cdot \sigma_1(t U_2))_g \overset{\sigma_1}{\rightarrow}_{\varepsilon_\ell} \varepsilon_2 \mid \varepsilon_\ell \rangle \approx_{ol} \langle \phi_2, \varepsilon_1, (\lambda^{g_1} x U_1 \cdot \sigma_2(t U_2))_g \overset{\sigma_1}{\rightarrow}_{\varepsilon_\ell} \varepsilon_2 \mid \varepsilon_\ell \rangle : C(U_2') \]

Each \( v_1 \) is either a bare value \( u_1 \) or a casted value \( \varepsilon_{a_1} u_1 :: U_1' \). In the latter case, the application expression combines evidence, which may fail with error. If it succeeds, we call the combined evidence \( \varepsilon_{a_1}' \). The application rule then applies: it may fail with error if the evidence \( \varepsilon_{a_1}' \) cannot be combined with the evidence for the function parameter. Every time a failure is produced product of evidence combination, then the relation vacuously holds. We therefore consider the only interesting case, where reductions always succeed. Then:

\[
\varepsilon_1 (\lambda^{g_1} x U_1 \cdot \sigma_1(t U_2))_g \overset{\sigma_1}{\rightarrow}_{\varepsilon_\ell} \varepsilon_2 u_1 \mid \mu_1' \\
\overset{\phi_1'}{\rightarrow} \text{prot}_{\varepsilon_1} \phi_1'' (\varepsilon_1 (\varepsilon_{a_1} u_1 :: U_1 / x U_1) \sigma_1(t U_2))_g \mid \mu_1' \\
\overset{\phi_1''}{\rightarrow} \text{prot}_{\varepsilon_1} \phi_1'' (\varepsilon_1 (\varepsilon_{a_1} u_1 :: U_1 / x U_1) \sigma_1(t U_2))_g \mid \mu_1'
\]

where \( \phi'' = (\varepsilon_1' (\phi' \triangleright \gamma g), \phi_1'') \), \( \varepsilon_1' = (\phi' \triangleright \gamma \text{ilbl}(\varepsilon_1)) \circ \varepsilon_\ell \circ \varepsilon_{a_1} \circ \varepsilon_1 \). If \( \text{obs}_{ol} (\phi_1') \) do not hold, then by Lemma 218 \( \text{obs}_{ol} (\phi_1'') \) do not hold. Then \( \phi_1' \leq_{ol} \phi_1'' \), and by Lemma 225 \( \phi_1 \leq_{ol} \phi_1'' \).

\( \varepsilon_1, \varepsilon_p \) and \( \varepsilon_{a_1} \) are the new evidences for the label, return value and argument, respectively. We then extend the substitutions to map \( x U_1 \) to the casted arguments:

\[ \sigma_1' = \sigma_1 \{ x U_1 \mapsto \varepsilon_{a_1} u_1 :: U_1 \} \]
We know that \( \langle \phi_1, v_1, \mu'_1 \rangle \approx_{ol}^j \langle \phi_2, v_2, \mu'_2 \rangle \) and consider \( \phi \triangleright u \in \text{TERM}_{U_1} \) then \( \varepsilon_{ai} \vdash U_1 \preceq U_1 \) and \( \varepsilon_{ai} = (\varepsilon_{a1} \circ^< \varepsilon_{2}) \circ^< \text{idom}(\varepsilon_1) \), therefore using Lemma 208 \( \langle \phi_1, (\varepsilon_{a1} u_1 :: U_1), \mu'_1 \rangle \approx_{ol}^j \langle \phi_2, (\varepsilon_{a2} u_2 :: U_1), \mu'_2 \rangle : U_1 \)

So as \( \mu_i \rightarrow \mu'_i \) then by Lemma 204 \( \Gamma, x^{U_1} \vdash \langle \phi_1, \sigma'_1, \mu'_1 \rangle \approx_{ol}^j \langle \phi_2, \sigma'_2, \mu'_2 \rangle \).

We also know that \( \phi'' \triangleright \sigma_1(t^{U_2}) \in \text{TERM}_{U_2} \). Then by induction hypothesis:

\[
\langle \phi_1, \sigma'_1(t^{U_2}), \mu'_1 \rangle \approx_{ol}^{j-1} \langle \phi_2, \sigma'_2(t^{U_2}), \mu'_2 \rangle : C(U_2)
\]

Finally, by Lemma 223

\[
\langle \phi_1, \text{prot}_{sg}^\sigma \phi''(e_{sg} \sigma'_1(t^{U_2})), \mu'_1 \rangle \\
\approx_{ol}^j \langle \phi_2, \text{prot}_{sg}^\sigma \phi''(e_{sg} \sigma'_2(t^{U_2})), \mu'_2 \rangle : C(U'_2)
\]

and finally the result holds by backward preservation of the relations (Lemma 206).

\textbf{Case } (!) \textbf{.} \( t^U = \text{Ref}_{g} U_1 \in t^{U_1} \). Then \( U = U_1 \overset{\gamma}{\rightarrow} g \).

By definition of substitution:

\[
\sigma_1(t^U) = \text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1})
\]

We have to show that

\[
\langle \phi_1, \text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1}), \mu_1 \rangle \\
\approx_{ol}^k \langle \phi_2, \text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1}), \mu_2 \rangle : C(U_1 \overset{\gamma}{\rightarrow} g)
\]

By Lemma 205

\[
\phi'_1 \triangleright \text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1}) \in \text{TERM}_{U_1 \overset{\gamma}{\rightarrow} g}
\]

By induction hypotheses on the subterm:

\[
\langle \phi_1, \sigma_1(t^{U_1}), \mu_1 \rangle \approx_{ol}^k \langle \phi_2, \sigma_2(t^{U_1}), \mu_2 \rangle : C(U'_1)
\]

Consider \( j < k \); then by definition of related computations

\[
\sigma_1(t^{U_1}) | \mu_i \overset{\phi_i^j}{\rightarrow} j_{t_i}^{U_1} | \mu'_i \quad \Rightarrow \quad \mu'_i \approx_{ol}^{k-j} \mu_2 \wedge (\text{irred}(t^{U_1}) \Rightarrow \langle \phi_1, t_1^{U_1}, \mu'_1 \rangle \approx_{ol}^{k-j} \langle \phi_2, t_2^{U_1}, \mu'_2 \rangle : U'_1)
\]

Where \( U'_1 = \text{Ref}_{g'} U'' \). If terms \( t_1^{U_1} \) are reducible after \( j = k - 1 \) steps, then

\[
\text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1}) | \mu_i \overset{\phi_i^j}{\rightarrow} j_{t_i}^{U_1} | \mu'_i \quad \text{and the result holds}.
\]

If after at most \( j \) steps \( t_i^{U_1} \) is irreducible it means that for some \( j' \leq j \), \( \text{Ref}_{g} U_1 \varepsilon \sigma_1(t^{U_1}) | \mu_i \overset{\phi_i^{j'}}{\rightarrow} j'_{t_i}^{U_1} | \varepsilon_{v_i} \quad \text{If } j' = j \text{ then we use the same argument for reducible terms and the result holds}.
\]

Let us consider now \( j' < j \). Then \( \langle \phi_1, v_1, \mu'_1 \rangle \approx_{ol}^{k-j'} \langle \phi_2, v_2, \mu'_2 \rangle : U'_1 \). By Lemma 274 each \( v_i \) is either a location \( (\alpha_{v_i}^{U'}_{g'}) \) or a casted location \( \varepsilon_1(\alpha_{v_i}^{U'}_{g'}) :: U'_1 \). Let us assume they both are a casted location (the other cases are analogous). In case a value \( v_{ij} \) is a casted value, then the
whole term $\sigma_1(t^U)$ can take a step by (Rg), combining $\varepsilon$ with $\varepsilon_1$. Such a step either fails, or succeeds with a new combined evidence. Therefore, either:

$$\sigma_1(t^U) \mid \mu_i \xrightarrow{\phi_i'} j_i \text{error}$$

in which case we do not care since we only consider termination-insensitive noninterference, or:

$$\sigma_1(t^U) \mid \mu \xrightarrow{\phi_i'} j_i+1 \quad \text{tRef}_{\mu_i} \varepsilon_i' \phi_i^{U''} \mid \mu_i'$$

and finally the result holds by backward preservation of the relations (Lemma 206).

By definition of substitution:

$$\varepsilon_1 = \varepsilon_1 \phi_i^g \approx \varepsilon_2 \phi_i^g$$

with $\varepsilon_1' = \phi_i^g U'' = \varepsilon_1 U''$, $\phi_i'' = \langle (\phi_i^g \varepsilon \text{bl}(\varepsilon_i)) (\phi_i^g \varepsilon \text{bl}(\varepsilon_i)), \phi_i^g \varepsilon \text{bl}(\varepsilon_i) \rangle$. By Lemma 218, if $\neg \text{obs}_{\phi_i^g}$ then $\neg \text{obs}_{\phi_i''}$. Then by Lemma 225 $\phi_i \leq \phi_i''$. As $\langle \phi_i, v_1, \mu_1' \rangle \approx U'_1$, then by Lemma 214 either both $\text{obs}_{\text{bl}(\varepsilon_i)}(\text{label}(U) \varepsilon \varepsilon)$ holds or do not hold. Finally as $\langle \phi_i, v_1, \mu_1' \rangle \approx U'_1$, by Lemma 222:

$$\langle \phi_i, \text{prot}_{\text{bl}(\varepsilon_i)}(\varepsilon_i)angle \phi_i''(\text{iref}(\varepsilon_i) v_1'), \mu_i' \rangle$$

and finally the result holds by backward preservation of the relations (Lemma 206).

---

**Case (:=) $t^U = \varepsilon_1 t^U_1 := \varepsilon_2 t^U_2$. Then $U = \text{Unit}_\perp$.**

By definition of substitution:

$$\sigma_1(t^U) = \varepsilon_1 \sigma_1(t^U_1) := \varepsilon_2 \sigma_1(t^U_2)$$

and Lemma 205:

$$\phi_i' \triangleright \varepsilon_1 \sigma_1(t^U_1) := \varepsilon_2 \sigma_1(t^U_2) \in \text{TERM}_{\text{Unit}_\perp}$$

We have to show that

$$\langle \phi_i, \varepsilon_1 \sigma_1(t^U_1) := \varepsilon_2 \sigma_1(t^U_2) \rangle = \langle \phi_i, \varepsilon_1 \sigma_1(t^U_1), \mu_1 \rangle \approx U_1$$

By induction hypotheses

$$\langle \phi_i, \sigma_1(t^U_1), \mu_1 \rangle \approx \langle \phi_i, \sigma_2(t^U_1), \mu_2 \rangle : C(U_1)$$

Suppose $j_1 < k$, and that $\sigma_1(t^U_1)$ are irreducible after $j_1$ steps (otherwise, similar to case !, the result holds immediately). Then by definition of related computations:

$$\sigma_i(t^U_1) \mid \mu_i \xrightarrow{i} j_1 v_i \mid \mu_i' \implies \mu_i' \approx \langle \phi_i, v_1, \mu_1' \rangle \approx \langle \phi_i, v_2, \mu_2' \rangle : U_1$$
By Lemma 224, \( \mu_i \rightarrow \mu'_i \), and \( \mu'_1 \approx_{ol}^{k-j_1} \mu_2 \) then by Lemma 204, \( \langle \phi_1, \sigma_1, \mu'_1 \rangle \approx_{ol}^{k-j_1} \langle \phi_2, \sigma_2, \mu'_2 \rangle \). By induction hypotheses:

\[
\langle \phi_1, \sigma_1, \mu'_1 \rangle \approx_{ol}^{k-j_1} \langle \phi_2, \sigma_2(t^{U_2}), \mu'_2 \rangle : C(U_2)
\]

Again, consider \( j_2 = k - j_1 \), if after \( j_2 \) steps \( \sigma_1(t^{U_2}) \) is reducible or is a value, the result holds immediately. The interest case if after \( j_2' < j_2 \) steps \( \sigma_1(t^{U_2}) \) reduces to values \( v'_i \):\n
\[
\sigma_1(t^{U_2}) | \mu'_i \xrightarrow{\delta_i} j_2' v'_i | \mu''_i \implies \mu'_i \approx_{ol}^{k-j_1-j_2} \mu''_i \wedge \langle \phi_1, v'_1, \mu''_1 \rangle \approx_{ol}^{k-j_1-j_2} \langle \phi_2, v'_2, \mu''_2 \rangle : U_2
\]

Then

\[
\sigma_1(t^U) | \mu_i \xrightarrow{\delta_i} j_1 + j_2 \varepsilon_1 v_i := \varepsilon_2 v'_i | \mu''_i \wedge \mu_1 \approx_{ol}^{k-j_1-j_2} \mu''_i
\]

Now \( v_i \) and \( v'_i \) can be bare values or casted values. In the case of casted values we can combine evidence, which may fail with error. We assume that all evidence combinations succeed, otherwise the relation vacuously holds. As both values \( v_i \) are related at some reference type, then by canonical forms (Lemma 274), they both must be locations \( o^{U_i} \) for some \( U'_1 \approx U_1 \). We consider when the values are observable and the locations are identical (otherwise the result is trivial):

\[
\varepsilon_1 v_i := \varepsilon_2 v'_i | \mu''_i
\]

\[
\xrightarrow{\delta_i} 2 \varepsilon'_i o_{U'_1} := \varepsilon_{i'2} v'_i | \mu''_i
\]

\[
\xrightarrow{\delta_i} 1 \text{ unit}_{\approx} | \mu''_i
\]

Where \( \mu''_i = \mu''_i[o^{U'_1} \rightarrow \varepsilon''_i(u'_i \widetilde{\gamma} (\phi, g, \widetilde{\gamma} g)) : U'_1] \). As \( \langle \phi_1, v'_1, \mu''_1 \rangle \approx_{ol}^{k-j_1-j_2} \langle \phi_2, v'_2, \mu''_2 \rangle : U_2 \) then by Lemma 208, \( \langle \phi_1, \varepsilon''_1 u'_1 : U'_1, \mu''_1 \rangle \approx_{ol}^{k-j_1-j_2} \langle \phi_2, \varepsilon''_2 u'_2 : U'_1, \mu''_2 \rangle : U'_1 \). As \( \varepsilon'_1 \vdash \phi, g, \gamma g \sqsubseteq \text{label}(U'_1) \) and \( \varepsilon''_1 = \varepsilon''_{21} \widetilde{\gamma} \varepsilon'_1 \), by Lemma 221:

\[
\langle \phi_1, \varepsilon''_1 (u'_1 \widetilde{\gamma} (\phi, g, \widetilde{\gamma} g)) : U'_1, \mu''_1 \rangle \\
\approx_{ol}^{k-j_1-j_2} \langle \phi_2, \varepsilon''_2 (u'_2 \widetilde{\gamma} (\phi, g, \widetilde{\gamma} g)) : U'_1, \mu''_2 \rangle
\]

Also if \( \neg \text{obs}_{ol}(\phi_1) \Rightarrow \neg \text{obs}_{ol}(\phi'_1) \) and therefore by monotonicity of the join \( \neg \text{obs}_{ol}(\varepsilon''_1 \text{label}(U'_1)) \). Therefore if the values where different but context not observables, now the new values are going to be not observable as well, independently of the context. Then \( \forall, \phi''_1 \approx_{ol} \phi''_2 \):

\[
\approx_{ol}^{k-j_1-j_2} \langle \phi''_1, \varepsilon''_1 (u'_1 \widetilde{\gamma} (\phi, g, \widetilde{\gamma} g)) : U'_1, \mu''_1 \rangle \\
\approx_{ol}^{k-j_1-j_2} \langle \phi''_2, \varepsilon''_2 (u'_2 \widetilde{\gamma} (\phi, g, \widetilde{\gamma} g)) : U'_1, \mu''_2 \rangle
\]

As every values are related at type \text{Unit}, we only have to prove that \( \mu''_1 \approx_{ol}^{k-j_1-j_2-3} \mu''_2 \), but using monotonicity (Lemma 209), it is trivial to prove that because either both both stores update the same location \( o^{U'_1} \) to values that are related, therefore the result holds.
Case (ref). $t^U = \text{ref}_{\xi^1} u^U$. Then $U = \text{Ref}_1 U_1$.

By definition of substitution:

$$\sigma_1(t^U) = \text{ref}_{\xi^1} \varepsilon \sigma_1(u^U)$$

and Lemma 205

$$\phi'_1 \triangleright \text{ref}_{\xi^1} \varepsilon \sigma_1(u^U) \in \text{TERM}_{\text{Ref}_1 U_1}$$

We have to show that

$$\langle \phi_1, \text{ref}_{\xi^1} \varepsilon \sigma_1(u^U), \mu_1 \rangle \approx_{\text{ol}}^k \langle \phi_2, \text{ref}_{\xi^1} \varepsilon \sigma_2(u^U), \mu_2 \rangle : C(\text{Ref}_1 U_1)$$

By induction hypotheses:

$$\langle \phi_1, \sigma_1(u^U), \mu \rangle \approx_{\text{ol}}^j \langle \phi_2, \sigma_2(u^U), \mu \rangle : C(U'_1)$$

Consider $j < k$, by definition of related computations

$$\sigma_1(t^U) | \mu_i \xrightarrow{\phi'_1} j \approx_{\text{ol}}^i \mu'_1$$

If terms $t^U_i$ are reducible after $j = k - 1$ steps, then

$$\text{ref}_{\xi^1} \varepsilon \sigma_1(t^U) | \mu_i \xrightarrow{\phi'_1} j \approx_{\text{ol}}^i \mu'_1$$

If after at most $j$ steps $t^U_i$ is irreducible, it means that for some $j' \leq j \text{ ref}_{\xi^1} \varepsilon \sigma_1(t^U) | \mu_i \xrightarrow{\phi'_1} j' \approx_{\text{ol}}^i \mu'_1$. If $j' = j$ then we use the same same argument for reducible terms and the result holds.

Let us consider now $j' < j$. By Lemma 274, each $v_i$ is either a base value or a casted base value $\varepsilon_i u_i :: U'_1$. In case a value $v_{ij}$ is a casted value, then the whole term $\sigma_1(t^U)$ can take a step by $\text{Ref}_1 U'_1 \varepsilon_i v_i | \mu'_i$. Such a step either fails, or succeeds with a new combined evidence. Therefore, either:

$$\sigma_1(t^U) | \mu_i \xrightarrow{\phi'_1} j' \text{error}$$

in which case we do not care since we only consider termination-insensitive noninterference, or:

$$\sigma_1(t^U) | \mu \xrightarrow{\phi'_1} j' + 1 \approx_{\text{ol}}^i \text{ref}_{\xi^1} \varepsilon_i u_i | \mu'_i$$

with, $\mu''_i = \mu'_i[\xi^1 \mapsto \varepsilon'_i(u_i \bar{\gamma} \phi'_i g_c) :: U_1]$. Where $\varepsilon''_i = \varepsilon'_i \bar{\gamma} (\phi'_i \varepsilon \circ \varepsilon_i)$. We know that if $u_i \in \text{TERM}_{U_i}$, then $\varepsilon_i \vdash U_i \subseteq U_1$. Also, as $\langle \phi_1, v_1, \mu'_i \rangle \approx_{\text{ol}}^k \langle \phi_2, v_2, \mu'_2 \rangle : U'_1$ then by Lemma 208

$$\langle \phi_1, \varepsilon''_i u_i :: U_1, \mu'_i \rangle \approx_{\text{ol}}^k \langle \phi_2, \varepsilon'_2 u_2 :: U_1, \mu'_2 \rangle : U'_1$$

and by Lemma 211 and Lemma 209

$$\langle \phi_1, \varepsilon''_i (u_1 \bar{\gamma} \phi'_i g_c) :: U_1, \mu'_i \rangle \approx_{\text{ol}}^k \langle \phi_2, \varepsilon'_2 (u_2 \bar{\gamma} \phi'_i g_c) :: U_1, \mu'_2 \rangle : U'_1$$

Also if $\neg \text{obs}_{\text{ol}}(\phi_1) \Rightarrow \neg \text{obs}_{\text{ol}}(\phi'_i)$ and therefore by monotonicity of the join $\neg \text{obs}_{\text{ol}}(\varepsilon''_i \text{label}(U_1))$. Therefore if the values where different but context not observables, now the new values are going to be not observable as well, independently of the context. Then

$$\forall, \phi''_i \approx_{\text{ol}}^k \phi'_i, \langle \phi''_i, \varepsilon''_i (u_1 \bar{\gamma} \phi'_i g_c) :: U_1, \mu'_i \rangle \approx_{\text{ol}}^k \langle \phi'_i, \varepsilon' (u_2 \bar{\gamma} \phi'_i g_c) :: U_1, \mu'_2 \rangle : U'_1$$

By definition of related stores $\mu''_i \approx_{\text{ol}}^k \mu'_2$. Then by Monotonicity of the relation (Lemma 209)

$$\mu''_i \approx_{\text{ol}}^k \mu'_2$$

and the result holds.
Case (⊕). \( t^U = \varepsilon_1 t^{U_1} \oplus^g \varepsilon_2 t^{U_2} \)

By definition of substitution:
\[
\sigma_i(t^U) = \varepsilon_1 \sigma_i(t^{U_1}) \oplus^g \varepsilon_2 \sigma_i(t^{U_2})
\]
and Lemma 205:
\[\phi'_i \triangleright \varepsilon_1 \sigma_i(t^{U_1}) \oplus^g \varepsilon_2 \sigma_i(t^{U_2}) \in \text{TERM}_U\]

We use a similar argument to case := for reducible terms. The interest case is when we suppose some \( j_1 \) and \( j_2 \) such that \( j_1 + j_2 < k - 3 \) where:
\[
\sigma_i(t^{U_1}) \mid \mu_i \xrightarrow{\phi'_i} j_1 v_{i1} \mid \mu'_i \quad \mu'_i \approx_{\text{ol}}^{k-j_1} \mu_2' \land \left\langle \phi_1, v_{i1}, \mu_1' \right\rangle \approx_{\text{ol}}^{k-j_1} \left\langle \phi_2, v_{i21}, \mu_2' \right\rangle : U_1
\]
\[
\sigma_i(t^{U_2}) \mid \mu'_i \xrightarrow{\phi''_i} j_2 v_{i2} \mid \mu''_i \quad \mu''_i \approx_{\text{ol}}^{k-j_1-j_2} \mu_2'' \land \left\langle \phi_1, v_{i12}, \mu''_1 \right\rangle \approx_{\text{ol}}^{k-j_1-j_2} \left\langle \phi_2, v_{i22}, \mu_2'' \right\rangle : U_2
\]
By Lemma 274 each \( v_{ij} \) is either a boolean \((b_{ij})_{g_{ij}}\) or a casted boolean \( \varepsilon_{ij}(b_{ij})_{g'_{ij}} : U_j \). In case a value \( v_{ij} \) is a casted value, then the whole term \( \sigma_i(t^U) \) can take a step by \((Rg)\), combining \( \varepsilon_i \) with \( \varepsilon_{ij} \). Such a step either fails, or succeeds with a new combined evidence. Therefore, either:
\[
\sigma_i(t^U) \mid \mu_i \xrightarrow{\phi'_i} j_1 + j_2 \text{error}
\]
in which case we do not care since we only consider termination-insensitive noninterference, or:
\[
\xrightarrow{j_1 + j_2 + 2} \sigma_i(t^U) \mid \mu''_i \quad \varepsilon'_{i1}(b_{11})_{g'_{11}} \oplus^g \varepsilon'_{i2}(b_{12})_{g'_{12}} \mid \mu''_i
\]
\[
\xrightarrow{1} \varepsilon'_{i1}(b_{11})_{g'_{11}} : \text{Bool}_g \mid \mu''_i
\]
with \( b_i = b_{11} \oplus b_{12}, \varepsilon'_{i1} = \varepsilon'_{i1} \check{\gamma} \varepsilon'_{i2} \), and \( g'_{ij} = g'_{11} \check{\gamma} g'_{12} \). It remains to show that:
\[
\left\langle \phi_1, \varepsilon'_{i1}(b_{11})_{g'_{11}} : \text{Bool}_g, \mu''_i \right\rangle \approx_{\text{ol}}^{k-j_1-j_2-3} \left\langle \phi_2, \varepsilon'_{i2}(b_{12})_{g'_{12}} : \text{Bool}_g, \mu''_i \right\rangle : \text{Bool}_g
\]
If \( \neg \text{obs}_\text{ol}(\phi_i) \), then the result is trivial because the resulting booleans are also related as they are not observable.

If \( \text{obs}_\text{ol}(\phi_i) \), then by Lemma 208:
\[
\left\langle \phi_1, \varepsilon'_{i1}(b_{11})_{g'_{11}} : \text{Bool}_g, \mu''_i \right\rangle \approx_{\text{ol}}^{k} \left\langle \phi_2, \varepsilon'_{i2}(b_{12})_{g'_{12}} : \text{Bool}_g, \mu''_i \right\rangle.
\]
If \( \neg \text{obs}_\text{ol}(\text{ilbl}(\varepsilon'_{i1})_{g}) \) or \( \neg \text{obs}_\text{ol}(\text{ilbl}(\varepsilon'_{i2})_{g}) \), then by Lemma 218, \( \neg \text{obs}_\text{ol}(\varepsilon[i]_{g}) \) and the result holds.
If both \( \text{obs}_\text{ol}(\text{ilbl}(\varepsilon'_{i2})_{g}) \) then \( b_{11} = b_{21} \) and \( b_{12} = b_{22} \), so \( b_1 = b_2 \), and the result holds.

Case (app). \( t^U = \varepsilon_1 t^{U_1} \otimes_{\varepsilon_1}^{U_1} \otimes_{\varepsilon_2}^{U_1} \varepsilon_2 t^{U_2} \)

with \( \varepsilon_1 \vdash U_1 \subseteq S_11 \Rightarrow g, S_12, \varepsilon_2 \vdash U_2 \subseteq S_11, \) and \( U = U_12 \check{\gamma} g. \)

We omit the \( \otimes_{\varepsilon_1}^{U_1} \) operator in applications below.

316
By definition of substitution:
\[ \sigma_1(t^U) = \varepsilon_1 \sigma_1(t^{U_1}) \varepsilon_2 \sigma_1(t^{U_2}) \]
and Lemma 205:
\[ \phi'_1 \triangleright \varepsilon_1 \sigma_1(t^{U_1}) \varepsilon_2 \sigma_1(t^{U_2}) \in \text{TERM}_U \]
We use a similar argument to case \( c \) for reducible terms. The interest case is when we suppose some \( j_1 \) and \( j_2 \) such that \( j_1 + j_2 < k \) where by induction hypotheses and the definition of related computations:
\[ \sigma_1(t^{U_1}) | \mu_1 \xrightarrow{\phi'_1} j_1 v_{11} | \mu'_1 \implies \mu'_1 \approx_{\text{ol}}^{k-j_1} \mu_2' \land \langle \phi_1, v_{11}, \mu_1' \rangle \approx_{\text{ol}}^{k-j_1} \langle \phi_2, v_{21}, \mu_2' \rangle : U_1 \]
\[ \sigma_1(t^{U_2}) | \mu_1 \xrightarrow{\phi'_2} j_2 v_{12} | \mu''_1 \implies \mu''_1 \approx_{\text{ol}}^{k-j_1-j_2} \mu_2'' \land \langle \phi_1, v_{12}, \mu_1'' \rangle \approx_{\text{ol}}^{k-j_1-j_2} \langle \phi_2, v_{22}, \mu_2'' \rangle : U_2 \]
Then
\[ \sigma_1(t^U) | \mu_1 \xrightarrow{\phi'_1+j_1+j_2} \varepsilon_1 v_{11} \varepsilon_2 v_{12} | \mu''_1 \]
If \( \text{obs}_{\text{ol}}(\phi_1 \triangleright v_{11}) \) then, by definition of \( \approx_{\text{ol}} \) at values of function type, using \( \varepsilon_1 \) and \( \varepsilon_2 \) to justify the subtyping relations, we have:
\[ \langle \phi_1, (\varepsilon_1 v_{11} \varepsilon_2 v_{12}), \mu''_1 \rangle \approx_{\text{ol}}^{k-j_1-j_2} \langle \phi_2, (\varepsilon_1 v_{21} \varepsilon_2 v_{22}), \mu''_2 \rangle : C(U_1 \sim g) \]
Finally, by backward preservation of the relations (Lemma 206) the result holds.

If \( \neg \text{obs}_{\text{ol}}(\phi_1 \triangleright v_{11}) \), and we assume by canonical forms that \( v_{11} = \varepsilon_{11}(\lambda g'.x.t)_g :: U_1 \) and that \( v_{12} = \varepsilon_{12} u_{12} :: U_2 \) (and that evidence combination always succeed or the result holds immediately), then,
\[ \langle \varepsilon_{11} v_{11} \varepsilon_{2} v_{12}, \mu''_1 \rangle \xrightarrow{\phi'_1} \langle \varepsilon_{11}(\lambda g'.x.t), \varepsilon_{12} u_{12} \rangle | \mu''_1 \]
\[ \langle \varepsilon_{11} v_{11} \varepsilon_{2} v_{12}, \mu''_1 \rangle \xrightarrow{\phi'_2} \langle \text{prot}_{\text{ol},U_{12}}(\varepsilon_{11} g'_1 (\text{icod}(\varepsilon_{11} t'_1)) \mu''_1 \rangle | \mu''_1 \]
Where \( \varepsilon_{11} = \varepsilon_{11} \circ \varepsilon_1, \varepsilon_{12} = \varepsilon_{12} \circ \varepsilon_2, \) and \( \phi'_1 = (\varepsilon_{11}(\phi'_1 \sim g) \sim g_1), \phi'_2 = (\phi'_2 \sim g_2) \varepsilon_{11} \circ \varepsilon_1 \circ \varepsilon_2 \).

If \( \neg \text{obs}_{\text{ol}}(\phi_1) \) then \( \neg \text{obs}_{\text{ol}}(\phi'_1) \) and by Lemma 218 and 216, \( \neg \text{obs}_{\text{ol}}(\phi'_1) \). As \( \varepsilon_{11} = \varepsilon_{11} \circ \varepsilon_1, \) by Lemma 218 either both \( \text{icod}(\varepsilon_{11}) \) are observable or not (the latter when \( \neg \text{obs}_{\text{ol}}(\text{icod}Q_{\text{ol}}(\varepsilon_{11} \text{label}(U_1))) \)).

Finally by Lemma 222
\[ \langle \phi_1, \text{prot}_{\text{ol},U_{12}}(\varepsilon_{11} g'_1 (\text{icod}(\varepsilon_{11} t'_1)) \mu''_1 \rangle \approx_{\text{ol}}^{k-j_1-j_2} \langle \phi_2, \text{prot}_{\text{ol},U_{21}}(\varepsilon_{21} g'_2 (\text{icod}(\varepsilon_{21} t'_2)) \mu''_2 \rangle : C(U_{12} \sim g) \]
Finally, by backward preservation of the relations (Lemma 206) the result holds.
Case (if). \( t^U = \text{if}^g \, e_1 t^{U_1} \) then \( \varepsilon_2 t^{U_2} \) else \( \varepsilon_3 t^{U_3} \), with \( \phi_1^r \triangleright t^{U_1}, e_2' \in \text{TERM}_{U_1}, e_3' = \text{label}(U_1), e_4' = (\phi_r \triangleright \gamma \, \text{ilbl}(e_1)), \phi_2'' = (\varepsilon_2' \, (\phi_r \triangleright \gamma \, \gamma)) \) and \( \phi_3'' : t^{U_2} \in \text{TERM}_{U_2}, \phi_4'' \triangleright t^{U_3} \in \text{TERM}_{U_3} \) if \( U_1 \preceq \text{Bool}_{g'} \) and \( U = (U_2 \triangleright U_3) \triangleright \gamma \).

By definition of substitution:

\[
\sigma_i(t^U) = \text{if}^g \, \varepsilon_1 \sigma_i(t^{U_1}) \text{ then } \varepsilon_2 \sigma_i(t^{U_2}) \text{ else } \varepsilon_3 \sigma_i(t^{U_3})
\]

We use a similar argument to case := for reducible terms. The interest case is when we suppose some \( j_1 \) and \( j_2 \) such that \( j_1 + j_2 < k \) where by induction hypotheses and related computations we have that:

\[
\sigma_i(t^{U_1}) \mid \mu_i \xrightarrow{\phi_r} j_1 \nu_{i1} \mid \mu'_i \implies \mu'_i \approx_{ol} \mu_2' \land \langle \phi_1 \triangleright \nu_{i1}, \mu'_2 \rangle \approx_{ol} \langle \phi_2 \triangleright \nu_{i1}, \mu'_2 \rangle : U_1
\]

By Lemma 274, each \( \nu_{i1} \) is either a boolean \((b_{i1})_{g_1} \) or a casted boolean \((\varepsilon_{i1}(b_{i1}))_{g_1} \lift{g_1} U_1 \). In either case, \( U_1 \preceq \text{Bool}_{g_1} \) implies \( U_1 = \text{Bool}_{g_1} \). In case a value \( \nu_{i1} \) is a casted value, then the whole term \( \sigma_i(t^U) \) can take a step by \((Rg)\), combining \( \varepsilon_1 \) with \( \varepsilon_{i1} \). Such a step either fails, or succeeds with a new combined evidence. Therefore, either:

\[
\sigma_i(t^U) \mid \mu_i \xrightarrow{\phi_r} j_1+1 \text{error}
\]

in which case we do not care since we only consider termination-insensitive noninterference, or:

\[
\sigma_i(t^U) \mid \mu_i \xrightarrow{\phi_r} j_1+1 \text{if}^g \, \varepsilon_{i1}(b_{i1})_{g_1} \text{ then } \varepsilon_2 \sigma_i(t^{U_2}) \text{ else } \varepsilon_3 \sigma_i(t^{U_3}) \mid \mu'_i
\]

If \( \text{obs}_{ol}(\phi_1 \triangleright \nu_{i1}) \) does not hold, then by Lemma 226 \( \text{obs}_{ol}(\phi_1 \triangleright \nu_{i1} \lift{g_1} \text{Bool}_{g_1}) \) is not observable. Let us assume the worst case scenario and that both execution reduce via different branches of the conditional.

Consider \( \phi_1'' = ((\phi_r \triangleright \gamma \, \text{ilbl}(e_1'))(\phi_r \triangleright \gamma g_1'), (\phi_r \triangleright \gamma g)) \). It is easy to see that if \( \phi_1 \) is not observable, then as \( \leq_{ol} \phi_1 \) is not observable, and therefore by Lemma 218 \( \text{obs}_{ol}(\phi_{1''} \phi_{1''} g_r) \) does not hold. Therefore \( \phi_1 \leq_{ol} \phi_{1''} \). If \( \text{obs}_{ol}(\varepsilon_{i1} \text{Bool}_{g_1}) \) does not hold, then also by Lemma 218 \( \text{obs}_{ol}(\phi_{1''} \phi_{1''} g_r) \) does not hold as well. Then

\[
\sigma_1(t^U) \mid \mu_1 \xrightarrow{\phi_r} j_1+2 \text{prot}^{\gamma_{ilbl}(e_1')}_{g_1} \phi_1''(\sigma_2 \sigma_1(t^{U_2})) \mid \mu'_1
\]

\[
\sigma_2(t^U) \mid \mu_2 \xrightarrow{\phi_r} j_1+2 \text{prot}^{\gamma_{ilbl}(e_1')}_{g_1} \phi_2''(\sigma_3 \sigma_2(t^{U_3})) \mid \mu'_2
\]

But because \( \text{obs}_{ol}(\phi_{\triangleright \nu_{i1} \lift{g_1} \text{Bool}_{g_1}) \) does not hold then either \( \text{obs}_{ol}(\phi_r \phi_r g_r) \) or \( \text{obs}_{ol}(\text{ilbl}(\varepsilon_{i1} g)) \) does not hold. Then as \( \phi_1 \leq_{ol} \phi_{1''} \) by Lemma 222.

\[
\langle \phi_1, \text{prot}^{\gamma_{ilbl}(e_1')}_{g_1} \phi_1''(\sigma_2 \sigma_1(t^{U_2})), \mu'_1 \rangle \approx_{ol} \langle \phi_2, \text{prot}^{\gamma_{ilbl}(e_1')}_{g_1} \phi_2''(\sigma_3 \sigma_2(t^{U_3})), \mu'_2 \rangle
\]

and the result holds by backward preservation of the relations (Lemma 206).
Now consider if $\text{obs}_{ol}(\phi \triangleright \nu_{i1})$ holds, then $\text{obs}_{ol}(\phi \triangleright \nu_{i1} \mathbin{:} \text{Bool}_{g_1'})$ may hold or not. If its not observable we proceed like we just did for the non-observable case. Let us consider that $\text{obs}_{ol}(\phi \triangleright \nu_{i1} \mathbin{:} \text{Bool}_{g_1'})$ holds.

Then by definition of $\approx_{ol}$ on boolean values, $b_{11} = b_{21}$. Because $b_{11} = b_{21}$, both $\sigma_1(t_U)$ and $\sigma_2(t_U)$ step into the same branch of the conditional. Let us assume the condition is true (the other case is similar):

Then by induction hypotheses $\langle \phi_1, \sigma_1(t_U), \mu_1' \rangle \approx_{ol} k \langle \phi_2, \sigma_2(t_U), \mu_2' \rangle$, then as $\phi_i \leq_{ol} \phi''_i$, by Lemma 223

$$\langle \phi_1, \text{prot}_{\text{ilbl}}(\nu_{i1}) \phi_1' \nu_{i1} \sigma_1(t_U), \mu_1' \rangle \approx_{ol} k \langle \phi_2, \text{prot}_{\text{ilbl}}(\nu_{i1}) \phi_2' \nu_{i1} \sigma_2(t_U), \mu_2' \rangle$$

and the result holds by backward preservation of the relations (Lemma 206).

*Case (prot()).* Direct by using Lemma 223

\[ \Box \]
Appendix C

A Gradual interpretation of Union Types

C.1 Gradual Unions

In this section we present the full definition of GTFL⊕. §C.1.1 presents the full definition of the static language STFL. §C.1.2 presents some proofs about the meaning of gradual gradual unions. §C.1.3 presents the static semantics of GTFL⊕. §C.1.4 presents the dynamic semantics of GTFL⊕. Finally, §C.1.5 presents the proofs of properties of GTFL⊕ such as type safe and the dynamic gradual guarantee.

C.1.1 The Static Language: STFL

In this section we present the full definition of STFL. Figure C.1 presents the syntax and type system, and figure C.2 presents the dynamic semantics. §C.1.1 presents the proofs of type safety.

Properties

Lemma 228 (Substitution). If \( \Gamma, x : T_1 \vdash t : T \) and \( \Gamma \vdash v : T'_1 \) such that \( T'_1 = T_1 \), then \( \Gamma \vdash [v/x]t : T' \) such that \( T' = T \).

Proof. By induction on the derivation of \( \Gamma, x : T_1 \vdash t : T \).

Proposition 229 (\( \rightarrow \) is well defined). If \( \Gamma \vdash t : T \), \( t \rightarrow t' \) then, \( \Gamma \vdash t' : T' \), where \( T' = T \).

Proof.
$T \in \text{TYPE}, \ x \in \text{VAR}, \ t \in \text{TERM}, \ \Gamma \in \text{VAR \ fin \ TYPE}$

$T ::= \text{Bool} \mid \text{Int} \mid T \rightarrow T$ (types)

$v ::= n \mid b \mid (\lambda x : T.t)$ (values)

$t ::= v \mid x \mid t t \mid t + t \mid \text{if } t \text{ then } t \text{ else } t : T$ (terms)

\[ \begin{align*}
\text{(Tx)} & \quad \Gamma \vdash x : T \\
\text{(Tb)} & \quad \Gamma \vdash b : \text{Bool} \\
\text{(Tn)} & \quad \Gamma \vdash n : \text{Int} \\
\text{(T\lambda)} & \quad \Gamma, x : T_1 \vdash t : T_2 \\
& \quad \Gamma; \Sigma \vdash (\lambda x : T_1.t) : T_1 \rightarrow T_2 \\
\text{(Tif)} & \quad \Gamma \vdash t_1 : T_1 \\
& \quad \Gamma \vdash t_2 : T_2 \\
& \quad T_2 = \text{dom}(T_1) \\
& \quad \Gamma \vdash t_1 \ t_2 : \text{cod}(T_1) \\
\text{(Tapp)} & \quad \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{equate}(T_2, T_3) \\
\text{(T+)} & \quad \Gamma \vdash t_1 : T_1 \\
& \quad T_1 = \text{Int} \\
\text{(T\)} & \quad \Gamma \vdash t_1 \ t_2 : T_2 \\
& \quad T_2 = \text{Int} \\
\text{Dom} : \text{TYPE} \rightarrow \text{TYPE} & \quad \text{Cod} : \text{TYPE} \rightarrow \text{TYPE} & \quad \text{Equate} : \text{TYPE}^2 \rightarrow \text{TYPE} \\
\text{Dom}(T_1 \rightarrow T_2) = T_1 & \quad \text{Cod}(T_1 \rightarrow T_2) = T_2 & \quad \text{Equate}(T, T) = T \\
\text{Dom}(T) \text{ undefined o.w.} & \quad \text{Cod}(T) \text{ undefined o.w.} & \quad \text{Equate}(T_1, T_2) \text{ undefined o.w.} \\
\end{align*} \]

Figure C.1: STFL: Syntax and Type System

Case (T+). Then $t = b_1 + b_2$ and

\[ \begin{align*}
\text{(T+)} & \quad \Gamma \vdash n_1 : T_1 \\
\text{(T\)} & \quad \Gamma \vdash n_2 : T_2 \\
\text{(T\)} & \quad T_1 = \text{Int} \\
\text{(T\)} & \quad T_2 = \text{Int} \\
\text{Then} & \quad n_1 + n_2 \rightarrow (n_1 [+] n_2) \\
\end{align*} \]

But $\Gamma \vdash (n_1 [+] n_2) : \text{Int}$ and the result holds.

Case (Tapp). Then $t = (\lambda x : T_{11}.t_1) v$, suppose $\Gamma \vdash (\lambda x : T_{11}.t_1) : T_1$, and $\text{dom}(T_1) = T_{11}$ and $\text{cod}(T_1) = T_{12}$. Therefore

\[ \begin{align*}
\text{(T\lambda)} & \quad \Gamma, x : T_{11} \vdash t : T_{12} \\
\text{(T\)} & \quad \Gamma \vdash (\lambda x : T_{11}.t_1) : T_{11} \rightarrow T_{12} \\
\text{(Tapp)} & \quad \Gamma \vdash v : T_2 \\
\text{Then} & \quad (\lambda x : T_{11}.t_1) v \rightarrow [v/x]t_1 \\
\end{align*} \]

By Lemma 228, $\Gamma \vdash [v/x]t_1 : T_{12}'$, where $T_{12}' = T_{12}$, and the result holds.
\[ v ::= n \mid \lambda x.t \mid \text{true} \mid \text{false} \mid v :: T \quad \text{(values)} \]

\[ f ::= \square + t \mid v + \square \mid \square t \mid v \square \mid \square :: T \quad \text{(frames)} \]

Notions of Reduction

\[ n_1 + n_2 \rightarrow n_3 \quad \text{where} \quad n_3 = n_1 \mathbb{[+] n}_2 \]

\[ (\lambda x.t) v \rightarrow ([v/x]t) \]

\[ \text{if true then } t \text{ else } t_2 \rightarrow t_1 \]

\[ \text{if false then } t_1 \text{ else } t_2 \rightarrow t_2 \]

\[ v :: T \rightarrow v \]

Reduction

\[
\begin{array}{c}
\frac{t_1 \rightarrow t_2}{t_1 \mapsto t_2} & \quad \frac{t_1 \mapsto t_2}{f[t_1] \mapsto f[t_2]} \\
\end{array}
\]

Figure C.2: STFL: Dynamic Semantics

Case (Tif-true). Then \( t = \text{if true then } t_1 \text{ else } t_2 \) and

\[
\begin{array}{c}
\frac{D_0}{\Gamma \vdash \text{true} : \text{Bool}} & \quad \frac{D_1}{\Gamma \vdash t_1 : T_1} & \quad \frac{D_2}{\Gamma \vdash t_2 : T_2} \\
\end{array}
\]

Then if

\[ \text{if true then } t_1 \text{ else } t_2 \rightarrow t_1 \]

But

\[ \frac{D_1}{\Gamma \vdash t_1 : T_1} \]

and by definition of the \textit{equate} operator, \( T_1 = \text{equate}(T_1, T_2) \) and the result holds.

Case (Tif-false). Analogous to case (if-true).

Case (T::). Then \( t = v :: T \) and

\[
\begin{array}{c}
\frac{D}{\Gamma \vdash v : T_1} & \quad T_1 = T \\
\end{array}
\]

Then

\[ v :: T \rightarrow v \]

But \( T_1 = T \) and the result holds.

Proposition 230 (Canonical forms). Consider a value \( v \) such that \( \cdot \vdash v : T \). Then:

1. If \( T = \text{Bool} \) then \( v = b \) for some \( b \).
2. If $T = \text{Int}$ then $v = n$ for some $n$.

3. If $T = T_1 \rightarrow T_2$ then $v = (\lambda x : T_1.t_2)$ for some $t_2$.

Proof. By inspection of the type derivation rules. \hfill \qed

Lemma 231. Consider frame $f$, and term $t_1$, such as $\Gamma \vdash t_1 : T_1$ and $\Gamma \vdash f[t_1] : T$. Consider term $t'_1$, such that $\Gamma \vdash t'_1 : T'_1$ and $T'_1 = T_1$. Then $\Gamma \vdash f[t'_1] : T'$ such that $T' = T$.

Proof. By induction on the derivation of $f[t_1]$.

Case ($\square t$). Then $f = \square t_2$, $f[t_1] = t_1 t_2$ and

\[
\begin{array}{c}
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_2 = \text{dom}(T_1)}{
\Gamma \vdash t_1 t_2 : \text{cod}(T_1)}
\end{array}
\]

then $f[t'_1] = t'_1 t_2$. But as $T'_1 = T_1$ then $\text{cod}(T'_1) = \text{cod}(T_1)$ and $\text{dom}(T_1) = \text{dom}(T'_1)$. Therefore

\[
\begin{array}{c}
\frac{\Gamma \vdash t_1 : T'_1 \quad \Gamma \vdash t_2 : T_2 \quad T_2 = \text{dom}(T'_1)}{
\Gamma \vdash t'_1 t_2 : \text{cod}(T'_1)}
\end{array}
\]

and the result holds.

Case ($v \square, \square + t, v + \square$, if $\square$ then $t$ else $t$). Analogous to ($\square t$) \hfill \qed

Proposition 232 ($\rightarrow$ is well defined). If $\Gamma \vdash t : T$, $t \rightarrow t'$ then, $\Gamma \vdash t' : T'$, where $T' = T$.

Proof. By induction on the structure of a derivation of $t \rightarrow t'$.

Case ($\rightarrow$). Then $t \rightarrow t'$. By well-definedness of $\rightarrow$ (Prop 229), $\Gamma \vdash t' : T'$, where $T' = T$, and the result holds immediately.

Case ($f \square$). Then $t = f[t_1]$, $\Gamma \vdash t_1 : T_1$ and $t_1 \rightarrow t_2$. By induction hypothesis $\Gamma \vdash t_2 : T_2$ where $T_2 = T_1$. By Lemma 231, $\Gamma \vdash f[t_2] : T'$ such that $T' = T$, and the result holds immediately. \hfill \qed

Proposition 233 (Safety). If $\Gamma \vdash t : T$, then one of the following is true:

- $t$ is a value $v$;
- $t \rightarrow t'$, and $\Gamma \vdash t' : T'$ where $T' = T$.

Proof. By induction on the structure of $\Gamma \vdash t : T$.

Case (Tb, Tn, Tl, Tl). $t$ is a value.
Case $(T+)$: Then $t = t_1 + t_2$ and

\[
\begin{array}{c}
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 = \text{Int} \quad T_2 = \text{Int}}{\Gamma \vdash t_1 + t_2 : \text{Int}}
\end{array}
\]

By induction hypotheses, one of the following holds:

1. $t_1$ is a value. Then by induction on $t_2$ one of the following holds:
   
   (a) $t_2$ is a value. Then by Canonical Forms (Lemma 230)
   
   \[
   \frac{t \longrightarrow t'}{t \mapsto t'}
   \]

   and by Prop 229, $\Gamma \vdash t' : T'$, where $T' = T$, and therefore the result holds.

   (b) $t_2 \mapsto t'_2$. Then by induction hypothesis, $\Gamma \vdash t_2 : T'_2$, where $T'_2 = T_2$.

   Then by $(Tf)$, using $f = v + \Box$, $t \mapsto t_1 + t'_2$ and by Lemma 231, $\Gamma \vdash t_1 + t'_2 : \text{Int}$ and the result holds.

2. $t_1 \mapsto t'_1$. Then by induction hypotheses, $\Gamma' \vdash t'_1 : \text{Int}$. Then by $(Tf)$, using $f = \Box + t_2$, $t \mapsto t'_1 + t_2$ and by Lemma 231, $\Gamma \vdash t'_1 + t_2 : \text{Int}$ and the result holds.

Case $(\text{Sapp})$: Then $t = t_1 \cdot t_2$, $T = T_{12}$ and

\[
\begin{array}{c}
\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_2 = \text{dom}(T_1)}{\Gamma \vdash t_1 \cdot t_2 : \text{cod}(T_1)}
\end{array}
\]

By induction hypotheses, one of the following holds:

1. $t_1$ is a value. Then by Canonical Forms (Lemma 230), and induction on $t_2$ one of the following holds:
   
   (a) $t_2$ is a value. Then by Canonical Forms (Lemma 230)
   
   \[
   \frac{t \longrightarrow t'}{t \mapsto t'}
   \]

   and by Prop 229, $\Gamma \vdash t' : T'$, where $T' = T$, and therefore the result holds.

   (b) $t_2 \mapsto t'_2$. Then by induction hypothesis, $\Gamma \vdash t_2 : T'_2$, where $T'_2 = T_2$.

   Then by $(Tf)$, using $f = v \cdot \Box$, $t \mapsto t_1 \cdot t'_2$ and by Lemma 231, $\Gamma \vdash t_1 \cdot t'_2 : T'_{12}$ where $T'_{12} = \text{cod}(T_1)$ and the result holds.

2. $t_1 \mapsto t'_1$. Then by induction hypotheses, $\Gamma' \vdash t'_1 : T'_{11} \longrightarrow T'_{12}$ where $T'_{11} \longrightarrow T'_{12} = T_1$.

   Then by $(Tf)$, using $f = \Box \cdot t_2$, $t \mapsto t'_1 t_2$ and by Lemma 231, $\Gamma \vdash t'_1 t_2 : T'_{12}$ where $T'_{12} = \text{cod}(T_1)$ and the result holds.

Case $(\text{Tif})$: Then $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ and

\[
\begin{array}{c}
\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{equate}(T_2, T_3)}
\end{array}
\]

By induction hypotheses, one of the following holds:
1. \( t_1 \) is a value. Then by Canonical Forms (Lemma 230)

\[
(R \rightarrow) \quad \frac{t \rightarrow t'}{t \rightarrow t'}
\]

and by Prop 229 \( \Gamma \vdash t' : T' \), where \( T' = T \), and therefore the result holds.

2. \( t_1 \mapsto t'_1 \). Then by induction hypothesis, \( \Gamma \vdash t'_1 : T'_1 \), where \( T'_1 = T_1 \). Therefore \( T'_1 = \text{Bool} \). Then by \((Tf)\), using \( f = \text{if} \ \Box \ \text{then} \ t \ \text{else} \ t' \), \( t \mapsto t'_1 \text{then} \ t_2 \text{else} \ t_3 \) and by Lemma 231 \( \Gamma \vdash \text{if} \ t'_1 \text{then} \ t_2 \text{else} \ t_3 : T' \) where \( T' = \text{equate}(T_2, T_3) \) and the result holds.

Case (T::). Then \( t = t_1 :: T_2 \) and

\[
(T::) \quad \frac{\Gamma \vdash t_1 : T_1 \quad T_1 = T_2}{\Gamma \vdash t_1 :: T_2 : T_2}
\]

By induction hypotheses, one of the following holds:

1. \( t_1 \) is a value. Then

\[
(R \rightarrow) \quad \frac{t \rightarrow t'}{t \rightarrow t'}
\]

and by Prop 229 \( \Gamma \vdash t' : T' \), where \( T' = T \), and therefore the result holds.

2. \( t_1 \mapsto t'_1 \). Then by induction hypothesis, \( \Gamma \vdash t'_1 : T'_1 \), where \( T'_1 = T_1 \). Then by \((Tf)\), using \( f = \Box :: T \), \( t \mapsto t'_1 :: T_2 \), but \( \Gamma \vdash t'_1 :: T_2 : T_2 \) and the result holds.

\[\square\]

C.1.2 Syntax and Meaning of Gradual Unions

Proposition 32 (\( \alpha \gamma \) is Sound and Optimal). If \( \widehat{T} \) is not empty, then

\( (a) \) \( \widehat{T} \subseteq \gamma(\alpha(\widehat{T})) \).

\( (b) \) \( \widehat{T} \subseteq \gamma(G) \Rightarrow \alpha(\widehat{T}) \subseteq G \).

Proof. We proceed by induction on the structure of \( U \). Let us start by proving a).

Case (\{ Int \}). Then \( \alpha(\{ \text{Int} \}) = \text{Int} \). But \( \gamma(\text{Int}) = \{ \text{Int} \} \) and the result holds.

Case (\{ Bool \}). Analogous to case \{ Int \}.

Case (\( \widehat{T}_1 \Rightarrow \widehat{T}_2 \)). Then \( \alpha(\widehat{T}_1 \Rightarrow \widehat{T}_2) = \alpha(\widehat{T}_1) \Rightarrow \alpha(\widehat{T}_2) \). But by definition of \( \gamma, \gamma(\alpha(\widehat{T}_1) \Rightarrow \gamma(\alpha(\widehat{T}_1))) \Rightarrow \gamma(\alpha(\widehat{T}_2))) \). By induction hypotheses, \( \widehat{T}_1 \subseteq \gamma(\alpha(\widehat{T}_1)) \) and \( \widehat{T}_2 \subseteq \gamma(\alpha(\widehat{T}_2)) \), therefore \( \widehat{T}_1 \Rightarrow \widehat{T}_2 \subseteq \gamma(\alpha(\widehat{T}_1)) \Rightarrow \alpha(\widehat{T}_2) \) and the result holds.

Case (\( \emptyset \)). This case cannot happen because \( \alpha \gamma \) is restricted to non-empty sets of gradual intermediate types.

Case (\( \widehat{T} \)). Then \( \alpha(\widehat{T}) = ? \) and therefore \( \gamma(\alpha(\widehat{T})) = \text{TYPE} \), but \( \widehat{T} \subseteq \text{TYPE} \) and the result holds.

Now let us prove b).
Case (Int). Trivial because $\gamma_\text{Int}(\text{Int}) = \{\text{Int}\}$, and $\alpha_{\oplus}(\{\text{Int}\}) = \text{Int}$.

Case (Bool). Analogous to case Int.

Case ($G_1 \to G_2$). We have to prove that $\gamma_\tau(\alpha_\tau(\hat{T})) \subseteq \gamma_\tau(G_1 \to G_2)$. But we know that $\hat{T} \subseteq \gamma_\tau(G_1 \to G_2) = \gamma_\tau(G_1) \to \gamma_\tau(G_2)$, therefore $\hat{T}$ has the form $\{T_{i1} \to T_{i2}\}$, for some $\{T_{i1}\} \subseteq \gamma_\tau(G_1)$ and $\{T_{i2}\} \subseteq \gamma_\tau(G_2)$. But by definition of $\alpha_\tau$, $\alpha_\tau(\{T_{i1} \to T_{i2}\}) = \alpha_\tau(\{T_{i1}\}) \to \alpha_\tau(\{T_{i2}\})$ and therefore $\gamma_\tau(\alpha_\tau(\{T_{i1}\})) \to \gamma_\tau(\alpha_\tau(\{T_{i2}\})) = \gamma_\tau(\alpha_\tau(\{T_{i1}\})) \to \gamma_\tau(\alpha_\tau(\{T_{i2}\}))$. But by induction hypotheses $\gamma_\tau(\alpha_\tau(\{T_{i1}\})) \subseteq \gamma_\tau(G_1)$ and $\gamma_\tau(\alpha_\tau(\{T_{i2}\})) \subseteq \gamma_\tau(G_2)$ and the result holds.

