UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

GRADUAL REFINEMENT TYPES

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

NICOLÁS EMILIO LEHMANN MELÉNDEZ

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
ALEXANDRE BERGEL
LUIS MATEU BRULÉ
ISMAEL FIGUEROA PALET

Este trabajo ha sido parcialmente financiado por CONICYT-PCHA Magíster Nacional 2015-22150894

SANTIAGO DE CHILE
MARZO 2017

Resumen

Los tipos refinados (Refinement Types) son una técnica efectiva de verificación basada en lenguajes que extienden la expresividad de los sistemas de tipos tradicionales incluyendo la posibilidad de restringir valores usando predicados lógicos. Sin embargo, como cualquier otra disciplina de tipos expresiva, obligan a los programadores a lidiar con las estrictas restricciones impuestas por los tipos. Esto puede resultar ser demasiado tedioso para los programadores, especialmente en etapas tempranas de desarrollo donde el código cambia constantemente. Nosotros conjeturamos que esta rigidez no deseada puede obstaculizar la adopción de los tipos refinados.

Los tipos graduales (Gradual Typing) permiten combinar la flexibilidad de los lenguajes dinámicamente tipados con las garantías otorgadas por los lenguajes estáticamente tipados. Usando tipos graduales, los programadores pueden comenzar un desarrollo con código chequeado completamente de forma dinámica y aprovechar la flexibilidad de las construcciones idiomáticas típicas de los lenguajes dinámicos. A medida que el código se vuelve más estable, el programador o la programadora puede escoger verificar estáticamente ciertas porciones del programa, haciendo precisos los tipos en esos puntos. Finalmente, si lo desea, también puede decidir verificar el código completamente de forma estática. Los sistemas de tipos graduales aseguran una transición suave entre estos distintos niveles de precisión.

Esta tesis demuestra como los tipos graduales pueden ser extendidos para soportar los tipos refinados, permitiendo una evolución suave además de interoperabilidad entre tipos simples y tipos lógicamente refinados. Al hacer esto, se atacan dos desafíos previamente inexplorados en la literatura sobre tipos graduales: lidiar con información lógica imprecisa y con la presencia de tipos dependientes. El primer desafío condujo a una noción crucial de localidad para fórmulas, mientras que el segundo desafío terminó en la definición de novedosos operadores relacionados con substitución al nivel de tipos y términos, que identifican nuevas oportunidades para errores en tiempo de ejecución en lenguajes graduales con tipos dependientes.

Abstract

Refinement types are an effective language-based verification technique, extending the expressiveness of traditional type systems by allowing values to be constrained with logical predicates. However, as any other expressive typing discipline, they force programmers to deal with the strict restrictions imposed by types. This can be too restrictive for programmers, especially in early stages of development where a lot of prototyping happens. We conjecture that the undesired rigidity imposed by refinement types can hamper their wider adoption.

Gradual typing is an approach to combine the flexibility of dynamically typed languages with the safety of statically typed ones. With gradual typing, programmers can begin with a program that is checked fully dynamically using the flexibility of common dynamic idioms. As the code becomes more stable, the programmer can choose to statically verify some critical portions of the program by making precise the types at those points and ultimately verify the whole program statically. The gradual type system ensures a smooth transition between these different levels of type precision.

This thesis demonstrate how the idea of gradual typing can be extended to accommodate refinement types, allowing smooth evolution and interoperability between simple types and logically refined types. In doing so, we address two challenges unexplored in the gradual typing literature: dealing with imprecise logical information, and with dependent function types. The first challenge leads to a crucial notion of locality for refinement formulas, and the second yields novel operators related to type- and term-level substitution, identifying new opportunity for runtime errors in gradual dependently-typed languages.

A mis padres, a quienes debo esto y mucho más

Agradecimientos

A mi familia por todo su cariño y apoyo, en especial a mis padres, Claus y Fancy, quienes inculcaron en mi lo necesario para poder realizar este trabajo. Sin su formación esto no hubiese sido posible.

A mis amigos y a todos los que me acompañaron durante mi recorrido por la Universidad, por prestar siempre su apoyo, y por su compañía junto a un vaso de cerveza en los momentos oportunos.

A todos los que me apoyaron durante esta investigación. Gracias a Éric por enseñarme las maravillas de los lenguajes de programación, por su confianza y constante interés durante mi trabajo. A mis compañeros de PL por sus consejos y su interés al escuchar mis relatos durante nuestras reuniones semanales. También un especial agradecimiento a Jorge quien en una larga discusión me dio, sin saberlo, una definición importante para mi trabajo.

Finalmente, un especial agradecimiento a Mammal Hands, pues su música fue fundamental para mantenerme sereno y darme la inspiración necesaria para completar el trabajo a tiempo antes de enviarlo a POPL.

Contents

Chapter 1

Introduction

Software development is hard. Tight time constraints, coordination of large development groups, unexpected changes in requirements from clients, together with the fact that any line of code is a potential point of failure, rise software industry to the top in complexity. Tremendous amounts of resources are spent every year to guarantee the quality of software, but even though, is common to find bugs in almost any program.