Case (?). Then we have to prove that $\gamma_\tau(\alpha_\tau(\hat{T})) \subseteq \gamma_\tau(?) = \text{Type}$, but this is always true and the result holds immediately.

\[\square\]

**Proposition 33** ($\alpha_\gamma$ is Sound and Optimal). If $\hat{T}$ is not empty, then

\[a) \hat{T} \subseteq \gamma_\tau(\alpha_\gamma(\hat{T})). \quad \text{b) } \hat{T} \subseteq \gamma_\tau(G) \Rightarrow \alpha_\gamma(\hat{T}) \subseteq \hat{G}.\]

**Proof.** We start proving a). By definition of $\alpha_\gamma$, $\alpha_\gamma(\hat{T}) = \bigcup_{\hat{T} \in \hat{T}} \alpha_\tau(\hat{T})$. And by definition of $\hat{T}$, $\gamma_\tau(\alpha_\gamma(\hat{T})) = \{\gamma_\tau(\alpha_\tau(\hat{T})) \mid \hat{T} \in \hat{T}\}$. We have to prove that $\forall \hat{T} \in \hat{T}, \exists \hat{T}' \in \gamma_\tau(\alpha_\gamma(\hat{T}))$ such that $\hat{T} \subseteq \hat{T}'$. But now $\hat{T}$ is sound therefore $\forall \hat{T} \in \hat{T}, \hat{T} \subseteq \gamma_\tau(\alpha_\tau(\hat{T}))$, and the result holds immediately.

Now we prove b). We know that $\forall \hat{T} \in \hat{T}, \exists \hat{T}' \in \gamma_\tau(\hat{G})$ such that $\hat{T} \subseteq \hat{T}'$. We have to prove that $\gamma_\tau(\alpha_\gamma(\hat{T})) \subseteq \gamma_\tau(\hat{G})$. But $\gamma_\tau(\alpha_\gamma(\hat{T})) = \{\gamma_\tau(\alpha_\tau(\hat{T})) \mid \hat{T} \in \hat{T}\}$. But as $\alpha_\tau$ is optimal, $\forall$ non-empty $\hat{T}$, if $\hat{T} \subseteq G$ then $\gamma_\tau(\alpha_\tau(\hat{T})) \subseteq \gamma_\tau(G)$. Then we know that $\forall \hat{T} \in \hat{T}, \gamma_\tau(\alpha_\tau(\hat{T})) \subseteq \hat{T}'$, but $\hat{T}' \in \gamma_\tau(G)$ and the result holds.

\[\square\]

**Proposition 34** ($\alpha_{\oplus}$ is Sound and Optimal). If $\hat{G}$ is not empty, then

\[a) \hat{G} \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\hat{G})). \quad \text{b) } \hat{G} \subseteq \gamma_{\oplus}(U) \Rightarrow \alpha_{\oplus}(\hat{G}) \subseteq U.\]

**Proof.** We proceed by induction on the structure of $U$. Let us start by proving a).

Case ($\{\text{Int}\}$). Then $\alpha_{\oplus}(\{\text{Int}\}) = \text{Int}$. But $\gamma_{\oplus}(\text{Int}) = \{\text{Int}\}$ and the result holds.

Case ($\{\text{Bool}\}$). Analogous to case $\{\text{Int}\}$.

Case ($\{?\}$). Analogous to case $\{\text{Int}\}$.

Case ($\emptyset$). This case cannot happen because $\alpha_{\oplus}$ is restricted to non-empty sets of gradual intermediate types.

Case ($\hat{T}_1 \cup \hat{T}_2$). Then $\alpha_{\oplus}(\hat{T}_1 \cup \hat{T}_2) = \alpha_{\oplus}(\hat{T}_1) \oplus \alpha_{\oplus}(\hat{T}_2)$. But by definition of $\gamma_{\oplus}$, $\gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_1) \oplus \alpha_{\oplus}(\hat{T}_2)) = \gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_1)) \cup \gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_2))$. By induction hypotheses, $\hat{T}_1 \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_1))$ and $\hat{T}_2 \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_2))$, therefore $\hat{T}_1 \cup \hat{T}_2 \subseteq \gamma_{\oplus}(\alpha_{\oplus}(\hat{T}_1) \oplus \alpha_{\oplus}(\hat{T}_2))$ and the result holds.

Now let us prove b).

Case (Int). Trivial because $\gamma_{\oplus}(\text{Int}) = \{\text{Int}\}$, and $\alpha_{\oplus}(\{\text{Int}\}) = \text{Int}$.

326
Case (Bool). Analogous to case Int.

Case (?). Analogous to case ?.

Case \((U_1 \rightarrow U_2)\). We have to prove that \(\gamma_\oplus(\alpha_\oplus(\overline{G})) \subseteq \gamma_\oplus(U_1 \rightarrow U_2)\). But \(\gamma_\oplus(\alpha_\oplus(\overline{G})) = \gamma_\oplus(\overline{G}) = \overline{G}\) and the result follows.

Case \((U_1 \oplus U_2)\). Then \(\gamma_\oplus(U_1 \oplus U_2) = \gamma_\oplus(U_1) \cup \gamma_\oplus(U_2)\) and \(G = G_1 \cup G_2\) for some \(G_1\) and \(G_2\) such that \(G_1 \subseteq \gamma_\oplus(U_1)\) and \(G_2 \subseteq \gamma_\oplus(U_2)\). By definition of \(\alpha_\oplus\), \(\alpha_\oplus(G) = \alpha_\oplus(G_1) \cup \alpha_\oplus(G_2)\). By induction hypotheses, \(\gamma_\oplus(\alpha_\oplus(G_1)) \subseteq \gamma_\oplus(U_1)\) and \(\gamma_\oplus(\alpha_\oplus(G_2)) \subseteq \gamma_\oplus(U_2)\), therefore as \(\gamma_\oplus(G) = \gamma_\oplus(\alpha_\oplus(G_1)) \cup \gamma_\oplus(\alpha_\oplus(G_2))\), then \(\gamma_\oplus(G) \subseteq \gamma_\oplus(U_1) \cup \gamma_\oplus(U_2) = \gamma_\oplus(U)\) and the result holds.

\(\square\)

**Proposition 35** (\(\alpha\) is Sound and Optimal). If \(\widehat{T}\) is not empty, then

\[a) \widehat{T} \subseteq \gamma(\alpha(\widehat{T})). \quad b) \widehat{T} \subseteq \gamma(U) \Rightarrow \alpha(\widehat{T}) \subseteq U.\]

**Proof.** By propositions 34 and 33 and composition of sound and optimal abstractions. \(\square\)

### C.1.3 Static Semantics of GTFL\(^{\oplus}\)

This section presents the full static semantics of GTFL\(^{\oplus}\) in figure C.3. This section formally justifies the compositional lifting in § C.1.3. This section also presents the inductive definitions in § C.1.3, the proof of Static Gradual Guarantee for GTFL\(^{\oplus}\) in § C.1.3 and figure C.4 presents some type derivations of the examples in § 5.2.3

#### Consistent Predicates and Functions

**Proposition 36.** \(\overline{P}(U_1, U_2) \iff \exists G_1 \in \gamma_\oplus(U_1), \exists G_2 \in \gamma_\oplus(U_2), \overline{P}_\gamma(G_1, G_2)\)

where \(\overline{P}\) is the predicate \(P\) lifted with \(\gamma_\oplus\).

**Proof.** By definition of lifting predicates, \(\overline{P}(U_1, U_2) = \exists \overline{\gamma}_1 \in \gamma(U_1), \exists \overline{T}_1 \in \overline{\gamma}_1, \overline{\exists \gamma}_2 \in \gamma(U_2), \exists \overline{T}_2 \in \overline{\gamma}_2\). \(P(T_1, T_2)\).

If we unfold \(\overline{P}_\gamma(G_1, G_2)\), then \(\exists G_1 \in \gamma_\oplus(U_1), \exists G_2 \in \gamma_\oplus(U_2), \overline{P}_\gamma(G_1, G_2) \iff \exists G_1 \in \gamma_\oplus(U_1), \exists G_2 \in \gamma_\oplus(U_2), \exists T_1 \in \gamma_\oplus(G_1), \exists T_2 \in \gamma_\oplus(G_2)\). \(P(T_1, T_2)\).

We start with direction \(\Rightarrow\). As \(\gamma = \overline{\gamma}_\oplus \circ \gamma_\oplus\), then \(\gamma(U_1) = \overline{\gamma}_\oplus(\overline{G}_1)\) and \(\gamma(U_2) = \overline{\gamma}_\oplus(\overline{G}_2)\). As \(\overline{T}_1 \in \overline{\gamma}_\oplus(\overline{G}_1)\) then \(\overline{T}_1 \in \gamma_\oplus(G_1)\) for some \(G_1' \in \overline{G}_1\), and also \(\overline{T}_2 \in \overline{\gamma}_\oplus(\overline{G}_2)\) then \(\overline{T}_2 \in \gamma_\oplus(G_2)\) for some \(G_2' \in \overline{G}_2\). Then we can always choose \(G_1 = G_1'\) and \(G_2 = G_2'\) and the result holds.

Then we prove direction \(\Leftarrow\). As \(G_1 \in \gamma_\oplus(U_1)\) and \(T_1 \in \gamma_\oplus(G_1)\), then its easy to see that \(T_1 \in (\overline{\gamma}_\oplus \circ \gamma_\oplus)(U_1)\) as we know that \(G_1 \in \gamma_\oplus(U_1)\) and \((\overline{\gamma}_\oplus \circ \gamma_\oplus)(U_1) = \bigcup_{G \in \gamma_\oplus(U_1)} \gamma_\oplus(G)\). Analogous \(T_2 \in (\overline{\gamma}_\oplus \circ \gamma_\oplus)(U_2)\), but \(\gamma = \overline{\gamma}_\oplus \circ \gamma_\oplus\) and the result holds. \(\square\)
\[
U \in \text{UType}, \quad x \in \text{Var}, \quad \bar{t} \in \text{UTerm}, \Gamma \in \text{Var}^{\text{fin}} \text{ UType}
\]

\[
\begin{align*}
U & ::= U \oplus U \mid \text{Int} \mid \text{Bool} \mid U \to U \quad \text{(types)} \\
v & ::= n \mid \text{true} \mid \text{false} \mid (\lambda x : U \bar{t}) \quad \text{(values)} \\
\bar{t} & ::= v \mid x \mid \bar{t} \bar{t} \mid \bar{t} + \bar{t} \mid \text{if} \bar{t} \text{ then } \bar{t} \text{ else } \bar{t} \mid \bar{t} :: U \quad \text{(terms)}
\end{align*}
\]

\[\Gamma \vdash x : U \quad (Ux) \]
\[\Gamma, x : U_1 \vdash \bar{t} : U_2 \quad (U\lambda) \]
\[\Gamma \vdash \bar{t}_1 : U_1 \quad (Uapp) \]
\[\Gamma \vdash \bar{t}_2 : U_2 \quad U_2 \sim \text{dom}(U_1) \]
\[\Gamma \vdash \bar{t}_1 \bar{t}_2 : \text{cod}(U_1) \]
\[\Gamma \vdash \bar{t}_1 : U_1 \quad U_1 \sim \text{Bool} \quad \Gamma \vdash \bar{t}_2 : U_2 \quad \Gamma \vdash \bar{t}_3 : U_3 \quad (Uif) \]
\[\Gamma \vdash \text{if} \bar{t}_1 \text{ then } \bar{t}_2 \text{ else } \bar{t}_3 : U_2 \cap U_3 \]
\[\text{dom} : \text{UType} \to \text{UType} \quad \text{cod} : \text{UType} \to \text{UType} \]
\[\text{dom}(U) = \alpha(\text{dom}(\gamma(U))) \quad \text{cod}(U) = \alpha(\text{cod}(\gamma(U))) \]

\[\text{Figure C.3: GTFL}^\alpha: \text{Syntax and Type System} \]

**Proposition 234.** \(U_1 \sim U_2 \iff \exists G_1 \in \gamma(\beta(U_1)), \exists G_2 \in \gamma(\beta(U_2)), G_1 \sim \gamma G_2\) where \(\sim \gamma\) is the classic consistency operator defined in [44].

**Proof.** Direct by proposition [36]. \(\square\)

When a function \(f\) receives more than one argument, then the lifting is the pointwise application of \(f\) to every combinations of elements of each set of each powerset. Formally:

**Definition 109** (Lifting of type functions). Let \(f : \text{TYPE}^n \to \text{TYPE}\), then \(\bar{f} : \text{UType}^n \to \text{UType}\) is defined as:

\[
\bar{f}(U_1, ..., U_2) = \alpha(\{ f(T_1, ..., T_n) \mid T_1 \in \gamma_\beta(G_1), ..., T_n \in \gamma_\beta(G_n) \})
\]

\[G_1 \in \gamma_\beta(U_1), ..., G_n \in \gamma_\beta(U_n) \}

Note that we can also define the lifting of a function using its intermediate lifting:

**Proposition 235.** Let \(\bar{f} : \text{UType}^n \to \text{UType}\) and \(\bar{f}_\tilde{n} : \text{UType}^n \to \text{UType}\) then,

\[
\bar{f}(U_1, ..., U_2) \iff \alpha_\beta \circ \hat{\bar{f}} \circ \gamma_\beta(U_1, ..., U_n) \equiv \alpha_\beta(\{ \bar{f}(G_1, ..., G_n) \mid G_1 \in \gamma_\beta(U_1), ..., G_n \in \gamma_\beta(U_n) \})
\]

328
where $\tilde{f}_f$ is the pointwise application of $f_f$, and $f_f$ is the lifting of $f$ in the intermediate abstraction.

**Proof.** Trivial by unfolding of the definition of $\tilde{f}_f(G_1, \ldots, G_n)$. \qed

**Proposition 236.** Let $F : \text{TYPE} \to \text{TYPE}$ be a partial function, and define the predicate $P(T_1, T_2) \equiv T_1 = F(T_2)$. Then $\tilde{P}(U_1, U_2)$ implies $U_1 \sim \tilde{F}(U_2)$.

**Proof.** Suppose $\tilde{P}(U_1, U_2)$. Then $T_1 = F(T_2)$ for some $T_1 \in \tilde{T}_1, T_1 \in \gamma(U_1)$ and $T_2 \in \tilde{T}_2, T_2 \in \gamma(U_2)$. Therefore $F(T_2) \in \tilde{T}_F, \tilde{T}_F \in \tilde{F}(\gamma(U_2))$ and by Prop. 35 $\tilde{F}(\gamma(U_2)) \subseteq \gamma(\alpha(\tilde{F}(\gamma(U_2))))$. But $\tilde{F}(U_2) = \alpha(\tilde{F}(\gamma(U_2)))$, so $F(T_2) \in \tilde{T}_F, \tilde{T}_F \in \gamma(\alpha(\tilde{F}(\gamma(U_2)))) = \gamma(\tilde{F}(U_2))$. Then by definition of consistency, we can choose $(T_1, F(T_2)) \in (\tilde{T}_1, \tilde{T}_F), (\tilde{T}_1, \tilde{T}_F), \in \gamma(U_1, F(U_2))$ such that $T_1 = F(T_2)$, therefore $U_1 \sim \tilde{F}(U_2)$. \qed

**Proposition 237.** Let $P(T_1, T_2) \equiv T_1 = \text{dom}(T_2)$. Then $U_1 \sim \text{dom}(U_2)$ implies $\tilde{P}(U_1, U_2)$.

**Proof.** Suppose $U_1 \sim \text{dom}(U_2)$. Then exists $T_1 \in \tilde{T}_1, T_1 \in \gamma(U_1)$ and $T_2 \in \tilde{T}_2, T_2 \in \gamma(\text{dom}(U_2))$ such that $T_1 = T_2$, which implies that $\exists T_2' \in \tilde{T}_2, T_2' \in \gamma(G_2)$, such that $T_2 = \text{dom}(T_2')$, which is by definition $\tilde{P}(U_1, U_2)$. \qed

**Proposition 238.** Let $P(T_1, T_2) \equiv T_1 = \text{dom}(T_2)$. Then $U_1 \sim \text{dom}(U_2)$ if and only if $\tilde{P}(U_1, U_2)$.

**Proof.** Direct consequence of Prop. 236 and 237. \qed

**Inductive definitions**

This section presents some inductive definitions of metafunctions presented in §5.

**Proposition 37.**

<table>
<thead>
<tr>
<th>$U \sim U_1$</th>
<th>$U \sim U_2$</th>
<th>$U_1 \sim U$</th>
<th>$U_2 \sim U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U \sim U_1 \oplus U_2$</td>
<td>$U \sim U_1 \oplus U_2$</td>
<td>$U_1 \oplus U_2 \sim U$</td>
<td>$U_1 \oplus U_2 \sim U$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$U \sim U$</th>
<th>$U \sim \text{?}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$U_1 \sim U_1$</td>
<td>$U_1 \sim U_2$</td>
</tr>
</tbody>
</table>

**Proof.** Straightforward from the definition of consistency. \qed

**Definition 110** (Gradual Meet). Let $\sqcap : \text{U} \text{TYPE} \to \text{U} \text{TYPE}$ be defined as:

1. $U \sqcap U = U$

2. $\text{?} \sqcap U = U \sqcap \text{?} = U$
3. \( U \cap (U_1 \oplus U_2) = (U_1 \oplus U_2) \cap U = \begin{cases} U \cap U_1 & \text{if } U \cap U_2 \text{ is undefined} \\ U \cap U_2 & \text{if } U \cap U_1 \text{ is undefined} \\ (U \cap U_1) \oplus (U \cap U_2) & \text{otherwise} \end{cases} \)

4. \( (U_{11} \rightarrow U_{12}) \cap (U_{21} \rightarrow U_{22}) = (U_{11} \cap U_{21}) \rightarrow (U_{12} \cap U_{22}) \)

5. \( U_1 \cap U_2 \) is undefined otherwise.

**Definition 111** (Equate lifting).

\[
equate(U_1, U_2) = U_1 \cap U_2 = \alpha (\{ \overline{T_1} \cap \overline{T_2} \mid \overline{T_1} \in \gamma(U_1), \overline{T_2} \in \gamma(U_2) \}) = \alpha (\overline{\alpha \oplus \{ \equate\gamma(G_1, G_2) \mid G_1 \in \gamma_\oplus(U_1), G_2 \in \gamma_\oplus(U_2) \}})
\]

where \( \overline{\equate\gamma} \) is the lifting of the \( \equate \) function in the intermediate abstraction (defined in [44]).

**Proposition 38.** \( \sqcap = \alpha \circ \overline{\equate} \circ \gamma \)

*Proof. Direct by the definition of meet as intersection of sets of sets.*

**Proposition 239.**

\[
\frac{U \subseteq U}{? \subseteq U} \quad \frac{U_1 \subseteq U_3 \quad U_2 \subseteq U_4}{U_1 \rightarrow U_2 \subseteq U_3 \rightarrow U_4} \quad \frac{U_1 \subseteq U_2 \quad U_1 \subseteq U_3}{U_1 \oplus U_2 \subseteq U_3 \quad U_2 \subseteq U_3 \quad U_1 \oplus U_2 \subseteq U_3}
\]

*Proof. Direct by definition of the concretization function.*

**Examples of type derivations**

In this section we present type derivations of the examples presented in §5.2.3.

**Static Gradual Guarantee**

**Proposition 40** (Equivalence for fully-annotated terms).

For any \( t \in \text{TERM} \), \( \vdash_S t : T \) if and only if \( \vdash t : T \)

*Proof. By induction over the typing derivations. The proof is trivial because static types are given singleton meanings via concretization.*
\[
\begin{array}{c}
x : \text{Int} \vdash x : \text{Int} \oplus \text{Bool} \\
x : \text{Int} \vdash 1 : \text{Int} \quad \text{Int} \oplus \text{Bool} \sim \text{Int} \quad \text{Int} \sim \text{Int} \\
x : \text{Int} \vdash (x + 1) : \text{Int} \\
\vdash (\lambda x : \text{Int}.(x + 1)) : (\text{Int} \oplus \text{Bool}) \rightarrow \text{Int}
\end{array}
\]

\[
\begin{array}{c}
x : \text{Int} \vdash x : \text{Int} \oplus \text{Bool} \\
x : \text{Int} \vdash 1 : \text{Int} \quad \text{Int} \oplus \text{Bool} \sim \text{Int} \quad \text{Int} \sim \text{Int} \\
x : \text{Int} \vdash (x + 1) : \text{Int} \\
\vdash (\lambda x : \text{Int}.(x + 1)) : (\text{Int} \oplus \text{Bool}) \rightarrow \text{Int} \\
\vdash 1 : \text{Int} \quad \text{Int} \sim \text{Int} \oplus \text{Bool}
\end{array}
\]

\[
\begin{array}{c}
x : \text{Int} \vdash x : \text{Int} \oplus \text{Bool} \\
x : \text{Int} \vdash 1 : \text{Int} \quad \text{Int} \oplus \text{Bool} \sim \text{Int} \quad \text{Int} \sim \text{Int} \\
x : \text{Int} \vdash (x + 1) : \text{Int} \\
\vdash (\lambda x : \text{Int}.(x + 1)) : (\text{Int} \oplus \text{Bool}) \rightarrow \text{Int} \\
\vdash \text{true} : \text{Bool} \quad \text{Bool} \sim \text{Int} \oplus \text{Bool}
\end{array}
\]

\[
\begin{array}{c}
x : \text{Bool} \vdash 1 : \text{Int} \quad \text{Int} \sim \text{Int} \oplus \text{Bool} \\
x : \text{Bool} \vdash \text{false} : \text{Bool} \quad \text{Bool} \sim \text{Int} \oplus \text{Bool} \\
(\lambda x : \text{Int} \oplus \text{Bool} : \text{Int} \oplus \text{Bool}) \vdash \text{false} : \text{Int} \oplus \text{Bool} \\
(\lambda x : \text{Int} \oplus \text{Bool} : \text{Int} \oplus \text{Bool}) \vdash \text{false} : \text{Int} \oplus \text{Bool} \\
\vdash (\lambda x : \text{Bool}.(\text{if } x \text{ then } (1 :: \text{Int} \oplus \text{Bool}) \text{ else } (\text{false} :: \text{Int} \oplus \text{Bool})) : \text{Bool} \rightarrow (\text{Int} \oplus \text{Bool})
\end{array}
\]

Figure C.4: Examples of intrinsic type derivations

**Definition 112** (Term precision).

\[
\begin{align*}
\text{(Px)} & \quad x \sqsubseteq x \\
\text{(Pb)} & \quad b \sqsubseteq b \\
\text{(Pn)} & \quad n \sqsubseteq n \\
\text{(Pλ)} & \quad \overline{t} \sqsubseteq \overline{t}' \quad U_1 \sqsubseteq U'_1 \\
& \quad (\lambda x : U_1.\overline{t}) \sqsubseteq (\lambda x : U'_1.\overline{t}') \\
\text{(P+)} & \quad \overline{t}_1 \sqsubseteq \overline{t}'_1 \quad \overline{t}_2 \sqsubseteq \overline{t}'_2 \\
& \quad \overline{t}_1 + \overline{t}_2 \sqsubseteq \overline{t}'_1 + \overline{t}'_2 \\
\text{(Papp)} & \quad \overline{t}_1 \sqsubseteq \overline{t}'_1 \quad \overline{t}_2 \sqsubseteq \overline{t}'_2 \\
& \quad \overline{t}_1 \, \overline{t}_2 \sqsubseteq \overline{t}'_1 \, \overline{t}'_2 \\
\text{(Pif)} & \quad \overline{t} \sqsubseteq \overline{t}' \quad \overline{t}_1 \sqsubseteq \overline{t}'_1 \\
& \quad \overline{t}_2 \sqsubseteq \overline{t}'_2 \\
& \quad \text{if } \overline{t} \text{ then } \overline{t}_1 \text{ else } \overline{t}_2 \sqsubseteq \text{if } \overline{t}' \text{ then } \overline{t}'_1 \text{ else } \overline{t}'_2 \\
\text{(P::)} & \quad \overline{t} \sqsubseteq \overline{t}' \quad U \sqsubseteq U' \\
& \quad \overline{t} : U \sqsubseteq \overline{t}' :: U'
\end{align*}
\]

**Definition 113** (Type environment precision).

\[
\begin{array}{c}
\sqsubseteq. \\
\Gamma \sqsubseteq \Gamma' \quad U \sqsubseteq U' \\
\Gamma, x : U \sqsubseteq \Gamma', x : U'
\end{array}
\]

**Lemma 240.** If \( \Gamma \vdash \overline{t} : U \) and \( \Gamma \sqsubseteq \Gamma' \), then \( \Gamma' \vdash \overline{t} : U' \) for some \( U \sqsubseteq U' \).

331
Proof. Simple induction on typing derivations. \[\square\]

**Lemma 241.** If \(U_1 \sim U_2\) and \(U_1 \subseteq U'_1\) and \(U_2 \subseteq U'_2\) then \(U'_1 \sim U'_2\).

*Proof.* By definition of \(\sim\), there exists \(\langle T_1, T_2 \rangle \in \langle \tilde{T}_1, \tilde{T}_2 \rangle \in \gamma^2(U_1, U_2)\) such that \(T_1 = T_2, U_1 \subseteq U'_1\) and \(U_2 \subseteq U'_2\) mean that \(\gamma(U_1) \subseteq \gamma(U'_1)\) and \(\gamma(U_2) \subseteq \gamma(U'_2)\), therefore \(\langle T_1, T_2 \rangle \in \langle \tilde{T}_1, \tilde{T}_2 \rangle \in \gamma^2(U'_1, U'_2)\).

\[\square\]

**Proposition 41** (Static gradual guarantee). If \(\vdash \bar{t}_1 : U_1\) and \(\bar{t}_1 \subseteq \bar{t}_2\), then \(\vdash \bar{t}_2 : U_2\), for some \(U_2\) such that \(U_1 \subseteq U_2\).

*Proof.* We prove the property on opens terms instead of closed terms: If \(\Gamma \vdash \bar{t}_1 : U_1\) and \(\bar{t}_1 \subseteq \bar{t}_2\) then \(\Gamma \vdash \bar{t}_2 : U_2\) and \(U_1 \subseteq U_2\).

The proof proceed by induction on the typing derivation.

**Case \((Ux, Ub)\).** Trivial by definition of \(\subseteq\) using \((Px), (Pb)\) respectively.

**Case \((U\lambda)\).** Then \(\bar{t}_1 = (\lambda x : U_1, \bar{t})\) and \(U_1 = U'_1 \rightarrow U'_2\). By \((U\lambda)\) we know that:

\[
\begin{align*}
\frac{\Gamma, x : U'_1 \vdash \bar{t} : U'_2}{\Gamma \vdash (\lambda x : U'_1, \bar{t}) : U'_1 \rightarrow U'_2} \quad (C.1)
\end{align*}
\]

Consider \(\bar{t}_2\) such that \(\bar{t}_1 \subseteq \bar{t}_2\). By definition of term precision \(\bar{t}_2\) must have the form \(\bar{t}_2 = (\lambda x : U''_1, \bar{t}')\) and therefore

\[
\begin{align*}
\frac{\bar{t} \subseteq \bar{t}' \quad U'_1 \subseteq U''_1}{(\lambda x : U'_1, \bar{t}) \subseteq (\lambda x : U''_1, \bar{t}')} \quad (C.2)
\end{align*}
\]

Using induction hypotheses on the premise of \(D.2\), \(\Gamma, x : U'_1 \vdash \bar{t}' : U''_2\) with \(U'_2 \subseteq U''_2\). By Lemma 267 \(\Gamma, x : U''_1 \vdash \bar{t}' : U''_2\) where \(U''_2 \subseteq U''_2\). Then we can use rule \((U\lambda)\) to derive:

\[
\begin{align*}
\frac{\Gamma, x : U''_1 \vdash \bar{t}' : U''_2}{\Gamma \vdash (\lambda x : U'_1, \bar{t}') : U'_1 \rightarrow U''_2}
\end{align*}
\]

Where \(U_2 \subseteq U''_2\). Using the premise of \(D.2\) and the definition of type precision we can infer that

\(U'_1 \rightarrow U'_2 \subseteq U''_1 \rightarrow U''_2\)

and the result holds.

**Case \((U+)\).** Then \(\bar{t}_1 = \tilde{t}_1 + \tilde{t}_2\) and \(U_1 = \text{Int}\). By \((U+)\) we know that:

\[
\begin{align*}
\frac{\Gamma \vdash \bar{t}_1 : U_1 \quad \Gamma \vdash \bar{t}_2 : U_2 \quad U_1 \sim \text{Int} \quad U_2 \sim \text{Int}}{\Gamma \vdash \bar{t}_1 + \bar{t}_2 : \text{Int}} \quad (C.3)
\end{align*}
\]

Consider \(\bar{t}_2\) such that \(\bar{t}_1 \subseteq \bar{t}_2\). By definition of term precision \(\bar{t}_2\) must have the form \(\bar{t}_2 = \tilde{t}'_1 + \tilde{t}'_2\) and therefore

\[
\begin{align*}
\frac{\tilde{t}'_1 \subseteq \tilde{t}'_1 \quad \tilde{t}'_2 \subseteq \tilde{t}'_2}{\tilde{t}'_1 + \tilde{t}'_2 \subseteq \tilde{t}'_1 + \tilde{t}'_2} \quad (C.4)
\end{align*}
\]

332
Using induction hypotheses on the premises of \( \text{D.9} \) \( \Gamma \vdash \widetilde{t}_1' : U_1' \) and \( \Gamma \vdash \widetilde{t}_2' : U_2' \), where \( U_1 \subseteq U_1' \) and \( U_2 \subseteq U_2' \). By Lemma \( \text{268} \) \( U_1' \sim \text{Int} \) and \( U_2' \sim \text{Int} \). Therefore we can use rule \( (U+) \) to derive:

\[
\Gamma \vdash \widetilde{t}_1' : U_1' \quad \Gamma \vdash \widetilde{t}_2' : U_2' \quad U_1' \sim \text{Int} \quad U_2' \sim \text{Int} \\
\hline
\Gamma \vdash \widetilde{t}_1' + \widetilde{t}_2' : \text{Int}
\]

and the result holds.

**Case** \( (U\text{app}) \). Then \( \widetilde{t}_1 = \widetilde{t}_1' \widetilde{t}_2' \) and \( U_1 = U_{12} \). By \( (U\text{app}) \) we know that:

\[
\Gamma \vdash \widetilde{t}_1' : U_1' \quad \Gamma \vdash \widetilde{t}_2' : U_2' \quad U_2' \sim \text{dom}(U_1')
\]

\[
\hline
\Gamma \vdash \widetilde{t}_1' \widetilde{t}_2' : \text{dom}(U_1')
\]

(C.5)

Consider \( \widetilde{t}_2 \) such that \( \widetilde{t}_1 \subseteq \widetilde{t}_2 \). By definition of term precision \( \widetilde{t}_2 \) must have the form \( \widetilde{t}_2 = \widetilde{t}_1' \widetilde{t}_2'' \) and therefore

\[
\text{(Papp)} \frac{\widetilde{t}_1 \subseteq \widetilde{t}_1' \widetilde{t}_2 \subseteq \widetilde{t}_2''}{\widetilde{t}_1' \widetilde{t}_2 \subseteq \widetilde{t}_1' \widetilde{t}_2''}
\]

(C.6)

Using induction hypotheses on the premises of \( \text{D.11} \) \( \Gamma \vdash \widetilde{t}_1'' : U_1'' \) and \( \Gamma \vdash \widetilde{t}_2'' : U_2'' \), where \( U_1' \subseteq U_1'' \) and \( U_2' \subseteq U_2'' \). By definition precision (Def. \( \text{27} \)) and the definition of \( \text{dom} \), \( \text{dom}(U_1') \subseteq \text{dom}(U_1'') \) and, therefore by Lemma \( \text{268} \) \( U_2'' \sim \text{dom}(U_1'') \). Also, by the previous argument \( \text{cod}(U_1') \subseteq \text{cod}(U_1'') \). Then we can use rule \( (U\text{app}) \) to derive:

\[
\Gamma \vdash \widetilde{t}_1'' : U_1'' \quad \Gamma \vdash \widetilde{t}_2'' : U_2'' \quad U_2'' \sim \text{dom}(U_1'')
\]

\[
\hline
\Gamma \vdash \widetilde{t}_1'' \widetilde{t}_2'' : \text{cod}(U_1'')
\]

and the result holds.

**Case** \( (U\text{if}) \). Then \( \widetilde{t}_1 = \text{if } \widetilde{t}_1' \text{ then } \widetilde{t}_2' \text{ else } \widetilde{t}_3' \) and \( U_1 = (U_2' \cap U_3') \). By \( (U\text{if}) \) we know that:

\[
\Gamma \vdash \widetilde{t}_1' : U_1' \quad \Gamma \vdash \widetilde{t}_2' : U_2' \quad \Gamma \vdash \widetilde{t}_3' : U_3'
\]

\[
\hline
\Gamma \vdash \text{if } \widetilde{t}_1' \text{ then } \widetilde{t}_2' \text{ else } \widetilde{t}_3' : (U_2' \cap U_3')
\]

(C.7)

Consider \( \widetilde{t}_2 \) such that \( \widetilde{t}_1 \subseteq \widetilde{t}_2 \). By definition of term precision \( \widetilde{t}_2 \) must have the form \( \widetilde{t}_2 = \text{if } \widetilde{t}_1'' \text{ then } \widetilde{t}_2'' \text{ else } \widetilde{t}_3'' \) and therefore

\[
\text{(Pif)} \frac{\widetilde{t}_1' \subseteq \widetilde{t}_1'' \quad \widetilde{t}_1 \subseteq \widetilde{t}_1'' \quad \widetilde{t}_2 \subseteq \widetilde{t}_2''}{\text{if } \widetilde{t}_1' \text{ then } \widetilde{t}_2'' \text{ else } \widetilde{t}_3'' \subseteq \text{if } \widetilde{t}_1'' \text{ then } \widetilde{t}_2'' \text{ else } \widetilde{t}_3''}
\]

(C.8)

Then we can use induction hypotheses on the premises of \( \text{C.7} \) and derive:

\[
\Gamma \vdash \widetilde{t}_1'' : U_1'' \quad \Gamma \vdash \widetilde{t}_2'' : U_2'' \quad \Gamma \vdash \widetilde{t}_3'' : U_3''
\]

\[
\hline
\Gamma \vdash \text{if } \widetilde{t}_1'' \text{ then } \widetilde{t}_2'' \text{ else } \widetilde{t}_3'' : (U_2'' \cap U_3'')
\]

Where \( U_1' \subseteq U_1'' \) and \( U_2' \subseteq U_2'' \). Using the definition of type precision (Def. \( \text{27} \)) we can infer that

\( (U_1' \cap U_2') \subseteq (U_1'' \cap U_2'') \)

and the result holds.
Case \((U::)\). Then \(\bar{t}_1 = \bar{t} :: U_1\). By \((U::)\) we know that:

\[
(U::) \quad \frac{ \Gamma \vdash \bar{t} : U'_1 \quad U'_1 \sim U_1 }{ \Gamma \vdash \bar{t} :: U'_1 : U_1 }
\]  
(C.9)

Consider \(\bar{t}_2\) such that \(\bar{t}_1 \subseteq \bar{t}_2\). By definition of term precision \(\bar{t}_2\) must have the form \(\bar{t}_2 = \bar{t}' :: U_2\) and therefore

\[
(P::) \quad \frac{ \bar{t} \subseteq \bar{t}' \quad U_1 \subseteq U_2 }{ \bar{t} :: U_1 \subseteq \bar{t}' :: U_2 }  
\]  
(C.10)

Using induction hypotheses on the premises of \([D.7]\) \(\Gamma \vdash \bar{t}' : U'_2\) where \(U'_1 \subseteq U'_2\). We can use rule \((U::)\) and Lemma \([268]\) to derive:

\[
(U::) \quad \frac{ \Gamma \vdash \bar{t}' : U'_2 \quad U'_2 \sim U_2 }{ \Gamma \vdash \bar{t}' :: U_2 : U_2 }
\]

Where \(U_1 \subseteq U_2\) and the result holds.

\[\square\]

\section{C.1.4 Dynamic Semantics of GTFL\(\oplus\)}

One of the salient features of the AGT methodology is that it provides a direct dynamic semantics for gradual programs \([44]\), instead of the typical translational semantics through an intermediate cast calculus \([109]\). The key idea is to apply proof reduction on gradual typing derivations \([63]\); by the Curry-Howard correspondence, this gives a notion of relation for gradual terms. We call such semantics the reference semantics.

The main insight of AGT is that gradual typing derivations need to be augmented with evidence to support consistent judgments. Evidence reflects the justification of why a given consistent judgment holds. Therefore, the dynamic semantics mirrors the type preservation argument of the static language, combining evidences at each reduction step in order to determine whether the program can reduce or should halt with a runtime error.

A consistency judgment \(U_1 \sim U_2\) is supported by an evidence \(\varepsilon\) that denotes the most precise knowledge about \(U_1\) and \(U_2\) gained by knowing that they are related by consistency. It is written \(\varepsilon \vdash U_1 \sim U_2\). In the case of consistency, which is symmetric (as opposed to, say, consistent subtyping), evidence boils down to a single gradual type, i.e. \(\varepsilon \in \text{UType}\), which is precisely the least upper bound of both types, i.e. \(U_1 \cap U_2\) \([44]\). For instance, \(\text{Int} \vdash \text{Int} \oplus \text{Bool} \sim \text{Int}\).

Consider the simple program: \((\lambda x : \text{Int} \oplus \text{Bool}. x + 1) \text{ true}\). It is a well-typed gradual program, and its typing derivation includes two consistent judgments: \(\text{Bool} \sim \text{Int} \oplus \text{Bool}\) to accept passing \(\text{true}\) as argument, and \(\text{Int} \oplus \text{Bool} \sim \text{Int}\) to accept using \(x\) in the addition. When simplifying the typing derivation itself (replacing the use of the \(x\) hypothesis with the typing derivation of the argument) it becomes necessary to combine the two consistent judgments.
in order to justify the reduction. In general, in the safety proof of the static language, this corresponds to a use of the fact that type equality is transitive. Here, transitivity demands that the respective evidences for the consistent judgments can be combined. In the example, \texttt{Int} and \texttt{Bool} cannot be combined (their meet is undefined), therefore the program halts with \texttt{error}.

To formalize this approach while avoiding writing down reduction rules on actual (bi-dimensional) derivation trees, Garcia \textit{et al.} adopt \textit{intrinsic terms} \cite{25}, which are a flat notation that is isomorphic to typing derivations. Specifically, the typing derivation for the judgment $\Gamma \vdash \bar{i} : U$ is represented by an intrinsic term $\bar{i} \in \text{TERM}_U$. (The reversed hat on $\bar{i}$ is meant to suggest the derivation tree.)

To illustrate how intrinsic terms are formed, consider addition and ascription:

\[
\frac{\bar{t}_1 \in \text{TERM}_{U_1} \quad \varepsilon_1 \vdash U_1 \sim \text{Int} \quad \bar{t}_2 \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \sim \text{Int}}{\varepsilon_1 \bar{t}_1 + \varepsilon_2 \bar{t}_2 \in \text{TERM}_{\text{Int}}}
\]
\[
\frac{\bar{t} \in \text{TERM}_U \quad \varepsilon \vdash U \sim U'}{\varepsilon \bar{t} :: U' \in \text{TERM}_{U'}}
\]

The rules describe how a derivation tree for a compound expression is formed from the sub-derivations of the subterms together with the evidences that support the consistency judgments. Note how the involved evidences show up in the term representation. The syntax of intrinsic terms follows the same pattern as that illustrated with addition and ascription. Also, intrinsic values $\bar{v}$ can either be simple values $\bar{u}$ or ascribed values $\varepsilon \bar{u} :: U$. With this notational device, the reduction rules on derivation trees can be written as reduction rules on intrinsic terms, possibly failing with an \texttt{error} when combining evidences, for instance:

\[
\varepsilon_1 (\varepsilon_2 \bar{v} :: U) \longrightarrow_c \begin{cases} (\varepsilon_2 \circ^= \varepsilon_1)\bar{v} \\ \texttt{error} \quad \text{if } (\varepsilon_2 \circ^= \varepsilon_1) \text{ is not defined} \end{cases}
\]

The definition of consistent transitivity for a type predicate $P$, $\circ^P$, is given by the abstract interpretation framework \cite{44}; in particular, for type equality, $\circ^=$ corresponds to the meet of gradual types $\sqcap$.

Notice that for convenience, from this point forward we use the notation $t^U \in \text{TERM}_U$ to refer to an intrinsic term $\bar{t} \in \text{TERM}_U$, $u$ to refer to an $\bar{u}$, $v$ to refer to an $\bar{v}$, and finally we sometimes omit the type notation in $t^U$ when the type is not important in that context. Figure C.5 presents the gradual intrinsic terms of GTFL$^\oplus$. Figures C.6 and C.7 present some intrinsic type derivations of the examples presented in §5.2.3. Figure C.8 presents the syntax and notions of reductions. Figure C.9 presents the intrinsic reduction of GTFL$^\oplus$. Recall that the online implementation provides step-by-step reduction traces of arbitrary source programs (with the examples of §5.3 pre-loaded).

C.1.5 Properties of the Gradual Security Language

This section present the proof of type safety and the proof of the dynamic gradual guarantee in terms of intrinsic terms.
Proof. By induction on the structure of a derivation of \( t^U \rightarrow r \), considering the last rule used in the derivation.

**Case (IU\(?\)).** Then \( t^U = \varepsilon_1 n_1 + \varepsilon_2 n_2 \). Then

\[
(IU+) \quad n_1 \in \text{TERM}_{\text{Int}} \quad \varepsilon_1 \vdash n_1 \sim \text{Int} \quad n_2 \in \text{TERM}_{\text{Int}} \quad \varepsilon_2 \vdash n_2 \sim \text{Int} \quad \varepsilon_1 n_1 + \varepsilon_2 n_2 \in \text{TERM}_{\text{Int}}
\]

Figure C.5: Gradual Intrinsic Terms for GTFL\(^\oplus\)

**Type Safety**

In this section we present the proof of type safety for GTFL\(^\oplus\).

**Lemma 242** (Canonical forms). Consider a value \( v \in \text{TERM}_U \). Then either \( v = u \), or \( v = \varepsilon u :: U \) with \( u \in \text{TERM}_U \) and \( \varepsilon \vdash U' \sim U \). Furthermore:

1. If \( U = \text{Bool} \) then either \( v = b \) or \( v = \varepsilon b :: \text{Bool} \) with \( b \in \text{TERM}_{\text{Bool}} \).
2. If \( U = \text{Int} \) then either \( v = n \) or \( v = \varepsilon n :: \text{Int} \) with \( n \in \text{TERM}_{\text{Int}} \).
3. If \( U = U_1 \rightarrow U_2 \) then either \( v = (\lambda x^{U_1}.t^{U_2}) \) with \( t^{U_2} \in \text{TERM}_{U_2} \) or \( v = \varepsilon (\lambda x^{U_1'.U_2}) :: U_1 \rightarrow U_2 \) with \( t^{U_2} \in \text{TERM}_{U_2} \) and \( \varepsilon \vdash U_1' \sim U_1 \rightarrow U_2 \).

**Proof.** By direct inspection of the formation rules of gradual intrinsic terms (Figure C.5). \(\square\)

**Lemma 243** (Substitution). If \( t^U \in \text{TERM}_U \) and \( v \in \text{TERM}_{U_1} \), then \([v/x^{U_1}]t^U \in \text{TERM}_U\).

**Proof.** By induction on the derivation of \( t^U \). \(\square\)

**Proposition 244** (\( \rightarrow \) is well defined). If \( t^U \rightarrow r \), then \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \).

**Proof.** By induction on the structure of a derivation of \( t^U \rightarrow r \), considering the last rule used in the derivation.

Case (IU\(?\)). Then \( t^U = \varepsilon_1 n_1 + \varepsilon_2 n_2 \). Then

\[
(IU+) \quad n_1 \in \text{TERM}_{\text{Int}} \quad \varepsilon_1 \vdash n_1 \sim \text{Int} \quad n_2 \in \text{TERM}_{\text{Int}} \quad \varepsilon_2 \vdash n_2 \sim \text{Int} \quad \varepsilon_1 n_1 + \varepsilon_2 n_2 \in \text{TERM}_{\text{Int}}
\]
\[ x \in \text{TERM}_{\text{Int} \oplus \text{Bool}} \quad 1 \in \text{TERM}_{\text{Int}} \]
\[
\begin{array}{c}
\langle \text{Int} \rangle \vdash \text{Int} \oplus \text{Bool} \sim \text{Int} \\
\langle \text{Int} \rangle \vdash \text{Int} \sim \text{Int}
\end{array}
\]
\[
(\langle \text{Int} \rangle x + \langle \text{Int} \rangle 1) \in \text{TERM}_{\text{Int} \oplus \text{Bool} \rightarrow \text{Int}}
\]
\[
\begin{array}{c}
\lambda x.((\langle \text{Int} \rangle x + \langle \text{Int} \rangle 1)) \in \text{TERM}_{\langle \text{Int} \rangle \oplus \text{Bool} \rightarrow \langle \text{Int} \rangle 1} \in \text{TERM}_{\text{Int}}
\end{array}
\]

Therefore
\[
\varepsilon_1 n_1 + \varepsilon_2 n_2 \rightarrow n_3 \text{ where } n_3 = n_1 [+] n_2
\]

But \( n_3 \in \text{TERM}_{\text{Int}} \) and the result holds.

**Case (I\(U\) app).** Then \( t^U = \varepsilon_1 (\lambda x^{U_1} . t_1^{U_1} ) \#^{U_1 \rightarrow U_2} (\varepsilon_2 u) \) and \( U = U_2 \). Then

If \( \varepsilon' = (\varepsilon_2 \circ idom(\varepsilon_1)) \) is not defined, then \( t^U \rightarrow \text{error} \), and then the result hold immediately. Suppose that consistent transitivity does hold, then

\[ \varepsilon_1 (\lambda x^{U_1} . t_1^{U_1} ) \#^{U_1 \rightarrow U_2} \varepsilon_2 u \in \text{TERM}_{U_2} \rightarrow idom(\varepsilon_1)([\varepsilon' u :: U_{11}] / x^{U_1} )t) :: U_2 \]

As \( \varepsilon_2 :: U'_2 \sim U_1 \) and by inversion lemma \( idom(\varepsilon_1) \vdash U_1 \sim U_{11} \), then \( \varepsilon' \vdash U'_2 \sim U_{11} \). Therefore \( \varepsilon' u :: U_{11} \in \text{TERM}_{U_{11}} \), and by Lemma 275, \( t^{U_{12}} = ([\varepsilon' u :: U_{11}] / x^{U_{11}} )t^{U_{12}} \in \text{TERM}_{U_{12}} \).

Then
\[ (I\#) \quad t^{U_{12}} \in \text{TERM}_{U_{12}} \quad idom(\varepsilon_1) t^{U_{12}} :: U_2 \in \text{TERM}_{U_2} \]

337
We know that \( \varepsilon \) have there exists \( U \) Case (R Term Case (R (R Case (R

\[\begin{array}{ll}
1 \in \text{TERM}_{\text{Int}} & \langle \text{Int} \rangle \vdash \text{Int} \sim \text{Int} \oplus \text{Bool} \\
((\langle \text{Int} \rangle 1 :: \text{Int} \oplus \text{Bool}) \in \text{TERM}_{\text{Int} \oplus \text{Bool}} & x \in \text{TERM}_{\text{Bool}} \\
\langle \text{Int} \oplus \text{Bool} \rangle \vdash \text{Int} \oplus \text{Bool} \sim \text{Int} \oplus \text{Bool} \\
\end{array}\]

\[(\langle \text{Bool} \rangle \text{false} :: \text{Int} \oplus \text{Bool}) \in \text{TERM}_{\text{Int} \oplus \text{Bool}}
\]

\[\begin{array}{ll}
\langle \text{Bool} \rangle \vdash \text{Bool} \sim \text{Int} \oplus \text{Bool} \\
\langle \text{Int} \oplus \text{Bool} \rangle \vdash \text{Int} \oplus \text{Bool} \sim \text{Int} \oplus \text{Bool} \\
\end{array}\]

Therefore

\[\lambda x \cdot \langle \langle \text{Term} \rangle 1 :: \text{Int} \oplus \text{Bool} \rangle \vdash \langle \langle \text{Term} \rangle \rangle \sim (U_2 \cap U_3) \]

\[\varepsilon_1 b \vdash U_1 \sim \text{Bool} \quad U = (U_2 \cap U_3) \]

\[t^{U_2} \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \sim U \]

\[t^{U_3} \in \text{TERM}_{U_3} \quad \varepsilon_3 \vdash U_3 \sim U \]

\[\text{if } \varepsilon_1 b \text{ then } \varepsilon_2 t^{U_2} \text{ else } \varepsilon_3 t^{U_3} \in \text{TERM}_U \]

Therefore

\[\text{if } \varepsilon_1 b \text{ then } \varepsilon_2 t^{U_2} \text{ else } \varepsilon_3 t^{U_3} \in \text{TERM}_U \]

But

\[\varepsilon_2 t^{U_2} :: U_2 \cap U_3 \in \text{TERM}_{U_2 \cap U_3} \]

and the result holds.

Case (IUif-false). Analogous to case (if-true).

Figure C.7: Examples of intrinsic type derivations (part 2)

\[\text{Proposition 245 (} \rightarrow \text{ is well defined). If } t^U \rightarrow r, \text{ then } r \in \text{CONFIG}_U \cup \{ \text{error} \}. \]

\[\begin{array}{ll}
\text{Proof.} \text{ By induction on the structure of a derivation of } t^U \rightarrow r. \\
\text{Case} (R \rightarrow). t^U \rightarrow r. \text{ By well-definedness of } \rightarrow (\text{Prop \#280}), r \in \text{CONFIG}_U \cup \{ \text{error} \}. \\
\text{Case} (R f). t^U = f[t^{U''}], f[t^{U''}] \in \text{TERM}_U, t_1^{U''} \rightarrow t_2^{U''}, t^{U''} \in \text{TERM}_U, \text{ and } f : \text{TERM}_U \rightarrow \text{TERM}_U. \text{ By induction hypothesis, } t_2^{U''} \in \text{TERM}_U, \text{ so } f[t_2^{U''}] \in \text{TERM}_U. \\
\text{Case} (R f err, R g err). r = \text{error}. \\
\text{Case} (R g). t^U = g[et], g[t^{U'}] \in \text{TERM}_U, \text{ and } g : \text{EvTERM} \rightarrow \text{TERM}_U, \text{ and } et \rightarrow_c et'. \text{ Then there exists } U_v, U_x \text{ such that } et = \varepsilon_v t^{U_v} \text{ and } \varepsilon_v \vdash U_v \sim U_x. \text{ Also, } t_e = \varepsilon_v v :: U_e, \text{ with } v \in \text{TERM}_U \text{ and } \varepsilon_v \vdash U_v \sim U_e. \text{ We know that } \varepsilon_c = \varepsilon_v o^= \varepsilon_e \text{ is defined, and } et = \varepsilon_v t_e \rightarrow_c \varepsilon_c v = et'. \text{ By definition of } o^= \text{ we have } \varepsilon_c \vdash U_v \sim U_x, \text{ so } g[et'] \in \text{TERM}_U. \]

338
Evidence, \( et \in \text{EvTerm}, \ ev \in \text{EvValue}, \ t \in \text{Term}_*, \)
\( v \in \text{Value}, \ u \in \text{SimpleValue}, g \in \text{EvFrame}, \ f \in \text{TmFrame} \)

- \( et ::= \varepsilon t \)
- \( ev ::= \varepsilon u \)
- \( u ::= x \mid n \mid b \mid \lambda x.t \)
- \( v ::= u \mid \varepsilon u \)
- \( g ::= \square + et \mid ev + \square \mid \square @ U \mid ev @ U \mid \square :: U \mid \square @ U \square :: U \)
- \( f ::= g[\varepsilon \square] \)

**Notions of Reduction**

- \( \rightarrow: \text{Term}_U \times (\text{Term}_U \cup \{ \text{error} \}) \)
- \( \rightarrow_e: \text{EvTerm} \times (\text{EvTerm} \cup \{ \text{error} \}) \)

\[
\varepsilon_1n_1 + \varepsilon_2n_2 \rightarrow n_3 \text{ where } n_3 = n_1 \mathcal{[}+] \ n_2
\]

\[
\varepsilon_1(\lambda x^{U_1}.t) @ U \rightarrow \varepsilon_2u \rightarrow \begin{cases} \text{icod}(\varepsilon_1)([((\varepsilon_2 o= \text{idom}(\varepsilon_1))u :: U_1)/x^{U_1}])t :: U_2 & \text{if not defined} \\
\end{cases}
\]

\[
\text{if } \varepsilon_1 b \text{ then } \varepsilon_2t^U \text{ else } \varepsilon_3t^U \rightarrow \begin{cases} \varepsilon_2t^U :: U_2 \mathcal{[} \land \ U_3 \ b = \text{true} \\
\varepsilon_3t^U :: U_2 \mathcal{[} \land \ U_3 \ b = \text{false} \\
\end{cases}
\]

\[
\varepsilon_1(\varepsilon_2v :: U) \rightarrow_e \begin{cases} \text{(icod}(\varepsilon_1)v & \text{if not defined} \\
\end{cases}
\]

Figure C.8: Syntax and notions of Reduction

Now we can establish type safety: programs do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

**Proposition 246** (Type Safety). If \( t^U \in \text{Term}_U \) then either \( t^U \) is a value \( v; t^U \rightarrow \text{error} \); or \( t^U \rightarrow t'^U \) for some term \( t'^U \in \text{Term}_U \).

**Proof.** By induction on the structure of \( t^U \).

**Case (Iu,In, Ib, Ix, I\lambda).** \( t^U \) is a value.

**Case (I::).** \( t^U = \varepsilon_1t^{U_1} :: U_2 \), and

\[
(\text{I::}) \begin{array}{c}
\text{t}^{U_1} \in \text{Term}_{U_1} \\
\varepsilon_1t^{U_1} :: U_2 \in \text{Term}_{U_2}
\end{array}
\]

By induction hypothesis on \( t^{U_1} \), one of the following holds:

1. \( t^{U_1} \) is a value, in which case \( t^U \) is also a value.

2. \( t^{U_1} \rightarrow r_1 \) for some \( r_1 \in \text{Term}_{U_1} \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{Config}_U \cup \{ \text{error} \} \) by Prop \( \ref{prop:282} \) and either (Rf), or (Rferr).
Reduction

\[ \rightarrow: \text{TERM}_U \times (\text{TERM}_U \cup \{ \text{error} \}) \]

\[ \text{(R→)}\quad t^U \rightarrow r \quad r \in (\text{TERM}_U \cup \{ \text{error} \}) \]

\[ \text{(Rg)}\quad et \rightarrow_c et' \quad g[et] \rightarrow g[et'] \]

\[ \text{(Rerr)}\quad et \rightarrow_c \text{error} \quad g[et] \rightarrow \text{error} \]

\[ \text{(Rf)}\quad t_1^U \rightarrow t_2^U \quad f[t_1^U] \rightarrow f[t_2^U] \]

\[ \text{(Rferr)}\quad t_1^U \rightarrow \text{error} \quad f[t_1^U] \rightarrow \text{error} \]

Figure C.9: Intrinsic Reduction

Case (Iif). \( t^U = \varepsilon_1 t^U_1 \) then \( \varepsilon_2 t^U_2 \) else \( \varepsilon_3 t^U_3 \) and

\[ t^U_1 \in \text{TERM}_{U_1} \quad \varepsilon_1 \vdash U_1 \sim \text{Bool} \quad U = (U_2 \cap U_3) \]

\[ t^U_2 \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \sim U \]

\[ t^U_3 \in \text{TERM}_{U_3} \quad \varepsilon_3 \vdash U_3 \sim U \]

By induction hypothesis on \( t^U_1 \), one of the following holds:

1. \( t^U_1 \) is a value \( u \), then by (R→), \( t^U \rightarrow r \) and \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282]

2. \( t^U_1 \) is an ascribed value \( v \), then, \( \varepsilon_1 t^U_1 \rightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282] and either (Rg), or (Rgerr).

3. \( t^U_1 \rightarrow r_1 \) for some \( r_1 \in \text{TERM}_{U_1} \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282] and either (Rf), or (Rferr).

Case (Iapp). \( t^U = (\varepsilon_1 t^U_1) @^U_1 \rightarrow U_1 \) (\( \varepsilon_2 t^U_2 \))

\[ t^U_1 \in \text{TERM}_{U_1} \quad \varepsilon_1 \vdash U_1 \sim U_1 \rightarrow U_1 \]

\[ t^U_2 \in \text{TERM}_{U_2} \quad \varepsilon_2 \vdash U_2 \sim U_1 \]

By induction hypothesis on \( t^U_1 \), one of the following holds:

1. \( t^U_1 \) is a value (\( \lambda x t^U_1, t^U_1 \)) (by canonical forms Lemma [274], posing \( U_1 = U'_1 \rightarrow U'_1 \). Then by induction hypothesis on \( t^U_2 \), one of the following holds:

   (a) \( t^U_2 \) is a value \( u \), then by (R→), \( t^U \rightarrow r \) and \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282]

   (b) \( t^U_2 \) is an ascribed value \( v \), then, \( \varepsilon_2 t^U_2 \rightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282] and either (Rg), or (Rgerr).

   (c) \( t^U_2 \rightarrow r_2 \) for some \( r_2 \in \text{CONFIG}_U \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282] and either (Rf), or (Rferr).

2. \( t^U_1 \) is an ascribed value \( v \), then, \( \varepsilon_1 t^U_1 \rightarrow_c et' \) for some \( et' \in \text{EvTERM} \cup \{ \text{error} \} \). Hence \( t^U \rightarrow r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop [282] and either (Rg), or (Rgerr).
Ω \cup \{ x^{U_1} \sqsubseteq x^{U_2} \} \vdash x^{U_1} \sqsubseteq x^{U_2} \quad \Omega \vdash b \sqsubseteq b \quad \Omega \vdash n \sqsubseteq n

\Omega \vdash t^{U_1} \sqsubseteq t^{U_2} \quad U_1 \sqsubseteq U_2 \quad \Omega \vdash (\lambda x^{U_1} . t^{U_{12}}) \sqsubseteq (\lambda x^{U_{21}} . t^{U_{22}})

\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{21}} \quad U_{12} \sqsubseteq U_{22} \quad \epsilon_{12} \sqsubseteq \epsilon_{22} \quad \epsilon_{11} \sqsubseteq \epsilon_{21}

\Omega \vdash (\epsilon_{11} t^{U_{11}} + \epsilon_{12} t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} + \epsilon_{22} t^{U_{22}})

\Omega \vdash \epsilon_{11} \sqsubseteq \epsilon_{21} \quad \Omega \vdash \epsilon_{12} \sqsubseteq \epsilon_{22}

\Omega \vdash \epsilon_{11} t^{U_{11}} @^{U_1} \epsilon_{12} t^{U_{12}} \sqsubseteq \epsilon_{21} t^{U_{21}} @^{U_2} \epsilon_{22} t^{U_{22}}

\Omega \vdash (\epsilon_{11} t^{U_{11}} + \epsilon_{12} t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} + \epsilon_{22} t^{U_{22}})

\Omega \vdash (\epsilon_{11} t^{U_{11}} \cdot t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} \cdot t^{U_{22}})

\Omega \vdash (\epsilon_{11} t^{U_{11}} @^{U_1} \epsilon_{12} t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} @^{U_2} \epsilon_{22} t^{U_{22}})

\Omega \vdash (\epsilon_{11} t^{U_{11}} + \epsilon_{12} t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} + \epsilon_{22} t^{U_{22}})

\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{12}} \quad \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}}

\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{12}} \quad \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}}

\Omega \vdash t^{U_{11}} \sqsubseteq t^{U_{12}} \quad \Omega \vdash t^{U_{12}} \sqsubseteq t^{U_{22}}

\Omega \vdash (\epsilon_{11} t^{U_{11}} + \epsilon_{12} t^{U_{12}}) \sqsubseteq (\epsilon_{21} t^{U_{21}} + \epsilon_{22} t^{U_{22}})

Figure C.10: Intrinsic term precision

3. \( t^{U_1} \mapsto r_1 \) for some \( r_1 \in \text{CONFIG}_U \cup \{ \text{error} \} \). Hence \( t' \mapsto r \) for some \( r \in \text{CONFIG}_U \cup \{ \text{error} \} \) by Prop 282 and either (Rf), or (Rferr).

Case (IU+). Similar case to (IUapp)

Dynamic Gradual Guarantee

In this section we present the proof the Dynamic Gradual Guarantee for GTFL\textsuperscript{®}.

Definition 114 (Intrinsic term precision). Let
\( \Omega \in \mathcal{P}(\text{VAR}_* \times \text{VAR}_*) \) be defined as \( \Omega := \{ x^{U_{11}} \sqsubseteq x^{U_{12}} \} \) We define an ordering relation \( (\cdot \vdash \cdot \cdot) \in (\mathcal{P}(\text{VAR}_* \times \text{VAR}_*) \times \text{TERM}_* \times \text{TERM}_* \) shown in Figure C.10

Definition 115 (Well Formedness of \( \Omega \)). We say that \( \Omega \) is well formed iff \( \forall \{ x^{G_{11}} \sqsubseteq x^{G_{12}} \} \in \Omega, G_{11} \sqsubseteq G_{12} \)

Before proving the gradual guarantee, we first establish some auxiliary properties of precision. For the following propositions, we assume Well Formedness of \( \Omega \) (Definition 115).

Proposition 247. If \( \Omega \vdash t^{U_1} \sqsubseteq t^{U_2} \) for some \( \Omega \in \mathcal{P}(\text{VAR}_* \times \text{VAR}_*) \), then \( U_1 \sqsubseteq U_2 \).

Proof. Straightforward induction on \( \Omega \vdash t^{U_1} \sqsubseteq t^{U_2} \), since the corresponding precision on types is systematically a premise (either directly or transitively). \( \square \)
Proposition 248. Let $g_1, g_2 \in \text{EvFrame}$ such that $g_1[\varepsilon_1^{t_{11}}] \in \text{Term}_{U_1'}, g_2[\varepsilon_2^{t_{12}}] \in \text{Term}_{U_2'}$, with $U_1' \subseteq U_2'$. Then if $g_1[\varepsilon_1^{t_{11}}] \subseteq g_2[\varepsilon_2^{t_{12}}]$, $\varepsilon_{12} \subseteq \varepsilon_{22}$ and $t_{11} \subseteq t_{22}$, then $g_1[\varepsilon_1^{t_{11}}] \subseteq g_2[\varepsilon_2^{t_{22}}]$.

Proof. We proceed by case analysis on $g_1$.

Case ($\square \oplus U \text{ et}$). Then for $i \in \{1, 2\}$ $g_i$ must have the form $\square \oplus U'' \varepsilon_i^{t_i} t_i$ for some $U''$, $\varepsilon_i'$ and $t_i'$. As $g_1[\varepsilon_1^{t_{11}}] \subseteq g_2[\varepsilon_2^{t_{12}}]$ then by $\square \oplus \square, \varepsilon_1 \subseteq \varepsilon_2, \varepsilon_1' \subseteq \varepsilon_2'$, $U_1'' \subseteq U_2''$ and $t_1' \subseteq t_2'$.

As $\varepsilon_{12} \subseteq \varepsilon_{22}$ and $t_{12} \subseteq t_{22}$, then by $\square \oplus \square, \varepsilon_{12} t_{12} \oplus U'' \varepsilon_1^{t_1} t_1 \subseteq \varepsilon_{22} t_{22} \oplus U'' \varepsilon_2^{t_2} t_2$, and the result holds.

Case ($\square + \text{et, ev} \circ \square, \varepsilon \circ @ U \square, \square : U, \text{if } \square \text{ then et else et}$). Straightforward using similar argument to the previous case.

Proposition 249. Let $g_1, g_2 \in \text{EvFrame}$ such that $g_1[\varepsilon_1^{t_{11}}] \in \text{Term}_{U_1'}, g_2[\varepsilon_2^{t_{22}}] \in \text{Term}_{U_2'}$, with $U_1' \subseteq U_2'$. Then if $g_1[\varepsilon_1^{t_{11}}] \subseteq g_2[\varepsilon_2^{t_{22}}]$ then $t_{11} \subseteq t_{22}$ and $\varepsilon_1 \subseteq \varepsilon_2$.

Proof. We proceed by case analysis on $g_1$.

Case ($\square \oplus U \text{ et}$). Then there must exist some $U''$, $\varepsilon_1'$ and $t_1'$ such that $g_1[\varepsilon_1^{t_{11}}] = \varepsilon_1^{t_{11}} \oplus U'' \varepsilon_1^{t_1} t_1$ and $g_2[\varepsilon_2^{t_{22}}] = \varepsilon_2^{t_{22}} \oplus U'' \varepsilon_2^{t_2} t_2$. Then by the hypothesis and the premises of ($\square \oplus \square$), $t_1' \subseteq t_2'$ and $\varepsilon_1 \subseteq \varepsilon_2$, and the result holds immediately.

Case ($\square + \text{et, ev} \circ \square, \varepsilon \circ @ U \square, \square : U, \text{if } \square \text{ then et else et}$). Straightforward using similar argument to the previous case.

Proposition 250. Let $f_1, f_2 \in \text{EvFrame}$ such that $f_1[t_1^{t_{11}}] \in \text{Term}_{U_1'}, f_2[t_2^{t_{12}}] \in \text{Term}_{U_2'}$, with $U_1' \subseteq U_2'$. Then if $f_1[t_1^{t_{11}}] \subseteq f_2[t_2^{t_{12}}]$ and $t_{11} \subseteq t_{22}$, then $f_1[t_1^{t_{11}}] \subseteq f_2[t_2^{t_{12}}]$.

Proof. Suppose $f_1[t_1^{t_{11}}] = g_1[\varepsilon_1^{t_{11}}]$. We know that $g_1[\varepsilon_1^{t_{11}}] \in \text{Term}_{U_1'}, g_2[\varepsilon_2^{t_{12}}] \in \text{Term}_{U_2'}$ and $U_1' \subseteq U_2'$. Therefore if $g_1[\varepsilon_1^{t_{11}}] \subseteq g_1[\varepsilon_1^{t_{11}}]$, by Prop 249 $\varepsilon_1 \subseteq \varepsilon_2$. Finally by Prop 248 we conclude that $g_1[\varepsilon_1^{t_{12}}] \subseteq g_1[\varepsilon_1^{t_{12}}]$.

Proposition 251. Let $f_1, f_2 \in \text{EvFrame}$ such that $f_1[t_1^{t_{11}}] \in \text{Term}_{U_1'}, f_2[t_2^{t_{12}}] \in \text{Term}_{U_2'}$, with $U_1' \subseteq U_2'$. Then if $f_1[t_1^{t_{11}}] \subseteq f_2[t_2^{t_{12}}]$ then $t_{11} \subseteq t_{22}$.

Proof. Suppose $f_1[t_1^{t_{11}}] = g_1[\varepsilon_1^{t_{11}}]$. We know that $g_1[\varepsilon_1^{t_{11}}] \in \text{Term}_{U_1'}, g_2[\varepsilon_2^{t_{12}}] \in \text{Term}_{U_2'}$ and $U_1' \subseteq U_2'$. Therefore if $g_1[\varepsilon_1^{t_{11}}] \subseteq g_2[\varepsilon_2^{t_{12}}]$, then using Prop 249 we conclude that $t_{11} \subseteq t_{22}$.

Proposition 252 (Substitution preserves precision). If $\Omega \cup \{x^{t_{33}} \subseteq x^{t_{33}}\} \vdash t_{11} \subseteq t_{22}$ and $\Omega \vdash t_{33} \subseteq t_{44}$, then $\Omega \vdash [t_{33} / x^{t_{33}}] t_{11} \subseteq [t_{33} / x^{t_{33}}] t_{22}$.
Proof. By induction on the derivation of \( t^{U_1} \sqsubseteq t^{U_2} \), and case analysis of the last rule used in the derivation. All cases follow either trivially (no premises) or by the induction hypotheses.

\[ \vdash \]

**Proposition 253** (Monotone precision for \( \circ^= \)). If \( \varepsilon_1 \sqsubseteq \varepsilon_2 \) and \( \varepsilon_3 \sqsubseteq \varepsilon_4 \) then \( \varepsilon_1 \circ^= \varepsilon_3 \sqsubseteq \varepsilon_2 \circ^= \varepsilon_4 \).

Proof. By definition of consistent transitivity for = and the definition of precision.

\[ \vdash \]

**Proposition 254.** If \( U_{11} \sqsubseteq U_{12} \) and \( U_{21} \sqsubseteq U_{22} \) then \( U_{11} \sqcap U_{21} \sqsubseteq U_{12} \sqcap U_{22} \).

Proof. By induction on the type derivation of the types and meet.

\[ \vdash \]

**Proposition 255** (Dynamic guarantee for \( \rightarrow \)). Suppose \( \Omega \vdash t^{U_1}_1 \sqsubseteq t^{U_2}_1 \). If \( t^{U_1}_1 \rightarrow t^{U_1}_2 \) then \( t^{U_2}_1 \rightarrow t^{U_2}_2 \), where \( \Omega' \vdash t^{U_1}_2 \sqsubseteq t^{U_2}_2 \) for some \( \Omega' \supseteq \Omega \).

Proof. By induction on the structure of \( t^{U_1}_1 \sqsubseteq t^{U_2}_1 \). For simplicity we omit the \( \Omega \vdash \) notation on precision relations when it is not relevant for the argument.

**Case** (\( \rightarrow + \)). We know that \( t^{U_1}_1 = (\varepsilon_{11}(n_1) + \varepsilon_{12}(n_2)) \) then by (\( \sqcup + \)) \( t^{U_2}_1 = (\varepsilon_{21}(n_1) + \varepsilon_{22}(n_2)) \) for some \( \varepsilon_{21}, \varepsilon_{22} \) such that \( \varepsilon_{11} \subseteq \varepsilon_{21} \) and \( \varepsilon_{12} \subseteq \varepsilon_{22} \).

If \( t^{U_1}_1 \rightarrow n_3 \) where \( n_3 = (n_1 \sqcup n_2) \), then \( t^{U_2}_1 \rightarrow n'_3 \) where \( n'_3 = (n_1 \sqcup n_2) \). But \( n_3 = n'_3 \) and therefore \( t^{U_2}_1 \sqsubseteq t^{U_2}_2 \) and the result holds.

**Case** (\( \rightarrow \text{app} \)). We know that \( t^{U_1}_1 = \varepsilon_{11}(\lambda x^{U_{11}}. t^{U_{12}}) \at U_1 \rightarrow t^{U_2}_1 \varepsilon_{12} u \) then by (\( \sqcup \text{app} \)) \( t^{U_2}_1 \) must have the form \( t^{U_2}_1 = \varepsilon_{21}(\lambda x^{U_{21}}. t^{U_{22}}) \at U_3 \rightarrow t^{U_2}_2 \varepsilon_{22} u_2 \) for some \( \varepsilon_{21}, \varepsilon_{22} \) and \( U_{12}, U_{21}, U_{13}, U_{22}, U_3, U_4 \).

Let us pose \( \varepsilon_1 = \varepsilon_{12} \circ^= \text{idom}(\varepsilon_{11}) \). Then

\[ t^{U_1}_1 \rightarrow \text{icod}(\varepsilon_{11}) t^{U_1}_1 : U_2 \text{ with } t^{U_1}_1 = \{((\varepsilon_{11} u_1 :: U_{11})/x^{U_{11}}) t^{U_{12}}\}. \]

Also, let us pose \( \varepsilon_2 = \varepsilon_{22} \circ^= \text{idom}(\varepsilon_{21}) \). Then

\[ t^{U_2}_1 \rightarrow \text{icod}(\varepsilon_{21}) t^{U_2}_1 : U_4 \text{ with } t^{U_2}_1 = \{((\varepsilon_{22} u_2 :: U_{22})/x^{U_{21}}) t^{U_{22}}\}. \]

As \( \Omega \vdash \vdash t^{U_1}_1 \sqsubseteq t^{U_2}_1 \), then \( u_1 \sqsubseteq u_2 \), \( \varepsilon_{12} \subseteq \varepsilon_{22} \) and \text{idom}(\varepsilon_{11}) \subseteq \text{idom}(\varepsilon_{21}) \) as well, then by Prop 253 \( \varepsilon_1 \sqsubseteq \varepsilon_2 \). Then \( \varepsilon_1 u_1 :: U_{11} \sqsubseteq \varepsilon_2 u_2 :: U_{22} \) by (\( \sqsubseteq \)).

We also know by (\( \sqsubseteq \text{app} \)) and (\( \square \)) \( \Omega \sqcup \{x^{U_{21}} \sqsubseteq x^{U_{21}}\} \vdash t^{U_{12}} \sqsubseteq t^{U_{22}} \). By Substitution preserves precision (Prop 252), \( t^{U_1}_1 \sqsubseteq t^{U_2}_1 \) therefore \( \text{icod}(\varepsilon_{11}) t^{U_1}_1 :: U_2 \sqsubseteq \text{icod}(\varepsilon_{21}) t^{U_2}_1 :: U_4 \) by (\( \sqsubseteq \)). Then \( t^{U_1}_2 \sqsubseteq t^{U_2}_2 \).

**Case** (\( \rightarrow \text{if-true} \)). \( t^{U_1}_1 = \text{if} \varepsilon_{11} \text{true} \text{then} \varepsilon_{12} t^{U_{12}} \text{else} \varepsilon_{13} t^{U_{13}} \) then by (\( \sqsubseteq \text{if} \)) \( t^{U_2}_1 \) has the form \( t^{U_2}_1 = \text{if} \varepsilon_{21} \text{true} \text{then} \varepsilon_{22} t^{U_{22}} \text{else} \varepsilon_{23} t^{U_{23}} \) for some \( \varepsilon_{21}, \varepsilon_{22}, \varepsilon_{23}, \) and \( t^{U_{12}}, t^{U_{13}} \). Then \( t^{U_1}_1 \rightarrow \varepsilon_{12} t^{U_{12}} :: (U_{12} \sqcap U_{13}) \), and \( t^{U_2}_1 \rightarrow \varepsilon_{22} t^{U_{22}} :: (U_{22} \sqcap U_{23}) \). Using the fact that \( t^{U_1}_1 \sqsubseteq t^{U_2}_1 \) we know that \( \varepsilon_{12} \subseteq \varepsilon_{22} \), \( t^{U_{12}} \sqsubseteq t^{U_{22}} \) and by Prop 247 \( U_{12} \sqsubseteq U_{22} \) and \( U_{13} \sqsubseteq U_{23} \). Therefore by Prop 254 \( (U_{12} \sqcap U_{13}) \sqsubseteq (U_{22} \sqcap U_{23}) \). Then using (\( \sqsubseteq \)), \( t^{U_1}_2 \sqsubseteq t^{U_2}_2 \).

**Case** (\( \rightarrow \text{if-false} \)). Same as case \( \rightarrow \text{if-true} \), using the fact that \( \varepsilon_{13} \subseteq \varepsilon_{23} \) and \( t^{U_{13}} \sqsubseteq t^{U_{23}} \).

\[ \vdash \]

**Proposition 256** (Dynamic guarantee). Suppose \( t^{U_1}_1 \sqsubseteq t^{U_2}_1 \). If \( t^{U_1}_1 \rightarrow t^{U_1}_2 \) then \( t^{U_2}_1 \rightarrow t^{U_2}_2 \) where \( t^{U_1}_2 \sqsubseteq t^{U_2}_2 \).

343
Proof. We prove the following property instead: Suppose Ω ⊢ t′ 1 ⊑ t′ 2. If t′ 1 →→ t′ 1′ then t′ 2 →→ t′ 2′ where Ω′ ⊢ t′ 1′ ⊑ t′ 2′, for some Ω′ ⎷ Ω.

By induction on the structure of a derivation of t′ 1 ⊑ t′ 2. For simplicity we omit the Ω ⊢ notation on precision relations when it is not relevant for the argument.

Case (R→). Ω ⊢ t′ 1 ⊑ t′ 2, t′ 1 →→ t′ 2. By dynamic guarantee of → (Prop 255), t U 1, t U 2 →→ t U 1′ where Ω′ ⊢ t′ U 1 ⊑ t′ U 2 for some Ω′ ⎷ Ω. And the result holds immediately.

Case (Rf). t′ 1 = f 1[t′ 1], t′ 2 = f 2[t′ 2]. We know that Ω ⊢ f 1[t′ 1] ⊑ f 2[t′ 2]. By using Prop 247, U′ 1 ⊑ U′ 2. By Prop 251 we also know that Ω ⊢ t′ U 1 ⊑ t′ U 2. By induction hypothesis, t′ U 1 →→ t′ U 2, t′ U 2 →→ t′ U 2′, Ω′ ⊢ t′ U 1′ ⊑ t′ U 2′ for some Ω′ ⎷ Ω.

Then by Prop 250 then Ω′ ⊢ f 1[t′ U 1] ⊑ f 2[t′ U 2] and the result holds.

Case (Rg). t′ 1 = g 1[et 1], t′ 2 = g 2[et 2], where Ω ⊢ g 1[et 1] ⊑ g 2[et 2]. Also et 1 →→ et′ 1 and et 2 →→ et′ 2.

Then there exists U 1, ε 11, ε 12 and v 1 such that et 1 = ε 11(ε 12v 1 :: U 1). Also there exists U 2, ε 21, ε 22 and v 2 such that et 2 = ε 21(ε 22v 2 :: U 2). By Prop 249, ε 11 ⊑ ε 21, and by (⊔ v) ε 12 ⊑ ε 22, v 1 ⊑ v 2 and U 1 ⊑ U 2. Then as et 1 →→ (ε 11 ⊕ ε 12)v 1 and et 2 →→ (ε 21 ⊕ ε 22)v 2 then, by Prop 253 we know that ε 12 v 1 ⊑ ε 22 v 2. Then using this information, and the fact that v 1 ⊑ v 2, by Prop 248 it follows that Ω ⊢ g 1[et 1] ⊑ g 1[et 2].

\[ \square \]

C.2 Compiling GTFL$^\oplus$ to Threesomes

This sections presents the translational semantics of GTFL$^\oplus$. § C.2.1 presents the intermediate language. § C.2.2 presents the cast insertion rules. And § C.2.3 presents the full formalization of the correctness of the translational semantics.

C.2.1 Intermediate language: GTFL$^\oplus$️

Figure C.11 presents the syntax of GTFL$^\oplus$. Figure C.12 presents the full type system of GTFL$^\oplus$. Figure C.13 presents the full dynamic semantics of GTFL$^\oplus$. Applications of a non-casted function are standard. The reduction rule for additions and conditionals use the $rval$ metafunction, which strips away the surrounding cast, if any, to access the underlying value. The reduction of the application of a casted function is standard, splitting the function cast into a cast on the argument and a cast on the result. Two threesomes that coincide on their source/target types are combined by meeting their middle types. If the meet is undefined then the term steps to $error$. Otherwise both casts are merged to a new cast where the middle type is now the meet between the middle types. Note that new casts are introduced using the following cast metafunction, which avoids producing useless threesomes:

344
\[ T \in \text{TYPE}, \quad x \in \text{VAR}, \quad t \in \text{TERM}, \quad \Gamma \in \text{VAR} ^{\text{fin}} \to \text{TYPE} \]

\[
T ::= \text{Bool} \mid \text{Int} \mid T \to T \quad \text{(types)}
\]

\[
u ::= n \mid b \mid (\lambda x : U.t) \quad \text{(simple values)}
\]

\[
v ::= u \mid \langle U \Downarrow U \rangle u \quad \text{(values)}
\]

\[
t ::= v \mid x \mid tt \mid t + t \mid \text{if } t \text{ then } t \text{ else } t \mid t :: T \mid \langle U \Downarrow U \rangle t \quad \text{(terms)}
\]

Figure C.11: Syntax of the intermediate language

\[
\begin{align*}
(IT_x) & \quad \frac{x : U \in \Gamma}{\Gamma \vdash x : U} \\
(IT_b) & \quad \frac{b : \text{Bool}}{\Gamma \vdash b : \text{Bool}} \\
(IT_n) & \quad \frac{n : \text{Int}}{\Gamma \vdash n : \text{Int}} \\
(IT\lambda) & \quad \frac{\Gamma, x : U_1 \vdash t : U_2}{\Gamma \vdash (\lambda x : U_1.t) : U_1 \to U_2} \\
(IT\langle\rangle) & \quad \frac{U_1 \sim U_3 \quad U_3 \sim U_2}{\Gamma \vdash \langle U_2 \Downarrow U_1 \rangle t : U_2} \\
(IT\text{if}) & \quad \frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : U_2 \quad \Gamma \vdash t_3 : U_2}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : U_2}
\end{align*}
\]

Figure C.12: GTFL\textsuperscript{⊕}⇒: Type system for the intermediate language

\[
\langle\langle U_2 \Downarrow U_1 \rangle \rangle t = \begin{cases} t & \text{if } U_1 = U_2 = U_3 \\ \langle U_2 \Downarrow U_1 \rangle t & \text{otherwise} \end{cases}
\]

C.2.2 Cast Insertion

We now briefly describe the cast insertion translation from a GTFL\textsuperscript{⊕} term \( \tilde{t} \) to a GTFL\textsuperscript{⊕}⇒ term \( t \). The cast insertion rules are presented in Figure C.14. Cast insertion rules use twosomes to ease readability; a twosome \( \langle U_2 \Downarrow U_1 \rangle \) is equal to \( \langle\langle U_2 \Downarrow U_1 \rangle \rangle \) \( t \): the initial middle type is the meet of both ends [112]. Note that useless casts are not introduced by translation due to the use of the cast metafunction.

The key idea of the transformation is to insert casts in places where consistency is used to justify the typing derivation. For instance, if a term \( \tilde{t} \) of type \text{Int}⊕\text{Bool} is used where an \text{Int} is required, the translation inserts a cast \( \langle\text{Int} \Downarrow \text{Int}⊕\text{Bool} \rangle \) \( t \), where \( t \) is the recursive translation of \( \tilde{t} \). This cast plays the role of the implicit projection from the gradual union type. Dually, when a term of type \text{Int} is used where a gradual union is expected, the translation adds a cast
\[ u ::= \text{true} | \text{false} | n | \lambda x.t \]
\[ v ::= u | \langle U \xleftarrow{=} U \rangle u \]  \hspace{1cm} \text{(values)}
\[ f ::= \Box + t | v + \Box | \Box t | v \Box | \langle U \xleftarrow{=} U \rangle \Box \text{ if } \Box \text{ then } t \text{ else } t \]  \hspace{1cm} \text{(frames)}

### Notions of Reduction

\[ n_3 = \text{rval}(v_1) \langle[+]\rangle \text{rval}(v_2) \]
\[ v_1 + v_2 \rightarrow n_3 \]
\[ \lambda x.t \rightarrow [v/x]t \]  \hspace{1cm} \text{if } v \text{ then } t_1 \text{ else } t_2 \rightarrow \begin{cases} t_2 \text{ if } \text{rval}(v) = \text{true} \\ t_3 \text{ if } \text{rval}(v) = \text{false} \end{cases}

\[ \langle U_{21} \rightarrow U_{22} \xleftarrow{=} U_{11} \rightarrow U_{12} \rangle u v \rightarrow \langle U_{22} \xleftarrow{=} \text{idom}(U_2) \rangle \langle U_{11} \xleftarrow{=} \text{idom}(U_1) \rangle \langle U_{21} \xrightarrow{=} U_1 \rangle v \]

\[ \langle U_3 \xleftarrow{=} U_2 \rangle \langle U_2 \xleftarrow{=} U_1 \rangle v \rightarrow \begin{cases} \langle U_3 \xleftarrow{=} \text{U}32\cap U_{21} \rangle \text{U}1 \rangle v \text{ if } U_{32} \cap U_{21} \text{ is undefined} \end{cases} \]

### Reduction

\[ t_1 \rightarrow t_2 \]
\[ f[t_1] \rightarrow f[t_2] \]
\[ f[\text{error}] \rightarrow \text{error} \]

Figure C.13: GTFL\textsuperscript{⇒}. Dynamic Semantics of GTFL\textsuperscript{⇒}

that performs the implicit injection to the gradual union, \textit{e.g.} \langle \text{int} \oplus \text{bool} \xleftarrow{=} \text{int} \rangle 10. Note that a value with a cast that loses precision is like a tagged value in tagged union type systems; the difference again is that the “tag” is inserted implicitly.

Rule (C::) may insert a cast from the type of the body to the ascribed type. Rule (Capp) may insert two casts. By declaring that the resulting type of the application is \text{cod}(U_1) and that the argument is consistent with \text{dom}(U_1), we are implicitly assuming that \( U_1 \) is consistent with some function type, which justifies the cast on \( t_1 \). The second cast on \( t_2 \) comes from the consistent judgment \( U_2 \sim \text{dom}(U_1) \). Rule (C+) is similar.
(Cx) \[ \frac{x : U \in \Gamma}{\Gamma \vdash x : U} \]

(Cb) \[ \frac{}{\Gamma \vdash b : \text{Bool}} \]

(Cu) \[ \frac{}{\Gamma \vdash n \Rightarrow n : \text{Int}} \]

(Ca) \[ \frac{\Gamma, x : U_1 \vdash \bar{t} \Rightarrow \bar{t}' : U_2}{\Gamma \vdash (\lambda x : U_1.t) \Rightarrow (\lambda x : U_1.t') : U_1 \rightarrow U_2} \]

(Ca) \[ \frac{\Gamma \vdash \bar{t} \Rightarrow \bar{t}' : U \quad U \sim U_1}{\Gamma \vdash (\bar{t} :: U_1) \Rightarrow (U_1 \leftarrow U) \bar{t}' : U_1} \]

(Capp) \[ \frac{\Gamma \vdash \bar{t}_1 \Rightarrow \bar{t}_1' : U_1 \quad \Gamma \vdash \bar{t}_2 \Rightarrow \bar{t}_2' : U_2 \quad U_2 \sim \text{dom}(U_1)}{\Gamma \vdash \bar{t}_1 \bar{t}_2 \Rightarrow} \]

\[
\begin{align*}
\langle \text{dom}(U_1) \rightarrow \text{cod}(U_1) \leftarrow U_1 \rangle \bar{t}_1' & \quad \langle \text{dom}(U_1) \leftarrow U_2 \rangle \bar{t}_2' : \text{cod}(U_1)
\end{align*}
\]

(C+) \[ \frac{\Gamma \vdash \bar{t}_1 \Rightarrow \bar{t}_1' : U_1 \quad \Gamma \vdash \bar{t}_2 \Rightarrow \bar{t}_2' : U_2 \quad U_1 \sim \text{Int} \quad U_2 \sim \text{Int}}{\Gamma \vdash \bar{t}_1 + \bar{t}_2 \Rightarrow} \]

\[
\langle \text{Int} \leftarrow U_1 \rangle \bar{t}_1' + \langle \text{Int} \leftarrow U_2 \rangle \bar{t}_2' : \text{Int}
\]

(Cif) \[ \frac{\Gamma \vdash \bar{t}_1 \Rightarrow \bar{t}_1' : U_1 \quad U_1 \sim \text{Bool} \quad \Gamma \vdash \bar{t}_2 \Rightarrow \bar{t}_2' : U_2 \quad \Gamma \vdash \bar{t}_3 \Rightarrow \bar{t}_3' : U_3}{\Gamma \vdash \text{if } \bar{t}_1 \text{ then } \bar{t}_2 \text{ else } \bar{t}_3 \Rightarrow}} \]

\[
\begin{align*}
\text{if } \langle \text{Bool} \leftarrow U_1 \rangle \bar{t}_1' & \text{ then } \langle U_2 \cap U_3 \leftarrow U_2 \rangle \bar{t}_2' \text{ else } \langle U_2 \cap U_3 \leftarrow U_3 \rangle \bar{t}_3' : U_2 \cap U_3
\end{align*}
\]

Figure C.14: Cast insertion rules
**C.2.3 Correctness of the Translational Semantics**

This section presents all the definitions and properties used to prove the correctness of translational semantics.

One of the novelty of this work is to establish that the translational semantics of the gradual language enjoys both type safety and the gradual guarantees, without relying on the usual proof techniques. The typical approach is to prove type safety of a gradual language by first establishing type safety of the target cast calculus and second proving that the cast insertion translation preserves typing. Proving the gradual guarantees is a separate effort [**III**].

Here, we instead directly establish that the translation semantics is equivalent to the reference semantics derived with AGT. Because the reference semantics describes a type-safe gradual language that satisfies the gradual guarantees, so does the translational semantics. We establish the equivalence between the semantics using step-indexed: logical relations. We use step-indexed logical relations so the relation is well-founded: the definition without indexes may contain some vicious cycles in presence of gradual unions. Equivalence between two terms is acknowledged when either both evaluate to the same value, or both lead to an error.

Figure C.15 presents the logical relations between simple values, values and computations, which are defined mutually recursively. The logical relations are defined for pairs composed of an intrinsic term $\bar{t}$, which denotes the typing derivation for a GTFL $\beta$ term $\bar{t}$, and a GTFL $\beta$ term $t$. For simplicity, we write $t : U$ for $\cdot \vdash t : U$.

A pair of simple values $(\bar{u}_1, u_2)$ are related for $k$ steps at type $U$, notation $(\bar{u}_1, u_2) \in U_k[U]$. Figure C.15 presents the logical relations between intrinsic terms and cast calculus terms.

\[
\begin{align*}
(b_1, b_2) \in U_k[\mathbf{Bool}] & \iff b_1 \in \text{TERM}_{\mathbf{Bool}} \land b_2 : \mathbf{Bool} \land b_1 = b_2 \\
(n_1, n_2) \in U_k[\mathbf{Int}] & \iff n_1 \in \text{TERM}_{\mathbf{Int}} \land n_2 : \mathbf{Int} \land n_1 = n_2 \\
(\bar{u}_1, u_2) \in U_k[U_1 \rightarrow U_2] & \iff \bar{u}_1 \in \text{TERM}_{U_1 \rightarrow U_2} \land u_2 : U_1 \rightarrow U_2 \land \\
& \forall U' = U'' \rightarrow U''', \varepsilon_1 \vdash U_1 \rightarrow U_2 \rightarrow U''', \text{ and} \\
& \varepsilon_2 \vdash U_1' \sim U''', \text{ we have: } \forall j \leq k, (\bar{v}_1, v_2) \in V_j[U_1'][j], \\
& (\varepsilon_1 \bar{u}_1 @ U' \vdash_2 \bar{v}_1, (\bar{u}' \vdash_2 U_1 \rightarrow U_2)u_2 (U_1'' \vdash_2 U_1')v_2) \in T_j[U_2'][j] \\
(\varepsilon \bar{u}_1 :: U, u_2) \in V_k[U] & \iff \varepsilon \bar{u}_1 :: U \in \text{TERM}_U \land \varepsilon = U \land (\bar{u}_1, u_2) \in U_k[U] \\
(\varepsilon \bar{u}_1 :: U, (U \vdash_2 U')u_2) \in V_k[U] & \iff \varepsilon \bar{u}_1 :: U \in \text{TERM}_U \land (\bar{u}_1, u_2) \in U_{k-1}[U][k] \\
(\bar{u}_1, u_2) \in V_k[U] & \iff (\bar{u}_1, u_2) \in U_k[U] \\
(\bar{t}_1, t_2) \in T_k[U] & \iff \bar{t}_1 \in \text{TERM}_U \vdash t_2 : U \land \forall j < k \\
& (\bar{t}_1 \rightarrow^j \bar{v}_1 \Rightarrow (t_2 \rightarrow^* v_2 \land (\bar{v}_1, v_2) \in V_{k-j}[U][k]) \land \\
& (t_2 \rightarrow^k v_2 \Rightarrow (\bar{t}_1 \rightarrow^* \bar{v}_1 \land (\bar{v}_1, v_2) \in V_{k-j}[U][k]) \land \\
& (\bar{t}_1 \rightarrow^j \text{error} \Rightarrow t_2 \rightarrow^* \text{error}) \land \\
& (\bar{t}_2 \rightarrow^j \text{error} \Rightarrow t_1 \rightarrow^* \text{error})
\end{align*}
\]

Figure C.15: Logical relations between intrinsic terms and cast calculus terms.
if they both have the same type \( U \) and, if \( U \) is either \texttt{Bool} or \texttt{Int}, then the values are also equal. If the simple values are functions, then they are related if their application to related arguments, for \( j \leq k \) steps, yields related computations, as explained below. Note that the relation between simple values need not consider the case of gradual types, as no literal values have gradual types.

A pair of values \((\bar{v}_1, v_2)\) are related for \( k \) steps at type \( U \), notation \((\bar{v}_1, v_2) \in \mathcal{U}_k[U]\), if both have the same type and their underlying simple values are related. One important point to notice is that we may only relate an ascribed value \( \varepsilon \bar{u}_1 :: U \) to a simple value \( v_2 \) if we do not learn anything new from the ascription, i.e. both the type of \( \bar{u}_1 \) and the evidence \( \varepsilon \) are \( U \). This corresponds to the case where the reference semantics carries useless evidence—recall that the cast insertion translation does not insert useless casts. Additionally, an ascribed value \( \varepsilon \bar{u}_1 :: U \) is related to a casted value if the evidence and ascription correspond to the threesome. More precisely, the evidence \( \varepsilon \) must be exactly the middle type of the threesome, and the source and target types of the threesome must correspond to the type of \( \bar{u}_1 \) and the ascribed type \( U \), respectively. Finally, in order to reason about the underlying simple values, the ascription and cast must be eliminated by combining them with an evidence and a cast respectively. Because of this extra step, the underlying simple values must be related for \( k - 1 \) steps instead.

A pair of terms \((\bar{t}_1, t_2)\) are related computations for \( k \) steps at type \( U \), notation \((\bar{t}_1, t_2) \in \mathcal{T}_k[U]\), if both terms have the same type \( U \), then either both terms reduce to related values at type \( U \), or both terms reduce to an error. Formally, for any \( j < k \), if the evaluation of the intrinsic term \( \bar{t}_1 \) terminates in a value \( v_1 \) at least in \( j \) steps, then the compiled term \( t_2 \) also reduces to a value \( v_2 \), and the resulting values are related values for \( k - j \) steps at type \( U \). Analogously, if the evaluation of the compiled term \( t_2 \) reduces to a value \( v_2 \) at least in \( j \) steps, then the intrinsic term \( \bar{t}_1 \) also reduces to a related value \( v_1 \). Finally, if either term reduces to an error in at least \( j \) steps, then the other also reduces to an error. Note that this last condition is only required because we do not assume type safety of \( \text{GTFL}^\oplus \).

Armed with these logical relations we can state the notion of semantic equivalence between a \( \text{GTFL}^\oplus \) intrinsic term and a \( \text{GTFL}^\oplus \) term.

**Definition 116** (Semantic equivalence). Let \( \bar{t} \in \text{TERM}_U \), \( \Gamma = FV(\bar{t}) \) and a \( \text{GTFL}^\oplus \) term \( t \) such that \( \Gamma \vdash_1 t : U \). We say that \( \bar{t} \) and \( t \) are semantically equivalent, notation \( \bar{t} \equiv t : U \), if and only if for any \( k \geq 0 \), \((\sigma_1, \sigma_2) \in \mathcal{G}_k[\Gamma]\), we have \((\sigma_1(\bar{t}), \sigma_2(t)) \in \mathcal{T}_k[U]\).

The definition of semantic equivalence appeals to the notion of related substitutions. Two substitutions \( \sigma_1 \) and \( \sigma_2 \) are related for \( k \) steps at type environment \( \Gamma \), notation \((\sigma_1, \sigma_2) \in \mathcal{G}_k[\Gamma]\), if they map each variable in \( \Gamma \) to related values (full definition in [C.2.3]).

Note that we write \( \bar{t} \) instead of \( t^U \) when it is clear from the context that it is an intrinsic term. Also note that \( t : U \equiv \cdot \vdash_1 t : U \).

**Definition 117.** Let \( \sigma \) be a substitution and \( \Gamma \) a type substitution. We say that substitution \( \sigma \) satisfy environment \( \Gamma \), written \( \sigma \models \Gamma \), if and only if \( \text{dom}(\sigma) = \text{dom}(\Gamma) \).

**Definition 118** (Related substitutions). Let \( \sigma_1 \) be a substitution function from intrinsic
variables to intrinsic values, and let $\sigma_2$ be a substitution function from variables to values from the intermediate language. Then we define related substitution as follows:

$$(\sigma_1, \sigma_2) \in G_k[\Gamma] \iff \sigma_1 \vdash \Gamma \land \forall x \in \Gamma. (\sigma_1(x^{\Gamma(x)}), \sigma_2(x)) \in V_k[\Gamma(x)]$$

**Lemma 257 (Reduction preserves relations).** Consider $\Gamma \vdash \tilde{t} : U$, $t^U \in \text{TERM}_U$ and $\Gamma \vdash \tilde{t} \Rightarrow t : U$. Consider $k, j > 0$, if $t^U \rightarrow^j t'^U$ and $t \rightarrow^j t'$, then we have $(t^U, t) \in T_k[U]$ if and only if $(t'^U, t') \in T_{k-j}[U]$.

**Proof.** The $\Rightarrow$ direction relies on the determinism of the reduction relation and the definition of related computations. The $\Leftarrow$ direction follows direct from the definition of $(t^U, t) \in T_k[U]$ and transitivity of $\rightarrow$.

**Lemma 258.** If $(\tilde{t}_1, t_2) \in T_k[U]$ then if $\varepsilon \vdash U \sim U'$, then $(\varepsilon \tilde{t}_1 :: U', (U' \approx U)t_2) \in T_{k+1}[U']$.

**Proof.** If either term reduce to an error then the lemma trivially holds. If either one of the term reduce to a value, then it holds by definition of related values and Lemma 257.

**Lemma 259.** Consider $k > 0$. If $(\tilde{t}_1, t_2) \in T_k[U]$ then $(\tilde{t}_1, t_2) \in T_{k-1}[U]$.

**Proof.** Trivial by definition of related computations, as $(\tilde{t}_1, t_2) \in T_k[U]$ is a stronger property than $(\tilde{t}_1, t_2) \in T_{k-1}[U]$.

Finally, semantic equivalence between the reference and the translational semantics says that given a well-typed term $\tilde{t}$ from the gradual source language, its corresponding intrinsic term $\tilde{t}$ is semantically equivalent to the cast calculus term $t$ obtained after the cast insertion translation.

**Proposition 260 (Equivalence of reference and translational semantics).** If $\Gamma \vdash \tilde{t} : U$, represented as the intrinsic term $\tilde{t} \in \text{TERM}_U$, and $\Gamma \vdash \tilde{t} \Rightarrow t : U$, then $\tilde{t} \approx t : U$.

We open the proposition to prove this instead:

If $\Gamma \vdash \tilde{t} : U$, $t^U \in \text{TERM}_U$, $\Gamma \vdash \tilde{t} \Rightarrow t : U$, then $\forall k \geq 0, (\sigma_1, \sigma_2) \in G_k[\Gamma], (\sigma_1(t^U), \sigma_2(t)) \in T_k[\Gamma]$.  

**Proof.** By induction on the type derivation of $\tilde{t}$.

**Case (Ub).** Then $\tilde{t} = b$ and therefore:

$$\Gamma \vdash b : \text{Bool}$$

where $U = \text{Bool}$. Then the corresponding intrinsic term is:

$$b \in \text{TERM}_{\text{Bool}}$$
and the type derivation of the compiled term is:

\[(ITb) \quad \Gamma \vdash b : \text{Bool}\]

But \(\sigma_1(b) = b, \sigma_2(b) = b\), and \(b = b\) and the result holds immediately.

**Case \((Un)\).** Then \(\tilde{t} = n\) and therefore:

\[(Un) \quad \Gamma \vdash n : \text{Int}\]

where \(U = \text{int}\). Then the corresponding intrinsic term is:

\[(IU/n) \quad n \in \text{TERM}_{\text{int}}\]

and the type derivation of the compiled term is:

\[(ITn) \quad \Gamma \vdash \tilde{t} : \text{Int}\]

But \(\sigma_1(n) = n, \sigma_2(n) = n\), and \(n = n\) and the result holds immediately.

**Case \((Ux)\).** Then \(\tilde{t} = x\) and therefore:

\[(Ux) \quad x : U \in \Gamma \quad \Gamma \vdash x : U\]

Then the corresponding intrinsic term is:

\[(IUx) \quad x \in \text{TERM}_U\]

and the type derivation of the compiled term is:

\[(ITx) \quad \Gamma \vdash \tilde{t} : U\]

As \((\sigma_1, \sigma_2) \in G_k[\Gamma]\) and \(x \in \text{dom}(\Gamma)\), then \((x^U, x) \in \mathcal{V}_k[U]\) and the result holds immediately.

**Case \((U\lambda)\).** Then \(\tilde{t} = (\lambda x : U_1, \tilde{t}_2)\) and therefore:

\[(U\lambda) \quad \Gamma, x : U_1 \vdash \tilde{t}_2 : U_2 \quad \Gamma \vdash (\lambda x : U_1, \tilde{t}_2) : U_1 \rightarrow U_2\]

where \(U = U_1 \rightarrow U_2\). Then the corresponding intrinsic term is:

\[(IU\lambda) \quad \tilde{t}_{U_2} \in \text{TERM}_{U_2} \quad (\lambda x^{U_1}, \tilde{t}_2^{U_2}) \in \text{TERM}_{U_1 \rightarrow U_2}\]

and the type derivation of the compiled term is:

\[(IT\lambda) \quad \Gamma, x : U_1 \vdash \tilde{t}_2 : U_2 \quad \Gamma \vdash (\lambda x : U_1, \tilde{t}_2) : U_1 \rightarrow U_2\]

where \(\Gamma, x : U_1 \vdash \tilde{t}_2 \Rightarrow t'_2 : U_2\). Consider \(j \leq k, U' = U''_1 \rightarrow U''_2, \varepsilon_1 \vdash U_1 \rightarrow U_2 \sim U'_1 \sim U''_2, \varepsilon_2 \vdash U'_1 \sim U''_2, \tilde{v}_1\), and \(\tilde{v}_2\), such that \((\tilde{v}_1', \tilde{v}_2') \in \mathcal{V}_j[U'_1]\). We have to prove that:

\[\langle \varepsilon_1(\lambda x^{U_1}, \sigma_1(t'_2)) \rangle @U' \varepsilon_2 \tilde{v}_1', \langle U' \Downarrow U_1 \rightarrow U_2 \rangle (\lambda x : U_1, \sigma_2(t_2)) \langle U'_1 \Downarrow U'_2 \rangle \tilde{v}_2' \in \mathcal{V}_j[U''_1]\]

Then we proceed depending on the structure of \((\tilde{v}_1', \tilde{v}_2')\), but ultimately we converge to analogous cases where the argument are just related simple values.
1. If \((\bar{v}_1, v_2') = (\bar{u}_1', u_2') \in V_k[U'_1]\), then \((\bar{u}_1', u_2') \in \mathcal{U}_k[U'_1]\). Therefore

\[
\langle U' \xleftarrow{\varepsilon_2} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'_1 \xleftarrow{\varepsilon_1} U'_1 \rangle u_2 \\
\longrightarrow (U'' \xleftarrow{\text{icod}(\varepsilon_1)} U_2)(\lambda x : U_1.\sigma_2(t_2)) \langle U'_1 \xleftarrow{\text{idom}(\varepsilon_1)} U'_1 \rangle (U'' \xleftarrow{\varepsilon_1} U'_1) u_2)
\]

(a) If \(\varepsilon_2 \circ \varepsilon_1 \neq \text{idom}(\varepsilon_1)\) is not defined, then

\[
\varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) @^{U'} \varepsilon_2 \bar{u}_1' \longrightarrow \text{error.}
\]

But by definition of consistent transitivity \(\varepsilon_2 \cap \text{idom}(\varepsilon_1) = \emptyset\) and therefore

\[
\langle U' \xleftarrow{\varepsilon_2} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'_1 \xleftarrow{\varepsilon_1} U'_1 \rangle u_2' \longrightarrow \text{error and the result holds.}
\]

(b) If \(\varepsilon' = \varepsilon_2 \circ \varepsilon_1 \neq \text{idom}(\varepsilon_1)\) is defined, then

\[
\varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) @^{U'} \varepsilon_2 \bar{u}_1' \\
\longrightarrow \text{icod}(\varepsilon_1)(\varepsilon' \bar{u}_1' :: U_1/x^{U_1}\sigma_1(t_2^{U_2})) :: U'' \\
= \text{icod}(\varepsilon_1)(\sigma_1[x^{U_1} \mapsto \varepsilon' \bar{u}_1'] :: U_1(t_2^{U_2})) :: U''
\]

and, let us suppose than \(\neg(U'_1 = U''_1 = \varepsilon')\) (the other case is similar modulo one step of evaluation)

\[
\langle U' \xleftarrow{\varepsilon_2} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'_1 \xleftarrow{\varepsilon_1} U'_1 \rangle u_2' \\
\longrightarrow^2 (U'' \xleftarrow{\text{icod}(\varepsilon_1)} U_2)[(U_1 \xleftarrow{\varepsilon'} U'_1) u_2'/x] \sigma_2(t_2) \\
= (U'' \xleftarrow{\text{icod}(\varepsilon_1)} U_2) \sigma_2[x \mapsto (U_1 \xleftarrow{\varepsilon'} U'_1) u_2'](t_2)
\]

As \((\bar{u}_1', u_2') \in U_j[U'_1]\) then by definition of related values,

\((\varepsilon' \bar{u}_1' :: U_1, \langle U_1 \xleftarrow{\varepsilon'} U'_1 \rangle u_2') \in V_{j+1}[U],\) then by Lemma 259

\((\varepsilon' \bar{u}_1' :: U_1, \langle U_1 \xleftarrow{\varepsilon'} U'_1 \rangle u_2') \in V_j[U].\) Therefore by definition of related substitutions,

\((\sigma_1[x^{U_1} \mapsto \varepsilon' \bar{u}_1'] :: U_1), \sigma_2[x \mapsto (U_1 \xleftarrow{\varepsilon'} U'_1) u_2']) \in G_j[\Gamma, x : U_1].\) Then by induction hypothesis on pair \((t_2^{U_2}, \Gamma, x : U_1 \vdash t_2 : U_2),\)

\(((\sigma_1[x^{U_1} \mapsto \varepsilon' \bar{u}_1'] :: U_1)(t_2^{U_2})), \sigma_2[x \mapsto (U_1 \xleftarrow{\varepsilon'} U'_1) u_2'](t_2)) \in \mathcal{F}_j[U'2]
\]

and the result holds by Lemma 259 and backward preservation of the relations (Lemma 257).

2. If \((\bar{v}_1', v_2') = (\bar{u}_1', u_2') \in V_j[U'_1]\), then \(\varepsilon = U'_1\) and \((\bar{u}_1', u_2') \in \mathcal{U}_j[U'_1]\).

Then \(\varepsilon' = \varepsilon \circ \varepsilon_2\) is defined because \(U'_1 \sim U''_1\), and therefore:

\(\varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) @^{U''} \varepsilon_2 \bar{u}_1' \longrightarrow \varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) @^{U'} \varepsilon_2 \bar{u}_1'.\) Then we proceed analogous to (1) and the result holds.

3. If \((\bar{v}_1', v_2') = (\bar{u}_1', \langle U'_1 \xleftarrow{\varepsilon} U'' \rangle u_2') \in V_j[U'_1]\), then \((\bar{u}_1', u_2') \in \mathcal{U}_j[U']\), for some \(U'\) such that \(\varepsilon \vdash U'' \sim U'_1\).

(a) If \(\varepsilon \circ \varepsilon_2\) is not defined then \(\varepsilon \vdash \varepsilon_2 = \emptyset,\)

\(\varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) @^{U'} \varepsilon_2 \bar{u}_1' :: U_1 \longrightarrow \text{error and}\)
\( \langle U' \xleftarrow{u_1} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'' \xleftarrow{u_2} U'_1 \rangle \langle U'_1 \xleftarrow{v_1} U' \rangle u'_2 \rightarrow \text{error}, \) and the result holds.

(b) If \( \varepsilon'_2 = \varepsilon \circ^= \varepsilon_2 \) is defined then

\[ \varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) \@^{U'} \varepsilon_2 \varepsilon u'_1 :: U'_1 \rightarrow \varepsilon_1(\lambda x^{U_1}.\sigma_1(t_2^{U_2})) \@^{U'} \varepsilon'_2 u'_1 \]

and

\[ \langle U' \xleftarrow{u_1} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'' \xleftarrow{u_2} U'_1 \rangle \langle U'_1 \xleftarrow{v_1} U' \rangle u'_2 \rightarrow \]

\[ \langle U' \xleftarrow{u_1} U_1 \rightarrow U_2 \rangle (\lambda x : U_1.\sigma_2(t_2)) \langle U'' \xleftarrow{u_2} U'_1 \rangle \]

and we proceed analogous to (1) and the result holds.

**Case** \((U ::)\). Then \( \bar{t} = t_1 :: U \) and therefore:

\[
(\bar{t} ::) \quad \Gamma \vdash t_1 : U' \quad U' \sim U \quad \Gamma \vdash t_1 :: U : U
\]

Then the corresponding non ascribed intrinsic term is:

\[
(\bar{t} ::) \quad \Gamma \vdash t_1^{U'} \in \text{TERM}_{U'} \quad \varepsilon \vdash U' \sim U \quad \varepsilon t_1^{U'} :: U \in \text{TERM}_U
\]

Let us assume \( U' \neq U \) (the other case is analogous). The type derivation of the compiled term is:

\[
(\text{IT}<>) \quad \Gamma \vdash t_1 : U' \quad \Gamma \vdash (U \xleftarrow{u_1} U') t_1
\]

Then we have to prove that

\[
(\varepsilon \sigma_1(t_1^{U'})) :: U, (U \xleftarrow{u_1} U') \sigma_2(t_1') \in \mathcal{F}_k[U]
\]

By induction hypotheses on \( \bar{t}_1 \) \((\sigma_1(t_1^{U'}), \sigma_2(t_1')) \in \mathcal{F}_k[U']\). If either term reduces to an error in less than \( k \) steps, then the result holds immediately. The interesting case if they reduce to related values in less than \( k \) steps. Then suppose \( \sigma_1(t_1^{U_1}) \rightarrow^j \bar{v}_1, \sigma_2(t_1') \rightarrow^* v_1' \), where \( j < k \) \((\bar{v}_1, v_1') \in \mathcal{V}_{k-j}[U_1]\).

Let us assume \( \bar{v}_1 = \bar{u}_1 \) then \( v_1' = u_1' \) (if they are ascribed values, then the argument is similar modulo one extra step of evaluation, where a runtime error may be produced).

\[
(\varepsilon \sigma_1(t_1^{U''})) :: U \rightarrow^j \varepsilon \bar{u}_1 :: U
\]

and

\[
(\varepsilon \bar{u}_1 :: U, (U \xleftarrow{u'_1} U') u'_1) \in \mathcal{V}_{k-j}[U]
\]

but \((\bar{v}_1, v_1') \in \mathcal{V}_{k-j}[U_1]\) and by Lemma \(259\) \((\bar{v}_1, v_1') \in \mathcal{V}_{k-j-1}[U_1]\), and the result holds by backward preservation lemma.
Case \((U_{\text{app}})\). Then \(\overline{t} = \overline{t}_1 \overline{t}_2\) and therefore:

\[
\begin{align*}
\frac{
\Gamma \vdash \overline{t}_1 : U_1 & \quad \Gamma \vdash \overline{t}_2 : U_2 & \quad U_2 \sim \text{dom}(U_1)
}{\Gamma \vdash \overline{t}_1 \overline{t}_2 : \text{cod}(U_1)}
\end{align*}
\]

and \(U = \text{cod}(U_1)\). Then the corresponding intrinsic term is:

\[
\begin{align*}
\frac{
\Gamma \vdash t^{U_1} \in \text{TERM}_{U_1} & \quad \varepsilon_1 \vdash U_1 \sim U_{11} \rightarrow U_{12} & \quad t^{U_2} \in \text{TERM}_{U_2} & \quad \varepsilon_2 \vdash U_2 \sim U_{11}
}{\langle \varepsilon_1 t^{U_1} \rangle \rightarrow^{U_{11} \rightarrow U_{12}} \langle \varepsilon_2 t^{U_2} \rangle \in \text{TERM}_{U_{12}}}
\end{align*}
\]

where \(U_{11} = \text{dom}(U_1)\) and \(U_{12} = \text{cod}(U_1)\).