Different programming languages offer different features to ease development, and we have many at our disposal, each being suitable for a particular task. For instance, *dynamic* programming languages like Python or Javascript grant agility and flexibility, which programmers can leverage to rapidly develop and test new features. This turns out very useful at early stages of development when a lot of prototyping is done. However, as a project grows in complexity, code written in those dynamic languages becomes fragile and difficult to maintain, consequence of the lack of static guarantees.

On the contrary, *statically typed* programming languages like Java, Scala or C# offer some guarantees about the correct execution of programs. These programming languages force programmers to correct a certain kind of errors before the code is actually executed, precluding the possibility of failures at runtime due to errors like adding values that are not numbers. These statically typed languages provide more reliable software at the expense of extra effort transferred to programmers who have to deal with the strict requirements imposed by the language.

Dealing with the requirements of statically typed languages can be too restrictive in early stages of development. Maintaining millions of line of dynamic code could become a really hard task. It is never clear which path to follow and the software industry repeatedly face this dilemma. A notable example is the case of Facebook, which spent significant amounts of resources to migrate its codebase, written in PHP, to an entirely new ecosystem for development. This journey ended up in the creation of a new programming language with (some) static typing called Hack and a sophisticated virtual machine with an optimizing just-in-time compiler called HHVM. Certainly we do not want, nor have the resources, to develop a new programming language whenever things become difficult.

The idea of combining dynamic and static features in the same language, gaining the best of both worlds, has been largely studied in academia. Even the industry has already realized that this integration is essential to ease development and today we can find many tools pointing in that direction. There are notorious examples proposed by leading companies: the Dart programming language developed by Google, Microsoft's TypeScript and the already mentioned Hack proposed by Facebook.

A prominent line of research concerned about the integration of static and dynamic features is *gradual typing* [32]. The idea of gradual typing is to provide a smooth evolution between dynamically and statically typed code. Thus, programmers can obviate types in some portions of programs gaining flexibility there and choose to annotate types in critical portions requiring stronger static guarantees. By doing this, the programmer can fine tune the level of static enforcement she is willing to deal with. Some of the ideas blended by the static-dynamic languages already mentioned can be tracked back to gradual typing.

Being able to smoothly transition from Python to Java, but actually remaining in the same language, is a powerful idea. Alas, not powerful enough. Despite offering useful guarantees, traditional programming languages cannot reason about other extremely common errors like dividing by zero or accessing an incorrect index in an array.

Refinement types [34, 30, 35] are a recent and prominent line of research, which allow reasoning about common errors not handled by traditional systems. The basic idea of refinement types is enrich traditional systems with the possibility to specify tighter restrictions on values using *logical formulas*. In traditional programming languages, we are able to restrict a variable to only contain values of type Int. In programming languages with refinement we can enrich this information to say, for example, that besides being an integer the variable can only contain values less than 10. We do this by giving the variable the type $\{\, \nu : \mathsf{Int} \mid \nu < 10 \,\}$. In general the type $\{\, \nu : B \mid p \,\}$ intuitively corresponds to the values of type B that also satisfy the constraint p.

More interestingly, refinement types can be used to give more expressive preconditions for functions than those allowed by traditional systems. For instance, traditional systems could have a division operator ensuring it can only be used with integers and also that it produces an integer as a result. We denote this with the type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$ saying that it is a function taking two arguments of type Int and returning an Int. In a system with refinement types we could give this operator a much more specific type saying that the second argument must be non-zero: $\mathsf{Int} \to \{\, \nu : \mathsf{Int} \mid \nu \neq 0 \,\} \to \mathsf{Int}$. By using this type, the system will guarantee that no "division-by-zero" errors are raised at runtime.

Gradual typing for simple typing disciplines has been largely studied. Recently, some effort has also been done to extend the idea to richer system like type-and-effects [2] and security typing [15]. The goal of this thesis is to push the current barrier of gradual typing and allow the transition to even stronger typing disciplines. Particularly, we focus in the transition between a simply-typed language and a language with refinement types. The result is a gradually-typed language with refinement types that we call *gradual refinement types*.

1.1 Overview of Gradual Refinement Types

Our intent is to develop a gradual language that ranges from simple types to logically-refined types. As such, the imprecision of the system is reflected in the logical information expressed in refinements for which we introduce the notion of *gradual formulas*. This brings the usability benefits of gradual typing to refinement types. First, gradual refinement types accommodate flexible idioms and interaction between refined and unrefined code. For instance, assume the following external library functions:

$$\begin{aligned} \mathit{get} &:: \{\, \nu : \mathsf{Int} \mid \nu > 0 \,\} \to \mathsf{Int} \\ \mathit{check} &:: \mathsf{Int} \to \mathsf{Bool} \end{aligned}$$