We proceed assuming that the compilation always inserts casts (the other cases are similar because then the evidences are equal to the types in the judgment. Therefore the next combinations of those evidences are redundant and never fails). Then suppose \(U_1 \neq U_{11} \rightarrow U_{12}\) and \(U_2 \neq U_{11}\). As \(\varepsilon_1 = U_1 \cap U_{11} \rightarrow U_{12}\), and \(\varepsilon_2 = U_2 \cap U_{11}\), the type derivation of the compiled term is:

\[
\begin{align*}
\frac{
\Gamma \vdash t'_1 : U_1 & \quad \Gamma \vdash t'_2 : U_{11}
}{\langle \Gamma \vdash U_{11} \rightarrow U_{12} \vdash U_1 t'_1 & \quad \langle \Gamma \vdash U_{11} \vdash U_2 t'_2 : U_{12}\rangle}
\end{align*}
\]

Then we have to prove that

\[
\langle \langle \varepsilon_1 \sigma_1(t^{U_1}) \rangle \rangle \rightarrow^{U_{11} \rightarrow U_{12}} \langle \varepsilon_2 \sigma_1(t^{U_2}) \rangle, \langle \langle U_{11} \vdash U_{12} \vdash U_1 \sigma_2(t'_1) \rangle \rangle \langle \langle U_{11} \vdash U_2 \vdash U_2 \rangle \rangle \rangle \in \mathcal{J}_k[U_{12}]
\]

By induction hypotheses on \(\overline{t}_1\) and \(\overline{t}_2\) \((\sigma_1(t^{U_1}), \sigma_2(t'_1)) \in \mathcal{J}_k[U_1]\) and \((\sigma_1(t^{U_2}), \sigma_2(t'_2)) \in \mathcal{J}_k[U_2]\). If either term reduces to an error in less than \(k\) steps, then the result holds immediately. The interesting case if they reduce to related values in less than \(k\) steps. Then suppose \(\sigma_1(t^{U_1}) \rightarrow^j \hat{v}_1, \sigma_1(t^{U_2}) \rightarrow^j \hat{v}_2, \sigma_2(t'_1) \rightarrow^\ast v'_1, \sigma_2(t'_2) \rightarrow^\ast v'_2\), where \(j < k\) \((\hat{v}_1, v'_1) \in \mathcal{V}_{k-j}[U_{11}]\) and \((\hat{v}_2, v'_2) \in \mathcal{V}_{k-j}[U_{12}]\). If \(\hat{v}_1 = \hat{u}_1\) then \(v'_1 = v'_1\), by canonical forms the simple values must be lambdas and the proof follows from Case\((U_{\lambda})\). If \(\hat{v}_1 = \hat{u}'_1 \hat{u}_1 :: U_1\) and \(\hat{u}_1 \in \text{TERM}_{U_1}\), then suppose \(v'_1 = \langle \langle U_1 \vdash U_1' \rangle \rangle u'_1\) (the other case is similar but the evidence combination never fails). Also \((\hat{u}_1, u'_1) \in \mathcal{V}_{k-j}[U_{11}]\). Suppose \(\varepsilon'_1 \circ \varepsilon_1\) is not defined (which is equivalent to \(\varepsilon'_1 \cap \varepsilon_1 = \emptyset\)), therefore

\[
\langle \langle \varepsilon_1 \varepsilon'_1 \hat{u}_1 :: U_1 \rangle \rangle \rightarrow^{U_{11} \rightarrow U_{12}} \langle \varepsilon_2 \hat{v}_2 \rangle \implies \text{error}
\]

and the result follows. Suppose \(\varepsilon''_1 = \varepsilon'_1 \circ \varepsilon_1\) is defined. Then

\[
\langle \langle \varepsilon_1 \varepsilon'_1 \hat{u}_1 :: U_1 \rangle \rangle \rightarrow^{U_{11} \rightarrow U_{12}} \langle \varepsilon_2 \hat{v}_2 \rangle \implies \langle \langle \hat{u}_1 \rangle \rangle \rightarrow^{U_{11} \rightarrow U_{12}} \langle \varepsilon'_2 \hat{u}_1 \rangle \rightarrow^{U_{11} \rightarrow U_{12}} v'_2
\]

which is exactly the definition of related functions and then the result holds by backward preservation of the relations (Lemma \[257\]) and Lemma \[259\].
Case (Uif). Then \( \bar{t} = \bar{t}_1 \bar{t}_2 \) and therefore:

\[
(U\text{if}) \quad \Gamma \vdash \bar{t}_1 : U_1 \quad U_1 \sim \text{Bool} \quad \Gamma \vdash \bar{t}_2 : U_2 \quad \Gamma \vdash \bar{t}_3 : U_3 \\
\therefore \Gamma \vdash \text{if } \bar{t}_1 \text{ then } \bar{t}_2 \text{ else } \bar{t}_3 : U_2 \cap U_3
\]

and \( U = U_2 \cap U_3 \). Then the corresponding intrinsic term is:

\[
U \in \text{TERM}_{U_1} \quad \epsilon_1 \vdash U_1 \sim \text{Bool} \quad U = (U_2 \cap U_3) \\
U \in \text{TERM}_{U_2} \quad \epsilon_2 \vdash U_2 \sim U \quad U \in \text{TERM}_{U_3} \quad \epsilon_3 \vdash U_3 \sim U \\
\therefore \text{if } \epsilon_1 \bar{t}_1 \text{ then } \epsilon_2 \bar{t}_2 \text{ else } \epsilon_3 \bar{t}_3 \in \text{TERM}_U
\]

We proceed assuming that the compilation always inserts casts (the other cases are similar because then the evidences are equal to the types in the judgment. Therefore the next combinations of those evidences are redundant and never fails). Then suppose \( U_1 \neq U_{11} \rightarrow U_{12} \) and \( U_2 \neq U_{11} \). As \( \epsilon_1 = U_1 \cap U_{11} \rightarrow U_{12} \), and \( \epsilon_2 = U_2 \cap U_{11} \), the type derivation of the compiled term is:

\[
(C\text{if}) \quad \Gamma \vdash \bar{t}_1 : U_1 \quad U_1 \sim \text{Bool} \quad \Gamma \vdash \bar{t}_2 : U_2 \quad \Gamma \vdash \bar{t}_3 : U_3 \\
\therefore \text{if } (\text{Bool } \equiv U_1) \bar{t}_1 \text{ then } (U_2 \cap U_3 \equiv U_2) \bar{t}_2 \text{ else } (U_2 \cap U_3 \equiv U_3) \bar{t}_3 \in U_2 \cap U_3
\]

where \( \Gamma \vdash \bar{t}_1 \Rightarrow \bar{t}_1 ': U_1 \), \( \Gamma \vdash \bar{t}_2 \Rightarrow \bar{t}_2 ': U_2 \), and \( \Gamma \vdash \bar{t}_3 \Rightarrow \bar{t}_3 ': U_3 \).

But by definition of substitution, \( \sigma_1(t'U) = \text{if } \epsilon_1 \sigma_1(tU') \text{ then else } \epsilon_2 \sigma_1(tU') \) and \( \sigma_2(t') = \text{if } (\text{Bool } \equiv U_1) \sigma_2(t_1') \text{ then } (U_2 \cap U_3 \equiv U_2) \sigma_2(t_2') \text{ else } (U_2 \cap U_3 \equiv U_3) \sigma_2(t_3') \).

By induction hypotheses on \( \bar{t}_1, \bar{t}_2 \) and \( \bar{t}_3 \), \( (\sigma_1(tU_1), \sigma_2(t_1')) \in \mathcal{F}_k[\{U_1\}] \) and \( (\sigma_1(tU_2), \sigma_2(t_2')) \in \mathcal{F}_k[\{U_2\}] \). If either \( \sigma_1(tU_1) \) or \( \sigma_2(t_1') \) term reduces to an error then the result holds immediately. The interesting case is when they reduce to related values. Then suppose \( \sigma_1(tU_1) \rightarrow j \bar{v}_1, \sigma_2(t_1') \rightarrow j \bar{v}_1 \), where \( (\bar{v}_1, \bar{v}_1') \in \mathcal{V}_{k-j}[\{U_1\}] \) and \( (\bar{v}_2, \bar{v}_2') \in \mathcal{V}_{k-j}[\{U_2\}] \).

1. If \( \bar{v}_1 = \bar{v}_1 \) then \( \bar{v}_1 = \bar{v}_1 \), by canonical forms, \( \bar{v}_1 \) must be booleans \( b^{\text{Bool}} \) and \( b \). Suppose that \( b = \text{true} \) (the other case is analogous). Then \( tU \rightarrow j \epsilon_2 \sigma_1(tU') \) :: \( U_2 \cap U_3 \) and \( t' \rightarrow j (U_2 \cap U_3 \equiv U_2) \sigma_2(t_2') \). Then as \( (\sigma_1(tU_2), \sigma_2(t_2')) \in \mathcal{F}_k[\{U_2\}] \) and Lemma 258, and backward preservation of the relation the result holds.

2. If \( \bar{v}_1 = \epsilon_1' \bar{u}_1 \) and \( \bar{u}_1 \in \text{TERM}_{U_1'} \), then suppose \( \bar{v}_1 = \epsilon_1' \bar{u}_1 \) (the other case is similar but the evidence combination never fails). Also \( (\bar{u}_1, \bar{u}_1') \in \mathcal{U}_{k-j}[\{U_1\}] \). Suppose \( \epsilon_1' \circ = \epsilon_1 \) is not defined (which is equivalent to \( \epsilon_1' \cap \epsilon_1 = \emptyset \)), therefore

\[
\text{if } \epsilon_1 \epsilon_1' \bar{u}_1 : U_1 \text{ then } \epsilon_2 \sigma_1(tU') \text{ else } \epsilon_3 \sigma_1(tU') \rightarrow \text{error} \iff
\]

\[
\text{if } (\text{Bool } \equiv U_1) \epsilon_1 \bar{u}_1 \text{ then ... else ... } \rightarrow \text{error}
\]

and the result follows. Suppose \( \epsilon_1'' = \epsilon_1' \circ = \epsilon_1 \) is defined. Then

\[
\text{if } \epsilon_1 \epsilon_1' \bar{u}_1 : U_1 \text{ then } \epsilon_2 \sigma_1(tU') \text{ else } \epsilon_3 \sigma_1(tU') \rightarrow \iff
\]

\[
\text{if } \epsilon_1 \epsilon_1' \bar{u}_1 \text{ then } \epsilon_2 \sigma_1(tU') \text{ else } \epsilon_3 \sigma_1(tU')
\]

Then as \( (\bar{u}_1, \bar{u}_1') \in \mathcal{U}_{k-j}[\{U_1\}] \) we proceed analogous to (1) and the result holds.
Case $(U+)$: Similar to the $(U_{app})$ and $(U_{if})$ case.
Appendix D

Gradual Parametricity, Revisited

In this appendix we present additional definitions that were not included in the main body of §6. Proofs are in the companion technical report [121].

D.1 SF: Well-formedness

In this section we present auxiliary definitions for well-formedness of type name stores, and well-formedness of types.

**Definition 119** (Well-formedness of the type name store).

\[ \vdash . \quad \alpha \notin \Sigma \quad \vdash \Sigma, \alpha : T \]

**Definition 120** (Well-formedness of types).

\[
\begin{align*}
\vdash \Sigma & \quad \Sigma; \Delta \vdash B & \quad \Sigma; \Delta \vdash T_1 & \quad \Sigma; \Delta \vdash T_2 & \quad \Sigma; \Delta, X \vdash T & \quad \Sigma; \Delta \vdash \forall X. T & \quad \Sigma; \Delta \vdash T_1 & \quad \Sigma; \Delta \vdash T_2 & \quad \vdash \Sigma \quad X \in \Delta & \quad \vdash \Sigma \quad \alpha : T \in \Sigma & \quad \vdash \Sigma \quad \Delta \vdash X & \quad \vdash \Sigma \quad \Delta \vdash \alpha
\end{align*}
\]
D.2 GSF: Statics

In this section we present auxiliary definitions of the statics semantics of GSF not presented in the § 6.

D.2.1 Syntax and Syntactic Meaning of Gradual Types

Proposition 44 (Precision, inductively). The inductive definition of type precision given in Figure 6.3 is equivalent to Definition 45.

Proof. Direct by induction on the type structure of $G_1$ and $G_2$. We only present representative cases to illustrate the reasoning used in the proof. We prove first that $C(G_1) \subseteq C(G_2) \Rightarrow G_1 \sqsubseteq G_2$, where $G_1 \sqsubseteq G_2$ stands for the inductive definition given in Figure 6.6.

Case ($G_1 = B, G_2 = B$). Then $\{ B \} \subseteq \{ B \}$, but we already know that $B \sqsubseteq B$ and the result holds.

Case ($G_1 = G, G_2 = ?$). Then $C(G) \subseteq C(?) = \text{TYPE}$, but $G \sqsubseteq ?$ is an axiom and the result holds.

Then we prove the other direction, i.e. $G_1 \sqsubseteq G_2 \Rightarrow C(G_1) \subseteq C(G_2)$.

Case ($G_1 = B, G_2 = B$). Then $B \sqsubseteq B$, but we already know that $\{ B \} \subseteq \{ B \}$ and the result holds.

Case ($G_1 = G, G_2 = ?$). Then $G \sqsubseteq ?$, but $C(G) \subseteq C(?) = \text{TYPE}$ and the result holds.

Case ($G_1 = \forall X.G'_1, G_2 = \forall X.G'_2$). Then we know that $\{ \forall X.T \mid T \in C(G'_1) \} \subseteq \{ \forall X.T \mid T \in C(G'_2) \}$, then it must be the case that $C(G'_1) \subseteq C(G'_2)$. Then by induction hypothesis $G_1 \sqsubseteq G_2$, then by inductive definition of precision for type abstractions, $\forall X.G_1 \sqsubseteq \forall X.G_2$ and the result holds.

Then we know that $\forall X.T \mid T \in C(G'_1) \} \subseteq \forall X.T \mid T \in C(G'_2) \}$, which is direct from $C(G'_1) \subseteq C(G'_2)$. 

\[ \square \]

Proposition 45 (Galois connection). $(C, A)$ is a Galois connection, i.e.:

a) (Soundness) for any non-empty set of static types $S = \{ T \}$, we have $S \subseteq C(A(S))$

b) (Optimality) for any gradual type $G$, we have $A(C(G)) \subseteq G$.

Proof. We first proceed to prove a) by induction on the structure of the non-empty set $S$.

Case ($\{ B \}$). Then $A(\{ B \}) = B$. But $C(B) = \{ B \}$ and the result holds.

Case ($\{ T_{i1} \rightarrow T_{i2} \}$). Then $A(\{ T_{i1} \rightarrow T_{i2} \}) = A(\{ T_{i1} \}) \rightarrow A(\{ T_{i2} \})$. But by definition of $C$,

$C(A(\{ T_{i1} \})) \rightarrow A(A(\{ T_{i2} \})) = T_{i} \rightarrow T_{2} \mid T_{1} \in C(A(\{ T_{i1} \}), T_{2} \in C(A(\{ T_{i2} \})))$. By induction hypotheses, $\{ T_{i1} \} \subseteq C(A(\{ T_{i1} \}))$ and $\{ T_{i2} \} \subseteq C(A(\{ T_{i2} \}))$, therefore $\{ T_{i1} \rightarrow T_{i2} \} \subseteq$
\{T_1 \to T_2 \mid T_1 \in \{\overline{T_1}\}, T_2 \in \{\overline{T_2}\}\} \subseteq \{T_1 \to T_2 \mid T_1 \in C(A(\{\overline{T_1}\})), T_2 \in C(A(\{\overline{T_2}\}))\}
and the result holds.

**Case** (\{\overline{T_{11}} \times \overline{T_{12}}\}). We proceed analogous to case \{\overline{T_{11}} \to \overline{T_{12}}\}.

**Case** (\{X\}, \{\alpha\}). We proceed analogous to case \{B\}.

**Case** (\{\forall X.T_i\}). Then \(A(\{\forall X.T_i\}) = \forall X.A(\{\overline{T_i}\})\). But by definition of \(C\), \(C(\forall X.A(\{\overline{T_i}\})) = \{\forall X.T \mid T \in C(A(\{\overline{T_i}\}))\}\). By induction hypothesis, \(\{\overline{T_i}\} \subseteq C(A(\{\overline{T_i}\}))\), therefore \(\{\forall X.T \mid T \in C(A(\{\overline{T_i}\}))\}\) \subseteq \(\{\forall X.T \mid T \in C(A(\{\overline{T_i}\}))\}\) and the result holds.

**Case** (\{\overline{T_i}\} heterogeneous). Then \(A(\{\overline{T_i}\}) = ?\) and therefore \(C(A(\{\overline{T_i}\})) = \text{TYPE}\), but \(\{\overline{T_i}\} \subseteq \text{TYPE}\) and the result holds.

Now let us proceed to prove b) by induction on gradual type \(G\).

**Case** (\(B\)). Trivial because \(C(B) = \{B\}\), and \(A(\{B\}) = B\).

**Case** (\(G_1 \to G_2\)). We have to prove that \(A(C(G_1 \to G_2)) \subseteq G_1 \to G_2\), which is equivalent to prove that \(C(A(\overline{T})) \subseteq \overline{T}\), where \(T = C(G_1 \to G_2) = \{T_1 \to T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2)\}\). Then \(\overline{T}\) has the form \(\{\overline{T_{11}} \to \overline{T_{12}}\}\), such that \(\forall i, T_{11} \in C(G_1)\) and \(T_{12} \in C(G_2)\). Also note that \(\{\overline{T_{11}}\} = C(G_1)\) and \(\{\overline{T_{12}}\} = C(G_2)\). But by definition of \(A\), \(A(\{\overline{T_{11}} \to \overline{T_{12}}\}) = A(\{\overline{T_{11}}\}) \to A(\{\overline{T_{12}}\})\) and therefore \(C(A(\{\overline{T_{11}}\}) \to A(\{\overline{T_{12}}\})) = \{T_1 \to T_2 \mid T_1 \in C(A(\{\overline{T_{11}}\})), T_2 \in C(A(\{\overline{T_{12}}\}))\}\). But by induction hypotheses \(C(A(\{\overline{T_{11}}\})) \subseteq C(G_1)\) and \(C(A(\{\overline{T_{12}}\})) \subseteq C(G_2)\) and the result holds.

**Case** (\(G_1 \times G_2\)). We proceed analogous to case \(G_1 \to G_2\).

**Case** (\(X\), \(\alpha\)). We proceed analogous to case \(B\).

**Case** (\(\forall X.G\)). We have to prove that \(A(C(\forall X.G)) \subseteq \forall X.G\), which is equivalent to prove that \(C(A(\overline{T})) \subseteq \overline{T}\), where \(\overline{T} = C(\forall X.G) = \{\forall X.T \mid T \in C(G)\}\). Then \(\overline{T}\) has the form \(\{\overline{\forall X.T_i}\}\), such that \(\forall i, T_i \in C(G)\). Also note that \(\{\overline{T_i}\} = C(G)\). But by definition of \(A\), \(A(\{\overline{\forall X.T_i}\}) = \forall X.A(\{\overline{T_i}\})\) and therefore \(C(A(\{\overline{T_i}\})) \subseteq C(\forall X.A(\{\overline{T_i}\})) = \forall X.T \mid T \in C(A(\{\overline{T_i}\}))\). But by induction hypothesis \(C(A(\{\overline{T_i}\})) \subseteq C(G)\) and the result holds.

**Case** (?). Then we have to prove that \(C(A(?)) \subseteq C(?) = \text{TYPE}\), but this is always true and the result holds immediately.

\[\square\]

### D.2.2 Lifting the Static Semantics

**Definition 121** (Store precision). \(\Xi_1 \subseteq \Xi_2\) if and only if \(\text{dom}(\Xi_1) = \text{dom}(\Xi_2)\) and \(\forall \alpha \in \text{dom}(\Xi_1), \Xi_1(\alpha) \subseteq \Xi_2(\alpha)\).

**Lemma 261.** If \(\Xi_1 \subseteq \Xi_2, \vdash \Xi_i, G_1 \subseteq G_2,\) and \(\Xi_1; \Delta \vdash G_1,\) then \(\Xi_2; \Delta \vdash G_2\).

**Proof.** Straightforward induction on relation \(G_1 \subseteq G_2\). We only present interesting cases.

**Case** (\(G_1 = \forall X.G'_1, G_2 = \forall X.G'_2\)). By definition of precision \(G'_1 \subseteq G'_2\). By definition of well-formedness of types, \(\Xi_1; X \vdash G'_1\) and then by induction hypothesis \(\Xi_2; \Delta, X \vdash G'_2\). Then by definition of well-formedness of types \(\Xi_2; \Delta \vdash \forall X.G'_2\) and the result holds.

**Case** (\(G_2 = ?\)). This is trivial because as \(\vdash \Xi_2\), then \(\Xi_2; \Delta \vdash ?\).
Case \((G_1 = \alpha, G_2 = \alpha)\). Trivial by definition of \(\Xi_1 \subseteq \Xi_2\), \(\alpha \in \text{dom}(\Xi_2)\), therefore \(\alpha : G'_2 \in \Xi_2\) and then \(\Xi_2 ; \Delta \vdash \alpha\).

Lemma 262. Let \(\Xi_1 \subseteq \Xi_2\), then \(\vdash \Xi_1 \Rightarrow \Xi_2\).

Proof. By induction on relation \(\Xi_1 \subseteq \Xi_2\).
Case \((\cdot \subseteq \cdot)\). Trivial as \(\vdash \cdot\).
Case \((\Xi'_1, \alpha : G_1 \subseteq \Xi'_2, \alpha : G_2)\). By definition of store precision we know that \(\Xi'_1 \subseteq \Xi'_2\) and that \(G_1 \subseteq G_2\). By definition of well-formedness, \(\vdash \Xi'_1, \alpha : G_1 \Rightarrow \Xi'_1\), therefore by induction hypothesis \(\vdash \Xi'_2\). We only have left to prove is that \(\Xi'_2 ; \cdot \vdash G_2\), which follows directly from Lemma 261.

Lemma 263. If \(\Sigma \in C(\Xi)\) and \(\vdash \Sigma\), then \(\vdash \Xi\)

Proof. Corollary of Lemma 262 as \(\Sigma \subseteq \Xi\).

Lemma 264. If \(\Sigma ; \Delta \vdash T_1 = T_2\), then \(\Sigma ; \Delta \vdash T_1\) and \(\Sigma ; \Delta \vdash T_2\).

Proof. By induction on relation \(\Sigma ; \Delta \vdash T_1 = T_2\). Most cases are straightforward, so we present only the interesting cases.
Case \((T_1 = \forall X.T'_1, T_2 = \forall X.T'_2)\). As \(\Sigma ; \Delta \vdash \forall X.T'_1 = \forall X.T'_2\), by inspection of the derivation rule, \(\Sigma ; \Delta, X \vdash T'_1 = T'_2\). By induction hypotheses we know that \(\Sigma ; \Delta, X \vdash T'_1\), and that \(\Sigma ; \Delta, X \vdash T'_2\). Therefore by well-formedness of types we know that \(\Sigma ; \Delta \vdash \forall X.T'_1\) and that \(\Sigma ; \Delta \vdash \forall X.T'_2\) and the result holds.
Case \((T_1 = X, T_2 = X)\). As \(\Sigma ; \Delta \vdash X = X\), then we know by inspection of the derivation rule that \(\vdash \Sigma\) and that \(X \in \Delta\). Then as \(\vdash \Sigma\) and that \(X \in \Delta\), \(\Sigma ; \Delta \vdash X\) and the result holds.

Proposition 46 (Consistency, inductively). The inductive definition of type consistency given in Figure 6.4 is equivalent to Definition 46.

Proof. First we prove that \(\Sigma ; \Delta \vdash T_1 = T_2\) for some \(\Sigma \in C(\Xi)\), \(T_1 \in C(G_i)\) implies that \(\Xi ; \Delta \vdash G_1 \sim G_2\), where \(\Xi ; \Delta \vdash G_1 \sim G_2\) stands for the inductive definition of consistency. We proceed by straightforward induction on \(G_i\) such that the predicate holds (we only show interesting cases). By Lemma 263 we know that if \(\vdash \Sigma\) then \(\vdash \Xi\), which will be assumed to be true whenever is needed.
Case \((G_1 = B, G_2 = B)\). Then \(\Sigma ; \Delta \vdash B = B\), but we already know that \(\Xi \vdash B \sim B\) and the result holds.
Case \((G_1 = G, G_2 = ?)\). We know that \(\Sigma; \Delta \vdash T_1 = T_2\) for some \(T_1 \in C(G)\) and \(T_2 \in C(\Sigma)\). Then by Lemma 264, \(\Sigma; \Delta \vdash T_1\), and as \(\Sigma \sqsubseteq \Xi\) and \(T_1 \sqsubseteq G\), by Lemma 261 \(\Xi; \Delta \vdash G\). Then as \(\Xi; \Delta \vdash G, G \sim ? = \text{TYPE}\) and the result holds.

Case \((G_1 = \forall X.G'_1, G_2 = \forall X.G'_2)\). Then we know that \(\Sigma; \Delta \vdash \forall X.T_1 = \forall X.T_2\) where \(\forall X.T_1 \in C(\forall X.G'_1), \forall X.T_2 \in C(\forall X.G'_2)\). Notice that \(T_1 \in C(G'_1)\), \(T_2 \in C(G'_2)\), and that \(\Sigma; \Delta, X \vdash T_1 = T_2\). Then by induction hypotheses, \(\Xi \vdash G'_1 \sim G'_2[\Delta, X]\), and therefore \(\Xi; \Delta \vdash \forall X.G'_1 \sim \forall X.G'_2\) and the result holds.

Then we prove the other direction, i.e. \(G_1 \sqsubseteq G_2 \Rightarrow C(G_1) \sim C(G_2)\).

Case \((G_1 = B, G_2 = B)\). Then \(B \sqsubseteq B\), but we already know that \(B \in C(B)\) and \(\Sigma; \Delta \vdash B = B\), and the result holds immediately.

Case \((G_1 = G, G_2 = ?)\). Then \(G \sqsubseteq ?\). Let \(T_1 \in C(G)\) and \(\Sigma \in C(\Xi)\) such that \(\Sigma; \Delta \vdash T_1\). As \(C(\Sigma) = \text{TYPE}\), we can choose \(T_1 \in \text{TYPE}\), so \(\Sigma; \Delta \vdash T_1 = T_1\), and the result holds.

Case \((G_1 = \forall X.G'_1, G_2 = \forall X.G'_2)\). Then we know that \(\Xi; \Delta \vdash \forall X.G'_1 \sim \forall X.G'_2\), then by looking at the premise of the corresponding definition, \(\Xi; \Delta, X \vdash G'_1 \sim G'_2\). Then by induction hypotheses \(\exists T_1 \in C(G'_1), T_2 \in C(G'_2), \Sigma \in C(\Xi), \) such that \(\Sigma; \Delta, X \vdash T_1 = T_2\). By definition of consistency \(\forall X.T_1 \in C(G_1)\). Then by definition of equality, \(\Sigma; \Delta \vdash \forall X.T_1 = \forall X.T_2\) and the result holds.

\[\square\]

Definition 47 (Consistent lifting of functions). Let \(F_n\) be a function of type \(\text{TYPE}^n \rightarrow \text{TYPE}\). Its consistent lifting \(F^\sharp_n\), of type \(\text{GTYPE}^n \rightarrow \text{GTYPE}\), is defined as: \(F^\sharp_n(G) = A(\{ F_n(T) \mid T \in C(G) \})\)

Lemma 265. \(G = A(C(G))\)

Proof. Then we have to prove that \(G = A(C(G))\). By optimality (Prop 45(b), we know that \(A(C(G)) \sqsubseteq G\), and by soundness (Prop 45(a), \(C(G) \sqsubseteq C(A(C(G)))\), i.e. \(G \sqsubseteq A(C(G))\). Therefore \(G \sqsubseteq A(C(G))\) and \(A(C(G)) \sqsubseteq G\), thus \(G = A(C(G))\) and the result holds.

Lemma 266. \(G[G'/X] = A(\{ T[T'/X] \mid T \in C(G), T' \in C(G') \})\).

Proof. We proceed by induction on \(G\). We only present interesting cases.

Case \((G = X)\). Then \(G[G'/X] = G', \) and \(C(G) = \{ X \}\). Then we have to prove that \(G' = A(\{ T' \mid T' \in C(G') \})\). But notice that \(A(\{ T' \mid T' \in C(G') \}) = A(C(G'))\) and by Lemma 265 the result holds immediately.

Case \((G = ?)\). Then \(G[G'/X] = ?, \) and \(C(G) = \text{TYPE}\). Then we have to prove that \(? = A(\{ T[T'/X] \mid T \in \text{TYPE}, T' \in C(G') \})\). But notice that \(A(\{ T[T'/X] \mid T \in \text{TYPE}, T' \in C(G') \}) = A(C(\text{TYPE}))\) and by Lemma 265 the result holds immediately.

Case \((G = \forall Y.G'')\). Then \(G[G'/X] = \forall Y.G''[G'/X]\), and \(C(G) = \forall Y.C(G'')\). Then we have to prove that \(\forall Y.G''[G'/X] = A(\{ \forall Y.T''[T'/X] \mid T'' \in C(G''), T' \in C(G') \})\). But notice that by definition of abstraction \(A(\{ \forall Y.T''[T'/X] \mid T'' \in C(G''), T' \in C(G') \}) = \forall Y.A(\{ T''[T'/X] \mid T'' \in C(G''), T' \in C(G') \})\). Then by induction hypothesis on \(G''\), \(G''[G'/X] = \forall Y.G''[G'/X] = \forall Y.A(\{ T''[T'/X] \mid T'' \in C(G''), T' \in C(G') \})\).
A(T''[T'/X] \mid T'' \in C(G''), T' \in C(G'))$, therefore
\[ \forall Y.G''[G'/X] = \forall Y.A(\{ T''[T'/X] \mid T'' \in C(G''), T' \in C(G') \}) \] and the result holds.

\[ \square \]

**Proposition 47** (Consistent type functions). The definitions of \( \text{dom}^\sharp, \text{cod}^\sharp, \text{inst}^\sharp, \) and \( \text{proj}^\sharp_i \) given in Fig. 6.5 are consistent liftings, as per Def. 47, of the corresponding functions from Fig. 6.1.

**Proof.** We present the proof for \( \text{inst}^\sharp \) and \( \text{dom}^\sharp \) (the other proofs are analogous).

First we prove that \( \text{inst}^\sharp(G, G') = A(\text{inst}(\text{C}^2(G, G'))) \), where \( \text{inst}^\sharp(G, G') \) correspond to the algorithmic definitions presented in Fig. 6.6. Notice that
\[
A(\text{inst}(\text{C}^2(G, G')))
\]
\[
= A(\text{inst}(\{ (T, T') \mid T \in C(G), T' \in C(G') \}))
\]
\[
= A(\{ T[T'/X] \mid \forall X.T \in C(G), T' \in C(G') \})
\]
But then the result follows immediately from Lemma 266.

Then we prove that \( \text{dom}^\sharp(G) = A(\text{dom}(C(G))) \), where \( \text{dom}^\sharp(G) \) correspond to the algorithmic definitions presented in Fig. 6.6. We proceed by induction on \( G \).

**Case** \( (G = G_1 \to G_2) \). Notice that
\[
A(\text{dom}(C(G')))
\]
\[
= A(\text{dom}(C(G_1 \to G_2)))
\]
\[
= A(\text{dom}(\{ T_1 \to T_2 \mid T_1 \in C(G_1), T_2 \in C(G_2) \}))
\]
\[
= A(\{ T_1 \mid T_1 \in C(G_1) \})
\]
\[
= A(C(G_1))
\]
But \( \text{dom}^\sharp(G_1 \to G_2) = G_1 \). Then we have to prove that \( G_1 = A(C(G_1)) \) which holds immediately by Lemma 265.

**Case** \( (G = ?) \). Notice that
\[
A(\text{dom}(C(G)))
\]
\[
= A(\text{dom}(C(?)))
\]
\[
= A(\text{dom}(\text{TYPE}))
\]
\[
= A(\text{TYPE})
\]
\[
= ?
\]
and the result holds immediately as \( \text{dom}^\sharp(?) = ? \).
Case \((G \neq ? \neq G_1 \to G_2)\). If \(G\) has not the form \(G_1 \to G_2\), or is not \(?\), then \(\text{dom}^s(G)\) is undefined. Then as \(\exists \beta, T \in C(G)\) such that \(T = T_1 \to T_2\) the result holds immediately as \(\text{dom}(T)\) is undefined \(\forall T \in C(G)\).

\[
\text{D.2.3 Well-formedness}
\]

In this section we present auxiliary definitions of the statics semantics of GSF.

**Definition 122** (Well-formedness of type name store).

\[
\begin{align*}
\vdash : & \quad \alpha \notin \Xi, \Sigma; \vdash G \\
\Sigma; \vdash, \alpha : G & \quad \vdash : \Sigma, \alpha : G
\end{align*}
\]

**Definition 123** (Well-formedness of types).

\[
\begin{align*}
\Xi; \vdash & \\
\Xi; \Delta \vdash B & \quad \Xi; \Delta \vdash G_1, \Xi; \Delta \vdash G_2 & \quad \Xi; \Delta, X \vdash G \\
\Xi; \Delta \vdash G_1 \times G_2 & \quad \Xi; \Delta \vdash X \in \Delta & \quad \Xi; \Delta \vdash \alpha : G \in \Xi \\
\Xi; \Delta \vdash ? &
\end{align*}
\]

\[
\Xi; \Delta \vdash \forall X.G
\]

\[
\Xi; \Delta \vdash W
\]

**D.2.4 Static Properties**

In this section we present two static properties of GSF and the proof: the static equivalence for static terms and the static gradual guarantee.

**Static Equivalence for Static Terms**

**Proposition 48** (Static equivalence for static terms). Let \(t\) be a static term and \(G\) a static type \((G = T)\). We have \(\vdash_S t : T\) if and only if \(\vdash t : T\).

**Proof.** We prove this proposition for open terms instead. The proof is direct thanks to the equivalence between the typing rules and the equivalence between type equality and type consistency rules for static types. We only present one case to illustrate the reasoning.

First we prove \(\Sigma; \Delta \vdash_S t : T \Rightarrow \Sigma; \Delta \vdash t : T\) by induction on judgment \(\Sigma; \Delta \vdash_S t : T\).

**Case** \((\Sigma; \Delta \vdash_S t'[T'''] : \text{inst}(\forall X.T', T'''))\). Then \(\Sigma; \Delta \vdash_S t' : \forall X.T'\), and by induction hypothesis \(\Sigma; \Delta \vdash t' : \forall X.T'\). Then \(\text{inst}^s(\forall X.T, T'') = T[T''/X] = \text{inst}(\forall X.T', T'')\), and as \(\Sigma; \Delta \vdash T''\), therefore \(\Sigma; \Delta \vdash t'[T'''] : T[T''/X]\) and the result holds.
Then we prove $\Sigma;\Delta \vdash t: T \Rightarrow \Sigma;\Delta \vdash_s t: T$.

Case $(\Sigma;\Delta \vdash t'[T'']: \text{inst}^2(\forall X.T',T''))$. Then $\Sigma;\Delta \vdash t': \forall X.T'$, and by induction hypothesis $\Sigma;\Delta \vdash t': \forall X.T'$. Then $\text{inst}(\forall X.T,T'') = T'[T''/X] = \text{inst}^2(\forall X.T',T'')$, and as $\Sigma;\Delta \vdash T''$, therefore $\Sigma;\Delta \vdash_s t'[T''] : T'[T''/X]$ and the result holds.

\[ \square \]

**Static Gradual Guarantee**

In this section we present the proof of the static gradual guarantee property. In the Definition[124] and Definition[125] we present term precision and type environment precision.

**Definition 124** (Term precision).

- $(P_x) \frac{x \sqsubseteq x}{x \sqsubseteq x}$
- $(P_b) \frac{b \sqsubseteq b}{b \sqsubseteq b}$
- $(P_{\lambda}) \frac{t \sqsubseteq t'}{(\lambda x : G.t) \sqsubseteq (\lambda x : G'.t')}$
- $(P\lambda) \frac{t \sqsubseteq t'}{(\Lambda X.t) \sqsubseteq (\Lambda X.t')}$
- $(P_{\text{pair}}) \frac{\vec{t} \sqsubseteq \vec{t}'}{\text{op}(\vec{t}) \sqsubseteq \text{op}(\vec{t})}$
- $(P_{\text{app}}) \frac{t_1 \sqsubseteq t'_1 \quad t_2 \sqsubseteq t'_2}{(t_1,t_2) \sqsubseteq (t'_1,t'_2)}$
- $(P_{\text{app}}^\triangleleft) \frac{t \sqsubseteq t'}{t[G] \sqsubseteq t'[G]}$
- $(P_{\text{app}}^\triangleleft) \frac{t \sqsubseteq t'}{t[G] \sqsubseteq t'[G]}$
- $(P_{\text{app}}^\triangleleft) \frac{t \sqsubseteq t'}{t[G] \sqsubseteq t'[G]}$
- $(P_{\text{app}}^\triangleleft) \frac{t \sqsubseteq t'}{t[G] \sqsubseteq t'[G]}$
- $(P_{\text{app}}^\triangleleft) \frac{t \sqsubseteq t'}{t[G] \sqsubseteq t'[G]}$

**Definition 125** (Type environment precision).

- $\Gamma \subseteq \Gamma'$
- $\Gamma, x : G \sqsubseteq \Gamma', x : G'$
- $\Gamma \subseteq \Gamma'$
- $\Gamma \sqsubseteq \Gamma'$
- $\Gamma \subseteq \Gamma'$
- $\Gamma \subseteq \Gamma'$
- $\Gamma \subseteq \Gamma'$
- $\Gamma \subseteq \Gamma'$
- $\Gamma \subseteq \Gamma'$

**Lemma 267.** If $\Xi;\Delta;\Gamma \vdash t : G$ and $\Gamma \subseteq \Gamma'$, then $\Xi;\Delta;\Gamma' \vdash t : G'$ for some $G \sqsubseteq G'$.

**Proof.** Simple induction on typing derivations.

**Lemma 268.** If $\Xi;\Delta \vdash G_1 \sim G_2$ and $G_1 \sqsubseteq G'_1$ and $G_2 \sqsubseteq G'_2$ then $\Xi;\Delta \vdash G'_1 \sim G'_2$.

**Proof.** By definition of $\Xi;\Delta \vdash \cdot \sim \cdot$, there exists $(T_1,T_2) \in C^2(G_1,G_2)$ such that $T_1 = T_2$. $G_1 \sqsubseteq G'_1$ and $G_2 \sqsubseteq G'_2$ mean that $C(G_1) \subseteq C(G'_1)$ and $C(G_2) \subseteq C(G'_2)$, therefore $(T_1,T_2) \in C^2(G'_1,G'_2)$, and the result follows.

**Lemma 269.** If $G_1 \sqsubseteq G'_1$ and $G_2 \sqsubseteq G'_2$ then $\text{inst}^2(G_1,G_2) \sqsubseteq \text{inst}^2(G'_1,G'_2)$.

**Proof.** Trivial by definition of type precision, definition of $\text{inst}^2(\cdot,\cdot)$ and induction on the structure of $G_1$.

**Lemma 270.** If $G_1 \sqsubseteq G_2$ and $\Xi;\Delta \vdash G_1$ then $\Xi;\Delta \vdash G_2$.

**Proof.** Trivial by definition of type precision and definition of $\text{proj}^2(\cdot)$.
Lemma 271. If \( G_1 \subseteq G_2 \) then \( \text{proj}_i^*(G_1) \subseteq \text{proj}_i^*(G_2) \).

**Proof.** Trivial by definition of type precision, definition of type well-formedness and induction on the structure of \( G_1 \).

\[ \square \]

**Proposition 272** (Static gradual guarantee for open terms). If \( \Xi; \Delta; \Gamma \vdash t_1 : G_1 \) and \( t_1 \subseteq t_2 \), then \( \Xi; \Delta; \Gamma \vdash t_2 : G_2 \), for some \( G_2 \) such that \( G_1 \subseteq G_2 \).

**Proof.** We prove the property on opens terms instead of closed terms: If \( \Xi; \Delta; \Gamma \vdash t_1 : G_1 \) and \( t_1 \subseteq t_2 \) then \( \Xi; \Delta; \Gamma \vdash t_2 : G_2 \) and \( G_1 \subseteq G_2 \).

The proof proceed by induction on the typing derivation.

**Case (Gx, Gb)**. Trivial by definition of term precision (\( \sqsubseteq \)) using \((Px),(Pb)\) respectively.

**Case (Gλ)**. Then \( t_1 = (\lambda x : G'_1.t) \) and \( G_1 = G'_1 \Rightarrow G'_2 \). By \((G\lambda)\) we know that:

\[
\begin{array}{c}
\Xi; \Delta; \Gamma, x : G'_1 \vdash t : G'_2 \\
\Xi; \Delta; \Gamma \vdash \lambda x : G'_1.t : G'_1 \rightarrow G'_2
\end{array}
\]

Using induction hypotheses on the premises of \([\text{D.1}]\) and \([\text{D.2}]\), \( \Xi; \Delta; \Gamma, x : G'_1 \vdash t' : G''_2 \) with \( G'_2 \subseteq G''_2 \). By Lemma 267 \( \Xi; \Delta; \Gamma, x : G'_1 \vdash t' : G''_2 \) where \( G''_2 \subseteq G''_2 \). Then we can use rule \((G\lambda)\) to derive:

\[
\Xi; \Delta; \Gamma, x : G''_1 \vdash t' : G''_2
\]

Using induction hypotheses on the premises of \([\text{D.1}]\) and \([\text{D.2}]\), \( \Xi; \Delta; \Gamma, x : G''_1 \vdash t' : G''_2 \) with \( G''_2 \subseteq G''_2 \). Then we can use rule \((G\lambda)\) to derive:

\[
\Xi; \Delta; \Gamma \vdash (\lambda x : G''_1.t') : G''_1 \Rightarrow G''_2
\]

Where \( G_2 \subseteq G''_2 \). Using the premise of \([\text{D.2}]\) and the definition of type precision we can infer that \( G'_1 \Rightarrow G'_2 \subseteq G''_2 \Rightarrow G''_2 \) and the result holds.

**Case (GA)**. Then \( t_1 = (\Lambda X.t) \) and \( G_1 = \forall X.G'_1 \). By \((GA)\) we know that:

\[
\begin{array}{c}
\Xi; \Delta, X; \Gamma \vdash t : G'_1 \\
\Xi; \Delta; \Gamma \vdash \Lambda X.t : \forall X.G'_1
\end{array}
\]

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = (\Lambda X.t') \) and therefore

\[
\begin{array}{c}
\Xi; \Delta, X; \Gamma \vdash t' : G''_1 \\
\Xi; \Delta; \Gamma \vdash (\Lambda X.t') : \forall X.G''_1
\end{array}
\]

Using induction hypotheses on the premises of \([\text{D.3}]\) and \([\text{D.4}]\), \( \Xi; \Delta, X; \Gamma \vdash t' : G''_1 \) with \( G'_1 \subseteq G''_1 \). Then we can use rule \((GA)\) to derive:

\[
\Xi; \Delta, X; \Gamma \vdash t' : G''_1
\]
Using the definition of type precision we can infer that
\[ \forall X. G'_1 \sqsubseteq \forall X. G''_1 \]
and the result holds.

Case (Gpair). Then \( t_1 = \langle t'_1, t'_2 \rangle \) and \( G_1 = G'_1 \times G'_2 \). By (Gpair) we know that:
\[
\frac{\Xi; \Delta; \Gamma \vdash t'_1 : G'_1 \quad \Xi; \Delta; \Gamma \vdash t'_2 : G'_2}{\Xi; \Delta; \Gamma \vdash \langle t'_1, t'_2 \rangle : G'_1 \times G'_2}
\] (D.5)

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision, \( t_2 \) must have the form \( \langle t''_1, t''_2 \rangle \) and therefore
\[
\frac{t'_1 \sqsubseteq t''_1 \quad t'_2 \sqsubseteq t''_2}{\langle t'_1, t'_2 \rangle \sqsubseteq \langle t''_1, t''_2 \rangle}
\] (D.6)

Using induction hypotheses on the premises of (D.5) and (D.6), \( \Xi; \Delta; \Gamma \vdash t''_1 : G''_1 \) and \( \Xi; \Delta; \Gamma \vdash t''_2 : G''_2 \), where \( G'_1 \sqsubseteq G''_1 \) and \( G'_2 \sqsubseteq G''_2 \). Then we can use rule (Gpair) to derive:
\[
\frac{\Xi; \Delta; \Gamma \vdash t''_1 : G''_1 \quad \Xi; \Delta; \Gamma \vdash t''_2 : G''_2}{\Xi; \Delta; \Gamma \vdash \langle t''_1, t''_2 \rangle : G''_1 \times G''_2}
\]

Finally, using the definition of type precision we can infer that
\[ G'_1 \times G'_2 \sqsubseteq G''_1 \times G''_2 \]
and the result holds.

Case (Gasc). Then \( t_1 = \mathit{op}(t) \) and \( G_1 = G^* \). By (Gasc) we know that:
\[
\frac{\Xi; \Delta; \Gamma \vdash t : G \quad \Xi; \Delta \vdash G \sim G_1}{\Xi; \Delta; \Gamma \vdash \mathit{op}(t) : G^*}
\] (D.7)

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( \mathit{op}(t') \) and therefore
\[
\frac{\sqsubseteq t' \quad G_1 \sqsubseteq G_2}{t' : G^*}
\] (D.8)

Using induction hypotheses on the premises of (D.7) and (D.8), \( \Xi; \Delta; \Gamma \vdash t' : G' \) where \( G \sqsubseteq G' \). We can use rule (Gasc) and Lemma 268 to derive:
\[
\frac{\Xi; \Delta; \Gamma \vdash t' : G' \quad \Xi; \Delta \vdash G' \sim G_2}{\Xi; \Delta; \Gamma \vdash \mathit{op}(t') : G^*}
\]

Where \( G_1 \sqsubseteq G_2 \) and the result holds.

Case (Cop). Then \( t_1 = \mathit{op}(\overline{t}) \) and \( G_1 = G^* \). By (Gop) we know that:
\[
\frac{\Xi; \Delta; \Gamma \vdash \overline{t} : G \quad \overline{t}\mathit{op} = \overline{t'_{G_2}} \rightarrow G^*}{\Xi; \Delta \vdash \overline{G} \sim G^*}
\] (D.9)
Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = op(\bar{v}) \) and therefore
\[
(\text{Pop}) \quad \frac{\bar{t} \subseteq \bar{v}}{op(\bar{t}) \subseteq op(\bar{v})} \quad \text{(D.10)}
\]

Using induction hypotheses on the premises of (D.9) and (D.10), \( \Xi; \Delta; \Gamma \vdash \bar{v} : \overline{\mathcal{G}} \), where \( \overline{\mathcal{G}} \subseteq \overline{\mathcal{G}}' \). Using the Lemma 268 we know that \( \Xi; \Delta \vdash \overline{\mathcal{G}}' \sim G_2 \). Therefore we can use rule (Gop) to derive:
\[
(\text{Gop}) \quad \frac{\Xi; \Delta; \Gamma \vdash \bar{v} : \overline{\mathcal{G}}'}{\Xi; \Delta; \Gamma \vdash op(\bar{v}) : G^*}
\]
and the result holds.

Case (Gapp). Then \( t_1 = t'_1 t'_2 \) and \( G_1 = cod^\sharp(G'_1) \). By (Gapp) we know that:
\[
(\text{Gapp}) \quad \frac{\Xi; \Delta; \Gamma \vdash t'_1 : G'_1}{\Xi; \Delta; \Gamma \vdash \text{dom}^\sharp(G'_1) \sim G'_2}
\]
(\text{D.11})

Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t''_1 t''_2 \) and therefore
\[
(\text{Papp}) \quad \frac{t'_1 \subseteq t''_1}{t'_1 t'_2 \subseteq t''_1 t''_2} \quad \text{(D.12)}
\]
Using induction hypotheses on the premises of (D.11) and (D.12), \( \Xi; \Delta; \Gamma \vdash t''_1 : G''_1 \) and \( \Xi; \Delta; \Gamma \vdash t''_2 : G''_2 \), where \( G''_1 \subseteq G'_1 \) and \( G''_2 \subseteq G'_2 \). By definition type precision and the definition of \( \text{dom}^\sharp \), \( \text{dom}^\sharp(G'_1) \subseteq \text{dom}^\sharp(G''_1) \) and, therefore by Lemma 268 \( \Xi; \Delta \vdash \text{dom}^\sharp(G''_1) \sim G''_2 \). Also, by the previous argument \( \text{cod}^\sharp(G'_1) \subseteq \text{cod}^\sharp(G''_1) \). Then we can use rule (Gapp) to derive:
\[
(\text{Gapp}) \quad \frac{\Xi; \Delta; \Gamma \vdash t''_1 : \text{cod}^\sharp(G''_1)}{\Xi; \Delta; \Gamma \vdash \text{dom}^\sharp(G''_1) \sim G''_2}
\]
and the result holds.

Case (GappG). Then \( t_1 = t \cdot [G] \). By (GappG) we know that:
\[
(\text{GappG}) \quad \frac{\Xi; \Delta; \Gamma \vdash t \cdot [G]}{\Xi; \Delta; \Gamma \vdash \text{inst}^\sharp(G'_1, G)} \quad \text{(D.13)}
\]
where \( G_1 = \text{inst}^\sharp(G'_1, G) \). Consider \( t_2 \) such that \( t_1 \subseteq t_2 \). By definition of term precision \( t_2 \) must have the form \( t_2 = t' \cdot [G'] \) and therefore
\[
(\text{PappG}) \quad \frac{t \subseteq t'}{t \cdot [G] \subseteq t' \cdot [G']} \quad \text{(D.14)}
\]
Using induction hypotheses on the premises of (D.13) and (D.14), \( \Xi; \Delta; \Gamma \vdash t' : G'_2 \) where \( G'_1 \subseteq G'_2 \). We can use rule (GappG) and Lemma 270 to derive:
\[
(\text{Gasc}) \quad \frac{\Xi; \Delta; \Gamma \vdash t' : G'_2}{\Xi; \Delta; \Gamma \vdash \text{inst}^\sharp(G'_2, G')} \quad \text{(G.270)}
\]
Finally, by the Lemma 269 we know that \( \text{inst}^\sharp(G'_1, G) \subseteq \text{inst}^\sharp(G'_2, G') \) and the result holds.

367
Case (Cpairi). Then \( t_1 = \pi_i(t) \) and \( G_1 = \text{proj}^i(G) \). By (Gpair) we know that:

\[
\frac{\Xi; \Delta; \Gamma \vdash t : G}{\Xi; \Delta; \Gamma \vdash \pi_i(t) : \text{proj}^i(G)} \tag{D.15}
\]

Consider \( t_2 \) such that \( t_1 \sqsubseteq t_2 \). By definition of term precision, \( t_2 \) must have the form \( \pi_i(t') \) and therefore

\[
\frac{t \sqsubseteq t'}{\pi_i(t) \sqsubseteq \pi_i(t')} \tag{D.16}
\]

Using induction hypotheses on the premises of (D.15) and (D.16), \( \Xi; \Delta; \Gamma \vdash t' : G' \) where \( G \sqsubseteq G' \). Then we can use rule (Cpairi) to derive:

\[
\Xi; \Delta; \Gamma \vdash \pi_i(t') : \text{proj}^i(G')
\]

Finally, by the Lemma \( \text{Lemma 271} \) we can infer that \( \text{proj}^i(G) \sqsubseteq \text{proj}^i(G') \) and the result holds.

\[\square\]

**Proposition 49** (Static gradual guarantee). Let \( t \) and \( t' \) be closed GSF terms such that \( t \sqsubseteq t' \) and \( \vdash t : G \). Then \( \vdash t' : G' \) and \( G \sqsubseteq G' \).

**Proof.** Direct corollary of Prop. \( \text{272} \) \[\square\]
D.3 GSF: Dynamics

In this section, we expose auxiliary definitions of the dynamic semantics of GSF. First, we present type precision, interior and consistent transitivity definitions for evidence types. Then we show some important definitions, used in the dynamic semantics of GSFε. Finally, we present the translation semantics from GSF to GSFε.

D.3.1 Evidence Type Precision

Figure D.1 presents the definition of the evidence type precision.

\[
\begin{array}{c}
E \sqsubseteq E' \\
B \sqsubseteq B \\
X \sqsubseteq X \\
E_1 \sqsubseteq E_2' \quad E_2 \sqsubseteq E_2' \\
E_1 \to E_2 \sqsubseteq E_1' \to E_2' \\
\forall X.E_1 \sqsubseteq \forall X.E_2 \\
E_1 \sqsubseteq E_2 \\
E_1 \sqsubseteq E_1' \quad E_2 \sqsubseteq E_2' \\
E_1 \times E_2 \sqsubseteq E_1' \times E_2' \\
\alpha_{E_1} \sqsubseteq \alpha_{E_2} \\
E \sqsubseteq ?
\end{array}
\]

Figure D.1: Evidence Type Precision

D.3.2 Initial Evidence

In Figure D.2 we present the interior function, used to compute the initial evidence.

D.3.3 Consistent Transitivity

In Figures D.3 and D.4 we present the definition of consistent transitivity for evidence types.

D.3.4 GSFε: Dynamic Semantics

In this section, f of GSFε, specifically in the type application rule (RappG).

Definition 126.

\( \varepsilon_{out} \triangleq \langle E_\ast[\alpha^E], E_\ast[E'] \rangle \) where \( E_\ast = \text{lift}_\varepsilon(\text{unlift}(\pi_2(\varepsilon))) \), \( \alpha^E = \text{lift}_\varepsilon(\alpha) \), \( E' = \text{lift}_\varepsilon(G') \)

Definition 127. \( \langle E_1, E_2 \rangle [E_3] = \langle E_1[E_3], E_2[E_3] \rangle \)
\[ g : \text{EType} \times \text{EType} \rightarrow \text{Evidence} \]

\[
\begin{align*}
    \text{Definition 128.} \\
    t[\alpha^E/X] = \begin{cases} 
    c & t = c \\
    \lambda x : G_1[\alpha/X].t[\alpha^E/X] & t = \lambda x : G_1.t \\
    \Lambda Y.t'[\alpha^E/X] & t = \Lambda Y.t' \\
    \langle t_1[\alpha^E/X], t_2[\alpha^E/X] \rangle & t = \langle t_1, t_2 \rangle \\
    x & t = x \\
    \varepsilon[\alpha^E/X]t'[\alpha^E/X] :: G[\alpha/X] & t = \varepsilon t' :: G \\
    \text{op}(t'[\alpha^E/X]) & t = \text{op}(t') \\
    t_1[\alpha^E/X] t_2[\alpha^E/X] & t = t_1 t_2 \\
    \pi_i(t'[\alpha^E/X]) & t = \pi_i(t') \end{cases}
\end{align*}
\]

\[
\begin{align*}
    \text{Definition 129.} \\
    \text{lift}_\Xi(G_1) \rightarrow \text{lift}_\Xi(G_2) & G = G_1 \rightarrow G_2 \\
    \forall X.\text{lift}_\Xi(G_1) & G = \forall X. G_1 \\
    \text{lift}_\Xi(G_1) \times \text{lift}_\Xi(G_2) & G = G_1 \times G_2 \\
    \alpha^{\text{lift}_\Xi(\Xi(\alpha))} & G = \alpha \\
    G & \text{otherwise} \\
\end{align*}
\]

Figure D.2: GSF: Computing Initial Evidence
Definition 130.

\[
\begin{align*}
\text{base} & : \langle B, B \rangle \circ \langle B, B \rangle = \langle B, B \rangle \\
\text{idR} & : \langle ?, ? \rangle \circ \langle E, E \rangle = \langle E, E \rangle \\
\text{sealL} & : \langle E_1, E_2 \rangle \circ \langle E_3, E_4 \rangle = \langle E_1', E_2' \rangle \\
\text{func} & : \langle E_1 \rightarrow E_2, E_3 \rightarrow E_4 \rangle \circ \langle E_1, E_2 \rangle = \langle E_1', E_2' \rangle \\
\text{abst} & : \langle \forall X. E_1, \forall X. E_2 \rangle \circ \langle \forall X. E_3, \forall X. E_4 \rangle = \langle \forall X. E_1', \forall X. E_2' \rangle \\
\text{pair} & : \langle E_1, E_3 \rangle \circ \langle E_2, E_4 \rangle = \langle E_1 \times E_2, E_3 \times E_4 \rangle \\
\text{func?L} & : \langle E_1 \rightarrow E_2, E_3 \rightarrow E_4 \rangle \circ \langle ?, ?, \rightarrow \rangle = \langle E_1', E_2', E_3', E_4' \rangle \\
\end{align*}
\]

Figure D.3: GSF: Consistent Transitivity part 1

D.3.5 Translation from GSF to GSF\(\varepsilon\)

In this section we present the translation from GSF to GSF\(\varepsilon\) (Figure D.5), which inserts ascriptions to ensure that top-level constructors match in every elimination form. We use the following normalization metafunction:

\[
norm(t, G_1, G_2, \Xi) = \begin{cases} 
E = B & \text{if } G_1 = G_2 \\
\text{unlift}(E_1) \rightarrow \text{unlift}(E_2) & \text{if } G_1 \neq G_2 \land \varepsilon = g_\Xi(G_1, G_2) \\
\forall X. \text{unlift}(E_1) & E = \forall X. E_1 \\
\text{unlift}(E_1) \times \text{unlift}(E_2) & E = E_1 \times E_2 \\
\alpha & E = \alpha E_1 \\
X & E = X \\
? & E = ?
\end{cases}
\]

\[
g_\Xi(G_1, G_2) = g(\text{lift}_\Xi(G_1), \text{lift}_\Xi(G_2))
\]

Theorem 273 (Translation Preserves Typing). If \(\Xi; \Delta; \Gamma \vdash t : G\), then \(\Xi; \Delta; \Gamma \vdash t \sim t' : G\) and \(\Xi; \Delta; \Gamma \vdash t' : G\).
Proof. The proof follows by induction on the typing derivation of $\Xi; \Delta; \Gamma \vdash t : G$, exploiting the fact that the term produced by $\text{norm}(t, G_1, G_2, \Xi)$ has type $G_2$. $\square$
Figure D.5: GSF to GSF\(\varepsilon\) translation.
D.4 GSF: Properties

In this section we present some properties of GSF. §D.4.1 presents Type Safety and its proof. §D.4.2 shows the property and proof about static terms do not fail.

D.4.1 Type Safety

In this section we present the proof of type safety for GSFε.

We define what it means for a store to be well typed with respect to a term. Informally, all free locations of a term and of the contents of the store must be defined in the domain of that store. Also, the store must preserve types between intrinsic locations and underlying values.

**Lemma 274** (Canonical forms). Consider a value Ξ;·;· ⊢ v : G. Then v = εu :: G with Ξ;·;· ⊢ u : G′ and ε ⊢ G′ ∼ G. Furthermore:

1. If G = B then v = εb :: B with Ξ;·;· ⊢ b : B and ε ⊢ B ∼ B.

2. If G = G_1 → G_2 then v = ε(λx : G_1′t) :: G_1 → G_2 with Ξ;·;· ⊢ t : G_2′ and ε ⊢ G_1′ → G_2′ ∼ G_1 → G_2.

3. If G = ∀X.G_1 then v = ε(ΛX.t) :: ∀X.G_1 with Ξ;·;· ⊢ t : G_1′ and ε ⊢ ∀X.G_1′ ∼ ∀X.G_1.

4. If G = G_1 × G_2 then v = ε⟨u_1,u_2⟩ :: G_1 × G_2 with Ξ;·;· ⊢ u_1 : G_1′,Ξ;·;· ⊢ u_2 : G_2′ and ε ⊢ G_1′ × G_2′ ∼ G_1 × G_2.

**Proof.** By direct inspection of the formation rules of evidence augmented terms.

**Lemma 275** (Substitution). If Ξ;Δ;Γ,x : G_1 ⊢ t : G, and Ξ;·;· ⊢ v : G_1, then Ξ;Δ;Γ ⊢ t[v/x] : G.

**Proof.** By induction on the derivation of Ξ;·;· ⊢ t : G.

**Lemma 276.** If ε ⊦ Ξ;Δ,X;Γ ⊢ G_1 ∼ G_2, Ξ;Δ ⊢ G′, α ∉ dom(Ξ), and E = lift_{Ξ}(G′), then ε[α^{E}/X] ⊦ Ξ;Δ;Γ ⊢ G_1[α/X] ∼ G_2[α/X].

**Proof.** By induction on the judgment ε ⊦ Ξ;Δ,X;Γ ⊢ G_1 ∼ G_2, noticing that the special substitution only differentiates from the usual type substitution by adding α to ? types, and by definition of precision ? ⊑ ? (therefore evidences may only gain precision).

**Lemma 277** (Type Substitution). If Ξ;Δ,X;Γ ⊢ t : G, Ξ;Δ ⊢ G′, α ∉ dom(Ξ), and E = lift_{Ξ}(G′), then Ξ;Δ;Γ ⊢ t[α^{E}/X] : G[α/X].

**Proof.** By induction on the derivation of Ξ;·;· ⊢ t : G and Lemma 276.
Lemma 278. If $\varepsilon_1 \Vdash \Xi; \Delta \vdash G_1' \sim G_1$, and $\varepsilon_2 \Vdash \Xi; \Delta \vdash G_2' \sim G_2$, then $\varepsilon_1 \times \varepsilon_2 \Vdash \Xi; \Delta \vdash G_1' \times G_2' \sim G_1 \times G_2$.

Proof. Direct by definition of evidences and interior. 

Lemma 279. If $\varepsilon \Vdash \Xi; \Delta \vdash G' \sim G$ then $p_i(\varepsilon) \Vdash \Xi; \Delta \vdash \text{proj}_i^\sharp(G') \sim \text{proj}_i^\sharp(G)$.

Proof. Direct by definition of evidences and interior. 

Proposition 280 (is well defined). If $\Xi; \cdot \vdash t : G$, then either

- $\Xi \triangleright t \rightarrow \Xi' \triangleright t'$, $\Xi \subseteq \Xi'$ and $\Xi' ; \cdot \vdash t' : G$; or
- $\Xi \triangleright t \rightarrow \text{error}$

Proof. By induction on the structure of a derivation of $\Xi \triangleright t \rightarrow r$, considering the last rule used in the derivation.

Case (Rapp). Then $t = (\varepsilon_1(\lambda x : G_{11}.t_1) :: G_1 \rightarrow G_2) (\varepsilon_2 u :: G_1)$. Then

\[
E_{\text{asc}} \quad \frac{\Xi ; \cdot ; t : G_11 \vdash t_1 : G_12}{\Xi ; \cdot ; \vdash (\lambda x : G_{11}.t_1) : G_{11} \rightarrow G_12}
\]

\[
E_{\text{app}} \quad \frac{\Xi ; \cdot ; u : G_2' \quad \Xi ; \cdot ; \vdash (\varepsilon_2 u :: G_1) : G_1}{\Xi ; \cdot ; \vdash (\varepsilon_2 u :: G_1) : G_1}
\]

As $\varepsilon_2 \vdash G_2' \sim G_1$ and by inversion lemma $\text{dom}(\varepsilon_1) \vdash G_1 \sim G_{11}$, then $\varepsilon' \vdash G_2' \sim G_{11}$. Therefore $\Xi ; \cdot ; \vdash \varepsilon'_u :: G_{11} ; G_{11}$, and by Lemma 275 $\Xi ; \cdot ; \vdash t[(\varepsilon'_u :: G_{11})/x] : G_{12}$.

Let us call $t'' = t[(\varepsilon'_u :: G_{11})/x]$. Then

\[
E_{\text{asc}} \quad \frac{\Xi ; \cdot ; t[\varepsilon'_u :: G_{11})/x] : G_{12}}{\Xi ; \cdot ; \vdash \text{cod}(\varepsilon_1)(t_1[\varepsilon'_u :: G_{11})/x]) : G_2}
\]

and the result holds.

Case (RappG). Then $t = (\varepsilon \Lambda X.t_1 :: \forall X.G_x) [G']$. Consider $G_x = \text{schm}^\sharp(G)$, then

\[
E_{\text{appG}} \quad \frac{\Xi ; X ; \vdash t_1 : G_1 \quad \varepsilon \Vdash \Xi ; X ; \vdash G_1 \sim \forall X.G_x}{\Xi ; \cdot ; \vdash (\varepsilon \Lambda X.t_1 :: \forall X.G_x) :: \forall X.G_x}
\]

375
Then
\[ \Xi \vdash (\varepsilon \Delta x. t_1 :: G) [G'] \rightarrow \Xi' \vdash \varepsilon_G^{E'/\alpha^{E'}} (\varepsilon [\alpha^{E'}] t_1 [\alpha^{E'}/X] :: G_x[\alpha/X]) :: G_x[G'/X] \]
where \( \Xi' \equiv \Xi, \alpha := G', \alpha \notin \text{dom}(\Xi) \), and \( E' \equiv \text{lift}_E(G') \), and
\[ \varepsilon_G^{E'/\alpha^{E'}} \equiv (\text{lift}_E(G_x[\alpha^{E'}/X], \text{lift}_E(G_x[G'/X])). \]
Notice that \( \langle \text{lift}_E(G_x[\alpha/X], \text{lift}_E(G_x[G'/X])) \rangle = \mathcal{G}(G_x[\alpha/X], G_x[G'/X]) \), and by definition of the special substitution, \( \text{lift}_E(G_x[\alpha^{E'}/X] \subseteq \text{lift}_E(G_x[\alpha/X]) \) (because \( \text{lift}_E(\alpha) = \alpha^{E'} \), and the substitution on evidences just extend unknowns with \( \alpha \)). Therefore \( \varepsilon_G^{E'/\alpha^{E'}} \models (G_x[\alpha/X], G_x[G'/X]) \), and \( \varepsilon_G^{E'/\alpha^{E'}} \models \Xi; \vdash G_x[\alpha/X] \sim G_x[G'/X] \). Also by Lemma 276 \( \varepsilon [\alpha^{E'}] \models \Xi; \vdash G_1[\alpha/X] \sim G_x[G'/X] \), and by Lemma 277 \( \Xi; \vdash \varepsilon [\alpha^{E'}] : G_1[\alpha/X] \).

Then, as \( \Xi \subseteq \Xi' \),
\[
\Xi; \vdash \varepsilon_G^{E'/\alpha^{E'}} (\varepsilon [\alpha^{E'}] t_1 [\alpha^{E'}/X] :: G_x[\alpha/X]) :: G_x[G'/X]
\]
and the result holds.

Case (Rasc). Then \( t = \varepsilon_1(\varepsilon_2 u :: G_2) :: G \). Then
\[
\Xi; \vdash \varepsilon_1(\varepsilon_2 u :: G_2) :: G
\]
If \( \varepsilon_2 \circ \varepsilon_1 \) is not defined, then \( \Xi \vdash t \rightarrow \text{error} \), and then the result hold immediately.

Suppose that consistent transitivity does hold, then
\[
\Xi \vdash \varepsilon_1(\varepsilon_2 u :: G_2) :: G \rightarrow \Xi \vdash (\varepsilon_2 \circ \varepsilon_1) u :: G
\]
where \( (\varepsilon_2 \circ \varepsilon_1) \models \Xi; \vdash G_u \sim G \). Then
\[
\Xi; \vdash (\varepsilon_2 \circ \varepsilon_1) u :: G
\]
and the result follows.

Case (Rop). Then \( t = \text{op}(\varepsilon u :: B') \). Then
\[
\Xi; \vdash \text{op}(\varepsilon u :: B') :: B
\]
Let us assume that \( ty(op) : B' \to B \).
\[
\Xi \vdash op(\varepsilon u :: B') \quad \implies \quad \Xi \vdash \varepsilon B \, \delta(op, \overline{u}) :: B
\]
But as \( \varepsilon_B \vdash \Xi ; \cdot \vdash B \sim B \), then
\[
(E_{\text{asc}}) \quad \Xi ; \cdot \vdash \delta(op, \overline{u}) : B \quad \varepsilon_B \vdash \Xi ; \cdot \vdash B \sim B \quad \implies \quad \Xi ; \cdot \vdash \varepsilon_B \, \delta(op, \overline{u}) :: B : B
\]
and the result follows.

**Case (Rpair).** Then \( t = \langle \varepsilon_1 u_1 :: G_1, \varepsilon_2 u_2 :: G_2 \rangle \). Then
\[
\Xi \vdash \langle \varepsilon_1 u_1 :: G_1, \varepsilon_2 u_2 :: G_2 \rangle \quad \implies \quad \Xi \vdash (\varepsilon_1 \times \varepsilon_2)(u_1, u_2) :: G_1 \times G_2
\]
By Lemma \ref{lem:278}, \( \varepsilon_1 \times \varepsilon_2 \vdash \Xi ; \cdot \vdash G_1' \times G_2' \sim G_1 \times G_2 \). Then
\[
(E_{\text{pair}}) \quad \Xi ; \cdot \vdash u_1 :: G_1' \quad \Xi ; \cdot \vdash u_2 :: G_2' \quad \Xi ; \cdot \vdash (u_1, u_2) :: G_1' \times G_2' \quad \Xi ; \cdot \vdash (\varepsilon_1 \times \varepsilon_2)(u_1, u_2) :: G_1 \times G_2
\]
and the result holds.

**Case (Rproji).** Then \( t = \pi_i(\varepsilon(u_1, u_2) :: G) \). Then
\[
\Xi ;\cdot \vdash u_i :: G_i' \quad \Xi ;\cdot \vdash \langle u_1, u_2 \rangle :: G_1' \times G_2' \quad \Xi ;\cdot \vdash \varepsilon(u_1, u_2) :: G \quad \Xi ;\cdot \vdash \pi_i(\varepsilon(u_1, u_2) :: G) :: \text{proj}_{i'}(G)
\]
By Lemma \ref{lem:279}, \( p_i(\varepsilon) \vdash \Xi ; \cdot \vdash \text{proj}_{i'}(G_1' \times G_2') \sim \text{proj}_{i'}(G) \). Then
\[
(E_{\text{app}}) \quad \Xi ;\cdot \vdash u_i :: G_i' \quad p_i(\varepsilon) \vdash \Xi ;\cdot \vdash \text{proj}_{i'}(G_1' \times G_2') \sim \text{proj}_{i'}(G) \quad \Xi ;\cdot \vdash p_i(\varepsilon) u_i :: \text{proj}_{i'}(G) :: \text{proj}_{i'}(G)
\]
and the result holds.

\( \square \)

**Proposition 281** (\( \langle \to \rangle \) is well defined). If \( \Xi ; \cdot \vdash t :: G \), then either
• $\Xi \triangleright t \mapsto \Xi' \triangleright t'$, $\Xi \subseteq \Xi'$ and $\Xi'; \cdot \vdash t' : G$; or

• $\Xi \triangleright t \mapsto \text{error}$

Proof.

Case (R→). Then $\Xi \triangleright t \mapsto \Xi' \triangleright t'$. By well-definedness of $\mapsto$ (Prop 280), $\Xi \subseteq \Xi'$ and $\Xi'; \cdot \vdash t' : G$; or the result holds.

Case (Rerr). If $\Xi \triangleright t \mapsto \text{error}$, then $\Xi \triangleright \epsilon \mapsto \text{error}$ and the result holds immediately.

Case (Rf). $t = f[t_1]$, $\Xi; \cdot \vdash f[t_1] : G$, $\Xi \triangleright t_1 \mapsto \Xi' \triangleright t_2$, and consider $f : G' \rightarrow G$. By induction hypothesis, $\Xi; \cdot \vdash t_2 : G'$, so $\Xi; \cdot \vdash f[t_2] : G$ and the result holds.

$\Xi \triangleright t \mapsto \Xi' \triangleright t'$ (→ is well defined). If $\Xi; \cdot \vdash t : G$, $t \leadsto t_\epsilon$, then $t_\epsilon$ is a value $v$; or $\Xi \triangleright t_\epsilon \mapsto \Xi' \triangleright t'_\epsilon$, $\Xi \subseteq \Xi'$ and $\Xi'; \cdot \vdash t'_\epsilon : G$; or $\Xi \triangleright t_\epsilon \mapsto \text{error}$.

Proof. By induction on the structure of $t$, using Lemma 282 and Canonical Forms (Lemma 274).

Now we can establish type safety: programs do not get stuck, though they may terminate with cast errors. Also the store of a program is well typed.

Proposition 246 (Type Safety). If $t^G \in G$ then either $t^G$ is a value $v$; $t^G \mapsto \text{error}$; or $t^G \mapsto t'^G$ for some term $t'^G \in G$.

Proof. Direct by 282.

D.4.2 Static Terms Do Not Fail

Lemma 283. If $\epsilon_1$ and $\epsilon_2$ two static evidences such that $\epsilon_1 \vdash \Xi; \Delta \vdash T_1 \sim T_2$ and $\epsilon_2 \vdash \Xi; \Delta \vdash T_2 \sim T_3$. Then $\epsilon_1 \circ \epsilon_2 = \langle p_1(\epsilon_1), p_2(\epsilon_2) \rangle$.

Proof. Straightforward induction on types $T_1, T_2, T_3$ ($\Xi; \Delta \vdash T_2 \sim T_3$ coincides with $\Xi; \Delta \vdash T_2 = T_3$), and optimality of evidences (Lemma 50), because the resulting evidence cannot gain precision as each component of the evidences are static.

Lemma 284. Let $T_1$ and $T_2$ two static types, and $\Xi$ a static store, such that $\Xi; \Delta \vdash T_1 \sim T_2$. Then $\iota(T_1, T_2) = \iota(\text{lift}_\Xi(T_1), \text{lift}_\Xi(T_2)) = \langle \text{lift}_\Xi(T_1), \text{lift}_\Xi(T_2) \rangle$.

Proof. Straightforward induction on types $T_1, T_2$, and noticing that we cannot gain precision from the types.
Proposition 285 (Static terms progress). Let \( t \) be a static term, \( \Xi \) a static store (\( \Xi = \Sigma \)), and \( G \) a static type (\( G = T \)). If \( \Sigma; \cdot \vdash t : T \), then either \( \Sigma \vdash t \rightarrow \Sigma' ; t' \), for some \( \Sigma, t' \) static; or \( t \) is a value \( v \).

**Proof.** By induction on the structure of a derivation of \( \Sigma; \cdot \vdash t : T \).

Note that \( \Xi; \Delta \vdash T_1 \sim T_2 \) coincides with \( \Xi; \Delta \vdash T_1 = T_2 \), so we use the latter notation throughout the proof.

**Case** \((t = \varepsilon u :: G)\). The result is trivial as \( t \) is a value.

**Case** \((t = (\varepsilon_1(\lambda x : T_{11} , t_1) :: T_1 \rightarrow T_2) (\varepsilon_2 u :: T_1))\). Then

\[
\frac{\Xi; \cdot \vdash t_1 : T_1 ; \Xi; \cdot \vdash t_1 : T_1 \varepsilon \vdash \Sigma; \Delta \vdash \varepsilon_1(\lambda x : T_{11} , t_1) :: T_1 \rightarrow T_1}{\Xi; \cdot \vdash (\varepsilon_1(\lambda x : T_{11} , t_1) :: T_1 \rightarrow T_1) \varepsilon \vdash \Sigma; \Delta \vdash (\varepsilon_2 u :: T_1) :: T_2}
\]

By Lemma 283 \( \varepsilon' = (\varepsilon_2 \circ \text{dom}(\varepsilon_1)) \) is defined and by Lemma 284 the new evidence is also static. Then

\[
\Xi \vdash (\varepsilon_1(\lambda x : T_{11} , t_1) :: T) (\varepsilon_2 u :: T_1) \rightarrow \Xi \vdash \text{cod}(\varepsilon_1)(t_1[\varepsilon' u :: T_{11}]/x)] :: T_2
\]

And the result holds immediately.

**Case** \((t = (\varepsilon_\Lambda x . t_1 :: \forall X . T_x) [T'])\). Then

\[
\frac{\Xi; X; \cdot \vdash t_1 : T_1 \varepsilon \vdash \Sigma; \Delta \vdash \varepsilon_\Lambda X . t_1 :: \forall X . T_x \forall X . T_x}{\Xi; \cdot \vdash (\varepsilon_\Lambda X . t_1 :: \forall X . T_x) [T'] :: T_x[T'/X]} \quad \Xi; \cdot \vdash T'
\]

Then

\[
(\varepsilon_\Lambda X . t_1 :: \forall X . T_x) [T'] \rightarrow \Xi' \vdash (\varepsilon_{\forall X . T_x}^{E'/\alpha E'} (\varepsilon [\alpha E'] t_1 [\alpha E'][X] :: T_x [\alpha X]) :: T_x[T'/X]
\]

where \( \Xi' \triangleq \Xi, \alpha := T', \alpha \notin \text{dom}(\Xi) \), and \( E' \triangleq \text{lift}_\Xi(T') \), and \( \varepsilon_{\forall X . T_x}^{E'/\alpha E'} = (\text{lift}_\Xi(T_x) [\alpha E'/X], \text{lift}_\Xi(T_x[T'/X])) \). Then, as \( \Xi \subseteq \Xi' \), and \( \Xi' \) is extended with a type name that maps to a static type the result holds immediately.

**Case** \((t = \Xi \vdash \varepsilon_1(\varepsilon_2 u :: T_2) :: T)\). Then

\[
\frac{\Xi; \cdot \vdash u : T_u \varepsilon_2 \vdash \Sigma; \Delta \vdash T_u = T_2}{\Xi; \cdot \vdash (\varepsilon_2 u :: T_2) :: T_2} \quad \Xi_1 \vdash \Sigma; \Delta \vdash T_2 = T
\]
By Lemma 283, $\varepsilon_2 \circ \varepsilon_1$ is defined and by Lemma 284, the new evidence is also static. Then

$$\Xi \triangleright \varepsilon_1(\varepsilon_2u :: T_2) :: T \rightarrow \Xi \triangleright (\varepsilon_2 \circ \varepsilon_1)u :: T$$

and the result holds.

**Case** ($t = \text{op}(\varepsilon u :: B')$). Then

$$\Xi \triangleright \varepsilon u :: B' \rightarrow B$$

Let us assume that $ty(op) : B' \rightarrow B$. Then

$$\Xi \triangleright \varepsilon u :: B' \rightarrow \Xi \triangleright \varepsilon B \delta(op, \overline{u}) :: B$$

And the result holds.

**Case** ($t = \langle \varepsilon_1 u_1 :: T_1, \varepsilon_2 u_2 :: T_2 \rangle$). Then

$$\Xi \triangleright \langle \varepsilon_1 u_1 :: T_1, \varepsilon_2 u_2 :: T_2 \rangle \rightarrow \Xi \triangleright (\varepsilon_1 \times \varepsilon_2)(u_1, u_2) :: T_1 \times T_2$$

and the result holds.

**Case** ($t = \pi_1(\varepsilon \langle u_1, u_2 \rangle :: T)$). Then

$$\Xi \triangleright \pi_1(\varepsilon \langle u_1, u_2 \rangle :: T) \rightarrow \Xi \triangleright \pi_1(\varepsilon)u_1 :: \text{proj}_1^\sharp(T)$$

And the result holds.

**Case** ($t = t_1 t_2$). Then by induction hypothesis $\Xi \triangleright t_1 \rightarrow \Xi \triangleright t'_1$, and $t'_1$ is static, and so $t'_1 t_2$.

**Case** ($t = v t_2$). Then by induction hypothesis $\Xi \triangleright t_2 \rightarrow \Xi \triangleright t'_2$, and $t'_2$ is static, and so $v t'_2$.

**Case** ($t = t_1[T], t = (t_1, t_2), t = \text{op}(\overline{t_1}), t = \pi_1(t_1)$). Similar inductive reasoning to application cases.

\[\Box\]
**Proposition 286** (Static terms do not fail). Let \( t \) be a static term, and \( G \) a static type \((G = T)\).
If \( \vdash t : T \) then \( \neg(t \Downarrow \text{error}) \).

*Proof.* Direct by Lemma 284. \( \square \)
D.5 GSF: Parametricity

In this section we present the logical relation for parametricity of GSF, the proof of the fundamental property, and the soundness of the logical relation w.r.t. contextual approximation.

D.5.1 Auxiliary Definitions

In this section we show function definitions used in the logical relation of GSF (Figure C.15).

**Definition 131.** \( ev(\varepsilon u :: G) = \varepsilon \)

**Definition 132.**

\[
const(E) = \begin{cases} 
B & E = B \\
? \to ? & E = E_1 \to E_2 \\
\forall X.? & E = \forall X.E_1 \\
? \times ? & E = E_1 \times E_2 \\
\alpha & E = \alpha^{E_1} \\
X & E = X \\
? & E = ?
\end{cases}
\]

D.5.2 Fundamental Property

**Theorem 53** (Fundamental Property). If \( \Xi; \Delta; \Gamma \vdash t : G \) then \( \Xi; \Delta; \Gamma \vdash t \preceq t : G \).

**Proof.** By induction on the type derivation of \( t \).

*Case (Easc).* Then \( t = \varepsilon s :: G \), and therefore:

\[
\Xi; \Delta; \Gamma \vdash s : G' \quad \Xi \vdash \Xi; \Delta \vdash G' \sim G
\]

\[
\Xi; \Delta; \Gamma \vdash \varepsilon s :: G' : G
\]

We follow by induction on the structure of \( s \).

- If \( s = b \) then:

\[
\Xi; \Delta; \Gamma \vdash b : B
\]

Then we have to prove that \( \Xi; \Delta; \Gamma \vdash \varepsilon b :: G \preceq \varepsilon b :: G : G \), but the result follows directly by Prop 287 (Compatibility of Constant).

- If \( s = \lambda x : G_1.t' \) then:

\[
\Xi; \Delta; \Gamma, x : G_1 \vdash t' : G_2
\]

\[
\Xi; \Delta; \Gamma \vdash \lambda x : G_1.t' : G_1 \to G_2
\]

382
Then we have to prove that:

\[ \Xi; \Delta; \Gamma \vdash \varepsilon(\lambda x : G_1.t') :: G \preceq \varepsilon(\lambda x : G_1.t') :: G \]

By induction hypotheses we already know that \( \Xi; \Delta; \Gamma, x : G_1 \vdash t' \preceq t' : G_2 \). But the result follows directly by Prop 288 (Compatibility of term abstraction).

- If \( s = \Lambda X.t' \) then:

\[
(E\Lambda) \quad \frac{\Xi; \Delta, X; \Gamma \vdash t' : G^* \quad \Xi; \Delta \vdash \Gamma}{\Xi; \Delta; \Gamma \vdash \Lambda X.t' : \forall X.G^*}
\]

Then we have to prove that:

\[ \Xi; \Delta; \Gamma \vdash \varepsilon(\Lambda X.t') :: G \preceq \varepsilon(\Lambda X.t') :: G \]

By induction hypotheses we already know that \( \Xi; \Delta, X; \Gamma \vdash t' \preceq t' : G^* \). But the result follows directly by Prop 289 (Compatibility of type abstraction).

- If \( s = \langle u_1, u_2 \rangle \) then:

\[
(E\text{pair}) \quad \frac{\Xi; \Delta; \Gamma \vdash u_1 : G_1 \quad \Xi; \Delta; \Gamma \vdash u_2 : G_2}{\Xi; \Delta; \Gamma \vdash \langle u_1, u_2 \rangle : G_1 \times G_2}
\]

Then we have to prove that:

\[ \Xi; \Delta; \Gamma \vdash \varepsilon(u_1, u_2) :: G \preceq \varepsilon(u_1, u_2) :: G \]

We know by premise that \( \Xi; \Delta; \Gamma \vdash \pi_1(\varepsilon)u_1 :: G_1 : G_1 \) and \( \Xi; \Delta; \Gamma \vdash \pi_2(\varepsilon)u_2 :: G_2 : G_2 \). Then by induction hypotheses we already know that: \( \Xi; \Delta; \Gamma \vdash \pi_1(\varepsilon)u_1 :: G_1 \preceq \pi_1(\varepsilon)u_1 :: G_1 \) and \( \Xi; \Delta; \Gamma \vdash \pi_2(\varepsilon)u_2 :: G_2 \preceq \pi_2(\varepsilon)u_2 :: G_2 \). But the result follows directly by Prop 290 (Compatibility of pairs).

- If \( s = t' \), and therefore:

\[
(E\text{asc}) \quad \frac{\Xi; \Delta; \Gamma \vdash t' : G' \quad \varepsilon \vdash \Xi; \Delta \vdash G' \sim G}{\Xi; \Delta; \Gamma \vdash \varepsilon t' :: G : G}
\]

By induction hypotheses we already know that \( \Xi; \Delta; \Gamma \vdash t' \preceq t' : G' \), then the result follows directly by Prop 293 (Compatibility of ascriptions).

Case (Epair). Then \( t = \langle t_1, t_2 \rangle \), and therefore:

\[
(E\text{pair}) \quad \frac{\Xi; \Delta; \Gamma \vdash t_1 : G_1 \quad \Xi; \Delta; \Gamma \vdash t_2 : G_2}{\Xi; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle : G_1 \times G_2}
\]

where \( G = G_1 \times G_2 \) Then we have to prove that:

\[ \Xi; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle \preceq \langle t_1, t_2 \rangle : G_1 \times G_2 \]

By induction hypotheses we already know that: \( \Xi; \Delta; \Gamma \vdash t_1 \preceq t_1 : G_1 \) and \( \Xi; \Delta; \Gamma \vdash t_2 \preceq t_2 : G_2 \). But the result follows directly by Prop 291 (Compatibility of pairs).

Case (Ex). Then \( t = x \), and therefore:

\[
(Ex) \quad \frac{x : G \in \Gamma \quad \Xi; \Delta \vdash \Gamma}{\Xi; \Delta; \Gamma \vdash x : G}
\]

Then we have to prove that \( \Xi; \Delta; \Gamma \vdash x \preceq x : G \). But the result follows directly by Prop 292 (Compatibility of variables).
Case (Eop). Then \( t = op(\overline{t}) \), and therefore:

\[
\begin{align*}
\text{(Eop)} & \quad \Xi; \Delta; \Gamma \vdash \overline{t} : \overline{\mathcal{G}} && \text{ty}(op) = \overline{\mathcal{G}}' \rightarrow G \\
\Xi; \Delta; \Gamma & \vdash op(\overline{t}) : G
\end{align*}
\]

Then we have to prove that: \( \Xi; \Delta; \Gamma \vdash op(\overline{t}) \preceq op(\overline{t}) : G \). By the induction hypothesis we obtain that: \( \Xi; \Delta; \Gamma \vdash \overline{t} \preceq \overline{t} : \overline{\mathcal{G}} \). Then the result follows directly by Prop 294 (Compatibility of app operator).

Case (Eapp). Then \( t = t_1 t_2 \), and therefore:

\[
\begin{align*}
\text{(Eapp)} & \quad \Xi; \Delta; \Gamma \vdash t_1 : G_{11} \\
\Xi; \Delta; \Gamma & \vdash t_2 : G_{12}
\end{align*}
\]

where \( G = G_{12} \). Then we have to prove that:

\[
\Xi; \Delta; \Gamma \vdash t_1 t_2 \preceq t_1 t_2 : G_{12}
\]

By the induction hypothesis we obtain that: \( \Xi; \Delta; \Gamma \vdash t_1 \preceq t_1 : G_{11} \) and \( \Xi; \Delta; \Gamma \vdash t_2 \preceq t_2 : G_{11} \). Then the result follows directly by Prop 295 (Compatibility of term application).

Case (EappG). Then \( t = t' [G_2] \), and therefore:

\[
\begin{align*}
\text{(EappG)} & \quad \Xi; \Delta; \Gamma \vdash t' \forall X. G_1 \\
\Xi; \Delta & \vdash G_2
\end{align*}
\]

where \( G = G_1[G_2/X] \). Then we have to prove that:

\[
\Xi; \Delta; \Gamma \vdash t' [G_2] \preceq t' [G_2] : G_1[G_2/X]
\]

By induction hypotheses we know that:

\[
\Xi; \Delta; \Gamma \vdash t' \preceq t' : \forall X. G_1
\]

Then the result follows directly by Prop 296 (Compatibility of type application).

Case (Epair1). Then \( t = \pi_1(t') \), and therefore:

\[
\begin{align*}
\text{(Epair1)} & \quad \Xi; \Delta; \Gamma \vdash t' : G_1 \times G_2 \\
\Xi; \Delta; \Gamma & \vdash \pi_1(t') : G_1
\end{align*}
\]

where \( G = G_1 \). Then we have to prove that: \( \Xi; \Delta; \Gamma \vdash \pi_1(t') \preceq \pi_1(t') : G_1 \). By the induction hypothesis we obtain that: \( \Xi; \Delta; \Gamma \vdash t' \preceq t' : G_1 \times G_2 \). Then the result follows directly by Prop 297 (Compatibility of access to the first component of the pair).

Case (Epair2). Then \( t = \pi_2(t') \), and therefore:

\[
\begin{align*}
\text{(Epair2)} & \quad \Xi; \Delta; \Gamma \vdash t' : G_1 \times G_2 \\
\Xi; \Delta; \Gamma & \vdash \pi_2(t') : G_2
\end{align*}
\]

where \( G = G_2 \). Then we have to prove that: \( \Xi; \Delta; \Gamma \vdash \pi_2(t') \preceq \pi_2(t') : G_2 \). By the induction hypothesis we obtain that: \( \Xi; \Delta; \Gamma \vdash t' \preceq t' : G_1 \times G_2 \). Then the result follows directly by Prop 298 (Compatibility of access to the second component of the pair).

\[\square\]
Proposition 287 (Compatibility-Eb). If \( b \in B, \varepsilon \vdash \Delta; \Gamma \vdash B \sim G \) and \( \Xi; \Delta \vdash \Gamma \) then:

\[
\Xi; \Delta; \Gamma \vdash \varepsilon b :: G \preceq \varepsilon b :: G : G
\]

Proof. As \( b \) is constant then it does not have free variables or type variables, then \( b = \rho(\gamma_i(b)) \). Then we have to prove that for all \( W \in S[\Xi] \) it is true that:

\[
(W, \rho_1(\varepsilon)b :: \rho(G), \rho_2(\varepsilon)b :: \rho(G)) \in \mathcal{T}_\rho[G]
\]

As \( \rho_1(\varepsilon)b :: G \) are values, then we have to proof that:

\[
(W, \rho_1(\varepsilon)b :: \rho(G), \rho_2(\varepsilon)b :: \rho(G)) \in \mathcal{V}_\rho[G]
\]

1. \( G = B \), we know that \( \langle B, B \rangle = \varepsilon \vdash \Xi; \Delta \vdash B \sim B \), then \( \rho_1(\varepsilon) = \varepsilon \) and the result follows immediately by the definition of \( \mathcal{V}_\rho[B] \).

2. If \( G \in \text{typeName} \) then \( \varepsilon = \langle H_3, \alpha^{E_1} \rangle \). Notice that as \( \alpha^{E_1} \) cannot have free type variables therefore \( H_3 \) neither. Then \( \varepsilon = \rho_1(\varepsilon) \). As \( \alpha \) is sync, then let us call \( G'' = W.\Xi_i(\alpha) \). We have to prove that:

\[
(W, \langle H_3, \alpha^{E_1} \rangle b :: \alpha, \langle H_3, \alpha^{E_1} \rangle b :: \alpha) \in \mathcal{V}_\rho[\alpha]
\]

which, by definition of \( \mathcal{V}_\rho[\alpha] \), is equivalent to prove that:

\[
(W, \langle H_3, E_4 \rangle b :: G'', \langle E_3, E_4 \rangle b :: G'') \in \mathcal{V}_\rho[G'']
\]

Then we proceed by case analysis on \( \varepsilon \):

- (Case \( \varepsilon = \langle H_3, \alpha^{\beta^{E_4}} \rangle \)). We know that \( \langle H_3, \alpha^{\beta^{E_4}} \rangle \vdash \Xi; \Delta \vdash B \sim \alpha \), then by Lemma \ref{lemma:313} \( \langle H_3, \beta^{E_4} \rangle \vdash \Xi; \Delta \vdash B \sim G'' \). As \( \beta^{E_4} \subseteq G'' \), then \( G'' \) can either be \( ? \) or \( \beta \).

If \( G'' = ? \), then by definition of \( \mathcal{V}_\rho[? \rangle \), we have to prove that the resulting values belong to \( \mathcal{V}_\rho[\beta] \). Also as \( \langle H_3, \beta^{E_4} \rangle \vdash \Xi; \Delta \vdash B \sim ? \), by Lemma \ref{lemma:311} \( \langle H_3, \beta^{E_4} \rangle \vdash \Xi; \Delta \vdash B \sim \beta \), and then we proceed just like this case once again (this is process is finite as there are no circular references by construction and it ends up in something different to a type name). If \( G'' = \beta \) we use an analogous argument as for \( G'' = ? \).

- (Case \( \varepsilon = \langle H_3, \alpha^{H_4} \rangle \)). Then using similar arguments as before, we have to prove that:

\[
(W, \langle H_3, H_4 \rangle b :: G'', \langle H_3, H_4 \rangle b :: G'') \in \mathcal{V}_\rho[G'']
\]

By Lemma \ref{lemma:313} \( \langle H_3, H_4 \rangle \vdash \Xi; \Delta \vdash B \sim G'' \). Then if \( G'' = ? \), we proceed as the case \( G = ? \), with the evidence \( \varepsilon = \langle H_3, H_4 \rangle \). If \( G'' \in \text{headType} \), we proceed as the previous case where \( G = B \), and the evidence \( \varepsilon = \langle H_3, H_4 \rangle \).

3. If \( G = ? \) we have the following cases:
Proposition 288. By the definition of \( V_\rho[\cdot] \) in this case we have to prove that:

\[(W, \rho_1(\varepsilon)b :: \text{const}(H_4), \rho_2(\varepsilon)b :: \text{const}(H_4)) \in V_\rho[\text{const}(H_4)]\]

but as \( \text{const}(H_4) = B \) (note that \( H_3 = B \) then since \( H_4 \in \text{HEADTYPE} \) has to be \( B \)). The result follows immediately since is part of the premise.

Proposition 288 (Compatibility-E\(\lambda\)). If \( \Xi; \Delta; \Gamma \vdash t \leq t' : G_2, \varepsilon \vdash \Xi; \Delta \vdash G_1 \rightarrow G_2 \sim G \) then:

\[\Xi; \Delta; \Gamma \vdash \varepsilon(\lambda x : G_1.t) :: G \leq \varepsilon(\lambda x : G_1.t') :: G : G\]

Proof. First, we are required to show that \( \Xi; \Delta; \Gamma \vdash \varepsilon(\lambda x : G_1.t) :: G : G \) and \( \Xi; \Delta; \Gamma \vdash \varepsilon(\lambda x : G_1.t') :: G : G \), which follow from \( \varepsilon \vdash \Xi; \Delta \vdash G_1 \rightarrow G_2 \sim G \) and \( \Xi; \Delta; \Gamma \vdash \lambda x : G_1.t : G_1 \rightarrow G_2 \) and \( \Xi; \Delta; \Gamma \vdash \lambda x : G_1.t' : G_1 \rightarrow G_2 \) respectively, which follow (respectively) from \( \Xi; \Delta; \Gamma \vdash t : G_2 \) and \( \Xi; \Delta; \Gamma \vdash t' : G_2 \), which follow from \( \Xi; \Delta; \Gamma \vdash t \leq t' : G_2 \).

Consider arbitrary \( W, \rho, \gamma \) such that \( W \in S[\Xi], (W, \rho) \in D[\Delta] \) and \( (W, \gamma) \in G_\rho[\Gamma] \). We are required to show that:

\[(W, \rho(\gamma_1(\varepsilon(\lambda x : G_1.t) :: G))), \rho(\gamma_2(\varepsilon(\lambda x : G_1.t) :: G))) \in J_\rho[G]\]

Consider arbitrary i, \( v_1 \) and \( \Xi_1 \) such that \( i < W.j \) and:

\[W.\Xi_1 \triangleright \rho(\gamma_1(\varepsilon(\lambda x : G_1.t) :: G)) \rightarrow^i \Xi_1 \triangleright v_1\]

Since \( \rho(\gamma_1(\varepsilon(\lambda x : G_1.t) :: G)) = \varepsilon_1(\lambda x : \rho(G_1), \rho(\gamma_1(t))) :: \rho(G) \) and \( \rho(G) \) is already a value, where \( \varepsilon_1 = \rho(\varepsilon) \), we have \( i = 0 \) and \( v_1 = \varepsilon_1(\lambda x : \rho(G_1), \rho(\gamma_1(t))) :: \rho(G) \) and \( \Xi_1 = W.\Xi_1 \). Since \( \varepsilon_2(\lambda x : \rho(G_1), \rho(\gamma_2(t))) :: \rho(G) \) is already a value, we are required to show that \( \Xi \in W'. \Xi \) such that \( W'.j + i = W.j, W' \geq W, W_1.\Xi = \Xi_1, W'.\Xi_2 = \Xi_2 \) and:

\[(W', \varepsilon_1(\lambda x : \rho(G_1), \rho(\gamma_1(t))) :: \rho(G), \varepsilon_2(\lambda x : \rho(G_1), \rho(\gamma_2(t))) :: \rho(G)) \in V_\rho[G]\]

Let \( W' = W \), then we have to show that:

\[(W, \varepsilon_1(\lambda x : \rho(G_1), \rho(\gamma_1(t))) :: \rho(G), \varepsilon_2(\lambda x : \rho(G_1), \rho(\gamma_2(t))) :: \rho(G)) \in V_\rho[G]\]

First we have to prove that:

\[W.\Xi_1; \Delta; \Gamma \vdash \varepsilon_1(\lambda x : \rho(G_1), \rho(\gamma_1(t))) :: \rho(G) : \rho(G)\]
As we know that $\Xi; \Delta; \Gamma \vdash \varepsilon(\lambda x : G_1.t) :: G : G$, by Lemma 309 the result follows immediately. The case $W.\Xi_2; \Delta; \Gamma \vdash \varepsilon'_2(\lambda x : \rho(G_1),\rho(\gamma(t'))) :: \rho(G) : \rho(G)$ is similar.

The type $G$ can be $G_1' \rightarrow G_2'$, for some $G_1'$ and $G_2'$, or a $\text{TYPE}$.

1. $G = G_1' \rightarrow G_2'$, we are required to show that $\forall W'', v_1' = \varepsilon'_1 u_1' :: \rho(G_1'), v_2' = \varepsilon'_2 u_2' :: \rho(G_1')$, such that $W'' \supseteq W$ and $(W'', v_1', v_2') \in V_G[G_1']$, it is true that:

$$(W'', \varepsilon'_1(\lambda x : \rho(G_1).\rho(\gamma_1(t)))) :: \rho(G_1' \rightarrow G_2') v_1',
\varepsilon'_2(\lambda x : \rho(G_1).\rho(\gamma_2(t'))) :: \rho(G_1' \rightarrow G_2') v_2') \in \mathcal{T}_\rho[G_2']$$

If $(\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))$ fails, then by Lemma 310 $(\varepsilon'_2 \circ \text{dom}(\varepsilon'_2))$ and the result follows immediately.

Else, if $(\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))$ follow we have to proof that:

$$\left(\downarrow W'', \text{cod}(\varepsilon'_1)(\rho(\gamma_1(t))[(\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))u_1' :: \rho(G_1)/x : \rho(G_1)]) :: \rho(G_2'),
\text{cod}(\varepsilon'_2)(\rho(\gamma_2(t'))[(\varepsilon'_2 \circ \text{dom}(\varepsilon'_2))u_2' :: \rho(G_1)/x : \rho(G_1)]) :: \rho(G_2') \in \mathcal{T}_\rho[G_2']\right)$$

Note that $\text{dom}(\varepsilon'_1) \vdash W''.\Xi_1 \vdash \rho(G_1') \sim \rho(G_1)$. By the Lemma 301 (with the type $G_1$ and the evidences $\text{dom}(\varepsilon'_1) \vdash W''.\Xi_1 \vdash \rho(G_1') \sim \rho(G_1)$) it is true that:

$$(W'', \text{dom}(\varepsilon'_1)v_1' :: G_1, \text{dom}(\varepsilon'_2)v_2' :: G_1) \in \mathcal{T}_\rho[G_1]$$

Since $(\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))$ does not fail, it is true that:

$$(W'', (\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))u_1' :: G_1, (\varepsilon'_2 \circ \text{dom}(\varepsilon'_2))u_2' :: G_1) \in V\rho[G_1]$$

We instantiate the hypothesis $\Xi; \Delta; \Gamma \vdash t \leq t' : G_2$, with $W''$, $\rho$ and $\gamma[x : \rho(G_1) \mapsto (v''_1, v''_2)]$, where $v''_1 = (\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))u'_1 :: \rho(G_1)$. Note that $\delta[\Xi] \ni W'' \supseteq W$ by the definition of $\delta[\Xi]$, $(W'', \rho) \in \mathcal{D}[\Delta]$ by the definition of $\mathcal{D}[\Delta]$ and $(W'', (v''_1, v''_2)) \in \mathcal{G}_\rho[\Gamma, x : \rho(G1)]$, which follow from: $(W'', \gamma) \in \mathcal{G}_\rho[\Gamma]$ and $(W'', v''_1, v''_2) \in V\rho[G_1]$ which follows from above. Then, we have that:

$$(W'', \rho(\gamma_1(t))[v''_1/x], \rho(\gamma_2(t'))[v''_2/x]) \in \mathcal{T}_\rho[G_2]$$

If the following term reduces to error, then the result follows immediately.

$$(W''.\Xi_1 \vdash \rho(\gamma_1(t))[v''_1/x])$$

If the above is not true, then the following terms reduce to values $(v_1f)$ and $\exists W'' \supseteq W''$ such that $(W'', v_1f, v_2f) \in V\rho[G_2]$.

$$(W''.\Xi_1 \vdash \rho(\gamma_1(t))[v''_1/x] \rightarrow^\ast W''.\Xi_1 \vdash v_1f)
(W''.\Xi_2 \vdash \rho(\gamma_2(t'))[v''_2/x] \rightarrow^\ast W''.\Xi_2 \vdash v_2f)$$
We instantiate the induction hypothesis in the previous result with the type $G_2'$ and the evidence $\text{cod}(\varepsilon') \vdash W'.\Xi_i \vdash G_2'' \sim G_2'$, then we obtain that:

$$(\uparrow W'', \text{cod}(\varepsilon')v_{1f} :: \rho(G_2'), \text{cod}(\varepsilon'')v_{2f} :: \rho(G_2')) \in \mathcal{J}_\rho[G_2']$$

and the result follows immediately.

Let $u_1 = \lambda x : \rho(G_1).\rho(\gamma_1(t))$, $u_2 = \lambda x : \rho(\rho(G_1).\rho(\gamma_2(t'')))$ and $G^* = G_1 \rightarrow G_2$, we have to prove that:

$$(W, \rho_1(\varepsilon)u_1 :: \rho(G), \rho_2(\varepsilon)u_2 :: \rho(G)) \in \mathcal{V}_\rho[G]$$

2. If $G \in \text{TYPE}_\text{NAME}$ then $\varepsilon = \langle H_3, \alpha^{E_4}\rangle$. Notice that as $\alpha^{E_4}$ cannot have free type variables therefore $H_3$ neither. Then $\varepsilon = \rho_1(\varepsilon)$. As $\alpha$ is sync, then let us call $G'' = W.\Xi_i(\alpha)$. We have to prove that:

$$(W, \langle H_3, \alpha^{E_4}\rangle u_1 :: \alpha, \langle H_3, \alpha^{E_4}\rangle u_2 :: \alpha) \in \mathcal{V}_\rho[\alpha]$$

which, by definition of $\mathcal{V}_\rho[\alpha]$, is equivalent to prove that:

$$(W, \langle H_3, E_4\rangle u_1 :: G'', \langle E_3, E_4\rangle u_2 :: G'') \in \mathcal{V}_\rho[G'']$$

Then we proceed by case analysis on $\varepsilon$:

- (Case $\varepsilon = \langle H_3, \alpha^{E_4}\rangle$). We know that $\langle H_3, \alpha^{E_4}\rangle \vdash \Xi; \Delta \vdash G^* \sim \alpha$, then by Lemma 313 $\langle H_3, \beta^{E_4}\rangle \vdash \Xi; \Delta \vdash G^* \sim G''$. As $\beta^{E_4} \sqsubseteq G''$, then $G''$ can either be $\beta$ or $\beta$.

  If $G'' = \beta$, then by definition of $\mathcal{V}_\rho[\beta]$, we have to prove that the resulting values belong to $\mathcal{V}_\rho[\beta]$. Also as $\langle H_3, \beta^{E_4}\rangle \vdash \Xi; \Delta \vdash G^* \sim \beta$, by Lemma 313 $\langle H_3, \beta^{E_4}\rangle \vdash \Xi; \Delta \vdash G^* \sim G''$, and then we proceed just like this case once again (this is process is finite as there are no circular references by construction and it ends up in something different to a type name). If $G'' = \beta$ we use an analogous argument as for $G'' = \beta$.

- (Case $\varepsilon = \langle H_3, \alpha^{H_4}\rangle$). Then using similar arguments as before, we have to prove that

  $$(W, \langle H_3, H_4\rangle u_1 :: G'', \langle H_3, H_4\rangle u_2 :: G'') \in \mathcal{V}_\rho[G'']$$

  By Lemma 313 $\langle H_3, H_4\rangle \vdash \Xi; \Delta \vdash G^* \sim G''$. Then if $G'' = \beta$, we proceed as the case $G = \beta$, with the evidence $\varepsilon = \langle H_3, H_4\rangle$. If $G'' \in \text{HEAD}_\text{TYPE}$, we proceed as the previous case where $G = G_1' \rightarrow G_2'$, and the evidence $\varepsilon = \langle H_3, H_4\rangle$.

3. If $G = \beta$ we have the following cases:

- $(G = \beta, \varepsilon = \langle H_3, H_4\rangle)$. By the definition of $\mathcal{V}_\rho[\beta]$ in this case we have to prove that:
(W, ρ₁(ε)u₁ :: ρ(G), ρ₂(ε)u₂ :: ρ(G′)) ∈ ⌈V_ρ[const(H₄)]⌉

but as \(const(H₄) = \? \rightarrow \?\), we proceed just like this case where \(G = \alpha'_1 \rightarrow \alpha_2\), where \(G'_1 = ?\) and \(G'_2 = ?\).

\(\ast\) \((G = ?, \varepsilon = \langle H_3, \alpha^{E_i} \rangle)\). Notice that as \(\alpha^{E_i}\) cannot have free type variables therefore \(E_3\) neither. Then \(\varepsilon = \rho_i(\varepsilon)\). By the definition of \(V_ρ[?]\) we have to prove that

\[(W, \langle H_3, \alpha^{E_i} \rangle) u_1 :: \alpha, \langle H_3, \alpha^{E_i} \rangle u_2 :: \alpha) \in ⌈V_ρ[\alpha]⌉\]

Note that by Lemma 311 we know that \(\varepsilon \vdash \Xi; \Delta \vdash \varepsilon; \Delta \vdash G^* \sim \alpha\). Then we proceed just like the case \(G \in \text{typeName}\).

\[\square\]

**Proposition 289 (Compatibility-ENA).** If \(\Xi; \Delta, X \vdash t_1 \leq t_2: G, \varepsilon \vdash \Xi; \Delta \vdash \forall X.G \sim G'\) and \(\Xi; \Delta \vdash \Gamma\) then \(\Xi; \Delta; \Gamma \vdash \varepsilon(\Lambda X.t_1) :: G' \leq \varepsilon(\Lambda X.t_2) :: G' : G'\).

**Proof.** First, we are required to prove that \(\Xi; \Delta, X \vdash t_1 : G\), therefore:

\[
\Xi; \Delta, X; \Gamma \vdash t_1 : G \quad \Xi; \Delta \vdash \Gamma
\]

Then we can conclude that:

\[
\Xi; \Delta; \Gamma \vdash \Lambda X.t_1 \in \forall X.G \quad \varepsilon \vdash \Xi; \Delta \vdash \forall X.G \sim G'
\]

\[
\Xi; \Delta; \Gamma \vdash \varepsilon(\Lambda X.t_1) :: G' : G'
\]

Consider arbitrary \(W, \rho, \gamma\) such that \(W \in S[\Xi], (W, \rho) \in D[\Delta]\) and \((W, \gamma) \in G_ρ[\Gamma]\). We are required to show that:

\[(W, \rho(\gamma_1(\varepsilon(\Lambda X.t_1)) :: G')), \rho(\gamma_2(\varepsilon(\Lambda X.t_2)) :: G'))) \in T_ρ[G']\]

First we have to prove that:

\[W, \Xi \vdash \rho(\gamma_1(\varepsilon(\Lambda X.t_1)) :: G') :: \rho(G')\]

As we know that \(\Xi; \Delta; \Gamma \vdash \varepsilon(\Lambda X.t_i) :: G' : G'\), by Lemma 309 the result follows immediately.

By definition of substitutions \(\rho(\gamma_1(\varepsilon(\Lambda X.t_1)) :: G') = \varepsilon'_1(\Lambda X.\rho(\gamma_1(t_i)))) :: \rho(G')\), where \(\varepsilon'_1 = \rho_i(\varepsilon)\), therefore we have to prove that:

\[(W, \varepsilon'_1(\Lambda X.\rho(\gamma_1(t_1)))) :: \rho(G'), \varepsilon'_2(\Lambda X.\rho(\gamma_1(t_2)))) :: \rho(G') \in T_ρ[G']\]

We already know that both terms are values and therefore we only have to prove that:

\[(W, \varepsilon'_1(\Lambda X.\rho(\gamma_1(t_1)))) :: \rho(G'), \varepsilon'_2(\Lambda X.\rho(\gamma_1(t_2)))) :: \rho(G') \in V_ρ[G']\]

389
The type $G'$ can be $\forall X. G'_1$ for some $G'_1$, or a Type Name. Let $u_1 = \Lambda X. \rho(\gamma_1(t_1))$, $u_2 = \Lambda X. \rho(\gamma_2(t_2))$ and $G^* = \forall X. G$, we have to prove that:

$$(W, \rho_1(\varepsilon)u_1 :: \rho(G), \rho_2(\varepsilon)u_2 :: \rho(G)) \in V'_\rho[G']$$

1. If $G' = \forall X. G'_1$, then consider $W' \succeq W$, and $G_1, G_2, R$ and $\alpha$, such that $W'.\Xi_i \vdash G_i$, and $R \in \text{REL}_{W'.\Xi_i}[G_1, G_2]$.

$$W'.\Xi_1 \triangleright \varepsilon'_1(\Lambda X. \rho(\gamma_i(t_i))) :: \forall X. \rho(G'_1) \ [G_i] \rightarrow$$

$$W'.\Xi_1, \alpha := G_i \triangleright \varepsilon^E_{\forall X, \rho(G'_1)}(\varepsilon'_1[\alpha^E]/\rho(\gamma_i(t_i)))[\alpha^E/X] :: \rho(G'_1)[\alpha/X] :: \rho(G'_1)[G_i/X]$$

where $E'_i = \text{lift}_{W'.\Xi_i}(G_i)$.

Note that $\varepsilon \vdash \Xi; \Delta \vdash \forall X. G \sim \forall X. G'_1$, then $\varepsilon = \langle \forall X. E_1, \forall X. E_2 \rangle$, for some $E_1$, $E_2$, $K$ and $L$. By the Lemma 308, we know that $\varepsilon'_1 \vdash W.\Xi_1; \Delta \vdash \forall X. \rho(G) \sim \forall X. \rho(G'_1)$, then $\varepsilon'_1 = \langle \forall X. E_{i1}, \forall X. E_{i2} \rangle$, where $\forall X. E_{i1} = \rho(E_i)$ and $E_{i2} = \rho(E_i)$.

Then we have to prove that:

$$(W'', (\varepsilon'_1[\alpha^E_j])\rho(\gamma_1(t_i))[\alpha^E_j/X] :: \rho(G'_1)[\alpha/X], (\varepsilon'_2[\alpha^E_j])\rho(\gamma_2(t_2))[\alpha^E_j/X] :: \rho(G'_1)[\alpha/X]) 
\in \mathcal{T}_{\rho[X \mapsto \alpha]}[G'_1]$$

where $W'' \downarrow (W' \boxtimes (\alpha, G_1, G_2, R))$.

Let $\rho' = \rho[X \mapsto \alpha]$. We instantiate the premise $\Xi; \Delta; \Gamma \vdash t_1 \preceq t_2 : G$ with $W''$, $\rho'$ and $\gamma$, such that $W'' \in \mathcal{S}[\Xi]$, as $\alpha \in \text{dom}(W'.\Xi_i[\alpha \mapsto R])$ then $(W'', \rho') \in \mathcal{D}[\Delta, X]$. Also note that as $X$ is fresh, then $\forall (v_1^*, v_2^*) \in \text{cod}(\gamma)$, such that $\Xi; \Delta; \Gamma \vdash v^*_i : G^*, X \notin \text{FV}(G^*)$, then it is easy to see that $(W'', \gamma) \in \mathcal{G}_{\rho[X \mapsto \alpha]}[\Gamma]$. Then we know that:

$$(W'', \rho'(\gamma_1(t_1)), \rho'(\gamma_2(t_2))) \in \mathcal{T}_{\rho'}[G]$$

But note that:

$$\rho'(\gamma_1(t_1)) = \rho[\alpha/X](\gamma_1(t_1)) = \rho(\gamma_1(t_1))[\alpha^E_j/X]$$

Then we have that:

$$(W'', \rho(\gamma_1(t_1))[\alpha^E_j/X], \rho(\gamma_2(t_2))[\alpha^E_j/X]) \in \mathcal{T}_{\rho[\alpha/X]}[G]$$

If the following term reduces to error, then the result follows immediately.

$$W''.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\alpha^E_j/X]$$

If the above is not true, then the following terms reduce to values $(v_{i1})$ and $\exists W''' \geq W''$ such that $(W''', v_{i1}, v_{i2}) \in V'_{\rho[\alpha \mapsto X]}[G]$. 

$$W''.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\alpha^E_j/X] \longrightarrow^* W'''.\Xi_i \triangleright v_{i1}$$
We instantiate the Lemma [301] with the type \( G'_1 \) and the evidence \( \langle E_1, E_2 \rangle \vdash \Xi; \Delta, X \vdash G \sim G'_1 \) (remember that \( \varepsilon = \langle \forall X. E_1, \forall X. E_2 \rangle \)). Note that \( \varepsilon \circ [\alpha_{E_1}] = \rho(X \mapsto \alpha) \vartriangleright \Xi, \langle (E_1, E_2) \rangle, \rho(X \mapsto \alpha) \rangle \{G'_1\} = \rho(G'_1) \alpha/X, W'' \in S[\Xi] \) and \((W'', \rho(X \mapsto \alpha)) \in D[\Delta, \Xi, X] \). Then we obtain that:

\[
(\uparrow W'', (\varepsilon_1 \circ [\alpha_{E_1}])_1 v_1 \varepsilon: \rho(G'_1) \alpha/X, (\varepsilon_2 \circ [\alpha_{E_2}])_2 v_2 \varepsilon: \rho(G'_1) \alpha/X) \in \mathcal{T}_\rho[G'_1]
\]

and the result follows immediately.

2. If \( G' \in \text{TypeName} \) then \( \varepsilon = \langle H_3, \alpha^E \rangle \). Notice that as \( \alpha^E \) cannot have free type variables therefore \( H_3 \) neither. Then \( \varepsilon = \rho_i(\varepsilon) \). As \( \alpha \) is sync, then let us call \( G'' = W.\Xi_i(\alpha) \). We have to prove that:

\[
(W, \langle H_3, \alpha^E \rangle u_1 :: \alpha, \langle H_3, \alpha^E \rangle u_2 :: \alpha) \in \mathcal{V}_\rho[\alpha]
\]

which, by definition of \( \mathcal{V}_\rho[\alpha] \), is equivalent to prove that:

\[
(W, \langle H_3, E_4 \rangle u_1 :: G'', \langle E_3, E_4 \rangle u_2 :: G'') \in \mathcal{V}_\rho[G'']
\]

Then we proceed by case analysis on \( \varepsilon \):

- (Case \( \varepsilon = \langle H_3, \alpha^E \rangle \)). We know that \( \langle H_3, \alpha^E \rangle \vdash \Xi; \Delta \vdash G^* \sim \alpha \), then by Lemma [313] \( \langle H_3, \beta^E \rangle \vdash \Xi; \Delta \vdash G^* \sim G'' \). As \( \beta^E \subseteq G'' \), then \( G'' \) can either be \( ? \) or \( \beta \).

If \( G'' = ? \), then by definition of \( \mathcal{V}_\rho[?] \), we have to prove that the resulting values belong to \( \mathcal{V}_\rho[?] \). Also as \( \langle H_3, \beta^E \rangle \vdash \Xi; \Delta \vdash G^* \sim ? \), by Lemma [311] \( \langle H_3, \beta^E \rangle \vdash \Xi; \Delta \vdash G^* \sim \beta \), and then we proceed just like this case once again (this is process is finite as there are no circular references by construction and it ends up in something different to a type name). If \( G'' = \beta \) we use an analogous argument as for \( G'' = ? \).

- (Case \( \varepsilon = \langle H_3, \alpha^H \rangle \)). Then using similar arguments as before, we have to prove that

\[
(W, \langle H_3, H_4 \rangle u_1 :: G'', \langle H_3, H_4 \rangle u_2 :: G'') \in \mathcal{V}_\rho[G'']
\]

By Lemma [313] \( \langle H_3, H_4 \rangle \vdash \Xi; \Delta \vdash G^* \sim G''. \) Then if \( G'' = ? \), we proceed as the case \( G' = ? \), with the evidence \( \varepsilon = \langle H_3, H_4 \rangle \). If \( G'' \in \text{HEADTYPE} \), we proceed as the previous case where \( G' = \forall X.G \), and the evidence \( \varepsilon = \langle H_3, H_4 \rangle \).

3. If \( G' = ? \) we have the following cases:

- (\( G' = ?, \varepsilon = \langle H_3, H_4 \rangle \)). By the definition of \( \mathcal{V}_\rho[?] \) in this case we have to prove that:

\[
(W, \rho_1(\varepsilon)u_1 :: \rho(G), \rho_2(\varepsilon)u_2 :: \rho(G)) \in \mathcal{V}_\rho[const(H_4)]
\]

but as \( const(H_4) = \forall X.\? \), we proceed just like the case where \( G' = \forall X.G'_1 \), where \( G'_1 = ? \).
• \((G' = ?, \varepsilon = \langle H_3, \alpha^{E_3} \rangle)\). Notice that as \(\alpha^{E_3}\) cannot have free type variables therefore \(E_3\) neither. Then \(\varepsilon = \rho_i(\varepsilon)\). By the definition of \(\nu_\rho[?]\) we have to prove that

\[
(W, \langle H_3, \alpha^{E_3} \rangle u_1 :: \alpha, \langle H_3, \alpha^{E_3} \rangle u_2 :: \alpha) \in \nu_\rho[\alpha]
\]

Note that by Lemma \[\text{311}\] we know that \(\varepsilon \vdash \Xi; \Delta \vdash G^* \sim \alpha\). Then we proceed just like the case \(G' \in \text{TYPE}\).

\[
\]

**Proposition 290 (Compatibility-EpairU).** If \(\Xi; \Delta; \Gamma \vdash \pi_1(\varepsilon)u_1 :: G_1 \leq \pi_1(\varepsilon)u'_1 :: G_1 \times G_2, \Xi; \Delta; \Gamma \vdash \pi_2(\varepsilon)u_2 :: G_2 \leq \pi_2(\varepsilon)u'_2 :: G_2 \times G_2\), and \(\varepsilon \models \Xi; \Delta \vdash G_1 \times G_2 \sim G\) then:

\[
\Xi; \Delta; \Gamma \vdash \varepsilon(u_1, u_2) :: G \leq \varepsilon(u'_1, u'_2) :: G : G
\]

**Proof.** By induction on the subterms.

\[
\]

**Proposition 291 (Compatibility-Epair).** If \(\Xi; \Delta; \Gamma \vdash t_1 \leq t'_1 : G_1\) and \(\Xi; \Delta; \Gamma \vdash t_2 \leq t'_2 : G_2\), then \(\Xi; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle \leq \langle t'_1, t'_2 \rangle : G_1 \times G_2\).

**Proof.** By induction on the subterms.

\[
\]

**Proposition 292 (Compatibility-Ex).** If \(x : G \in \Gamma\) and \(\Xi; \Delta; \Gamma \vdash \exists\) then \(\Xi; \Delta; \Gamma \vdash x \leq x : G\).

**Proof.** First, we are required to show \(\Xi; \Delta; \Gamma \vdash x : G\), which is immediate. Consider arbitrary \(W, \rho, \gamma\) such that \(W \in \delta[\Xi], (W, \rho) \in \delta[\Delta]\) and \((W, \gamma) \in \mathcal{G}_\rho[\Gamma]\). We are required to show that:

\[
(W, \rho(\gamma_1(x)), \rho(\gamma_2(x))) \in \mathcal{T}_\rho[G]
\]

Consider arbitrary \(i, v_1\) and \(\Xi_1\) such that \(i < W, j)\) and \(W.\Xi_1 \triangleright \rho(\gamma_1(x)) \rightarrow^! \Xi_1 \triangleright v_1\). Since \(\rho(\gamma_1(x)) = \gamma_1(x)\) and \(\gamma_1(x)\) is already a value, we have \(i = 0\) and \(\gamma_1(x) = v_1\). We are required to show that exists \(\Xi_2, v_2\) such that \(W.\Xi_2 \triangleright \gamma_2(x) \rightarrow^* \Xi_2 \triangleright v_2\) which is immediate (since \(\rho(\gamma_2(x)) = \gamma_2(x)\) is a value and \(\Xi_2 = W.\Xi_2\)). Also, we are required to show that \(\exists W'\), such that \(W'.j + i = W, j \wedge W' \subseteq W \wedge W'.\Xi_1 = \Xi_1 \wedge W'.\Xi_2 = \Xi_2 \wedge (W', \gamma_1(x), \gamma_2(x)) \in \nu_\rho[G]\). Let \(W' = W\), then \((W, \gamma_1(x), \gamma_2(x)) \in \nu_\rho[G]\) because of the definition of \((W, \gamma) \in \mathcal{G}_\rho[\Gamma]\).

**Proposition 293 (Compatibility-Easc).** If \(\Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : G\) and \(\varepsilon \models \Xi; \Delta \vdash G \sim G'\) then \(\Xi; \Delta; \Gamma \vdash \varepsilon t_1 :: G' \leq \varepsilon t_2 :: G' : G'\).

**Proof.** First we are required to prove that \(\Xi; \Delta; \Gamma \vdash \varepsilon t_1 :: G' : G'\), but by \(\Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : G\) we already know that \(\Xi; \Delta; \Gamma \vdash t_1 : G\), therefore:

\[
(E\text{asc}) \quad \Xi; \Delta; \Gamma \vdash t_1 :: G \quad \varepsilon \vdash \Xi; \Delta \vdash G \sim G' \quad \Xi; \Delta; \Gamma \vdash \varepsilon t_1 :: G' : G'
\]

\[
392
\]
Consider arbitrary $W, \rho, \gamma$ such that $W \in S[\Xi]$, $(W, \rho) \in D[\Delta]$ and $(W, \gamma) \in G_\rho[\Gamma]$. We are required to show that:

$$(W, \rho(\gamma_1(\varepsilon t_1 :: G')), \rho(\gamma_2(\varepsilon t_2 :: G'))) \in J_\rho[G']$$

But by definition of substitutions $\rho(\gamma_1(\varepsilon t_1 :: G')) = \rho(\varepsilon)\rho(\gamma_1(t_1)) :: \rho(G')$, therefore we have to prove that:

$$(W, \rho(\varepsilon)\rho(\gamma_1(t_1))) :: \rho(G'), \rho(\varepsilon)\rho(\gamma_2(t_2)) :: \rho(G')) \in J_\rho[G']$$

First we have to prove that:

$$W.\Xi_1 \vdash \rho(\varepsilon)\rho(\gamma_1(t_1)) :: \rho(G') : G'$$

As we know that $\Xi; \Delta; \Gamma \vdash t_1 :: G' : G'$, by Lemma 309 the result follows intermediately.

Second, consider arbitrary $i < W.j, \Xi_1$. Either there exist $v_1$ such that:

$$W.\Xi_1 \vdash \rho(\varepsilon)\rho(\gamma_1(t_1')) :: \rho(G') \longrightarrow^i \Xi_1 \vdash v_1$$

or

$$W.\Xi_1 \vdash \rho(\varepsilon)\rho(\gamma_1(t_1')) :: \rho(G') \longrightarrow^i \text{error}$$

Let us suppose that $W.\Xi_1 \vdash \rho(\gamma_1(t_1')) :: \rho(G') \longrightarrow^i \Xi_1 \vdash v_1$. Hence, by inspection of the operational semantics, it follows that there exist $i_1 + 1 < i, \Xi_{11}$ and $v_{11}$ such that:

$$W.\Xi_1 \vdash \rho(\varepsilon)\rho(\gamma_1(t_1')) :: \rho(G') \longrightarrow^{i_1} \Xi_{11} \vdash \rho(\varepsilon)v_{11} :: \rho(G')$$

We instantiate the hypothesis $\Xi; \Delta; \Gamma \vdash t_1' :: t_2' : G$ with $W, \rho$ and $\gamma$ to obtain that:

$$(W, \rho(\gamma_1(t_1')), \rho(\gamma_2(t_2'))) \in J_\rho[G]$$

We instantiate $J_\rho[G]$ with $i_1, \Xi_{11}$ and $v_{11}$ (note that $i_1 < i < W.j$), hence there exists $v_{12}$ and $W_1$, such that $W_1 \preceq W, W_1.j = W.j - i_1, W_1.w_2 \triangleright \rho(\gamma_2(t_2')) \longrightarrow^* W_1.\Xi_{12} \triangleright v_{12}, W_1.\Xi_1 = \Xi_{11}, v_{12}$ and $(W_1, v_{11}, v_{12}) \in \mathcal{V}_\rho[G]$.

Since we have that $(W_1, v_{11}, v_{12}) \in \mathcal{V}_\rho[G]$, then it is true that $(W_1, \rho(\varepsilon)v_{11} :: G', \rho(\varepsilon)v_{12} :: G') \in J_\rho[G']$ by the Lemma 301.

By the inspection of the operational semantics:

$$W.\Xi_1 \vdash \rho(\varepsilon)\rho(\gamma_1(t_1)) :: \rho(G') \longrightarrow^{i_1} W_1.\Xi_1 \vdash \rho(\varepsilon)v_{11} :: \rho(G') \longrightarrow^1 \Xi_1 \vdash v_1$$

We instantiate $(W_1, \rho(\varepsilon)v_{11} :: G', \rho(\varepsilon)v_{12} :: G') \in J_\rho[G']$ with $1, v_1$ and $\Xi_1$. Therefore there must exist $v_2$ and $W'$ such that $W' \succeq W_1$ (note that $W' \succeq W$), $W'.j = W_1.j - (i - i_1 - 2) = W.j - i_1 - 1 = W.j - i,$

$$W_1.\Xi_2 \triangleright \rho(\varepsilon)v_{12} :: \rho(G') \longrightarrow^* \Xi_2 \triangleright v_2$$

and $(W', v_1, v_2) \in \mathcal{V}_\rho[G']$ then the result follows. \qed
**Proposition 294 (Compatibility-Eop).** If $\Xi; \Delta; \Gamma \vdash t \preceq t' : G$ and $ty(op) = \overline{G} \rightarrow G$ then $\Xi; \Delta; \Gamma \vdash op(t) \preceq op(t') : G$.

*Proof.* Similar to the term application. \hfill $\Box$

**Proposition 295 (Compatibility-Eapp).** If $\Xi; \Delta; \Gamma \vdash t_1 \preceq t_1' : G_{11} \rightarrow G_{12}$ and $\Xi; \Delta; \Gamma \vdash t_2 \preceq t_2' : G_{11}$ then $\Xi; \Delta; \Gamma \vdash t_1 t_2 \preceq t_1' t_2' : G_{12}$.

*Proof.* First, we are required to show that:

$\Xi; \Delta; \Gamma \vdash t_1 t_2 : G_{12}$

which follows directly from (Eapp) as $\Xi; \Delta; \Gamma \vdash t_1 : G_1$, and $\Xi; \Delta; \Gamma \vdash t_2 : G_2$. Also, we are required to prove that:

$\Xi; \Delta; \Gamma \vdash t_1' t_2' : G_{12}$

which follows analogously.

Second, consider $\Delta$ and $\Gamma$ such that $\Gamma \supseteq FV(t_1 t_2)$, and $\Gamma \supseteq FV(t_1' t_2')$, and consider arbitrary $W, \rho, \gamma$ such that $W \in S[\Xi], (W, \rho) \in D[\Delta]$ and $(W, \gamma) \in G[\rho][\Pi]$. We are required to show that:

$\forall i \in \mathbb{N}, v \in \mathbb{V}, W.\Xi_1 \triangleright \rho(\gamma_1(t_1 t_2)) \triangleright^i \Xi_1 \triangleright v \lor W.\Xi_1 \triangleright \rho(\gamma_1(t_1 t_2)) \triangleright^i \text{error}$

Hence, by inspection of the operational semantics, it follows that there exist $i_1 < i$, $\Xi_{11}$ and $v_{11}$ such that:

$W.\Xi_{11} \triangleright \rho(\gamma_1(t_1)) \triangleright^{i_1} \Xi_{11} \triangleright v_{11} \lor W.\Xi_{11} \triangleright \rho(\gamma_1(t_1)) \triangleright^{i_1} \text{error}$

If $W.\Xi_1 \triangleright \rho(\gamma_1(t_1)) \triangleright^{i_1} \text{error}$ then $W.\Xi_{11} \triangleright \rho(\gamma_1(t_1)) \triangleright^{i_1} \text{error}$ and the result holds immediately. Let us assume that the reduction do not fail. We instantiate the hypothesis $\Xi; \Delta; \Gamma \vdash t_1 \preceq t_1' : G_{11} \rightarrow G_{12}$ with $W, \rho$ and $\gamma$ we obtain that:

$\forall i \in \mathbb{N}, v \in \mathbb{V}, W.\Xi_1 \triangleright \rho(\gamma_1(t_1)) \triangleright^* \Xi_{11} \triangleright v_{11}, W.\Xi_{11} = \Xi_{11}$ and $(W, v_{11}, v_{11}') \in V[\rho][G_{11} \rightarrow G_{12}]$.
Note that:

\[ W . \Xi_1 \triangleright \rho (\gamma_1 (t_1 t_2)) \rightarrow^{i_1} \Xi_{11} \triangleright v_{11}(\rho (\gamma_1 (t_2))) \rightarrow^{i_1 i_1} \Xi_1 \triangleright v_1 \]

or

\[ W . \Xi_1 \triangleright \rho (\gamma_1 (t_1 t_2)) \rightarrow^{i_1} \Xi_{11} \triangleright v_{11}(\rho (\gamma_1 (t_2))) \rightarrow^{i_1} \text{error} \]

Hence, by inspection of the operational semantics, it follows that there exist \( i_2 < i - i_1, \Xi_{22} \) and \( v_{22} \) such that:

\[ \Xi_{11} \triangleright \rho (\gamma_1 (t_2)) \rightarrow^{i_2} \Xi_{22} \triangleright v_{22} \lor \Xi_{11} \triangleright \rho (\gamma_1 (t_2)) \rightarrow^{i_2} \text{error} \]

We instantiate the hypothesis \( \Xi; \Delta; \Gamma \vdash t_2 \leq t'_2 : G_{11} \) with \( W_1, \rho \) and \( \gamma \), then we obtain that:

\[(W_1, \rho (\gamma_1 (t_2))), \rho (\gamma_2 (t_2))) \in \mathcal{J}_\rho [G_2] \]

If \( \Xi_{11} \triangleright \rho (\gamma_1 (t_2)) \rightarrow^{i_2} \text{error} \) then we instantiate with \( \Xi_{22} \) and \( \Xi_{22} \triangleright \rho (\gamma_1 (t'_2)) \rightarrow^{i_1} \text{error} \) and the result holds immediately. Let us assume that the reduction do not fail. We instantiate this with \( i_2 \) (note that \( i_2 < i - i_1 < W'.j = W.j - i_1 \)), \( \Xi_{22} \) and \( v_{22} \), hence there exists \( v'_{22} \) and \( W_2 \), such that \( W_2 \Xi_1 = \Xi_{22}, W_2 \succeq W_1, W_2.j = W_1.j - i_2 \) and

\[ W_1.\Xi_{2} \triangleright \rho (\gamma_2 (t'_2)) \rightarrow^{*} W_2.\Xi_{2} \triangleright v'_{22} \]

and \((W_2, v_{22}, v'_{22}) \in \mathcal{V}_\rho [G_{11}]\).

Note that:

\[ W . \Xi_1 \triangleright \rho (\gamma_1 (t_1 t_2)) \rightarrow^{i_1} \Xi_{11} \triangleright v_{11}(\rho (\gamma_1 (t_2))) \rightarrow^{i_2} \Xi_{22} \triangleright v_{22} \rightarrow^{i_1 i_1 i_2} \Xi_1 \triangleright v_1 \]

Since \((W_1, v_{11}, v'_{11}) \in \mathcal{V}_\rho [G_{11} \rightarrow G_{12}]\), we instantiate this with \( W_2, \rho (G_{11} \rightarrow G_{12}), v_{22} \) and \( v'_{22} \) (note that \((W_2, v_{22}, v'_{22}) \in \mathcal{V}_\rho [G_1] \) and \( W_2 \succeq W_1 \)). Then \((W_2, v_{11}, v_{22}, v'_{11}, v'_{22}) \in \mathcal{J}_\rho [G_{2}] \).

Since \((W_2, v_{11}, v_{22}, v'_{11}, v'_{22}) \in \mathcal{J}_\rho [G_{2}] \), we instantiate this with \( i - i_1 - i_2 \) (note that \( i - i_1 - i_2 < W_2.j = W.j - i_1 - i_2 \) since \( i < W.j \)), \( v_1 \) and \( \Xi_1 \).

If \( W_2.\Xi_{1} \triangleright v_{11}, v_{22} \rightarrow^{i_1 i_1 i_2} \text{error} \) then \( W_2.\Xi_{2} \triangleright v'_{11}, v'_{22} \rightarrow^{*} \text{error} \) and the result holds. Let us assume that the reduction does not fail. Hence there exists \( v_2 \) and \( W' \), such that \( W' \succeq W_2 \) (note that \( W' \succeq W \)), \( W'.j = W_2.j - (i - i_1 - i_2) = W.j - i \), \( W_2.\Xi_{2} \triangleright v'_{11}, v'_{22} \rightarrow^{*} W'.\Xi_{2} \triangleright v_{2}, W'.\Xi_{1} = \Xi_1 \) and \((W', v_1, v_2) \in \mathcal{V}_\rho [T_2] \), then the proof is complete. \( \Box \)

**Proposition 296 (Compatibility-EappG).** If \( \Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : \forall X.G \) and \( \Xi; \Delta \vdash G' \), then

\[ \Xi; \Delta; \Gamma \vdash t_1[G'] \leq t_2[G'] : G[G'/X] \]

**Proof.** First we are required to prove that \( \Xi; \Delta; \Gamma \vdash t_1[G'] : G[G'/X] \), but by \( \Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : \forall X.G \) we already know that \( \Xi; \Delta; \Gamma \vdash t_1 : \forall X.G \), therefore:

395
Consider $\Gamma = FV(t_1[ G'])$, and consider arbitrary $W, \rho, \gamma$ such that $W \in \delta[\Xi]$, $(W, \rho) \in D[\Delta]$ and $(W, \gamma) \in G_\rho[\Gamma]$. We are required to show that:

$$(W, \rho(\gamma_1(t_1[ G'])), \rho(\gamma_2(t_2[ G']))) \in J_\rho[G'[ G'/X]]$$

But by definition of substitutions $\rho(\gamma_1(t_1[ G'])) = \rho(\gamma_1(t_1))[\rho(G')]$, therefore we have to prove that:

$$(W, \rho(\gamma_1(t_1))[\rho(G')], \rho(\gamma_2(t_2))[\rho(G')]) \in J_\rho[G'[ G'/X]]$$

First we have to prove that:

$$W.\Xi_1 \vdash \rho(\gamma_1(t_1))[\rho(G')] : \rho(G)[\rho(G')/X]$$

As we know that $\Xi; \Delta; \Gamma \vdash t_1[ G']: G'[ G'/X]$, by Lemma 309 the result follows immediately. Second, consider arbitrary $i < W.j$ and $\Xi_1$. Either there exist $v_i$ such that $W.\Xi_1 \vdash \rho(\gamma_1(t_1))[\rho(G')] \longmapsto v_i$ or $W.\Xi_1 \vdash \rho(\gamma_1(t_1))[\rho(G')] \longmapsto error$. First, let us suppose that:

$$W.\Xi_1 \vdash \rho(\gamma_1(t_1))[\rho(G')] \longmapsto v_1$$

Hence, by inspection of the operational semantics, it follows that there exist $i_1 + 1 < i$, $\varepsilon_1$ and $v_{i_1}$ such that

$$W.\Xi_1 \vdash \rho(\gamma_1(t_1))[\rho(G')] \longmapsto^{\varepsilon_1} \Xi_1 \triangleright v_{i_1}$$

We instantiate the premise $\Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : \forall X.G$ with $W$, $\rho$ and $\gamma$ to obtain that:

$$(W, \rho(\gamma_1(t_1)), \rho(\gamma_2(t_2))) \in J_\rho[\forall X.G]$$

We instantiate $J_\rho[\forall X.G]$ with $i_1$, $\Xi_1$ and $v_{i_1}$ (note that $i_1 < i < W.j$), hence there exists $v_{i_2}$ and $W_1$, such that $W_1 \geq W$, $W_1.j = W.j - i_1$, $W.\Xi_2 \triangleright \rho(\gamma_2(t_2)) \longmapsto^{*} W_1.\Xi_2 \triangleright v_{i_2}$, $W_1.\Xi_1 = \Xi_1$, $v_{i_2}$ and:

$$(W_1, v_{i_1}, v_{i_2}) \in V_\rho[\forall X.G]$$

Then by inspection of the operational semantics:

$$W.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\rho(G')] \longmapsto^{*} W_1.\Xi_1 \triangleright v_{i_1}[\rho(G')]$$

$$\longmapsto W_1.\Xi_1, \alpha := \rho(G') \triangleright \varepsilon_1(\varepsilon_2' t_1' :: \rho(G)[\alpha/X]) :: \rho(G)[\rho(G')/X]$$

for some $\varepsilon_1$, $\varepsilon_2$, $\varepsilon_1'$, $\varepsilon_2'$, $t_1'$ and $\alpha \notin dom(W_1.\Xi_1)$. Let us call $t_1'' = (\varepsilon_1' t_1' :: \rho(G)[\alpha/X])$. We instantiate $V_\rho[\forall X.G]$ with $\alpha$, $t_1''$, $\rho(G')$, $R = V_\rho[\Gamma']$, $\varepsilon_1$, $\varepsilon_2$ and $W_1$. Then $(W_1', t_1''', t_2'') \in J_{\rho[\alpha \rightarrow \beta]}[\Gamma']$, where $W_1' = (\downarrow W_1) \boxtimes (\alpha, \rho(G'), \rho(G'), V_\rho[\Gamma'])$.

We instantiate $J_{\rho[\alpha \rightarrow \beta]}[\Gamma']$ with $i_2 = i - i_1 - 2$ (note that $i - i_1 - 2 < W_1.j = W.j - i_1 - 2$ since $i < W.j$), $\Xi_1$, $v_1'$, such that:

$$W_1.\Xi_1 \triangleright (\varepsilon_1' t_1' :: \rho(G)[\alpha/X]) \longmapsto^{i_2-1} \Xi_1 \triangleright (\varepsilon_1' v_1'' :: \rho(G)[\alpha/X]) \longmapsto \Xi_1 \triangleright v_1'$$
for some $v''$. Therefore there must exist $v''$, and $W'$ such that $W' \succeq W'_1$ (note that $W' \succeq W$), $W'.j = W'_1.j - (i - i_1 - 2) = W.j - i$.

$$W_1 \Xi_2 \triangleright (v'' \rho (G)[\alpha/X]) \mapsto^* W'.\Xi_2 \triangleright (v'' \rho (G)[\alpha/X]) \mapsto W'.\Xi_2 \triangleright v''$$

for some $v''$, $W'.\Xi_1 = \Xi_1$ and $(W', v'_1, v'_2) \in \mathcal{V}_{\rho[X \to \alpha]}[G]$.

Notice that $t_i$ reduce to a type abstraction of the form $v_{11} = \langle \forall X.E_{i_1}, \forall X.E_{i_2} \rangle X.t''$. Let us call $v'_i = ε'' \rho (G)[\alpha/X]$, as $π_2(ε''') \equiv π_2(ε'')$, then $G_{\rho} = \text{unlift}(π_2(ε'''))$, then $E_i = \text{lift}_{W_2}.(G_{\rho})$, and $E_i' = \text{lift}_{W_1}.(\rho(G'))$, and $ε_i = \langle E_i[α/X], E_i'[E'/X] \rangle$.

Then as $(W', v'_1, v'_2) \in \mathcal{V}_{\rho[X \to \alpha]}[G]$ by Lemma 300

$$(\downarrow W', (v'' \circ ε_1)u'' \rho (G)[\rho(G')/X], (v'' \circ ε_2)u'' \rho (G)[\rho(G')/X]) \in \mathcal{V}_{\rho}[G[G'/X]]$$

Let us call $v_i = (v'' \circ ε_i)u'' \rho (G)[\rho(G')/X]$. Where the theorem holds by instantiating $\mathcal{J}_P[G[G'/X]]$ with $\Xi_1, v_i, i = i_1 + i_2 + 2$ and therefore $W'.\Xi_1 \triangleright v'_i \rho (G)[\rho(G')/X] \mapsto W'.\Xi_1 \triangleright v_1$. Then there must exists some $v_2$ such that $W'.\Xi_2 \triangleright v_2 \rho (G)[\rho(G')/X] \mapsto W'.\Xi_2 \triangleright v_2$, and the result follows.

Now let us suppose that $W.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\rho(G')] \mapsto^1 \Xi_1 \triangleright \text{error}$. We instantiate the hypothesis $\Xi; Δ; Γ \triangleright t_1 \leq t_2 : \forall X.G$ with $W$, $\rho$ and $γ$ to obtain that $(W, \rho(\gamma_1(t_1)), \rho(\gamma_2(t_2))) \in \mathcal{J}_P[\forall X.G]$. If $W.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\rho(G')] \mapsto^1 \Xi_1 \triangleright \text{error}$, for some $\Xi_1$ and $i_1 < W.j$ then $W.\Xi_2 \triangleright \rho(\gamma_2(t_2))[\rho(G')] \mapsto^* \Xi_2 \triangleright \text{error}$, for some $\Xi_2$, and the result follows immediately.

If not, then there exists for some $i_1, \Xi_{11}$ and $v_{11}$, and therefore there exists $v_{12}$ and $W_1$, such that $W.\Xi_2 \triangleright \rho(\gamma_2(t_2)) \mapsto^* W_1.\Xi_2 \triangleright v_{12}, W_1.\Xi_1 = \Xi_{11}, v_{12}$ and $(W_1, v_{11}, v_{12}) \in \mathcal{V}_{\rho}[\forall X.G]$.

Then by inspection of the operational semantics:

$$W.\Xi_1 \triangleright \rho(\gamma_1(t_1))[\rho(G')] \mapsto^* W_1.\Xi_1 \triangleright v_{11}[\rho(G')]$$

$$\mapsto^* W_1.\Xi_1, α := \rho(G') \triangleright ε_1 t'_i \rho (G)[\rho(G')/X]$$

for some $ε_i, t'_i$, and $α \not\in \text{dom}(W_1.\Xi_1)$.

We instantiate $\mathcal{V}_{\rho}[\forall X.G]$ with $ev(v_{11}) α, t'_1, \rho(G'), \mathcal{V}_{\rho}[G'], ε_i$ and $\downarrow W_1$. Then $(W'_1, t'_1, t'_2) \in \mathcal{J}_P[\forall X.G]$, where $W'_1 = (\downarrow W_1) \boxdot (α, ρ(G'), \rho(G'), \mathcal{V}_{\rho}[G'])$.

Then if $W_1.\Xi_1 \triangleright t'_1 \mapsto^i \text{error}$ for some $i_2 < W'_1.j$, then $W_1.\Xi_2 \triangleright t'_2 \mapsto^* \text{error}$ and the result follows immediately.

□

**Proposition 297 (Compatibility-Epair1).** If $\Xi; Δ; Γ \vdash t_1 \preceq t_2 : G_1 \times G_2$ then $\Xi; Δ; Γ \vdash π_1(t_1) \preceq π_1(t_2) : G_1$.

**Proof.** By induction on the subterms. □

397
Proposition 298 (Compatibility-Epair2). If \( \Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : G_1 \times G_2 \) then \( \Xi; \Delta; \Gamma \vdash \pi_2(t_1) \leq \pi_2(t_2) : G_2 \).

Proof. By induction on the subterms. \( \square \)

Lemma 299. Let \( E_i = \text{lift}_{\Xi_i}(G_p) \) for some \( G_p \sqsubseteq G \), \( \langle E_{i1}, E_{i2} \rangle \vdash \Xi_i \vdash G_u \sim G \), and \( E_{12} \equiv E_{22} \), then
\[ \langle E_{11}, E_{12} \rangle \circ \langle E_1, E_1 \rangle \iff \langle E_{21}, E_{22} \rangle \circ \langle E_2, E_2 \rangle. \]

Proof. Note that by definition \( E_1 \equiv E_2 \). Also, \( \forall \alpha^E \in \text{FTN}(E_i), E = \text{lift}_{\Xi_i}(\Xi_i(\alpha)) \). Then we prove \( \Rightarrow \) direction (the other is analogous), by induction on the structure of the evidences \( \langle E_{i1}, E_{i2} \rangle \). We skip cases where \( E_i = ? \) or \( E_{i1} = ? \), as the result is trivial (combination never fails).

Case \( \langle E_{i1}, E_{i2} \rangle = \langle E_{i1}, \alpha^{E_{i2}} \rangle \). Then \( \langle E_{21}, E_{22} \rangle = \langle E_{21}, \alpha^{E_{22}} \rangle \), and \( E_i = \langle \alpha^{E_{i1}}, \alpha^{E_{i2}} \rangle \), where \( E_{i1} = \text{lift}_{\Xi_i}(\Xi_i(\alpha)) \), and therefore \( E_{12} \equiv E_{i1} \). And then by Lemma 310, the result holds immediately as both combinations are defined.

Case \( \langle E_{i1}, E_{i2} \rangle = \langle E_{i1}, B \rangle \). Then \( \langle E_{21}, E_{22} \rangle = \langle E_{12}, B \rangle \), and \( \langle E_1, E_1 \rangle = \langle B, B \rangle \), and the result trivially holds.

Case \( \langle E_{i1}, E_{i2} \rangle = \langle \alpha^{E_{i1}}, E_{i2} \rangle \). The result holds by de inspection of consistent transitivity rule (selR) and induction on evidence \( \langle E_{i1}, E_{i2} \rangle \).

Case \( \langle E_{i1}, E_{i2} \rangle = \langle E_{i11} \rightarrow E_{i12}, E_{i12} \rightarrow E_{i2} \rangle \). Then \( \langle E_{11}, E_{12} \rangle = \langle E_{111} \rightarrow E_{112}, E_{121} \rightarrow E_{122} \rangle \), and \( \langle E_1, E_i \rangle = \langle E_{11} \rightarrow E_{12}, E_{11} \rightarrow E_{122} \rangle \). As consistent transitivity is a symmetric relation, then the result holds by induction hypothesis on combinations of evidence \( \langle E_{i11} \rightarrow E_{i12} \rangle \circ \langle E_{i11}, E_{i11} \rangle \) and \( \langle E_{i12} \rightarrow E_{i2} \rangle \circ \langle E_{i12}, E_{i12} \rangle \).

For the other cases we proceed analogous to the function case. \( \square \)

Proposition 300 (Compositionality). Let \( \rho' = \rho[X \mapsto \alpha] \) and \( E_i' = \text{lift}_{W\Xi_i}(\rho(G')) \), \( W\Xi_i(\alpha) = \rho(G) \) and \( W\kappa(\alpha) = \psi_p[G'] \), \( E_i = \text{lift}_{W\Xi_i}(G_p) \) for some \( G_p \sqsubseteq \rho(G) \), \( e_i = \langle E_i(E_i(X)/X), E_i([E_i(X)]) \rangle \), \( e_i^{-1} = \langle E_i(E_i(X)/X), E_i([E_i(X)]) \rangle \), such that \( e_i \vdash W\Xi \vdash \rho(G[\alpha/X]) \sim \rho(G[G'/X]) \), and \( e_i^{-1} \vdash W\Xi \vdash \rho(G[\alpha/X]) \sim \rho(G[G'/X]) \) then

1.
\[
(W, e'_1u_1 :: \rho'(G), e'_2u_2 :: \rho'(G)) \in \psi_{\rho}[G] \Rightarrow \\
(\downarrow W, (e'_1 \circ e_1)u_1 :: \rho(G[G'/X]), (e'_2 \circ e_2)u_2 :: \rho(G[G'/X])) \in \psi_{\rho}[G] 
\]

2.
\[
(W, e'_1u_1 :: \rho(G[G'/X]), e'_2u_2 :: \rho(G[G'/X])) \in \psi_{\rho}[G[G'/X]] \Rightarrow \\
(\downarrow W, (e'_1 \circ e_1^{-1})u_1 :: \rho'(G), (e'_2 \circ e_2^{-1})u_2 :: \rho'(G)) \in \psi_{\rho}[G] 
\]

Proof. We proceed by induction on \( G \). Let \( v_1 = e'_1u_1 :: \rho'(G) \). We prove (1) first.
Case ($G = X$). Let $v_i = \langle H_{11}, \alpha^{E_{12}} \rangle u_i :: \alpha$. Then we know that

$$(W, \langle H_{11}, \alpha^{E_{12}} \rangle u_1 :: \alpha, \langle H_{21}, \alpha^{E_{22}} \rangle u_2 :: \alpha) \in \wp_{\rho[X \rightarrow \alpha]}[X]$$

which is equivalent to

$$(W, \langle H_{11}, \alpha^{E_{12}} \rangle u_1 :: \alpha, \langle H_{21}, \alpha^{E_{22}} \rangle u_2 :: \alpha) \in \wp_{\rho[X \rightarrow \alpha]}[\alpha]$$

As $W.\Xi_i(\alpha) = \rho(G')$ and $W.\kappa(\alpha) = \wp_\rho[G']$, we know that:

$$(W, \langle H_{11}, E_{12} \rangle u_1 :: \rho(G'), \langle H_{21}, E_{22} \rangle u_2 :: \rho(G')) \in \wp_\rho[G']$$

Then $\varepsilon_i \vdash W.\Xi_i \vdash \alpha \sim \rho(G')$, and $\varepsilon_i$ has to have the form $\varepsilon_i = \langle \alpha^{E'_1}, E'_1 \rangle$. As $E'_1 = \text{lift}_{W.\Xi_i}(\rho(G'))$ (initial evidence for $\alpha$), then $E_{i2} \sqsubseteq E'_1$, and therefore by Lemma 316 $\langle H_{11}, \alpha^{E_{12}} \rangle \circ \langle \alpha^{E'_1}, E'_1 \rangle = \langle H_{11}, E_{i2} \rangle$, and then we have to prove that

$$(\downarrow W, \langle H_{11}, E_{12} \rangle u_1 :: \rho(G'), \langle H_{21}, E_{22} \rangle u_2 :: \rho(G')) \in \wp_\rho[G']$$

which we already know, and the result holds.

Case ($G = Y$). Let $v_i = \langle H_{11}, \beta^{E_{12}} \rangle u_i :: \beta$, where $\rho'(Y) = \beta$. Then we know that

$$(W, \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \beta, \langle H_{21}, \beta^{E_{22}} \rangle u_2 :: \beta) \in \wp_{\rho[X \rightarrow \alpha]}[Y]$$

which is equivalent to

$$(W, \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \beta, \langle H_{21}, \beta^{E_{22}} \rangle u_2 :: \beta) \in \wp_{\rho[X \rightarrow \alpha]}[\beta]$$

Then $\varepsilon_i \vdash W.\Xi_i \vdash \beta \sim \beta$, and $\varepsilon_i$ has to have the form $\varepsilon_i = \langle \beta^{E'_1}, \beta^{E'_1} \rangle$, and $\beta^{E'_1} = \text{lift}_{W.\Xi_i}(\beta)$. By Lemma 299, we assume that both combinations of evidence are defined (otherwise the result holds immediately):

$$\langle H_{11}, \beta^{E_{12}} \rangle \circ \langle \beta^{E'_1}, \beta^{E'_1} \rangle = \langle H_{11}, \beta^{E_{12}} \rangle$$

Then we have to prove that

$$(\downarrow W, \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \beta, \langle H_{21}, \beta^{E_{22}} \rangle u_2 :: \beta) \in \wp_\rho[\beta]$$

which we already know by Lemma 303 and the result holds.

Case ($G = ?$). Let $v_i = \langle H_{11}, \alpha_{i2} \rangle u_i :: \?$. Then by definition of $\wp_\rho[?]$, let $G'' = \text{const}(E_{i2})$ (where $G'' \neq ?$). Then we know

$$(W, \langle H_{11}, E_{12} \rangle u_1 :: G'', \langle H_{21}, E_{22} \rangle u_2 :: G'') \in \wp_\rho[G'']$$

If $\varepsilon_i = \langle ?, ? \rangle$, then, by Lemma 301, the result holds immediately. If $\varepsilon_i = \langle E_i, E_i \rangle$, where $E_i \neq ?$, then we proceed similar to the other cases where $G \neq ?$. 

399
Case \((G = G_1 \rightarrow G_2)\). We know that

\[
(W, v_1, v_2) \in V_\rho[G_1 \rightarrow G_2]
\]

Then we have to prove that

\[
(\downarrow W, (\varepsilon'_1 \circ \varepsilon_1)(\lambda x : G'_1.t_1) :: \rho(G_1[G'/X]) \rightarrow \rho(G_2[G'/X])),
(\varepsilon'_2 \circ \varepsilon_2)(\lambda x : G'_2.t_2) :: \rho(G_1[G'/X]) \rightarrow \rho(G_2[G'/X])) \in V_\rho[G_1[G'/X] \rightarrow G_2[G'/X]]
\]

Let us call \(v''_1 = (\varepsilon'_1 \circ \varepsilon_1)(\lambda x : G'_1.t_1) :: \rho'(G_1) \rightarrow \rho'(G_2)\). By unfolding, we have to prove that

\[
\forall W'. \forall u'_1, u'_2. (W', v'_1, v'_2) \in V_\rho[G_1[G'/X]] \Rightarrow (W', v''_1, v'_1, v''_2, v'_2) \in V_\rho[G_2[G'/X]]
\]

Suppose that \(v'_1 = \varepsilon''_1 u'_1 :: \rho(G_1[G'/X])\), by inspection of the reduction rules, we know that

\[
W'. \Xi \triangleright v''_1 v'_1 \mapsto W'. \Xi \triangleright (\text{cod}(\varepsilon'_1) \circ \text{cod}(\varepsilon_1))t_1[(\varepsilon''_1 \circ (\text{dom}(\varepsilon_1) \circ \text{dom}(\varepsilon'_1)))u'_1 :: G'_1/x] :: \rho(G_2[G'/X])
\]

This is equivalent by Lemma \(302\).

\[
W'. \Xi \triangleright v''_1 v'_1 \mapsto W'. \Xi \triangleright (\text{cod}(\varepsilon'_1) \circ \text{cod}(\varepsilon_1))t_1[(\varepsilon''_1 \circ \text{dom}(\varepsilon_1)) \circ \text{dom}(\varepsilon'_1))u'_1 :: G'_1/x] :: \rho(G_2[G'/X])
\]

Notice that \(\text{dom}(\varepsilon_1) \vdash W. \Xi_i \vdash \rho(G_1[G'/X]) \sim \rho(G_1[\alpha/X])\), by Lemma \(299\) we assume that both combinations of evidence are defined (otherwise the result holds immediately), then let us assume that \((\varepsilon''_1 \circ \text{dom}(\varepsilon_1))\) is defined. We can use induction hypothesis on \(v'_1\), with evidences \(\text{dom}(\varepsilon_1)\). Then we know that \((\downarrow W', (\varepsilon''_1 \circ \text{dom}(\varepsilon_1))u'_1 :: \rho'(G_1), (\varepsilon''_2 \circ \text{dom}(\varepsilon_2))u'_2 :: \rho'(G_1)) \in V_\rho[G_1]\). Let us call \(v''''_1 = (\varepsilon''_1 \circ \text{dom}(\varepsilon_1))u'_1 :: \rho'(G_1)\).

Now we instantiate

\[
(W', v_1, v_2) \in V_\rho[G_1 \rightarrow G_2]
\]

with \(W'\) and \(v''''_1\), to obtain that either both executions reduce to an error (then the result holds immediately), or \(\exists W'' \succeq W'\) such that \((W'', v_{f1}, v_{f2}) \in V_\rho[G_2]\)

\[
W'. \Xi \triangleright v_1 v''''_1 \mapsto W'. \Xi \triangleright \text{cod}(\varepsilon_1) t_1[(\varepsilon''_1 \circ \text{dom}(\varepsilon_1)) \circ \text{dom}(\varepsilon'_1))u'_1 :: G'_1/x] :: \rho'(G_2)
\]

\[
\mapsto *W''. \Xi \triangleright v_{f1}
\]

Suppose that \(v_{f1} = \varepsilon_{f1} u_{f1} :: \rho'(G_2)\). Then we use induction hypothesis once again using evidences \(\text{cod}(\varepsilon_1)\) over \(v_{f1}\) (noticing that by Lemma \(299\) the combination of evidence either both fail or both are defined), to obtain that,

\[
(\downarrow W'', (\varepsilon_{f1} \circ \text{cod}(\varepsilon_1))u_{f1} :: \rho(G_2[G'/X]), (\varepsilon_{f2} \circ \text{cod}(\varepsilon_2))u_{f2} :: \rho(G_2[G'/X])) \in V_\rho[G_2[G'/X]]
\]

and the result holds.

Case \((\forall Y. G_1)\). We know that

\[
(W, v_1, v_2) \in V_\rho[\forall Y. G_1]
\]

Then we have to prove that

\[
(\downarrow W, (\varepsilon'_1 \circ \varepsilon_1)(\Lambda Y.t_1) :: \forall Y. \rho(G_1[G'/X]),
(\varepsilon'_2 \circ \varepsilon_2)(\Lambda Y.t_2) :: \forall Y. \rho(G_1[G'/X])) \in V_\rho[\forall Y. G_1[G'/X]]
\]

400
Let \( \varepsilon_1' = \langle \forall Y.E_{i1}, \forall Y.E_{i2} \rangle \) and \( \varepsilon_1 = \langle \forall Y.E_{i1}, \forall Y.E_{i2} \rangle = \langle \forall Y.E''_1[\alpha^{E_i}/X], \forall Y.E''_1[E_i'/X] \rangle \), where \( E_1 = \forall Y.E''_1 \). Let us call \( v''_1 = (\varepsilon_1' \circ \varepsilon_1)(\Lambda Y.t_1) : \forall Y.\rho(G_1[G'/X]) \). By unfolding, we have to prove that

\[
\forall W' \vdash \langle \forall Y.E''_1[\alpha^{E_i}/X], \forall Y.E''_1[E_i'/X] \rangle(W' \cdot \Xi_1 \triangleright v''_1[G'_1] ) \implies \langle \forall Y.E''_1[\alpha^{E_i}/X], \forall Y.E''_1[E_i'/X] \rangle(W' \cdot \Xi_1 \triangleright v''_1[G'_1] ) \implies \langle \forall Y.E''_1[\alpha^{E_i}/X], \forall Y.E''_1[E_i'/X] \rangle(W' \cdot \Xi_1 \triangleright v''_1[G'_1] )
\]

By inspection of the reduction rules we know that

\[
t''_1 = ((E_{i1}[\beta^{E_i}/Y], E_{i2}[\beta^{E_i}/Y]) \circ (E''_1[\alpha^{E_i}/X][\beta^{E_i}/Y], E''_1[E_i'/X][\beta^{E_i}/Y])t'_1[\beta^{E_i}/Y] : \rho(G_1[G'/X][\beta/Y])
\]

By the reduction rule of the type application we know that:

\[
W'.\Xi_1 \triangleright v_1[G'_1] \implies W'.\Xi_1, \beta := G'_1 \triangleright (E''_1[\alpha^{E_i}/X][\beta^{E_i}/Y], E''_1[E_i'/X][\beta^{E_i}/Y])t'_1[\beta^{E_i}/Y] : \rho(G_1[G'/X][\beta/Y])
\]

where \( E'_i = \text{lift}_{W'.\Xi_1}(G'_i) \), and \( t'_1 = ((E_{i1}[\beta^{E_i}/Y], E_{i2}[\beta^{E_i}/Y])t_1[\beta^{E_i}/Y] : \rho(G_1[G'/X][\beta/Y])) \).

Now we instantiate

\[
(W, v_1, v_2) \in \psi_\rho[\forall Y.G_1]
\]

with \( W' \), \( G'_1 \), \( G'_2 \), \( R \), \( t'_1 \), \( t'_2 \), \( \beta \), and evidences \( \langle E_{i1}[\beta^{E_i}/Y], E_{i2}[E_i'/Y] \rangle \), to obtain that

\[
(W^*, t'_1, t'_2) \in \tau_{\rho[\forall Y \cdot \beta]}[G_1]
\]

where \( W^* = \downarrow (W' \boxdot (\beta, G'_1, G'_2, R)) \) then either both executions reduce to an error (then the result holds immediately), or \( \exists W'' \geq W^*, v_{fi} \), such that \( (W'', v_{fi1}, v_{fi2}) \in \psi_\rho[\forall Y \cdot \beta][G_1] \)

Suppose that \( v_{fi} = (\varepsilon_{fi} \circ (E_{i1}[\beta^{E_i}/Y], E_{i2}[\beta^{E_i}/Y]))u_{fi} : \rho'(G_1[\beta/Y]) \). As \( E_{i2}[\beta^{E_i}/Y] \equiv E_{i2}[\beta^{E_i}/Y] \), then \( \text{unlift}(E_{i2}[\beta^{E_i}/Y]) = \text{unlift}(E_{i2}[\beta^{E_i}/Y]) \). Then we use induction hypothesis using \( \rho'[\beta/Y] \), evidences \( \langle E''_1[E_1]/Y \rangle, E''_1[E_1]/Y \rangle \), where

\[
\beta(lift_{W'' \cdot \Xi_1}(G_1[\beta/Y]), lift_{W'' \cdot \Xi_1}(G_1[\beta/Y])) = \langle E''_1[E_1]/Y \rangle, E''_1[E_1]/Y \rangle
\]

also we know that:

\[
\langle E''_1[E_1]/Y[\alpha^{E_i}/X], E''_1[E_1]/Y[E_1'/X] = \langle E''_1[\alpha^{E_i}/X][E_1'/Y], E''_1[E_1'/X][E_1'/Y] \rangle
\]

401
Note that $\rho(G_1[\beta/Y]) = \rho[Y \mapsto \beta](G_1)$. Then we know that

\[
\rho(\downarrow W'' \circ (\epsilon_{f1} \circ (E_{11} [\beta^{E_1'}/Y], E_{12} [\beta^{E_2'/Y}]))) \circ \rho(\downarrow W'' \circ ((E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}]))) \circ \rho(\downarrow W'' \circ ((E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}])).
\]

Then by inspection of the reduction rules:

\[
W^* \alpha \rho \Rightarrow \rho^* W^* \alpha \rho \Rightarrow ((\langle E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}] \rangle) \circ \rho(\downarrow W'' \circ (\langle E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}] \rangle)) \circ \rho(\downarrow W'' \circ ((\langle E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}]) \circ \rho(\downarrow W'' \circ ((\langle E_{11} [\beta^{E_1'/Y}], E_{12} [\beta^{E_2'/Y}]))).\]

and by Lemma \[302\] we know that those two values belong to the interpretation of $\rho(Y \mapsto \beta)[G_1[G'/X]]$, and the result holds.

Case \((G_1 \times G_2)\). Analogous to the function case.

Case \((B)\). Trivial.

Then we prove as (2):

Case \((G = X)\). Let $v_1 = \langle H_{11}, E_{12} \rangle u_1 :: X[G'/X] = \langle H_{11}, E_{12} \rangle u_1 :: G'$. Then we know that

\[
\rho(\downarrow W, (\langle H_{11}, E_{12} \rangle u_1 :: G', (H_{21}, E_{22}) u_2 :: G')) \in V_\rho[G'].
\]

and $\epsilon^{-1} = \alpha^{E_1'}$. Then we have to prove that

\[
(W, (\langle H_{11}, E_{12} \rangle \circ \langle E_1', \alpha^{E_1'} \rangle) u_1 :: \alpha, (\langle H_{21}, E_{22} \rangle \circ \langle E_2', \alpha^{E_2} \rangle) u_2 :: \alpha) \in V_\rho[X \mapsto \alpha][\alpha]
\]

By Lemma \[299\] we assume that both combinations of evidence are defined (otherwise the result holds immediately) Then by definition of transitivity $(\langle H_{11}, E_{12} \rangle \circ \langle E_1', \alpha^{E_1'} \rangle) = \langle H_{11}, \alpha^{E_{12}} \rangle$. Then we have to prove that

\[
(W, \langle H_{11}, \alpha^{E_{12}} \rangle u_1 :: \alpha, (H_{21}, \alpha^{E_{22}}) u_2 :: \alpha) \in V_\rho[X \mapsto \alpha][\alpha]
\]

but as $\alpha$ is sync, then that is equivalent to

\[
(W, \langle H_{11}, E_{12} \rangle u_1 :: G', (H_{21}, E_{22}) u_2 :: G') \in V_\rho[G']
\]

which is part of the premise by Lemma \[303\] and the result holds.

Case \((G = Y)\). Let $v_1 = \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \rho(Y[G'/X]) = \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \beta$ (where $\rho(Y) = \beta$). Then we know that

\[
(W, \langle H_{11}, \beta^{E_{12}} \rangle u_1 :: \beta, (H_{21}, \beta^{E_{22}}) u_2 :: \beta) \in V_\rho[\beta]
\]

402
Note that $\varepsilon_i^{-1} \vdash W.\Xi_i \vdash \beta \sim \beta$, and $\varepsilon_i$ has to have the form $\varepsilon_i = \langle \beta^{E_i'}, \beta^{E_i'} \rangle = \mathcal{J}(\text{lift}_{W,\Xi_i}(\beta), \text{lift}_{W,\Xi_i}(\beta))$. As $\varepsilon_i$ is the initial evidence for $\beta$, then $E_{i2} \subseteq E_{i1}$, and therefore by definition of the transtivity:

$$\langle H_{i1}, \beta^{E_{i2}} \rangle \circ \langle \beta^{E_i'}, \beta^{E_i'} \rangle = \langle H_{i1}, \beta^{E_{i2}} \rangle$$

Then we have to prove that:

$$(\downarrow W, (\langle H_{i1}, \beta^{E_{i2}} \rangle \circ \langle \beta^{E_i'}, \beta^{E_i'} \rangle)u_1 :: \beta, (\langle H_{21}, \beta^{E_{22}} \rangle \circ \langle E_2', \beta^{E_2'} \rangle)u_2 :: \beta) \in V_\rho[X \rightarrow \alpha][\beta]$$

or what is the same

$$(\downarrow W, \langle H_{i1}, \beta^{E_{i2}} \rangle u_1 :: \beta, (H_{21}, \beta^{E_{22}})u_2 :: \beta) \in V_\rho[\beta]$$

which is part of the premise and the result holds.

**Case ($G = ?$).** Let $v_i = \langle H_{i1}, E_{i2} \rangle u_i :: ?$. Then by definition of $V_\rho[?]$, let $G'' = \text{const}(E_{i2})$ (where $G'' \neq ?$). Then we know

$$(W, v_1, v_2) \in V_\rho[G][G'/X] \rightarrow G_2[\beta]$$

Then we have to prove that

$$(\downarrow W, (\varepsilon_1' \circ \varepsilon_1^{-1})(\lambda x^{G_1'}, t_1) :: \rho'(G_1) \rightarrow \rho'(G_2))$$

Let us call $v''_i = (\varepsilon_1' \circ \varepsilon_1^{-1})(\lambda x^{G_1'}, t_i) :: \rho'(G_1) \rightarrow \rho'(G_2)$. By unfolding, we have to prove that

$$\forall W'. \forall v'_1, v'_2. (W', v'_1, v'_2) \in V_\rho[G_1] \Rightarrow (W', v''_1, v''_2) \in V_\rho[G_2]$$

Suppose that $v'_1 = \varepsilon''_1 u'_1 :: \rho'(G_1)$, by inspection of the reduction rules, we know that

$$W', \Xi \vdash v''_1 \leftarrow W', \Xi \vdash (\text{cod}(\varepsilon') \circ \text{cod}(\varepsilon_1^{-1}))t_i[\varepsilon''_1 \circ (\text{dom}(\varepsilon_1^{-1}) \circ \text{dom}(\varepsilon'_1))]u'_1 :: G'/x] :: \rho'(G_2)$$

This is equivalent by Lemma 302

$$W', \Xi \vdash v''_1 \leftarrow W', \Xi \vdash (\text{cod}(\varepsilon') \circ \text{cod}(\varepsilon_1^{-1}))t_i[\varepsilon''_1 \circ \text{dom}(\varepsilon_1^{-1}) \circ \text{dom}(\varepsilon'_1)]u'_1 :: G'/x] :: \rho'(G_2)$$

Notice that $\text{dom}(\varepsilon_1^{-1}) \vdash W.\Xi_i \vdash \rho(G_1[\alpha/X]) \sim \rho(G_1[G'/X])$, and as $\text{dom}(\varepsilon_1^{-1})$ is constructed using the interior (and thus $\pi_2(\varepsilon''_1) \subseteq \pi_1(\text{dom}(\varepsilon_1^{-1}))$, then by definition of evidence $(\varepsilon''_1 \circ \text{dom}(\varepsilon_1^{-1}))$ is always defined. We can use induction hypothesis on $v'_1$, with evidences $\text{dom}(\varepsilon_1^{-1})$. Then we know that

$$(W', (\varepsilon''_1 \circ \text{dom}(\varepsilon_1^{-1}))u'_1 :: \rho(G_1[G'/X]), (\varepsilon''_2 \circ \text{dom}(\varepsilon_2^{-1}))u''_2 :: \rho(G_1[G'/X])) \in V_\rho[G_1[G'/X]]$$

403
Let us call $v''_i = (\varepsilon''_i \circ \text{dom}(\varepsilon^{-1}_i))u'_i :: \rho(G'_1[X])$.

Now we instantiate

$$(W, v_1, v_2) \in \mathcal{V}_{\rho}[G'_1[X] \rightarrow G'_2[X]]$$

with $W'$ and $v''_i$, to obtain that either both executions reduce to an error (then the result holds immediately), or $\exists W'' \geq W'$ such that $(W'', v_{f1}, v_{f2}) \in \mathcal{V}_{\rho}[G'_2[X]]$

$$W'.\Xi_i \triangleright v_i v''_i \mapsto W''.\Xi_i \triangleright \text{cod}(\varepsilon''_i) t_i[(\varepsilon''_i \circ \text{dom}(\varepsilon^{-1}_i)) \circ \text{dom}(\varepsilon'_i)]u'_i :: \rho(G'_2[X])$$

Suppose that $v_{f1} = \varepsilon_{f1}u_{f1} :: \rho(G'_2[X])$. Then we use induction hypothesis once again using evidences $\text{cod}(\varepsilon^{-1}_i)$ over $v_{f1}$ (noticing that the combination of evidence does not fail as the evidence is obtained via the interior function i.e. the less precise evidence possible), to obtain that,

$$(\downarrow W'', (\varepsilon_{f1} \circ \text{cod}(\varepsilon^{-1}_i))u_{f1} :: \rho(G'_2), (\varepsilon_{f2} \circ \text{cod}(\varepsilon^{-1}_i))u_{f2} :: \rho'(G'_2)) \in \mathcal{V}_{\rho}[G_2]$$

and the result holds.

The remaining cases are similar.

$\Box$

**Definition 133.** $\rho \vdash \varepsilon_1 \equiv \varepsilon_2$ if \text{unlift}(\pi_2(\varepsilon_1)) = \text{unlift}(\pi_2(\varepsilon_2))$

**Proposition 301.** If

- $(W, v_1, v_2) \in \mathcal{V}_{\rho}[G]$
- $\varepsilon \models \Xi; \Delta \vdash G \sim G'$
- $W \in \mathcal{S}[\Xi]$ and $(W, \rho) \in \mathcal{D}[\Delta]$
- $\forall \alpha \in \text{dom}(\Xi).\text{sync}(\alpha, W)$

then:

$$(W, \rho_1(\varepsilon)v_1 :: \rho(G'), \rho_2(\varepsilon)v_2 :: \rho(G')) \in \mathcal{J}_{\rho}[G']$$

where $\text{sync}(\alpha, W) \iff W.\Xi_1(\alpha) = W.\Xi_2(\alpha) \land W.\kappa(\alpha) = [V_0[W.\Xi_1(\alpha)] \mid W.j].$

**Proof.** We proceed by induction on $G$. We know that $u_i \in G_i$ for some $G_i$, notice that $G_1 \in \text{HEADTYPE} \cup \text{TYPEVAR}$. In every case we apply Lemma 310 to show that $(\varepsilon_1 \circ \varepsilon''_i) \iff (\varepsilon_2 \circ \varepsilon''_i)$, so in all cases we assume that the transitivity does not fail (otherwise the proof holds immediately). Let us call $\varepsilon''_i = \rho_1(\varepsilon)$ and $\varepsilon''_i = \rho_2(\varepsilon)$.

**Case** $(G = B \land G' = B)$. We know that $v_i$ has the form $(B, B)u :: B$, and we know that $(W, (B, B)u :: B, (B, B)u :: B) \in \mathcal{V}_{\rho}[B]$. Also as $\varepsilon \models \Xi; \Delta \vdash B \sim B$, then $\varepsilon = (B, B)$, then as $\rho_1(B) = B$, $\varepsilon_1 \circ \rho_1(\varepsilon) = \varepsilon_1$, and we have to prove that $(W, (B, B)u :: B, (B, B)u :: B) \in \mathcal{V}_{\rho}[B]$, which is part of the premise and the result holds.

404
Case \( G = G''_1 \to G''_2 \), and \( G' = G'_1 \to G'_2 \). We know that:
\[
(W', v_1, v_2) \in \mathcal{V}_p[G''_1 \to G''_2]
\]
Where \( v_i = \varepsilon_i(\lambda x^{G_{i1}}.t_i) :: G''_1 \to G''_2 \) and \( \varepsilon_i \vdash W.\Xi_i \vdash G_i \sim G''_1 \to G''_2 \).

We have to proof that:
\[
(\downarrow W, \varepsilon''_1 v_1 :: G'_1 \to G'_2, \varepsilon''_2 v_2 :: G'_1 \to G'_2) \in \mathcal{I}_p[G'_1 \to G'_2]
\]

First we suppose that \( (\varepsilon_i \circ \varepsilon''_i) \) does not fail, then we have to proof that:

\[
\forall W' \vdash \downarrow W.\forall v'_1, v'_2, (W', v'_1, v'_2) \in \mathcal{V}_p[G'_1] \Rightarrow
(W', [(\varepsilon_i \circ \varepsilon''_i)(\lambda x^{G_{i1}}.t_i) :: G'_1 \to G'_2] v'_1, [(\varepsilon_2 \circ \varepsilon''_2)(\lambda x^{G_{i2}}.t_2) :: G'_1 \to G'_2] v'_2) \in \mathcal{I}_p[G'_2]
\]

where \( v_i' = \varepsilon_i' u_i' :: G'_1 \). Note that by the reduction rule of application terms, we obtain that:

\[
W'.\Xi_1 \triangleright ((\varepsilon_i \circ \varepsilon''_i)(\lambda x^{G_{i1}}.t_i) :: G'_1 \to G'_2) (\varepsilon_i' u_i' :: G'_1) \rightarrow
W'.\Xi_1 \triangleright \text{cod}(\varepsilon_i \circ \varepsilon''_i)(((\varepsilon_i \circ \text{dom}(\varepsilon_i \circ \varepsilon''_i))u_i' :: G_{i1})/(\lambda x^{G_{i1}}.t_i)) :: G'_2
\]

We know by the Proposition 304 that \( \text{dom}(\varepsilon_i \circ \varepsilon''_i) = \text{dom}(\varepsilon''_i) \circ \text{dom}(\varepsilon_i) \). Then by the Proposition 302 we know that:

\[
\varepsilon_i' \circ (\text{dom}(\varepsilon_i \circ \varepsilon''_i)) = \varepsilon_i' \circ (\text{dom}(\varepsilon''_i) \circ \text{dom}(\varepsilon_i)) = (\varepsilon_i' \circ \text{dom}(\varepsilon''_i)) \circ \text{dom}(\varepsilon_i)
\]

Also, by the Proposition 305 it is follows that: \( \text{cod}(\varepsilon_i \circ \varepsilon''_i) = \text{cod}(\varepsilon_i) \circ \text{cod}(\varepsilon''_i) \).

Then the following result is true:

\[
W'.\Xi_1 \triangleright \text{cod}(\varepsilon_i \circ \varepsilon''_i)(((\varepsilon_i' \circ \text{dom}(\varepsilon_i \circ \varepsilon''_i))u_i' :: G_{i1})/(\lambda x^{G_{i1}}.t_i)) :: G'_2 =
W'.\Xi_1 \triangleright \text{cod}((\varepsilon_i \circ \varepsilon''_i)(((\varepsilon_i' \circ \text{dom}(\varepsilon_i))u_i' :: G_{i1})/(\lambda x^{G_{i1}}.t_i)) :: G'_2
\]

We instantiate the induction hypothesis in \((W', v'_1, v'_2) \in \mathcal{V}_p[G'_1] \) with the type \( G''_1 \) and the evidences \( \varepsilon_i \vdash \Xi_i \Delta \vdash G'_1 \sim G''_1 \). We obtain that:

\[
(W', \text{dom}(\varepsilon''_i)v'_i :: G''_1, \text{dom}(\varepsilon''_2)v'_2 :: G''_1) \in \mathcal{I}_p[G''_1]
\]

In particular we focus on a pair of values such that \( (\varepsilon_i' \circ \text{dom}(\varepsilon''_i)) \) does not fail (otherwise the result follows immediately). Then it is true that:

\[
(W', (\varepsilon_i' \circ \text{dom}(\varepsilon_i))u'_i :: G''_1, (\varepsilon_2' \circ \text{dom}(\varepsilon''_i))u'_2 :: G''_1) \in \mathcal{V}_p[G''_1]
\]

By the definition of \( \mathcal{V}_p[G''_1 \to G''_2] \) we know that:

\[
\forall W'' \vdash \downarrow W.\forall v''_1, v''_2, (W'', v''_1, v''_2) \in \mathcal{V}_p[G''_1] \Rightarrow (W'', v_1 v''_1, v_2 v''_2) \in \mathcal{I}_p[G''_2]
\]
We instantiate $v_i'' = (\varepsilon_i' \circ \text{dom}(\varepsilon_i'))u_i'' :: G_i''$ and $W'' = W'$, then we obtain that:

$$(W', ((\varepsilon_1(\lambda x^{G_1^1}.t_1)) :: G_1'' \rightarrow G_2'')) (((\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))u_i' :: G_i'') ,$$

$$(\varepsilon_2(\lambda x^{G_1^2}.t_2) :: G_1'' \rightarrow G_2'') (((\varepsilon_2' \circ \text{dom}(\varepsilon_2'))u_i' :: G_i'')) \in J_\rho[G_2'']$$

Then by Lemma 302, as $(\varepsilon_1' \circ \text{dom}(\varepsilon_1')) \circ \text{dom}(\varepsilon_1) = \varepsilon_1' \circ (\text{dom}(\varepsilon_1') \circ \text{dom}(\varepsilon_1))$, then if $(\text{dom}(\varepsilon_1') \circ \text{dom}(\varepsilon_1))$ is not defined and $(\text{dom}(\varepsilon_2') \circ \text{dom}(\varepsilon_2))$ is defined, we get a contradiction as both must behave uniformly as the terms belong to $J_\rho[G_2'']$. Then if both combination of evidence fail, then the result follows immediately. Let us suppose that the combination does not fail, then

$$W.'\Xi_1 \triangleright (\varepsilon_1(\lambda x^{G_1^1}.t_1)) :: G_1'' \rightarrow G_2'') (((\varepsilon'_1 \circ \text{dom}(\varepsilon'_1))u_i' :: G_i'') \rightarrow$$

$$W.'\Xi_1 \triangleright \text{cod}(\varepsilon_1)((((\varepsilon'_1 \circ \text{dom}(\varepsilon'_1)) \circ \text{dom}(\varepsilon_1))u_i' :: G_1)/x^{G_1^1}t_1) :: G_i'' \rightarrow W.''.\Xi_1 \triangleright v_{if}$$

The resulting terms reduce to values $(v_{if})$ and $\exists W'' \geq W''$ such that $(W'', v_{1f}, v_{2f}) \in V_\rho[G_2''].$

$$W.'\Xi_1 \triangleright \text{cod}(\varepsilon_1)((((\varepsilon'_1 \circ \text{dom}(\varepsilon'_1)) \circ \text{dom}(\varepsilon_1))u_i' :: G_1)/x^{G_1^1}t_1) :: G_i'' \rightarrow W.''.\Xi_1 \triangleright v_{if}$$

We instantiate the induction hypothesis in the previous result with the type $G_2'$ and the evidence $\text{cod}(\varepsilon) \vdash \Xi; \Delta \vdash G'' \sim G_2'$, then we obtain that:

$$(W'', \text{cod}(\varepsilon'')v_{1f} :: G_2', \text{cod}(\varepsilon_2')v_{2f} :: G_2')' \in J_\rho[G_2']$$

Then $v_{if}$ has to have the form: $v_{if} = (\varepsilon_1'' \circ \text{cod}(\varepsilon_1))u_i :: G_i''$ form some $\varepsilon_i'', u_{if}$. Then as $(\varepsilon'' \circ \text{cod}(\varepsilon_1)) \circ \text{cod}(\varepsilon_1') = \varepsilon'' \circ (\text{cod}(\varepsilon_1) \circ \text{cod}(\varepsilon_1'))$, then $(\text{cod}(\varepsilon_1) \circ \text{cod}(\varepsilon_1'))$ must behave uniformly (either the two of them fail, or the two of them does not fail), and the result immediately.

**Case ($G = \forall X.G_1''$ and $G' = \forall X.G_1'$).** We know that:

$$(W, v_1, v_2) \in V_\rho[\forall X.G_1'']$$

Where $v_1 = \varepsilon_1(\lambda x.t_1) :: \forall X.\rho(G_1'')$ and $\varepsilon_1 \vdash W.\Xi_1 \vdash \Delta \sim \forall X.\rho(G_1'').$

We have to proof that:

$$(\downarrow W, \varepsilon_1'v_1 :: \forall X.\rho(G_1'), \varepsilon_2'v_2 :: \forall X.\rho(G_1')) \in J_\rho[\forall X.G_1']$$

As $(\varepsilon_1 \circ \varepsilon_1')$ does not fail, then by the definition of $J_\rho[\forall X.G_1']$ we have to proof that:

$$(W, (\varepsilon_1 \circ \varepsilon_1')(\lambda X.t_1) :: \forall X.\rho(G_1'), (\varepsilon_2 \circ \varepsilon_2')(\lambda X.t_2) :: \forall X.\rho(G_1')) \in V_\rho[\forall X.G_1']$$
or what is the same:
\[
\forall W'' \succeq W_1 \forall t'_1, t'_2, G_1^*, G_2^*, \alpha, \varepsilon_{11}, \varepsilon_{21}, \forall R \in \text{Rel}_{W''}^R [G_1^*, G_2^*].
\]

\[
(W'' \cdot \Xi_1 \vdash G_1^* \land W'' \cdot \Xi_2 \vdash G_2^* \land \]
\[
W'' \cdot \Xi_1 \vdash ((\varepsilon_1 \circ \varepsilon_1') t_1 :: \forall X. G_1^*) [G_1^*] \rightarrow W'' \cdot \Xi_1, \alpha := G_1^* \triangleright \varepsilon_{11} t'_1 :: G_1^*[G_1^*/X] \land \]
\[
W'' \cdot \Xi_2 \vdash ((\varepsilon_2 \circ \varepsilon_2') t_2 :: \forall X. G_1^*) [G_2^*] \rightarrow W'' \cdot \Xi_2, \alpha := G_2^* \triangleright \varepsilon_{21} t'_2 :: G_1^*[G_2^*/X]) \Rightarrow \]
\[
(\downarrow (W'' \cdot (\alpha, G_1^*, G_2^*, R)), t'_1, t'_2) \in \mathcal{T}_{\rho[X \rightarrow o]} [G_1^*]
\]

For simplicity, let us call \( W'' = \downarrow (W'' \cdot (\alpha, G_1^*, G_2^*, R)) \). Note that by the reduction rule of type application, we obtain that:
\[
W'' \cdot \Xi_1 \vdash ((\varepsilon_1 \circ \varepsilon_1') \Lambda X.t_1 :: \forall X. \rho(G_1^*)) [G_1^*] \rightarrow \]
\[
W'' \cdot \Xi_1, \alpha := G_1^* \triangleright \varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X] :: \rho(G_1^*)[\alpha/X] :: \rho(G_1^*[G_1^*/X]
\]

where \( E_1 = \text{lift}_W(\Xi_1) (G_1^*) \). The resulting evidences \( \varepsilon_1 \circ \varepsilon_1' \) have the form: \( \forall X. E_{i1}, \forall X. E_{i2} \), then:
\[
\varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X] :: \rho(G_1^*)[\alpha/X] :: \rho(G_1^*[G_1^*/X] = \]
\[
\varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X] :: \rho(G_1^*)[\alpha/X] \]

Then we have to prove that:

\[
(W'', (\varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X], E_{i1}[\alpha_{E_1}/X]) t_1[\alpha_{E_1}/X] :: \rho(G_1^*)[\alpha/X]), \]
\[
((E_{i2}[\alpha_{E_2}/X], E_{i2}[\alpha_{E_2}/X]) t_2[\alpha_{E_2}/X] :: \rho(G_1^*)[\alpha/X])) \in \mathcal{T}_{\rho[X \rightarrow o]} [G_1^*]
\]

Also by the Proposition [300] we know that:
\[
(\varepsilon_1 \circ \varepsilon_1')[\alpha_{E_1}] = (\varepsilon_1[\alpha_{E_1}]) \circ (\varepsilon_1'[\alpha_{E_1}])
\]

Note that:
\[
(\varepsilon_1 \circ \varepsilon_1')[\alpha_{E_1}] = (\varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X], E_{i1}[\alpha_{E_1}/X]) = (\varepsilon_1[\alpha_{E_1}]) \circ (\varepsilon_1'[\alpha_{E_1}])
\]

Then we have to prove that:

\[
(W'', (\varepsilon_1[\alpha_{E_1}] \circ \varepsilon_1'[\alpha_{E_1}]) t_1[\alpha_{E_1}/X] :: G_1^*[\alpha/X]), (\varepsilon_2[\alpha_{E_2}] \circ \varepsilon_2'[\alpha_{E_2}]) t_2[\alpha_{E_2}/X] :: \rho(G_1^*)[\alpha/X]) \]
\[
\in \mathcal{T}_{\rho[X \rightarrow o]} [G_1^*]
\]

Note that by the reduction rule of type application, we obtain that:
\[
W'' \cdot \Xi_1 \triangleright (\varepsilon_1\Lambda X.t_1 :: \forall X. \rho(G_1'')) [G_1^*] \rightarrow \]
\[
W'' \cdot \Xi_1, \alpha := G_1^* \triangleright \varepsilon_{E_1/\alpha} t_1[\alpha_{E_1}/X] :: \rho(G_1''[\alpha/X]) :: \rho(G_1''[G_1^*/X]
\]

407
Note that the evidence $\varepsilon_1$ has the form: $\langle \forall X. E''_1, \forall X. E''_{12} \rangle$, then:

$$\varepsilon_{E_1/\alpha_1}^{E''_1/\alpha_1} (\varepsilon_1[\alpha_1] t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X]) : \rho(G''_2[\alpha/X]) =$$

$$\varepsilon^{E''_1/\alpha_1} (E''_1[\alpha_1/E_1][\alpha/X], E''_{12}[\alpha_1/E_1][\alpha/X]) t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X])$$

As we know that $(W, v_1, v_2) \in \nu_\rho \forall X. G''_1$, then we can instantiate with $\forall W'' \geq W$, $G''_1$, $G''_2$, $R$, $\varepsilon_1[\alpha_1] t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X])$, $\varepsilon_2[\alpha_2/E_2] t_2[\alpha_2/E_2][\alpha/X] : \rho(G''_2[\alpha/X])$, $\varepsilon^{E_1/\alpha_1}$ and $\varepsilon^{E_2/\alpha_2}$.

Then we know that:

$$(W'', \varepsilon_1[\alpha_1] t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X]), \varepsilon_2[\alpha_2/E_2] t_2[\alpha_2/E_2][\alpha/X] : \rho(G''_2[\alpha/X])) \in \nu_\rho [\mu X. \epsilon_1 \mapsto G''_1]$$

If the following term reduces to error, then the result follows immediately.

$$W''. \Xi_1 \triangleright \varepsilon_1[\alpha_1] t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X])$$

If the above is not true, then the following terms reduce to values $(v_1f)$ and $\exists W'' \geq W''$ such that $(W'', v_1f, v_2f) \in \nu_\rho [\mu X. \epsilon_1 \mapsto G''_1]$.

$$W''. \Xi_1 \triangleright \varepsilon_1[\alpha_1] t_1[\alpha_1/E_1][\alpha/X] : \rho(G''_1[\alpha/X]) \rightarrow^* W'''. \Xi_1 \triangleright v_1f$$

By definition of consistency and the evidence we know that $\varepsilon [X] \vdash W''. \Xi \vdash G''_1 \sim G''_1$. Then we instantiate the induction hypothesis in the previous result with $G = G''_1$ and $\varepsilon = \varepsilon [X]$. Calling $\rho' = \rho [X \mapsto \alpha]$, then we obtain that:

$$(W''', \rho'_1(\varepsilon [X]) v_1f : \rho'(G'')_1, \rho'_2(\varepsilon [X]) v_2f : \rho'(G'')_2) \in \nu_\rho [G'']$$

but as $\rho'_1(\varepsilon [X]) = \varepsilon'_1[\alpha_1/E_1]$ which is equivalent to

$$(W''', (\varepsilon'_1[\alpha_1]) v_1f : \rho(G''_1[\alpha/X]), (\varepsilon'_2[\alpha_2]) v_2f : \rho(G''_2[\alpha/X]) \in \nu_\rho [G'']$$

and the result follows immediately.

Case $(G = G_1 \times G_2)$. Similar to function case.

Case $(A)(G = \alpha)$. This means that $\alpha \in dom(\Xi)$. We know that $(W, \varepsilon_1 u_1 : \alpha, \varepsilon_2 u_2 : \alpha) \in \nu_\rho [\alpha]$ and $\varepsilon_1 \vdash W. \Xi_1 \vdash G_1 \sim \alpha$, then $\varepsilon_1 = \langle E_1, \alpha_1^E \rangle$.

We proceed by doing case analyze on $\varepsilon$. 

408
(A.iii) \( (\varepsilon = \langle \alpha^?, \alpha^? \rangle) \) Then by definition of the transitivity operator, \( \varepsilon_1 \circ \varepsilon = \langle E_1'', \alpha^{E''} \rangle \) (where \( \langle E_1, E_1'' \rangle \circ (?, ?) = \langle E_1, E''_1 \rangle \)). Then we have to prove that
\[
(\downarrow W, \langle E_1, \alpha^{E_1} \rangle u_1 :: G', \langle E_2, \alpha^{E_2} \rangle u_2 :: G') \in V_\rho[G']
\]
where \( G' \) is either \( ? \) or \( \alpha \). In any case this is equivalent to prove that
\[
(\downarrow W, \langle E_1, \alpha^{E_1} \rangle u_1 :: \alpha, \langle E_2, \alpha^{E_2} \rangle u_2 :: \alpha) \in V_\rho[\alpha]
\]
which is part of the premise and the result holds.

(A.ii) \( (\varepsilon = \langle \alpha^?, ? \rangle) \) then \( G' = ? \), and \( W_\varepsilon \Xi(\alpha) = ? \).

Then by definition of the transitivity operator, \( \varepsilon_1 \circ \varepsilon = \langle E_1, E_1'' \rangle \) (where \( \langle E_1, E_1'' \rangle \circ (?, ?) = \langle E_1, E''_1 \rangle \)). Then we have to prove that
\[
(\downarrow W, \langle E_1, \alpha^{E_1} \rangle u_1 :: ?, \langle E_2, \alpha^{E_2} \rangle u_2 :: ?) \in V_\rho[?]
\]
But by definition of \( V_\rho[\alpha] \), the result holds immediately.

(A.iii) \( (\varepsilon = \langle \alpha^{E_3}, E_4 \rangle) \). Then \( \beta \in dom(\Xi) \), and for transitivity to be defined, \( \varepsilon_1 = \langle E_1, \alpha^{E_3, \alpha} \rangle \). Then suppose that \( \varepsilon_1^p = \langle \alpha^{E_3}, E_4 \rangle \), then by definition of transitivity
\[
\langle E_1, \alpha^{E_3, \alpha} \rangle \circ \langle \alpha^{E_3}, E_4 \rangle = \langle E_1, \beta^{E_3, \alpha} \rangle \circ \langle \beta^{E_3}, E_4 \rangle
\]
Also notice that by Lemmas 312 and 311 \( \langle \beta^{E_3}, E_4 \rangle \vdash \Xi; \Delta \vdash \beta \sim G' \) where \( \beta \) is sync, and by definition of the logical relation
\[
(W, \langle E_1, \beta^{E_3, \alpha} \rangle u_1 :: \beta, \langle E_2, \beta^{E_2, \alpha} \rangle u_2 :: \beta) \in V_\rho[\beta]
\]
so we proceed just like case \( (G = \alpha) \) one more time but with \( G = \beta \) and \( \varepsilon = \langle \beta^{E_3}, E_4 \rangle \).

(A.iv) \( (\varepsilon = \langle \alpha^{H_3}, E_4 \rangle) \). So for transitivity to be defined, \( \varepsilon_1 = \langle H_1, \alpha^{H_1} \rangle \). Then suppose that \( \varepsilon_1^p = \langle \alpha^{H_3}, E_4 \rangle \), then by definition of transitivity
\[
\langle H_1, \alpha^{H_1} \rangle \circ \langle \alpha^{H_3}, E_4 \rangle = \langle H_1, H_1' \rangle \circ \langle H_3, E_4 \rangle
\]
Also, as \( \alpha \) is sync then \( W \Xi_1(\alpha) = W \Xi_2(\alpha) \). Let us call \( G_\alpha = W \Xi_1(\alpha) \). Then by definition of the interpretation for type names
\[
(W, \langle H_1, H_1' \rangle u_1 :: G_\alpha, \langle H_2, H_2' \rangle u_2 :: G_\alpha) \in V_\rho[G_\alpha]
\]
where \( G_\alpha \notin \text{TypeName} \).

Also notice that as \( \langle \alpha^{H_3}, E_4 \rangle \vdash \Xi; \Delta \vdash G \sim G' \), where \( \alpha \subseteq G \), then by Lemma 311 \( \langle \alpha^{H_3}, E_4 \rangle \vdash \Xi; \Delta \vdash \alpha \sim G' \). Also by Lemma 312 \( \langle H_3, E_4 \rangle \vdash \Xi; \Delta \vdash G_\alpha \sim G' \). Then we proceed just like case \( (G \neq \alpha) \) where \( G = G_\alpha \) and \( \varepsilon = \langle H_3, E_4 \rangle \).
Case (B) \((G = X)\). Suppose that \(\rho(X) = \alpha\). We know that \(\alpha \not\in \Xi\), i.e. \(\alpha\) may not be in sync, that \((W, \varepsilon_1 u_1 :: \alpha, \varepsilon_2 u_2 :: \alpha) \in \mathcal{V}_{\rho}[X]\) and that \(\varepsilon_1 \vdash W \Xi_1 \vdash G_1 \sim \alpha\), then \(\varepsilon_1 = \langle E_i, \alpha^{E_i} \rangle\).

Then by construction of evidences, \(\varepsilon\) must be either \(\langle X, X \rangle\) or \(\langle ?, ? \rangle\)(any other case will fail when the meet is computed).

- \((\varepsilon = \langle X, X \rangle)\). Then \(\varepsilon'' = \langle \rho_i(X), \rho_i(X) \rangle\). But \(\rho_i(X)\) is the type that contains the initial precision for \(\alpha\). Therefore \(\alpha^{E_i} \subseteq \rho_i(X)\), and by Lemma \[316\] \(\varepsilon_1 \circ \varepsilon'' = \varepsilon_1\) and the result holds immediately (notice that if \(G' = ?\) then we have to show that they are related to \(\alpha\) which is part of the premise).

Case (C) \((G = ?)\). We know that \((W, \varepsilon_1 u_1 :: ?, \varepsilon_2 u_2 :: ?) \in \mathcal{V}_{\rho}[?]\) and \(\varepsilon_1 \vdash W \Xi_1 \vdash G_1 \sim ?\). We are going to proceed by case analysis on \(\pi_2(\varepsilon_1)\) and \(\rho \vdash \varepsilon_1 \equiv \varepsilon_2\):

(C.i) \((\varepsilon_1 = \langle E_i, \alpha^{E_i} \rangle)\). Then this means we know that

\[(W, \varepsilon_1 u_1 :: \alpha, \varepsilon_2 u_2 :: \alpha) \in \mathcal{V}_{\rho}[\alpha]\]

and \(\varepsilon_1 \vdash W \Xi_1 \vdash G_1 \sim \alpha\), then \(\varepsilon_1 = \langle E_i, \alpha^{E_i} \rangle\).

(a) \((\varepsilon = \langle \alpha^{E_i}, E_4 \rangle)\). Then as \(\langle E_i, \alpha^{E_i} \rangle \vdash \Xi; \Delta \vdash G_1 \sim ?\), then by Lemma \[311\] \(\langle E_i, \alpha^{E_i} \rangle \vdash \Xi; \Delta \vdash G_1 \sim \alpha\). Also we know that \(? \not\subseteq G\), then \(G = ?\), and \(\alpha \not\subseteq G\). Finally, we reduce this case to the Case A if \(\alpha \in \Xi\) or Case B if \(\alpha \not\in \Xi\).

(b) \((\varepsilon = \langle ?, ? \rangle)\). Then \(G' = ?\), and does \(\varepsilon_1 \circ \varepsilon = \varepsilon_1\). Then we have to prove that \((\downarrow W, \varepsilon_1 u_1 :: ?, \varepsilon_2 u_2 :: ?) \in \mathcal{V}_{\rho}[?]\), and as \(\text{const}(\alpha^{E_i}) = \alpha\) that is equivalent to prove that \((\downarrow W, \varepsilon_1 u_1 :: \alpha, \varepsilon_2 u_2 :: \alpha) \in \mathcal{V}_{\rho}[\alpha]\) which is part of the premise by Lemma \[303\] and the result holds immediately.

(c) \((\varepsilon = \langle ?, \beta^{\beta^{\ldots}} \rangle)\). Where \(\beta\) cannot transitively point to some unsync variable. Then by definition of the transitivity operator, \(\varepsilon_1 \circ \varepsilon = \langle E_i', \beta^{E_i'} \rangle\) (where \(\langle E_i, \alpha^{E_i} \rangle \circ \langle ?, \beta^{\beta^{\ldots}} \rangle = \langle E_i', E_i'' \rangle\)). Then we have to prove that

\[(\downarrow W, \langle E_i', \beta^{E_i'} \rangle u_1 :: G', \langle E_i', \beta^{E_i'} \rangle u_2 :: G' \in \mathcal{V}_{\rho}[G']\]

where \(G'\) is either \(?\) or \(\beta\). In any case this is equivalent to prove that

\[(\downarrow W, \langle E_i', \beta^{E_i'} \rangle u_1 :: \beta, \langle E_i', \beta^{E_i'} \rangle u_2 :: \beta \in \mathcal{V}_{\rho}[\beta] \iff (\downarrow W, \langle E_i', \beta^{E_i'} \rangle u_1 :: G'', \langle E_i', \beta^{E_i'} \rangle u_2 :: G'' \in \mathcal{V}_{\rho}[G''])\]

where \(G'' = W.\Xi_1(\beta) = W.\Xi_2(\beta)\) (note that \(\beta\) is sync). As \(\langle E_i, \alpha^{E_i} \rangle \circ \langle ?, \beta^{\beta^{\ldots}} \rangle = \langle E_i', E_i'' \rangle\), then we can reduce the demonstration to proof that:

\[(\downarrow W, \langle E_i, \alpha^{E_i} \rangle \circ \langle ?, \beta^{\beta^{\ldots}} \rangle) u_1 :: G'', \langle E_i, \alpha^{E_i} \rangle \circ \langle ?, \beta^{\beta^{\ldots}} \rangle) u_2 :: G'' \in \mathcal{V}_{\rho}[G'']\]

Finally, we reduce this case to this same case (note that we have base case because the sequence ends in \(?\)).

(d) \((\varepsilon = \langle ?, \beta^3 \rangle)\). Then by definition of the transitivity operator, \(\varepsilon_1 \circ \varepsilon = \langle E_i', \beta^{E_i'} \rangle\) (where \(\langle E_i, \alpha^{E_i} \rangle \circ \langle ?, ? \rangle = \langle E_i', E_i'' \rangle\)). Then we have to prove that

\[(\downarrow W, \langle E_i', \beta^{E_i'} \rangle u_1 :: G', \langle E_i', \beta^{E_i'} \rangle u_2 :: G' \in \mathcal{V}_{\rho}[G']\]
where $G'$ is either $? \, \text{or} \, \beta$. In any case this is equivalent to prove that
\[
\langle \downarrow W, \langle E''_1, \beta^E'' \rangle u_1 :: \beta, \langle E''_2, \beta^E'' \rangle u_2 :: \beta \rangle \in V_\rho[\beta] \iff \\
\langle \downarrow W, \langle E''_1, E''_2 \rangle u_1 :: G'', \langle E''_2, E''_2 \rangle u_2 :: G'' \rangle \in V_\rho[G'']
\]
where $G'' = W \cdot \Xi_1(\beta) = W \cdot \Xi_2(\beta) = ?$ (note that $\beta$ is sync). As $\langle E_1, \alpha^E \rangle \circ (?, ?) = \langle E_1, E'_1 \rangle$, then we can reduce the demonstration to prove that:
\[
\langle \downarrow W, \langle E_1, \alpha^E \rangle u_1 :: \alpha, \langle E_2, \alpha^E \rangle u_2 :: \alpha \rangle \in V_\rho[\alpha]
\]
which is part of the premise and the result holds.

(C.ii) $\langle \varepsilon_1 = \langle H_{11}, H_{12} \rangle \rangle$. Then as $G = ?$ and $G \sqsubseteq G'$, then $G = ?$. Let $G'' = \text{const}(H_{12})$, and we know that $G'' \in \text{HEADTYPE}$. By unfolding of the logical relation for $?$, we also know that
\[
\langle W, \langle H_{11}, H_{12} \rangle u_1 :: G'', \langle H_{21}, H_{22} \rangle u_2 :: G'' \rangle \in V_\rho[G'']
\]
and we have to prove that
\[
\langle \downarrow W, \langle (H_{11}, H_{12}) \circ \varepsilon'_1 \rangle u_1 :: G', \langle (H_{21}, H_{22}) \circ \varepsilon'_2 \rangle u_2 :: G' \rangle \in V_\rho[G']
\]
Note that for consistent transitivity to hold, then $\varepsilon$ has to take the following forms:

(a) $\varepsilon = \langle H_3, E_4 \rangle$. Then as $\varepsilon \vdash \Xi; \Delta \vdash ? \sim G'$, by Lemma 311, $\varepsilon \vdash \Xi; \Delta \vdash \text{const}(H_3) \sim G'$, and we proceed just like Case D, where $G \in \text{HEADTYPE} (G = G'')$.

(b) $\varepsilon = (?, ?)$. Then $G' = ?$ (let us assume without loosing generality that $H_{ij} = E_{i1} \rightarrow E_{i2}$, and thus $G'' = ? \rightarrow ? \langle H_{11}, H_{12} \rangle \circ (?, ?) = \langle H_{11}, H_{12} \rangle$. Then we have to prove that the resulting values are in the interpretation of $G'' = ? \rightarrow ?$, which we already know as premise and the result holds immediately.

(c) $\varepsilon = (?, \alpha ?)$. Then (let us assume without loosing generality that $H_{ij} = E_{i1} \rightarrow E_{i2}$, and thus $G'' = ? \rightarrow ? \langle W_1 \cdot \Xi_{(\alpha)} = ?, \text{and by inspection of the consistent transitivity rules, } \langle H_{11}, H_{12} \rangle \circ (?, \alpha ?) = \langle H_{11}, \alpha^{E_{i2}} \rangle$. Then by definition of the interpretation of $G'$ (which may be $? \, \text{or} \, \alpha$), we have to prove that
\[
\langle \downarrow W, \langle H_{11}, H_{12} \rangle u_1 :: ?, \langle H_{21}, H_{22} \rangle u_2 :: ? \rangle \in V_\rho[?]\]
\]
which is part of the premise, and the result holds.

(d) $\varepsilon = (?, \alpha^{E_{i1}})$. Then (let us assume without loosing generality that $H_{ij} = E_{i1} \rightarrow E_{i2}$, and thus $G'' = ? \rightarrow ? \langle W_1 \cdot \Xi_{(\alpha)} = \{ \beta, ? \}, \text{and by inspection of the consistent transitivity rules, } \langle H_{11}, H_{12} \rangle \circ (?, \alpha^{E_{i1}}) = \langle H_{11}^{''}, \alpha^{E_{i1}} \rangle$, where $\langle H_{11}, H_{12} \rangle \circ (?, E_{i1}) = \langle H_{i1}, E_{i2} \rangle$. Then by definition of the interpretation of $\alpha$ (after one or two unfolding of $G' = ?$), we have to prove that
\[
\langle \downarrow W, \langle (H_{11}, H_{12}) \circ (?, \beta^{E_{i1}}) \rangle u_1 :: \beta, \\
\langle (H_{21}, H_{22}) \circ (?, \beta^{E_{i1}}) \rangle u_2 :: \beta \rangle \in V_\rho[\beta]
\]
and then we proceed to the same case one more time (notice that the recursion is finite, until we get to the previous sub case).
Case (D) ($G$ ∈ HeadType). We know that $(W, \varepsilon_1 u_1 :: \rho(G), \varepsilon_2 u_2 :: \rho(G)) \in \mathcal{V}_\rho[G]$ and $\varepsilon_1 \vdash W.\Xi_1 \vdash G_1 \sim G$. Also $\varepsilon_1 = \langle H_{11}, H_{12} \rangle$, for some $H_{11}, H_{12}$. We proceed by case analysis on $G'$ and $\varepsilon$.

(D.i) ($\varepsilon = \langle E_3, \alpha^{E_3} \rangle$). Then $G' = \alpha$, or $G' = \?$. Notice that as $\alpha^{E_3}$ cannot have free type variables therefore $E_3$ neither. Then $\varepsilon = \rho_i(\varepsilon)$. As $\alpha$ is sync, then let us call $G'' = W.\Xi_i(\alpha)$. In either case $G' = \alpha$, or $G' = \?$, what we have to prove boils down to

$$(\downarrow W, (\varepsilon_1 \circ \langle E_3, \alpha^{E_3} \rangle) u_1 :: \alpha, (\varepsilon_2 \circ \langle E_3, \alpha^{E_3} \rangle) u_2 :: \alpha) \in \mathcal{V}_\rho[\alpha]$$

which, by definition of consistent transitivity, is equivalent to prove that

$$(\downarrow W, (\varepsilon_1 \circ \langle E_3, E_4 \rangle) u_1 :: G'', (\varepsilon_2 \circ \langle E_3, E_4 \rangle) u_2 :: G'') \in \mathcal{V}_\rho[G'']$$

Then we proceed by case analysis on $\varepsilon$:

- (Case $\varepsilon = \langle E_3, \alpha^{E_3} \rangle$). We know that $\alpha \subseteq G'$ and that $\langle E_3, \alpha^{E_3} \rangle \triangleright \Xi; \Delta \vdash G \sim G'$, then by Lemma \ref{lemma311} we know that $\langle E_3, \alpha^{E_3} \rangle \triangleright \Xi; \Delta \vdash G \sim \alpha$. Also by Lemma \ref{lemma313}, $\langle E_3, \beta^{E_3} \rangle \triangleright \Xi; \Delta \vdash G \sim \alpha$. As $\beta^{E_3} \subseteq G''$, then $G''$ can either be $\alpha$ or $\beta$.

If $G'' = \?$, then by definition of $\mathcal{V}_\rho[?]$, we have to prove that the resulting values belong to $\mathcal{V}_\rho[\beta]$. Also as $\langle E_3, \beta^{E_3} \rangle \triangleright \Xi; \Delta \vdash G \sim \?$, by Lemma \ref{lemma311}, $\langle E_3, \beta^{E_3} \rangle \triangleright \Xi; \Delta \vdash G \sim \beta$, and then we proceed just like this case once again (this is process is finite as there are no circular references by construction and it ends up in something different to a type name). If $G'' = \beta$ we use an analogous argument as for $G'' = \?$. 

- (Case $\varepsilon = \langle E_3, \alpha^{E_3}, E_4 \notin SITypeName \rangle)$. Then we have to prove that

$$(\downarrow W, (\varepsilon_1 \circ \langle E_3, E_4 \rangle) u_1 :: G'', (\varepsilon_2 \circ \langle E_3, E_4 \rangle) u_2 :: G'') \in \mathcal{V}_\rho[G'']$$

By Lemma \ref{lemma313}, $\langle E_3, E_4 \rangle \triangleright \Xi; \Delta \vdash G \sim G''$. Then if $G'' = \?$, we proceed as the case $G \in$ HeadType, $G' = ?$ with $\varepsilon = \langle E_3, E_4 \rangle$, where $E_3, E_4 \notin SITypeName \cup \{?\}$ (Case (D.ii)). If $G'' \in$ HeadType, we proceed as the case $G \in$ HeadType, $G' \in$ HeadType with $\varepsilon = \langle E_3, E_4 \rangle$, where $E_3, E_4 \in$ HeadType (Case (D.iii)).

(D.ii) ($G' = ?, \varepsilon = \langle H_3, H_4 \rangle$). We have to prove that

$$(\downarrow W, (\varepsilon_1 \circ \rho_1(\varepsilon)) u_1 :: ?, (\varepsilon_2 \circ \rho_2(\varepsilon)) u_2 :: ?) \in \mathcal{V}_\rho[?]$$

which is equivalent to prove that

$$(\downarrow W, (\varepsilon_1 \circ \rho_1(\varepsilon)) u_1 :: H, (\varepsilon_2 \circ \rho_2(\varepsilon)) u_2 :: H) \in \mathcal{V}_\rho[H]$$

for $H = const(H_{12})$ (and $H \in$ HeadType). But notice that as $\varepsilon \vdash \Xi; \Delta \vdash G \sim ?, \text{then as } H_4 \subseteq H \subseteq ?, \text{then by Lemma } \ref{lemma311}, \varepsilon \vdash \Xi; \Delta \vdash G \sim H$, then we proceed just like the case $G \in$ HeadType and $G' \in$ HeadType (Case (D.iii)).

(D.iii) ($G' \in$ HeadType). This cases are already analyzed, by structural analysis of types, e.g. (Case $G = G_1'' \rightarrow G_2''$ and $G' = G_1' \rightarrow G_2'$), (Case $G = \forall X. G_1''$ and $G'' = \forall X. G_1'$), etc.
Case $(G = B$ and $G' = B)$. We know that $v_i$ has the form $\langle B, B \rangle u :: B$, and we know that $(W, \langle B, B \rangle u :: B, \langle B, B \rangle u :: B) \in V_\rho[B]$. Also as $\varepsilon \vdash \Xi; \Delta \vdash B \sim B$, then $\varepsilon = \langle B, B \rangle$, then as $\rho_i(B) = B$, $\varepsilon_i \circ \rho_i(\varepsilon) = \varepsilon_i$, and we have to prove that $(W, \langle B, B \rangle u :: B, \langle B, B \rangle u :: B) \in V_\rho[B]$, which is part of the premise and the result holds.

Lemma 302 (Associativity of the evidence).

$$(\varepsilon_1 \circ \varepsilon_2) \circ \varepsilon_3 = \varepsilon_1 \circ (\varepsilon_2 \circ \varepsilon_3)$$


Lemma 303. If $(W, t_1, t_2) \in T_\rho[G]$, then $(\downarrow W, t_1, t_2) \in T_\rho[G]$

Proof. By induction on $G$.

Proposition 304. $dom(\varepsilon_1 \circ \varepsilon_2) = dom(\varepsilon_2) \circ dom(\varepsilon_1)$

Proof. Direct by inspection on the inductive definition of consistent transitivity.

Proposition 305. $cod(\varepsilon_1 \circ \varepsilon_2) = cod(\varepsilon_1) \circ cod(\varepsilon_2)$

Proof. Direct by inspection on the inductive definition of consistent transitivity.

Proposition 306. $(\varepsilon_1 \circ \varepsilon_2)[E] = \varepsilon_1[E] \circ \varepsilon_2[E]$.

Proof. Direct by inspection on the inductive definition of consistent transitivity.

Lemma 307. (Optimality of consistent transitivity).

If $\varepsilon_3 = \varepsilon_1 \circ \varepsilon_2$ is defined, then $\pi_1(\varepsilon_3) \sqsubseteq \pi_1(\varepsilon_1)$ and $\pi_2(\varepsilon_3) \sqsubseteq \pi_2(\varepsilon_2)$.

Proof. Direct by inspection on the inductive definition of consistent transitivity.

Lemma 308. If $\varepsilon \vdash \Xi; \Delta \vdash G_1 \sim G_2$, $W \in S[\Xi]$ and $(W, \rho) \in D[\Delta]$ then $\varepsilon_i^\rho \vdash W.\Xi; \Delta \vdash \rho(G_1) \sim \rho(G_2)$, where $\varepsilon_i^\rho = \rho_i(\varepsilon)$.

Proof. Direct by induction on the structure of the type.

Lemma 309. If $\Xi; \Delta; \Gamma \vdash t : G$, $W \in S[\Xi]$, $(W, \rho) \in D[\Delta]$ and $(W, \gamma) \in G_\rho[\Gamma]$ then $W.\Xi_i \vdash \rho(\gamma_i((t)) : \rho(G)$.
Proof. Direct by induction on the structure of the term.

Lemma 310. If

- $\varepsilon_1 \vdash W \Xi_i \vdash G_i \sim \rho(G), \varepsilon_1 \equiv \varepsilon_2$
- $\varepsilon \vdash \Xi; \Delta \vdash G \sim G', G \sqsubseteq G$
- $W \in S[\Xi], (W, \rho) \in D[\Delta]$
- $\forall \alpha \in \Xi. \alpha^{E_i} \in p_2(\varepsilon_i) \Rightarrow E_1^* \equiv E_2^*$

then $\varepsilon_1 \circ \rho_1(\varepsilon) \iff \varepsilon_2 \circ \rho_2(\varepsilon)$.

Proof. We proceed by induction on the judgment $\varepsilon_1 \vdash W \Xi_i \vdash G_i \sim G$.

Case ($\varepsilon_1 = \langle B_1, B_2 \rangle$). Then the result is trivial as by definition of $\varepsilon_1 \equiv \varepsilon_2$, $B_1 = B_2$, therefore $\varepsilon_1 = \varepsilon_2$. As $\varepsilon$ cannot have free type variables (otherwise the result holds immediately), proving that $\varepsilon_1 \circ \varepsilon \iff \varepsilon_1 \circ \varepsilon$ is trivial.

Case ($\varepsilon_1 = \langle ?, ? \rangle$). As the combination with $\langle ?, ? \rangle$ never produce runtime errors, the result follows immediately as both operation never fail.

Case ($\varepsilon_1 = \langle E_{11}, \alpha^{E_{12}} \rangle$). We branch on two sub cases:

- Case $\alpha \in \Xi$. Then $\varepsilon$ has to have the form $\langle \alpha^{E_3}, E_4 \rangle$, $\langle ?, ? \rangle$ or $\langle ?, \beta^{-1} \rangle$ (otherwise the transitivity operator will always fails in both branches). Also $E_4$ cannot be a type variable $X$ for instance, because $X$ is consistent with only $X$ or $?$, and in either case the evidence gives you $X$ on both sides of the evidence. And $\alpha$ cannot point to a type variable by construction (e.g, type $\alpha^X$ does not exists). Then $\varepsilon$ cannot have free type variables, therefore $\rho_1(\varepsilon) = \varepsilon$, and therefore we have to prove: $\varepsilon_1 \circ \varepsilon \iff \varepsilon_2 \circ \varepsilon$. For cases where $\varepsilon = \langle ?, ? \rangle$ or $\varepsilon = \langle ?, \beta^{-1} \rangle$, then as they never produce runtime errors, the result follows immediately as both operation never fail.

The interesting case is $\varepsilon = \langle \alpha^{E_3}, E_4 \rangle$. By definition of transitivity $\langle E_{11}, \alpha^{E_{21}} \rangle \circ \langle \alpha^{E_3}, E_4 \rangle = \langle E_{11}, E_{21} \rangle \circ \langle E_3, E_4 \rangle$. By Lemma 313 $\langle E_{11}, E_{21} \rangle \vdash W \Xi_i \vdash G_i \sim \Xi(\alpha)$ and $\langle E_3, E_4 \rangle \vdash W \Xi_i \vdash \Xi(\alpha) \sim G'$. Also we know by premise that $E_{21} \equiv E_{21}$, then by induction hypothesis $\langle E_{11}, E_{21} \rangle \circ \langle E_3, E_4 \rangle \iff \langle E_{12}, E_{22} \rangle \circ \langle E_3, E_4 \rangle$, and the result follows immediately.

- Case $\alpha \notin \Xi$. In this case $\varepsilon$ has to have the form $\langle X, X \rangle$ (where $\rho_1(\varepsilon) = \langle \text{lift}_{W \Xi_i}(\alpha), \text{lift}_{W \Xi_i}(\alpha) \rangle$, $\langle ?, ? \rangle$ or $\langle ?, \beta^{-1} \rangle$, (otherwise the transitivity always fail in both cases). For cases where $\varepsilon = \langle ?, ? \rangle$ or $\varepsilon = \langle ?, \beta^{-1} \rangle$, by the definition of transitivity, they never produce runtime errors, then the result follows immediately as both operation never fail.

If $\varepsilon = \langle X, X \rangle$, by construction of evidence, $\alpha^{E_{12}} \sqsubseteq \text{lift}_{W \Xi_i}(\alpha) \sqsubseteq ?, \theta$ then by Lemma 316 we know that $\varepsilon_1 \circ \rho_1(\varepsilon) = \varepsilon_1$, and the result holds.

Case ($\varepsilon_1 = \langle \alpha^{E_{11}}, E_{12} \rangle$). Then $\varepsilon$ has the form $\langle E_3, E_4 \rangle$, where $\rho_1(\varepsilon) = \langle E_{13}, E_{14} \rangle$. By the definition of transitivity we know that:

$\langle \alpha^{E_{11}}, E_{12} \rangle \circ \langle E_{13}, E_{14} \rangle \iff \langle E_{11}, E_{12} \rangle \circ \langle E_{13}, E_{14} \rangle$

414
Then by the induction hypothesis with:
\[ \langle E_{i1}, E_{i2} \rangle \models W \Xi_1 \vdash W \Xi_1(\alpha) \sim \rho(G) \]
\[ \varepsilon \models \Xi; \Delta \vdash G \sim G', G \subseteq G \]
we know that:
\[ \langle E_{i1}, E_{i2} \rangle \circ \langle E_{i3}, E_{i4} \rangle \iff \langle E_{i21}, E_{i22} \rangle \circ \langle E_{i23}, E_{i24} \rangle \]
Then the result follows immediately.
\[ Case \ (\varepsilon_i = \langle E_{i1i} \rightarrow E_{i1i}, E_{i2i} \rightarrow E_{i2i} \rangle). \] We analyze cases for \( \varepsilon \):

- Case \( \varepsilon = \langle ?, ?, \rangle \) or \( \varepsilon = \langle ?, \beta, \cdots \rangle \), then transitivity never fails as explained in previous cases.

- Case \( \varepsilon = \langle E_{i1} \rightarrow E_{i2}, E_{i4i} \rightarrow E_{i4} \rangle \). Then \( \rho_i(\varepsilon) = \langle E_{i3i} \rightarrow E_{i3i}, E_{i4i} \rightarrow E_{i4} \rangle \). By definition of interior and meet, the definition of transitivity for functions, can be rewritten like this:
\[ \langle E_{i1i}, E_{i2i} \rangle = \langle E_{i3i}, E_{i4i} \rangle \]
\[ \langle E_{i21}, E_{i22} \rangle = \langle E_{i23}, E_{i24} \rangle \]
\[ \langle E_{i1i} \rightarrow E_{i2i}, E_{i2i} \rightarrow E_{i22} \rangle = \langle E_{i3i} \rightarrow E_{i3i}, E_{i4i} \rightarrow E_{i4i} \rangle \]
Also notice as the definition of interior is symmetrical (as consistency is symmetric), \( \langle E_{i1i}, E_{i3i} \rangle \circ \langle E_{i2i}, E_{i1i} \rangle = \langle E_{i3i}, E_{i1i} \rangle \) can be computed as \( \langle E_{i1i}, E_{i2i} \rangle \circ \langle E_{i3i}, E_{i4i} \rangle = \langle E_{i3i}, E_{i1i} \rangle \). Also \( \varepsilon_1 \equiv \varepsilon_2 \) implies that \( dom(\varepsilon_1) \equiv dom(\varepsilon_2) \) and \( cod(\varepsilon_1) \equiv cod(\varepsilon_2) \). And that \( dom(\varepsilon) \models \Xi; \Delta \vdash dom(G') \sim dom(G) \) is equivalent to:
\[ \langle \pi_2(dom(\varepsilon)), \pi_1(dom(\varepsilon)) \rangle \models \Xi; \Delta \vdash dom(G) \sim dom(G') \]
where \( dom(G) \subseteq dom(G) \) and that \( cod(\varepsilon) \models \Xi; \Delta \vdash cod(G) \sim cod(G') \) (where \( cod(G) \subseteq cod(G) \)). The result holds by applying induction hypothesis on:
\[ \langle E_{i1i}, E_{i2i} \rangle \models \Xi; \Delta \vdash dom(G_i) \sim dom(\rho(G)) \]
\[ \langle \pi_2(dom(\varepsilon)), \pi_1(dom(\varepsilon)) \rangle \models \Xi; \Delta \vdash dom(G) \sim dom(G') \]
and
\[ \langle E_{i2i}, E_{i2i} \rangle \models \Xi; \Delta \vdash cod(G_i) \sim cod(\rho(G)) \]
\[ cod(\varepsilon) \models \Xi; \Delta \vdash cod(G) \sim cod(G') \]
- Case \( \varepsilon = \langle E_{31} \rightarrow E_{32}, \alpha^{E_{31} \rightarrow E_{32}} \rangle \). Then \( \rho_i(\varepsilon) = \langle E_{31i} \rightarrow E_{32i}, \alpha^{E_{31i} \rightarrow E_{32i}} \rangle \). We use a similar argument to the previous item noticing that
\[ \langle E_{i1i}, E_{i3i} \rangle \circ \langle E_{i2i}, E_{i1i} \rangle = \langle E_{i3i}, E_{i4i} \rangle \]
\[ \langle E_{i2i}, E_{i2i} \rangle = \langle E_{i23}, E_{i24} \rangle \]
\[ \langle E_{i1i} \rightarrow E_{i2i}, E_{i2i} \rightarrow E_{i22i} \rangle = \langle E_{i3i} \rightarrow E_{i3i}, E_{i4i} \rightarrow E_{i4i} \rangle \]
\[ \langle E_{i1i} \rightarrow E_{i1i}, E_{i2i} \rightarrow E_{i22i} \rangle \circ \langle E_{i1i} \rightarrow E_{i3i}, E_{i4i} \rightarrow E_{i4i} \rangle \]
\[ \langle E_{i1i} \rightarrow E_{i2i}, E_{i2i} \rightarrow E_{i22i} \rangle = \langle E_{i1i} \rightarrow E_{i3i}, E_{i4i} \rightarrow E_{i4i} \rangle \]
and that if \( G' = \alpha \) by Lemma 313
\[ \langle E_{31} \rightarrow E_{32}, E_{31i} \rightarrow E_{32i} \rangle \models \Xi; \Delta \vdash G \sim \Xi(\alpha) \]
\[ \langle E_{31} \rightarrow E_{32}, \alpha^{E_{31} \rightarrow E_{32}} \rangle \models \Xi; \Delta \vdash G \sim \alpha \]
and if $G' = ?$ by Lemma 313.

\[
\frac{\langle E_{31} \rightarrow E_{32}, E_{41} \rightarrow E_{42} \rangle \vdash \Xi; \Delta \vdash G \sim ?}{\langle E_{31} \rightarrow E_{32}, \alpha^{E_{41} \rightarrow E_{42}} \rangle \vdash \Xi; \Delta \vdash G \sim ?}
\]

Case ($\varepsilon_i = \langle \forall X. E_{i1}, \forall X. E_{i2} \rangle$).

We proceed similar to the function case using induction hypothesis on the subtypes.

Case ($\varepsilon_i = \langle E_{i1} \times E_{i2}, E_{3i} \times E_{4i} \rangle$).

We proceed similar to the function case using induction hypothesis on the subtypes.

**Lemma 311.** If $\langle E_1, E_2 \rangle \vdash \Xi; \Delta \vdash G_1 \sim G_2$, then

1. $\forall G_3, \text{toGType}(E_2) \subseteq G_3 \subseteq G_2, \langle E_1, E_2 \rangle \vdash \Xi; \Delta \vdash G_1 \sim G_3$, and
2. $\forall G_3, \text{toGType}(E_1) \subseteq G_3 \subseteq G_1, \langle E_1, E_2 \rangle \vdash \Xi; \Delta \vdash G_3 \sim G_2$

**Proof.** By definition of evidence and interior noticing that always $E_i \subseteq G_i$. □

**Lemma 312.** If $\langle \alpha^{E_1}, E_2 \rangle \vdash \Xi; \Delta \vdash \alpha \sim G$, then $\langle E_1, E_2 \rangle \vdash \Xi; \Delta \vdash \Xi(\alpha) \sim G$.

**Proof.** Direct by definition of interior and evidence. □

**Lemma 313.** If $\langle E_1, \alpha^{E_2} \rangle \vdash \Xi; \Delta \vdash G \sim \alpha$, then $\langle E_1, E_2 \rangle \vdash \Xi; \Delta \vdash G \sim \Xi(\alpha)$.

**Proof.** Direct by definition of interior and evidence. □

**Lemma 314.** $\rho(\varepsilon_1) \circ \rho(\varepsilon_2) = \rho(\varepsilon_1 \circ \varepsilon_2)$

**Proof.** By definition of transitivity and substitution on evidences, noticing that the substitution is the same in both evidences. □

**Lemma 315.** If $\rho \vdash H_1 \equiv H_2, H_i \cap H_3$ is defined, and $H_3$ does not contain type variables, then $H_1 = H_2$.

**Proof.** Direct by induction on $H_3$. □

**Lemma 316.** If $E_2 \subseteq E_3$ then $\langle E_1, E_2 \rangle \circ \langle E_3, E_3 \rangle = \langle E_1, E_2 \rangle$.

**Proof.** We proceed by induction on $\langle E_1, E_2 \rangle$. If $\langle E_3, E_3 \rangle = \langle ?, ? \rangle$ by definition of transitivity the result holds immediately so we do not consider this case in the following.

Case ($\langle E_1, E_2 \rangle = \langle ?, ? \rangle$). Then we know that $E_3 = ?$, and the result follows immediately.
Case \((\langle E_1, E_2 \rangle = \langle E_1, \alpha^{E_2} \rangle)\). Then \(\langle E_3, E_3 \rangle = \langle \alpha^{E_3}, \alpha^{E_3} \rangle\). Then \(\langle E_1, \alpha^{E_2} \rangle \circ \langle \alpha^{E_3}, \alpha^{E_3} \rangle\) boils down to \(\langle E_1, E_2 \rangle \circ \langle E_3, E_3 \rangle\). If \(E_2' = \beta^{E_2}\), then \(E_3\) has to be \(\beta^{E_3}\) and we repeat this process. Let us assume that \(E_2' \notin \text{STYPENAME}\), then by definition of meet \(E_3 \notin \text{STYPENAME}\). By definition of precision if \(\alpha^{E_2} \subseteq \alpha^{E_1}\), then \(E_2' \subseteq E_3\). Then by induction hypothesis \(\langle E_1, E_2 \rangle \circ \langle E_3, E_3 \rangle = \langle E_1, E_2' \rangle\) and the result holds.

Case \((\langle E_1, E_2 \rangle = \langle \alpha^{E_1}, E_2 \rangle)\). Then \(\langle \alpha^{E_1}, E_2 \rangle \circ \langle E_3, E_3 \rangle\) boils down to \(\langle E_1', E_2 \rangle \circ \langle E_3, E_3 \rangle\). We know that \(E_2 \subseteq E_3\). Then by induction hypothesis \(\langle E_1', E_2 \rangle \circ \langle E_3, E_3 \rangle = \langle E_1, E_2' \rangle\), then \(\langle \alpha^{E_1}, E_2 \rangle \circ \langle E_3, E_3 \rangle = \langle \alpha^{E_1}, E_2 \rangle\) and the result holds.

Case \((\langle E_1, E_2 \rangle = \langle B, B \rangle)\). Then by definition of precision \(E_3\) is either \(\text{co-}\) (case we wont analyze) or \(B\). But \(\langle B, B \rangle \circ \langle B, B \rangle = \langle B, B \rangle\) and the result holds.

Case \((\langle E_1, E_2 \rangle = \langle E_{11} \rightarrow E_{12}, E_{21} \rightarrow E_{22} \rangle)\). Then \(E_3\) has to have the form \(E_{31} \rightarrow E_{32}\). By definition of precision, if \(E_{21} \rightarrow E_{22} \subseteq E_{31} \rightarrow E_{32}\) then \(E_{21} \subseteq E_{31}\) and \(E_{22} \subseteq E_{32}\). As \(\langle E_{31}, E_{31} \rangle \circ \langle E_{21}, E_{11} \rangle = \langle (E_{11}, E_{21}) \circ (E_{31}, E_{31}) \rangle^{-1}\). By induction hypothesis \(\langle E_{11}, E_{21} \rangle \circ \langle E_{31}, E_{31} \rangle = \langle E_{11}, E_{21} \rangle\) and \(\langle E_{12}, E_{22} \rangle \circ \langle E_{32}, E_{32} \rangle = \langle E_{12}, E_{22} \rangle\). Therefore \(\langle E_{11} \rightarrow E_{12}, E_{21} \rightarrow E_{22} \rangle \circ \langle E_{31} \rightarrow E_{32} \rangle = \langle E_{11} \rightarrow E_{12}, E_{21} \rightarrow E_{22} \rangle\) and the result holds.

Case \((\langle E_1, E_2 \rangle = \langle \forall X.E_{11}, \forall X.E_{21} \rangle)\) or \(\langle E_1, E_2 \rangle = \langle E_{11} \times E_{12}, E_{21} \times E_{22} \rangle\). Analogous to function case.

\[\] 

\[\] 

### D.5.3 Contextual Equivalence

In this section we show that the logical relation is sound with respect to contextual approximation (and therefore contextual equivalence). Figure D.6 presents the syntax and static semantics of contexts.

**Definition 317** (Contextual Approximation and Equivalence).

\[\Xi; \Delta; \Gamma \vdash t_1 \leq^{ctx} t_2 : G \triangleq \Xi; \Delta; \Gamma \vdash t_1 : G \land \Xi; \Delta; \Gamma \vdash t_2 : G \land \forall C, \Xi', G'.\]

\[\vdash C : (\Xi; \Delta; \Gamma \vdash G) \rightsquigarrow (\Xi'; \Gamma \vdash \exists \xi' \exists \xi \rightarrow C[t_1] \rightarrow * \Xi \rightarrow C[t_2] \rightarrow * \Xi \rightarrow \text{error})\]

\[\Xi; \Delta; \Gamma \vdash t_1 \approx^{ctx} t_2 : G \triangleq \Xi; \Delta; \Gamma \vdash t_1 \leq^{ctx} t_2 : G \land \Xi; \Delta; \Gamma \vdash t_2 \leq^{ctx} t_1 : G\]

**Theorem 317** (Soundness w.r.t. Contextual Approximation). If \(\Xi; \Delta; \Gamma \vdash t_1 \leq t_2 : G\) then \(\Xi; \Delta; \Gamma \vdash t_1 \leq^{ctx} t_2 : G\).

**Proof.** The proof follows the usual route of going through congruence and adequacy. \[\square\]
\[
\begin{align*}
C \ ::= & \ \emptyset \mid \varepsilon C_u \mid \{t\} \mid \{C, t\} \mid \{t, C\} \mid \emptyset \mid C \mid C \mid \varepsilon C \mid G \mid \begin{aligned}
\text{op}(t_1, C, t_2) \mid C \mid [G] \mid \pi_1(C)
\end{aligned} \\
C_u \ ::= & \ \lambda x : G. C \mid \Delta X. C \mid \{C_u, u\} \mid \langle u, C_u \rangle \\
C_s \ ::= & \ C \mid C_u
\end{align*}
\]

Well-typed contexts

\[
\begin{align*}
\vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G')
\end{align*}
\]

\[\frac{\Xi \subseteq \Xi' \quad \Delta \subseteq \Delta' \quad \Gamma \subseteq \Gamma' \quad \Xi \vdash \Gamma \quad \Xi' \vdash \Delta' \vdash \Gamma'}{\vdash \emptyset : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G')}\] (Cid)

\[\frac{\vdash C : (\Xi; \Delta; \Gamma, x : G_1 \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma', x : G_1 \vdash G_2)}{\vdash \lambda x : G_1. C : (\Xi; \Delta; \Gamma, x : G_1 \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \rightarrow G_2)}\] (CA)

\[\frac{\vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1)}{\vdash \Delta X. C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash \forall X. G')}\] (CpairL)

\[\frac{\vdash \langle t, C \rangle : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \times G_2)}{\vdash \langle C, t \rangle : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \times G_2)}\] (CpairR)

\[\frac{\vdash C_s : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G') \quad \varepsilon \vdash \Xi; \Delta \vdash G' \Rightarrow G''}{\vdash \varepsilon C_s : G'' : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G'')}\] (Case)

\[\frac{\Xi' : \Delta'; \Gamma' \vdash t_1 : G_1 \quad \Xi' : \Delta'; \Gamma' \vdash \Xi' : \Delta'; \Gamma' \vdash \Gamma' \vdash G_3}{\vdash \Xi' : \Delta'; \Gamma' \vdash t_2 : G_2 \quad ty(op) = (G_1, G_3, G_2) \Rightarrow G''} \] (Cop)

\[\frac{\vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \rightarrow G_2)}{\vdash C t : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \rightarrow G_2)}\] (CappL)

\[\frac{\Xi' : \Delta'; \Gamma' \vdash t : G_1 \rightarrow G_2 \quad \Xi' : \Delta'; \Gamma' \vdash t : G_1 \rightarrow G_2 \quad \Xi' : \Delta'; \Gamma' \vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1)}{\vdash \langle t, C \rangle : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_2)}\] (CappR)

\[\frac{\vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash \forall X. G') \quad \Xi' : \Delta' \vdash G''}{\vdash C \left[ G'' \right] : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G'[G''/X])} \] (CappG)

\[\frac{\vdash C : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1 \times G_2)}{\vdash \pi_1(C) : (\Xi; \Delta; \Gamma \vdash G) \Rightarrow (\Xi'; \Delta'; \Gamma' \vdash G_1)}\] (Cpair)

Figure D.6: GSF \(\varepsilon\): Syntax and Static Semantics - Contexts

418
D.6 GSF: Imprecise Termination

Throughout these proofs we assume that lift_Σ(T) = \hat{T} (we omit the Σ notation when obvious from the context).

**Proposition 318.** Let 𝑡 be a static term. If Σ; Δ; Γ ⊢ 𝑡 : T, T ⊑ G, ε ⊢ Σ; Δ ⊢ T ⇓ G, and ε = (\hat{T}, \hat{T})", then Σ; Δ; Γ ⊢ ε : G : T ⊑ G.

**Proof.** By induction on the type derivation of 𝑡. Note that all the subterms of 𝑡 are also static.

**Case (Eb).** If 𝑡 = ε' : B then:

\[(\text{Eb}) \quad \text{ty}(b) = B \quad \Sigma; Δ; Γ ⊢ b : B\]

**Case (Eλ).** If 𝑡 = ε(λ𝑥 : T₁, 𝑡') : T₂ → T₂' then we know that:

\[\Sigma; Δ; Γ, x : T₁ ⊢ τ : T₂ \quad ε_λ \vdash \Sigma; Δ ⊢ T₁ ⇓ T₂ \rightarrow T₂' \rightarrow T₂' \]

Then we have to prove that:

\[\Sigma; Δ; Γ \vdash ε(ε_λ(λ𝑥 : T₁, 𝑡')) : T₂' → T₂' \vdash G : T₁ → T₂' \subseteq G\]

Then after the usual unfoldings we have to prove that, for all ρ ∈ D_Σ[Δ], γ ∈ G_p_Σ[Γ]:

\[\rho(ε)(ρ(ε_λ)(λ𝑥 : T₁, ρ(⟨τ⟩))) : ρ(T₁') → ρ(T₂') \vdash ρ(G) ∈ C_p_Σ[T₁' → T₂' \subseteq G]\]

Suppose that \(T_{1'} = ρ(T₁), T_{12} = ρ(T₂), T_{21} = ρ(T₁')\), and \(T_{22} = ρ(T₂')\) then \(ρ(ε) = (T_{21} → T_{22}, T_{21} → T_{22})\), and \(ρ(ε_λ) = (T_{11} → T_{12}, T_{21} → T_{22})\). Then by Lemma 316, \(ρ(ε_λ) \circ ρ(ε) = ρ(ε_λ) = (T_{11} → T_{12}, T_{21} → T_{22})\). Then we have to prove that

\[\rho(ε_λ)(λ𝑥 : T₁, ρ(⟨τ⟩))) : ρ(G) ∈ \mathcal{N}_p_Σ[T₁' → T₂' \subseteq G]\]

Let \(G₁ = \text{dom}_T(ρ(G))\) and \(G₂ = \text{cod}^T(ρ(G))\). We have to proof that for all \(v' ∈ \mathcal{N}_p_Σ[T₁' \subseteq \text{dom}_T(G)]\) it is true that:

\[(ρ(ε_λ)(λ𝑥 : T₁, ρ(⟨τ⟩))) : G₁ → G₂ \quad v' ∈ \mathcal{C}_p_Σ[T₂' \subseteq \text{cod}^T(G)]\]

Let \(v = ε_v : G₁\). By the reduction rules we know that:

\[\Sigma \vdash ρ(ε_λ)(λ𝑥 : T₁, ρ(⟨τ⟩))) : G₁ → G₂ \quad v' \quad \text{cod}(ρ(ε_λ))(⟨ρ(⟨τ⟩)⟩)(ε_v \circ \text{dom}(ε_λ))u : ρ(T₁)/x) \vdash G₂\]
Note that $\gamma' = \gamma, (x : T_1 \rightarrow (\varepsilon \circ \text{dom}(\varepsilon_\lambda))u :: \rho(T_1)) \in G_\rho^\Sigma_\Gamma [x : T_1]$. Then by the induction hypothesis on $t'$, with $\rho, \gamma'$, and $(T_0', T_1')$, we know that:

$$\rho(\langle T_0', T_1' \rangle) \rho(\gamma'(t')) :: \rho(T_2) \in C_\rho^\Sigma[T_2 \sqsubseteq T_2]$$

Note that $\rho(\gamma(t'))[(\varepsilon \circ \text{dom}(\varepsilon_\lambda))u :: \rho(T_1)/x \mapsto \rho(\gamma'(t'))]$. Then by Lemma 325, the result holds.

**Case.** If $t = \varepsilon\Delta X. t' :: \forall X. T_1$ then:

$$\Sigma; \Delta, X ; \Gamma \vdash t' : T_2 \quad \varepsilon\Delta \models \Sigma; \Delta \vdash \forall X. T_2 \sim \forall X. T_1$$

Then we have to prove that:

$$\Sigma; \Delta; \Gamma \vdash \forall \varepsilon\Delta X. t' :: \forall X. T_1 :: G : \forall X. T_1 \sqsubseteq G$$

Then after the usual unfoldings we have to prove that, for some $\rho \in D^\Sigma[\Delta], \gamma \in G_\rho^\Sigma_\Gamma$:

$$\rho(\varepsilon)(\rho(\varepsilon)\Delta X. \rho(\gamma(t')) :: \forall X. \rho(T_1)) :: \rho(G) \in C_\rho^\Sigma[\forall X. T_1 \sqsubseteq G]$$

Suppose that $T'_1 = \rho(T_1)$, then $\rho(\varepsilon) = \langle \forall X. \hat{T}_1', \forall X. \hat{T}_1' \rangle$, and that $\rho(\varepsilon') = \langle \forall X. \hat{T}_2', \forall X. \hat{T}_1' \rangle$. Then by Lemma 316 $\rho(\varepsilon') \circ \rho(\varepsilon) = \langle \forall X. T_2', \forall X. T_1' \rangle$. Then we have to prove that

$$(\rho(\varepsilon')\Delta X. \rho(\gamma(t'))) :: \rho(G) \in N_\rho^\Sigma[\forall X. T_1 \sqsubseteq G]$$

Let some $T'$ such that $\Sigma \vdash T'$, posing $G_1 = \text{schm}'(\rho(G))$, then

$$\Sigma \triangleright (\rho(\varepsilon)\Delta X. \rho(\gamma(t'))) :: \forall X. G_1 [T'] \longmapsto$$

$$\Sigma, \alpha := T' \triangleright \langle E_1[\alpha^{T'} / X], E_1[\hat{T}_1' / X] \rangle \langle \hat{T}_2'[\alpha^{T} / X], \hat{T}_1'[\alpha^{T_1} / X] \rangle \rho(\gamma(t'))[\alpha^{T_1} / X] :: G_1[\alpha / X] :: G_1[T'/X]$$

where $E_1 = \text{lift}_2(G_1)$, and $\hat{T}' = \text{lift}_2(T')$. Note that $\text{schm}(\varepsilon) \models \Sigma; \Delta, X \vdash T_1 \sim \text{schm}^2(G)$. Now we have to prove that

$$(\langle \hat{T}_2'[\alpha^{T_1}/X], \hat{T}_1'[\alpha^{T_1}/X] \rangle \rho(\gamma(t'))[\alpha^{T_1}/X] :: G_1[\alpha / X]) \in C_\rho^\Sigma[T_1 \sqsubseteq G_1]$$

But note that $\rho(\gamma(t'))[\alpha^{T_1}/X] = \rho'(\gamma(t'))$, then we use induction hypothesis on $t'$, with $\rho'$, $\gamma$, and $\varepsilon = \langle \hat{T}_2, \hat{T}_2' \rangle$, where $\hat{T}_2 = \text{lift}_2(T_2)$. Then

$$(\rho'(\langle \hat{T}_2, \hat{T}_2' \rangle) \rho'(\gamma(t'))) :: \rho'(T_2) \in C_\rho^\Sigma[T_2 \sqsubseteq T_2]$$

and thus, posing $\hat{T}_2'' = \rho'(\hat{T}_2)$

$$\Sigma' \triangleright (\langle \hat{T}_2'', \hat{T}_2' \rangle) \rho'(\gamma(t')) :: \rho'(T_2) \longmapsto \Sigma'' \triangleright (\langle \hat{T}_3, \hat{T}_3'' \rangle u :: \rho'(T_2))$$

and $(\langle \hat{T}_3, \hat{T}_3'' \rangle u :: \rho'(T_2)) \in N_\rho^\Sigma[\forall T_2 \sqsubseteq T_2]$. Then by Lemma 325, as $\hat{T}_2'[\alpha^{T_1}/X] = \hat{T}_2''$, then

$$(\langle \hat{T}_3, \hat{T}_1'[\alpha^{T_1}/X] \rangle u :: G_1[\alpha / X] \in N_\rho^\Sigma[\forall T_1 \sqsubseteq G_1]$$

and the result holds.
Case. If \( t = \langle u_1, u_2 \rangle \) then:

\[
\begin{align*}
\text{(Epair)} & \quad \Sigma; \Delta; \Gamma \vdash u_1 : T_1 & \Sigma; \Delta; \Gamma \vdash u_2 : T_2 \\
& \quad \Sigma; \Delta; \Gamma \vdash \langle u_1, u_2 \rangle : T_1 \times T_2
\end{align*}
\]

Then we have to prove that:

\[
\Sigma; \Delta; \Gamma \vdash \epsilon \langle u_1, u_2 \rangle :: G : T_1 \times T_2 \subseteq G
\]

We know that \( p_1(\epsilon) \vdash \Sigma; \Delta \vdash T_1 \sim \text{proj}_1^2(G) \). Then by induction hypotheses we already know that: \( \Sigma; \Delta; \Gamma \vdash p_1(\epsilon) u_1 :: \text{proj}_1^2(G) : T_1 \subseteq \text{proj}_1^2(G) \) and \( \Sigma; \Delta; \Gamma \vdash p_2(\epsilon) u_2 :: \text{proj}_2^2(G) : T_2 \subseteq \text{proj}_2^2(G) \). But the result follows directly by Prop \[297\] and \[298\] (compatibility of pairs).

Case (Easc). Then \( t = \varepsilon \langle t' \rangle :: T \), and therefore:

\[
\begin{align*}
\text{(Easc)} & \quad \Sigma; \Delta; \Gamma \vdash \varepsilon \langle t' \rangle :: T' \sim T \\
& \quad \Sigma; \Delta; \Gamma \vdash \varepsilon \langle t' \rangle :: T : T
\end{align*}
\]

By induction hypotheses we already know that \( \Sigma; \Delta; \Gamma \vdash \varepsilon \langle t' \rangle :: T : T \subseteq T \), then the result follows directly by Prop \[293\] (Compatibility of ascriptions).

Case (Epair). Then \( t = \langle t_1, t_2 \rangle \), and therefore:

\[
\begin{align*}
\text{(Epair)} & \quad \Sigma; \Delta; \Gamma \vdash t_1 : G_1 & \Sigma; \Delta; \Gamma \vdash t_2 : G_2 \\
& \quad \Sigma; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle : G_1 \times G_2
\end{align*}
\]

where \( G = G_1 \times G_2 \) Then we have to prove that:

\[
\Xi; \Delta; \Gamma \vdash \langle t_1, t_2 \rangle \preceq \langle t_1, t_2 \rangle : G_1 \times G_2
\]

By induction hypotheses we already know that: \( \Xi; \Delta; \Gamma \vdash t_1 \preceq t_1 : G_1 \) and \( \Xi; \Delta; \Gamma \vdash t_2 \preceq t_2 : G_2 \). But the result follows directly by Prop \[291\] (Compatibility of pairs).

Case (Ex). Then \( t = x \), and therefore:

\[
\begin{align*}
\text{(Ex)} & \quad x : G \in \Gamma \\
& \quad \Xi; \Delta; \Gamma \vdash x : G
\end{align*}
\]

Then we have to prove that \( \Xi; \Delta; \Gamma \vdash x \preceq x : G \). But the result follows directly by Prop \[292\] (Compatibility of variables).

Case (Eop). Then \( t = \text{op}(\overline{v}) \), and therefore:

\[
\begin{align*}
\text{(Eop)} & \quad \Sigma; \Delta; \Gamma \vdash \overline{v} : \overline{G} & t_y(\text{op}) = G' \rightarrow G \\
& \quad \Sigma; \Delta; \Gamma \vdash \text{op}(\overline{v}) : G
\end{align*}
\]

Then we have to prove that: \( \Xi; \Delta; \Gamma \vdash \text{op}(\overline{v}) \preceq \text{op}(\overline{v}) : G \). By the induction hypothesis we obtain that: \( \Xi; \Delta; \Gamma \vdash \overline{v} \preceq \overline{v} : \overline{G} \). Then the result follows directly by Prop \[294\] (Compatibility of app operator).

Case (Eapp). Then \( t = t_1 \ t_2 \), and therefore:

\[
\begin{align*}
\text{(Eapp)} & \quad \Sigma; \Delta; \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} & \Sigma; \Delta; \Gamma \vdash t_2 : T_{11} \\
& \quad \Sigma; \Delta; \Gamma \vdash t_1 \ t_2 : T_{12}
\end{align*}
\]

where \( T = T_{12} \). Then we have to prove that:

\[
\Sigma; \Delta; \Gamma \vdash \varepsilon \langle t_1, t_2 \rangle :: G : T_{12} \subseteq G
\]

By the induction hypothesis we obtain that: \( \Sigma; \Delta; \Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \subseteq T_{11} \rightarrow G \) and \( \Sigma; \Delta; \Gamma \vdash t_2 : T_{11} \subseteq T_{11} \). Then the result follows directly by Prop \[295\] (Compatibility of term application).
Case (EappG). Then $t = t'[T_2]$, and therefore:

\[(EappG) \begin{array}{c}
\Sigma; \Delta; \Gamma \vdash t' : \forall X. T_1 \\
\Xi; \Delta \vdash T_2
\end{array} \]

where $T = T_1[T_2/X]$. Then after the usual unfoldings we have to prove that, for some $\rho \in D^{\Xi}[\Delta], \gamma \in G^{\Sigma}[\Gamma]$:

$$
\rho(\varepsilon)(\rho(\gamma(t'))[\rho(T_2)]) :: \rho(G) \in C_{\rho}^{\Sigma}[T_1[T_2/X] \subseteq G]
$$

Note that $\varepsilon_{\forall X. T_1} \vdash \Sigma; \Delta \vdash \forall X. T_1 \sim \forall X. T_1$. Then by induction hypothesis we know that $\varepsilon_{\forall X. \rho(T_1)}(\rho(\gamma(t')) :: \forall X. \rho(T_1) \in C_{\rho}^{\Sigma}[\forall X. T_1 \subseteq \forall X. T_1]$, let $T_1' = \rho(T_1)$, then

$$
\Sigma \triangleright \varepsilon_{\forall X. T_1} \rho(\gamma(t')) :: \forall X.T_1' :: \forall X.T_1'
$$

where $\varepsilon_{\forall} = (\forall X. T_1'', \forall X. \hat{T}_1'')$, for some $T_1''$, and $\varepsilon_{\forall}(\Lambda X.t'') :: \forall X. T_1' \in N_{\rho}^{\Sigma}[\forall X. T_1 \subseteq \forall X. T_1]$. If we instantiate the last interpretation with $T' = \rho(T_2)$, then we know that

$$
\Sigma', \alpha := \rho(T_2) \triangleright (\hat{T}_1'[\alpha^{\hat{T}}/X], \hat{T}_1'[T'/X]((\hat{T}_1''[\alpha^{\hat{T}}/X], \hat{T}_1'[\alpha^{\hat{T}}/X])t''[\alpha^{\hat{T}}/X] :: T_1[\alpha/X]) :: T_1'[T'/X]
$$

where $\hat{T}' = \text{lift}_{\Sigma}(T')$, $\hat{T}_1' = \text{lift}_{\Sigma}(T_1')$, and

$$
((\hat{T}_1''[\alpha^{\hat{T}}/X], \hat{T}_1'[\alpha^{\hat{T}}/X])t''[\alpha^{\hat{T}}/X] :: T_1[\alpha/X]) \in C_{\rho}^{\Sigma''}[T_1 \subseteq T_1]
$$

for $\Sigma'' = \Sigma', \alpha := T'$, and $\rho' = \rho, X \triangleright \alpha$. Let $t''' = ((\hat{T}_1''[\alpha^{\hat{T}}/X], \hat{T}_1'[\alpha^{\hat{T}}/X])t''[\alpha^{\hat{T}}/X] :: T_1[\alpha/X])$, then $\Sigma'' \triangleright t''' \mapsto \Sigma''' \triangleright (\hat{T}_3, \hat{T}_1'[\alpha^{\hat{T}}/X])u :: T_1'[\alpha/X]$, and $\langle \hat{T}_3, \hat{T}_1'[\alpha^{\hat{T}}/X] \rangle u :: T_1'[\alpha/X] \in N_{\rho}^{\Sigma'''}[T_1[T_2/X] \subseteq T_1[T_2/X]]$. Then we have to prove that

$$
\langle \hat{T}_3, \hat{T}_1' [T'/X] \rangle \circ \langle \hat{T}_1'[\alpha^{\hat{T}}/X], \hat{T}_1'[T'/X] \rangle u :: T_1'[T'/X] \in N_{\rho}^{\Sigma''''}[T_1[T_2/X] \subseteq T_1[T_2/X]]
$$

which follows the from compositionality (Prop[326]). Then

$$
\langle \hat{T}_3, \hat{T}_1'[T'/X] \rangle u :: T_1'[T'/X] \in N_{\rho}^{\Sigma''''}[T_1[T_2/X] \subseteq T_1[T_2/X]]
$$

But $\rho(\varepsilon) = \langle \hat{T}_1'[T'/X], \hat{T}_1'[T'/X] \rangle$, and the result holds by Lemma[325]

Case (Epair1). Then $t = \pi_1(t')$, and therefore:

\[(Epair1) \begin{array}{c}
\Sigma; \Delta; \Gamma \vdash t' : G_1 \times G_2 \\
\Sigma; \Delta; \Gamma \vdash \pi_1(t') : G_1
\end{array} \]

where $G = G_1$. Then we have to prove that: $\Xi; \Delta; \Gamma \vdash \pi_1(t') \triangleq \pi_1(t') : G_1$. By the induction hypothesis we obtain that: $\Xi; \Delta; \Gamma \vdash t' \triangleq t' : G_1 \times G_2$. Then the result follows directly by Prop[297] (Compatibility of access to the first component of the pair).

Case (Epair2). Then $t = \pi_2(t')$, and therefore:

\[(Epair2) \begin{array}{c}
\Sigma; \Delta; \Gamma \vdash t' : G_1 \times G_2 \\
\Sigma; \Delta; \Gamma \vdash \pi_2(t') : G_2
\end{array} \]

where $G = G_2$. Then we have to prove that: $\Xi; \Delta; \Gamma \vdash \pi_2(t') \triangleq \pi_2(t') : G_2$. By the induction hypothesis we obtain that: $\Xi; \Delta; \Gamma \vdash t' \triangleq t' : G_1 \times G_2$. Then the result follows directly by Prop[298] (Compatibility of access to the second component of the pair).
Lemma 319. If \( T \subseteq G \) and \( \varepsilon \vdash \Sigma; \Delta \vdash T \sim G \) then \( \varepsilon = \langle \text{lift}_\Sigma(T), \text{lift}_\Sigma(T) \rangle \).

Proof. By induction on the structure of the type \( T \), and the definition of \( \subseteq \) and \( \Sigma; \Delta \vdash \cdot \sim \cdot \).

\( \square \)

Lemma 320. If \( t \in C^\Sigma_\rho[T \subseteq G] \) and \( \Sigma \subseteq \Sigma' \) then \( t \in C^\Sigma'_\rho[T \subseteq G] \).

Proof. By induction on the structure of the \( t \), and the definition of \( C^\Sigma_\rho[\cdot \subseteq \cdot] \).

\( \square \)

Lemma 321. If \( t \in C^\Sigma_\rho[T \subseteq G] \) and \( \Sigma \subseteq \Sigma' \) then \( t \in C^\Sigma'_\rho[T \subseteq G] \).

Proof. By induction on the structure of the \( t \), and the definition of \( C^\Sigma_\rho[\cdot \subseteq \cdot] \).

\( \square \)

Lemma 322. \( g_\Sigma(G, G) = \langle \text{lift}_\Sigma(G), \text{lift}_\Sigma(G) \rangle \).

Proof. By induction on the structure of the type \( G \), and the definition of \( g_\Sigma(\cdot, \cdot) \).

\( \square \)

Lemma 323. If \( G_1 \subseteq G_2 \) then \( \text{lift}_\Sigma(G_1) \subseteq \text{lift}_\Sigma(G_2) \).

Proof. By induction on the structure of the type \( G \), and the definition of \( \subseteq \) and \( \text{lift}_\Sigma(\cdot) \).

\( \square \)

Lemma 324. \( \langle \hat{T}_1, \hat{T}_2 \rangle \circ \langle \hat{T}_2, \hat{T}_3 \rangle = \langle \hat{T}_1, \hat{T}_3 \rangle \).

Proof. By induction on the structure of the evidences, noticing that every evidence is static, so you cannot gain precision on the resulting values (Optimality Lemma 307).

\( \square \)

Lemma 325 (Lemma asc). If \( \langle \hat{T}_1, \hat{T}_2 \rangle \vdash \Sigma; \cdot \vdash \rho(T_1) \sim \rho(G), \langle \hat{T}_2, \hat{T}_3 \rangle \vdash \Sigma; \cdot \vdash \rho(G) \sim \rho(G'), T_3 \subseteq G' \), and \( \langle \hat{T}_1, \hat{T}_2 \rangle u :: \rho(G) \in N^\Sigma_\rho[T_2 \subseteq G] \) then \( \langle \hat{T}_1, \hat{T}_3 \rangle u :: \rho(G') \in N^\Sigma_\rho[T_3 \subseteq G'] \).

Proof. We proceed by induction on evidences \( \langle \hat{T}_1, \hat{T}_2 \rangle \) and \( \langle \hat{T}_2, \hat{T}_3 \rangle \). For simplicity, we omit type substitution \( \rho \), when is not important.

Case \( \hat{T}_2 = \alpha \hat{T}_2 \). Notice by inspection of the consistent transitivity rules and Lemma 324 \( \langle \hat{T}_1, \alpha \hat{T}_2 \rangle \circ \langle \alpha \hat{T}_2, \hat{T}_3 \rangle = \langle \hat{T}_1, \hat{T}_2 \rangle \circ \langle \hat{T}_2, \hat{T}_3 \rangle = \langle \hat{T}_1, \hat{T}_3 \rangle \).

As \( \alpha \subseteq G \) and \( \langle \hat{T}_1, \alpha \hat{T}_2 \rangle \vdash \Sigma; \cdot \vdash T_1 \sim G \), by Lemma 311 \( \langle \hat{T}_1, \alpha \hat{T}_2 \rangle \vdash \Sigma; \cdot \vdash T_1 \sim \alpha \). Then by Lemma 313 \( \langle \hat{T}_1, \hat{T}_2 \rangle \vdash \Sigma; \cdot \vdash T_1 \sim \Sigma(\alpha) \), and \( \langle \hat{T}_2, \hat{T}_3 \rangle \vdash \Sigma; \cdot \vdash \Sigma(\alpha) \sim G' \), then the result follows by induction hypothesis on \( \langle \hat{T}_1, \hat{T}_2 \rangle \) and \( \langle \hat{T}_2, \hat{T}_3 \rangle \).

Case \( \hat{T}_3 = \alpha \hat{T}_3 \). Notice by inspection of the consistent transitivity rules and Lemma 324 \( \langle \hat{T}_1, \hat{T}_2 \rangle \circ \langle \hat{T}_2, \alpha \hat{T}_3 \rangle = \langle \hat{T}_1, \alpha \hat{T}_3 \rangle \). Then we need to prove that \( \langle \hat{T}_1, \alpha \hat{T}_3 \rangle u :: G \in N^\Sigma_\rho[\alpha \subseteq G'] \), which is equivalent to prove that (posing \( T'_3 = \Sigma(\alpha) \)) \( \langle \hat{T}_1, \hat{T}_3' \rangle u :: G' \in N^\Sigma_\rho[T'_3 \subseteq G'' \), where \( T'_3 \subseteq G'' \). But also by Lemma 324 \( \langle \hat{T}_1, \hat{T}_2 \rangle \circ \langle \hat{T}_2, \hat{T}_3' \rangle = \langle \hat{T}_1, \hat{T}_3' \rangle \).

423
As $\alpha \subseteq G'$ and $(\hat{T}_2, \alpha^{\hat{T}_3}) \vdash \Sigma; \vdash G \sim G'$, by Lemma 311, $(\hat{T}_2, \alpha^{\hat{T}_3}) \vdash \Sigma; \vdash G \sim \alpha$. Then by Lemma 313 $(\hat{T}_2, \hat{T}_3) \vdash \Sigma; \vdash G \sim T'_\beta$, and by Lemma 311 $(\hat{T}_2, \hat{T}_3) \vdash \Sigma; \vdash G \sim G''$, then by induction hypothesis on $(\hat{T}_1, \hat{T}_2)$ and $(\hat{T}_2, \hat{T}_3)$ the result follows.

Case $(\hat{T}_1 = \alpha^{\hat{T}_1})$. This case can never happen as there are no values where the left component of an evidence is a type name.

Case $(\hat{T}_1 = \hat{T}_{11} \rightarrow \hat{T}_{12})$. We know that

\[
(\hat{T}_{11} \rightarrow \hat{T}_{12}, \hat{T}_{31} \rightarrow \hat{T}_{32})(\lambda x : T_{11}, t') :: G' \in \mathcal{H}_\rho^G[T_{31} \rightarrow T_{32} \sqsubseteq G']
\]

Where $(\hat{T}_{11} \rightarrow \hat{T}_{12}, \hat{T}_{31} \rightarrow \hat{T}_{32}) \vdash \Sigma; \Delta \vdash T_{11} \rightarrow T_{12} \sim G$.

We have to prove that:

\[
\forall(\varepsilon' u' :: G_1) \in \mathcal{H}_\rho^G[T_{31} \sqsubseteq G_1] \Rightarrow
(\forall(\varepsilon' u' :: G_1) \in \mathcal{H}_\rho^G[T_{31} \sqsubseteq G_1] \Rightarrow
\]

\[
(\langle \hat{T}_{11} \rightarrow \hat{T}_{12}, \hat{T}_{31} \rightarrow \hat{T}_{32} \rangle(\lambda x : T_{11}, t') :: G_1 \sim T_{21} \sqsubseteq G_2) \rightarrow (\varepsilon' u' :: G_1) \rightarrow (G_1 \rightarrow G_2) (\varepsilon' u' :: G_1) \rightarrow (G_1 \rightarrow G_2)
\]

Note that by the reduction rule of application terms, we obtain that:

$$
\Sigma \triangleright (\langle \hat{T}_{11} \rightarrow \hat{T}_{12}, \hat{T}_{31} \rightarrow \hat{T}_{32} \rangle(\lambda x : T_{11}, t') :: G_1 \sim T_{21} \sqsubseteq G_2) \rightarrow (\varepsilon' u' :: G_1) \rightarrow (G_1 \rightarrow G_2) (\varepsilon' u' :: G_1) \rightarrow (G_1 \rightarrow G_2)
$$

Note that by Lemma 324 $(\langle \hat{T}_{31}, \hat{T}_{21} \rangle \circ (\hat{T}_{21}, \hat{T}_{11}) \rangle) = (\hat{T}_{31}, \hat{T}_{11})$, and by Lemma 302, $\varepsilon' \circ ((\hat{T}_{31}, \hat{T}_{21}) \circ (\hat{T}_{21}, \hat{T}_{11}))$ = $\varepsilon' \circ ((\hat{T}_{31}, \hat{T}_{21}) \circ (\hat{T}_{21}, \hat{T}_{11}))$. Note that $\varepsilon' = (\hat{T}_u, \hat{T}_{31}) \vdash \Sigma; \cdot \vdash T_u \sim dom^\sharp(G')$ and $(\hat{T}_{31}, \hat{T}_{21}) \vdash \Sigma; \cdot \vdash dom^\sharp(G') \sim dom^\sharp(G)$, therefore we know that by the induction hypothesis that $(\varepsilon' \circ (\hat{T}_{31}, \hat{T}_{21})) u' :: dom^\sharp(G)) \in \mathcal{H}_\rho^G[T_{21} \sqsubseteq dom^\sharp(G)]$.

We instantiate $\mathcal{H}_\rho^G[T_{21} \sqsubseteq T_{22} \sqsubseteq G]$ with $u' = ((\varepsilon' \circ (\hat{T}_{31}, \hat{T}_{21})) u' :: dom^\sharp(G)) \in \mathcal{H}_\rho^G[T_{21} \sqsubseteq dom^\sharp(G)]$, and then we know that:

$$
\Sigma \triangleright (\langle \hat{T}_{11} \rightarrow \hat{T}_{12}, \hat{T}_{21} \rightarrow \hat{T}_{22} \rangle(\lambda x : T_{11}, t') :: dom^\sharp(G) \rightarrow cod^\sharp(G) \rightarrow ((\varepsilon' \circ (\hat{T}_{31}, \hat{T}_{21})) u' :: dom^\sharp(G)) \rightarrow (\varepsilon'' u'' :: cod^\sharp(G)) \in \mathcal{H}_\rho^G[T_{22} \sqsubseteq cod^\sharp(G)]$, for some $\Sigma'$, such that $\Sigma \subseteq \Sigma'$.

The resulting term reduce to value $(\varepsilon'' u'' :: cod^\sharp(G)) \in \mathcal{H}_\rho^G[T_{22} \sqsubseteq cod^\sharp(G)]$, for some $\Sigma'$.
But note that by Lemmas 302 and 324

\[ \Sigma \vdash \langle T_{12}, T_{32} \rangle (t'[(\varepsilon' \circ (T_{31}, T_{21}) \circ (T_{21}, T_{11})))u' :: T_{11}/x]) :: G_2 \]

\[ = \Sigma \vdash \langle T_{22}, T_{32} \rangle (\langle T_{12}, T_{22} \rangle (t'[(\varepsilon' \circ (T_{31}, T_{21}) \circ (T_{21}, T_{11})))u' :: T_{11}/x]) :: cod^2(G) :: G_2 \]

\[ \mapsto \ast \Sigma' \vdash \langle T_{22}, T_{32} \rangle (\varepsilon''u'' :: cod^2(G)) :: G_2 \]

Then the result follows immediately by using induction hypothesis on evidences \( \varepsilon'' \) and \( \langle T_{22}, T_{32} \rangle \).

Case (\( \hat{T}_1 = \forall X.\hat{T}_1 \)). We know that:

\[ \langle \forall X.\hat{T}_1, \forall X.\hat{T}_2 \rangle (\Delta X.\rho(t_1)) :: \forall X.\rho(T_2) \in n_\Sigma^\Delta[vX.T_1 \sqsubseteq G] \]

where \( \langle \forall X.\hat{T}_1, \forall X.\hat{T}_2 \rangle \vdash \Sigma; \Delta \vdash \forall X.\rho(T_1) \sim \rho(G) \).

We have to proof that:

\[ \langle \forall X.\hat{T}_1, \forall X.\hat{T}_3 \rangle (\Delta X.\rho(t_1)) :: \forall X.\rho(T_3) \in n_\Sigma^\Delta[vX.T_3 \sqsubseteq G'] \]

Let \( \forall X.\hat{G}_1 = \forall X.schm^\delta(G') \). Then for any \( T' \), such that \( \Sigma \vdash T' \), as

\[ \Sigma \vdash \langle \forall X.\hat{T}_1, \forall X.\hat{T}_3 \rangle (\Delta X.\rho(t_1)) :: \forall X.\rho(G_1)[T] \mapsto \]

\[ \Sigma, \alpha := T' \vdash \varepsilon'((\hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_3[\alpha^{\hat{T}_1}/X])\rho(t_1)[\alpha^{\hat{T}_1}/X] :: \rho(G_1'[\alpha/X]) :: \rho(G_1')[T'/X] \]

Then we have to proof that

\[ (\langle \hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_3[\alpha^{\hat{T}_1}/X] \rangle \rho(t_1)[\alpha^{\hat{T}_1}/X] :: \rho(G_1'[\alpha/X])) \in C_\Sigma^\alpha[T_3 \sqsubseteq G'] \]

where \( \Xi' = \Xi, \alpha := T' \), and \( \rho' = \rho[X \mapsto \alpha] \).

Note that \( \langle \hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_3[\alpha^{\hat{T}_1}/X] \rangle = (\hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_2[\alpha^{\hat{T}_1}/X]) \circ (\hat{T}_2[\alpha^{\hat{T}_1}/X], \hat{T}_3[\alpha^{\hat{T}_1}/X]) \).

Let \( \forall X.\hat{G}_1 = \forall X.schm^\delta(G) \), we instantiate \( n_\Sigma^\Delta[vX.T_1 \sqsubseteq G] \) with \( T' \), so:

\[ \Sigma \vdash \langle \forall X.\hat{T}_1, \forall X.\hat{T}_3 \rangle (\Delta X.\rho(t_1)) :: \forall X.\rho(G_1)[T] \mapsto \]

\[ \Sigma, \alpha := T' \vdash \varepsilon'((\hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_2[\alpha^{\hat{T}_1}/X])\rho(t_1)[\alpha^{\hat{T}_1}/X] :: \rho(G_1)[\alpha/X]) :: \rho(G_1)[T'/X] \]

and

\[ (\langle \hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_2[\alpha^{\hat{T}_1}/X] \rangle \rho(t_1)[\alpha^{\hat{T}_1}/X] :: \rho(G_1)[\alpha/X]) \in C_\Sigma^\alpha[T_2 \sqsubseteq G_1] \]

therefore

\[ \Sigma' \vdash \langle \hat{T}_1[\alpha^{\hat{T}_1}/X], \hat{T}_2[\alpha^{\hat{T}_1}/X] \rangle \rho(t_1)[\alpha^{\hat{T}_1}/X] :: \rho(G_1)[\alpha/X] \mapsto \]

\[ \Sigma'' \vdash \langle \hat{T}_u, \hat{T}_2[\alpha^{\hat{T}_1}/X] \rangle u :: \rho(G_1)[\alpha/X] \]

for some \( \hat{T}_u \), such that \( \langle \hat{T}_u, \hat{T}_2[\alpha^{\hat{T}_1}/X] \rangle u :: \rho'(G_1) \in n_\Sigma^\Delta[T_2 \sqsubseteq G_1] \). Then using analogous arguments as for the function case, as

\[ \langle \hat{T}_u, \hat{T}_2[\alpha^{\hat{T}_1}/X] \rangle \circ (\hat{T}_2[\alpha^{\hat{T}_1}/X], \hat{T}_3[\alpha^{\hat{T}_1}/X]) = (\hat{T}_u, \hat{T}_3[\alpha^{\hat{T}_1}/X]), \]

then by induction hypothesis using \( \rho' \) (as \( \hat{T}_1 = lift_\Sigma(\rho(T_1)) \), then \( \hat{T}_1[\alpha^{\hat{T}_1}/X] = lift_{\Sigma''}(\rho'(T_1)) \), for \( i \in \{2, 3\} \)) the result holds immediately.
Proposition 326 (Compositionality). Let \( \rho' = \rho[X \mapsto \alpha] \) and \( \hat{T}' = \text{lift}_\Sigma(\rho(T')) \), \( \Sigma(\alpha) = \rho(T') \), \( \theta(\text{lift}_\Sigma(\rho(T)), \text{lift}_\Sigma(\rho(T))) = \langle \hat{T}, \hat{T}' \rangle \), \( \varepsilon = \langle \hat{T}[\alpha^{T'}/X], \hat{T'}[\alpha^{T'}/X] \rangle, \varepsilon^{-1} = \langle T[T'/X], \hat{T}[\alpha^{T'}/X] \rangle \), such that \( \varepsilon \vdash \Sigma \vdash \rho(T[\alpha/X]) \sim \rho(T[T'/X]) \), and \( \varepsilon^{-1} \vdash \Sigma \vdash \rho(T[T'/X]) \sim \rho(T[\alpha/X]) \) then

1. \( \varepsilon'u :: \rho'(T) \in \Sigma^\Sigma[\rho'] [T \subseteq T] \Rightarrow (\varepsilon' \circ \varepsilon)u :: \rho'(T[T'/X]) \in \Sigma^\Sigma[\rho'] [T[T'/X] \subseteq T[T'/X]] \)

2. \( \varepsilon'u :: \rho'(T[T'/X]) \in \Sigma^\Sigma[\rho'] [T[T'/X] \subseteq T[T'/X]] \Rightarrow (\varepsilon' \circ \varepsilon^{-1})u :: \rho'(T) \in \Sigma^\Sigma[\rho'] [T \subseteq T] \)

Proof. As everything is static, then we proceed analogous to the compositionality proof for static terms, proving (1) and (2) by induction on \( T \). For instance:

Case \((1), T = X\). Let \( v = \langle \hat{T}_1, \alpha^{T'} \rangle u :: \alpha \). Then we know that

\[
\langle \hat{T}_1, \alpha^{T'} \rangle u :: \alpha \in \Sigma^\Sigma[\rho'] [X \subseteq X]
\]

which is equivalent to

\[
\langle \hat{T}_1, \alpha^{T'} \rangle u :: \alpha \in \Sigma^\Sigma[\rho'] [\alpha \subseteq \alpha]
\]

As \( \Sigma(\alpha) = \rho(T') \), we know that:

\[
\langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \in \Sigma^\Sigma[\rho'] [T' \subseteq T']
\]

And as the value does not have \( X \) free,

\[
\langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \in \Sigma^\Sigma[\rho'] [T' \subseteq T']
\]

Then \( \varepsilon \vdash \Sigma \vdash \alpha \sim \rho(T') \), and \( \varepsilon \) has to have the form \( \varepsilon = \langle \alpha^{T'}, \hat{T}' \rangle \). Therefore by Lemma 316 \( \langle \hat{T}_1, \alpha^{T'} \rangle \circ \langle \alpha^{T'}, \hat{T}' \rangle = \langle \hat{T}_1, \hat{T}' \rangle \), and then we have to prove that

\[
\langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \in \Sigma^\Sigma[\rho'] [T' \subseteq T']
\]

which we already know, and the result holds.

Case \((2), T = X\). Let \( v = \langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \). Then we know that

\[
\langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \in \Sigma^\Sigma[\rho'] [T' \subseteq T']
\]

and as \( X \) is not free:

\[
\langle \hat{T}_1, \hat{T}' \rangle u :: \rho(T') \in \Sigma^\Sigma[\rho'] [T' \subseteq T']
\]

As \( \langle \hat{T}_1, \hat{T}' \rangle \circ \langle \hat{T}', \alpha^{T'} \rangle = \langle \hat{T}_1, \alpha^{T'} \rangle \), then we have to prove that

\[
\langle \hat{T}_1, \alpha^{T'} \rangle u :: \alpha \in \Sigma^\Sigma[\rho'] [X \subseteq X]
\]

which is equivalent to prove that

\[
\langle \hat{T}_1, \alpha^{T'} \rangle u :: G' \in \Sigma^\Sigma[\rho'] [T' \subseteq G']
\]

where \( T' \subseteq G' \). But the result holds immediately by premise and Lemma 325 using \( \langle \hat{T}''', \hat{T}'' \rangle \vdash \Sigma; \Delta \vdash T' \sim G \), where \( \hat{T}''' = \text{lift}_\Sigma(T') \).
D.7 A Cheap Theorem in GSF

Definition 135. Let $XMode(t, \alpha)$ a predicate that holds if and only if in each evidence of term $t$, if $\alpha$ is present as a free type name, then it appears in both sides of the evidence and in the same structural position. This predicate is defined inductively as follows:

\[
\frac{\forall \varepsilon \in t, XMode(\varepsilon, \alpha)}{XMode(t, \alpha)}
\]

where

\[
\frac{XMode((\alpha^E, \alpha^E), \alpha)}{XMode(\langle E_1, E_3 \rangle, \alpha) \land XMode(\langle E_2, E_4 \rangle, \alpha)}
\]

\[
\frac{\alpha \notin FTN(E_1) \cup FTN(E_2)}{XMode((\langle E_1, E_2 \rangle, \alpha))}
\]

\[
\frac{XMode(\langle E_1, E_3 \rangle, \alpha) \land XMode(\langle E_2, E_4 \rangle, \alpha)}{XMode((\langle E_1 \rightarrow E_2, E_3 \rightarrow E_4 \rangle, \alpha))}
\]

\[
\frac{XMode((\langle E_1, E_2 \rangle, \alpha))}{XMode((\forall X. E_1, \forall X. E_2), \alpha)}
\]

Lemma 327. $\forall W \in S[\exists], \rho, \gamma.((W, \rho) \in D[\Delta] \land (W, \gamma) \in G_\rho[\Gamma])$, such that $\forall v \in cod(\gamma_i)$, $XMode(v, \alpha)$. If $XMode(\rho(\gamma_i(t_i)), \alpha)$, then $\Xi \triangleright \rho(\gamma_i(t_i)) \iff \Xi' \triangleright t_i'$ and $XMode(t', \alpha)$

Proof. By induction on the structure of $t_i$. The proof is direct by looking at the inductive definition of construction of evidences (interior), noticing that $\forall G, g(X, G) = g(G, X) = \langle X, X \rangle$. Then by inspection of consistent transitivity we know that, for any evidence of a value $\langle E_1, E_2 \rangle$

\[
\langle E_1, E_2 \rangle \circ (\alpha^E, \alpha^E) = \langle E'_1, \alpha^E' \rangle \land E'_1 \neq \alpha^* \iff E_2 = \alpha^{E''} \land E_1 \neq \alpha^*
\]

but if that is the case $\neg(XMode(\langle E_1, E_2 \rangle, \alpha))$, which contradicts the premise.

Theorem 58. Let $v \triangleq \Lambda X. \lambda x : ?. t$ for some $t$.
If $\vdash v : \forall X.? \rightarrow X$, then for any $\vdash v' : G$, we either have $v [G] v' \Downarrow \text{error}$ or $v [G] v' \Uparrow$.

Proof. Let $\vdash v \bowtie v_\gamma : \forall X.? \rightarrow X, \vdash v' \bowtie v_\gamma : ?$. Because $\vdash v_\gamma : \forall X.? \rightarrow X$ and $\vdash v_\gamma : ?$, by the fundamental property (Theorem 53) we know that

\[
(W_0, v_\gamma, v_\gamma) \in \psi[\forall X.? \rightarrow X]
\]

\[
(W_0, v_\gamma, v_\gamma) \in \psi[?]\]

Let $v_\gamma = \varepsilon(\Lambda X.(\lambda x : ?. t)) :: \forall X.? \rightarrow X$, where $\varepsilon \vdash \vdash \vdash \vdash \forall X.? \rightarrow X \sim \forall X.? \rightarrow X$, and therefore $\varepsilon = \langle \forall X.? \rightarrow X, \forall X.? \rightarrow X \rangle$.

Note that by the reduction rules we know that

\[
\Xi \triangleright v_\gamma [G] \rightarrow^* \Xi_1 \triangleright \varepsilon_2(\lambda x : ?. t') :: ? \rightarrow \alpha) :: ? \rightarrow G
\]
for some \( t', \) where \( \varepsilon_1 = \langle \alpha^E, ? \to E \rangle, \varepsilon_2 = \langle \alpha^E, ? \to \alpha^E \rangle, \ E = \text{lift}(G), \Xi'_1 = \Xi, \alpha = G. \)

By definition of \( \forall X. ? \to X \) if we pick \( G_1 = G_2 = G, \) and some \( R, \) then for some \( W_1 \) we know that \( (W_1, v_1, v_2) \in \mathcal{V}_X \to \alpha \Xi, \) where \( v_1 = \varepsilon_2(\lambda x : ?. t' :: ? \to \alpha). \)

Also, by the reduction rules we know that \( \Xi'_1 \vdash (\varepsilon_1 v_i :: ? \to G) v_7 \iff \Xi'_1 \vdash cod(\varepsilon_1)(v_i (dom(\varepsilon_1)v_7 :: ?)) :: G. \) As \( \text{dom}(\varepsilon_1) = \langle ?, ? \rangle, \) then \( \Xi'_1 \vdash \text{dom}(\varepsilon_1)v_7 :: ? \iff \Xi'_1 \vdash v_7 :: ?. \) As \( \alpha \notin \text{FTN}(v_7), \) then \( \text{XMode}(v_7, \alpha). \) Also we know that \( \text{XMode}(v_i, \alpha). \) Then by Lemma 327, if \( \Xi'_1 \vdash t'[v_i] \longrightarrow^* v', \) then \( \text{XMode}(v', \alpha), \) but that is a contradiction because if \( (W_4, v', v') \in \mathcal{V}_\alpha, \) then \( \neg \text{XMode}(v', \alpha) \) and the result holds. \( \square \)