The function *get* has a refined signature ensuring it can only be called with strictly positive arguments. On the contrary, the function *check* just has a simply-typed signature and it does not say anything about the relation between its input and output. However, knowing the expected behavior of *check*, namely, that it checks if its argument is positive, the programmer knows that depending on whether the result is true or false it is possible to gain some knowledge about the input. Because this relation between input and output is not reflected in the type signature, a static refinement type system cannot accept natural code like the following:

$$\textbf{if}\ \mathit{check}(x)\ \textbf{then}\ \mathit{get}(x)\ \textbf{else}\ \mathit{get}(-x)$$

Traditionally, the absence of a refinement is interpreted as the absence of static knowledge denoted by the trivial formula $\top$. This can be made explicit in the following signature for *check*.

$$\mathit{check} :: \{\, \nu : \mathsf{Int} \mid \top \,\} \to \{\, \nu : \mathsf{Bool} \mid \top \,\}$$

By refining the result type with $\top$ we are assuming that there is no special information about the connection between the input and the output. But the relation is actually there, we just fail to transfer it to the type system. In general, lack of knowledge in types can be due to simple type annotations, or to the limited expressiveness of the refinement logic.

With gradual refinement types we can understand the absence of a refinement as an *imprecise* logical annotation. By using the imprecise gradual formula $?$ we could give *check* the following imprecise type signature:

$$\mathit{check} :: \{\, \nu : \mathsf{Int} \mid ? \,\} \to \{\, \nu : \mathsf{Bool} \mid ? \,\}$$

Note that instead of saying that there is no additional logical information about the return type, we are just saying that the information is not yet fully known. This imprecision can be exploited by the gradual system to optimistically accept the previous code, subject to dynamic checks ensuring the correct use of the *get* function at runtime. In practice a language could provide a way to gradually import statically annotated code or provide a compilation flag to treat the absence of a refinement as the unknown formula instead of $\top$.

Gradual refinement types also support a smooth evolution on the way to static refinement checking. For instance, consider the challenge of using an existing library with a refined typed interface:

$$\begin{aligned} f &:: \{\, \nu : \mathsf{Int} \mid \nu < 0 \,\} \to \mathsf{Bool} \\ g &:: \{\, \nu : \mathsf{Int} \mid \nu < 10 \,\} \to \mathsf{Int} \end{aligned}$$

One can start using the library without worrying about refinements:

$$\textbf{let}\ ipa\ (x\colon \{\,\nu\!:\!\mathsf{Int} \mid ?\,\}) = \textbf{if}\ f(x)\ \textbf{then}\ 1/x\ \textbf{else}\ g(x)$$

Based on the unknown refinement of x, all uses of x are statically accepted, but subject to runtime checks. Clients of g have no static requirement beyond passing an Int. The evolution of the program can lead to strengthening the type of x to $\{\,\nu\!:\!\mathsf{Int} \mid \nu > 0 \wedge ?\,\}$ forcing clients to statically establish that the argument is positive. In the definition of g, this more precise gradual type pays off: the type system definitely accepts $1/x$, making the associated runtime check superfluous, and it still optimistically accepts $b(x)$, subject to a dynamic check. It now, however, definitely rejects $a(x)$. Replacing $a(x)$ with $a(x-2)$ again makes the type system optimistically accept the program. Hence, programmers can fine tune the level of static enforcement they are willing to deal with by adjusting the precision of type annotations, and get as much benefits as possible (both statically and dynamically).

1.2 Deriving Gradual Refinement Types

This work formalizes a core language with gradual refinement types and prove its crucial metatheoretical properties. Starting from a simple refinement types system we systematically derive the gradual refinement types system guided by the Abstracting Gradual Typing methodology [18]. The gradual language we present is type safe, type sound, and satisfies the refined criteria for gradually-typed languages of Siek et al. [33].

After reviewing the required background to understand this work (Chapter 2), we formalize the simple refinement type system used as the basis for our gradual system (Chapter 3). In Chapter 4 we review what counts as valid source programs and define a static semantics for the gradual refinement type system. The examination of the static semantics leads us to the most important contribution of this work, which is discussed in detail in Chapter 5 when we give an exact meaning to the crucial notion of gradual formula. Later in Chapter 6 we define the execution of gradual refinement types. Finally, in Chapter 7 we discuss the pragmatics of the resulting system and some considerations that must be studied to target a practical implementation.

At the time of writing, the work presented here has been accepted for publication at POPL 2017. As such, the most important results are included there and many of the text written here is taken verbatim from the paper. Nevertheless, we expand in some details that for space restrictions do not fit in the conference paper.

Chapter 2

Background

In this chapter we introduce several concepts and conventions to understand our work. We assume familiarity with these ideas throughout the rest of the document. The already informed reader may choose to skip this chapter, or to directly jump to those sections that result unfamiliar.

2.1 Language Semantics and Type Systems

The formal study of programming languages is based on the construction of models allowing us to capture the essence of the features under study. This section presents three fundamental tools to model the behavior of programming languages. First, in Section 2.1.1, we introduce the λ-calculus as a clean notion to define *programs*. Later, in Section 2.1.2, we refer to the semantics of programming languages by giving an *operational semantics* to define the evaluation of programs. Finally, we introduce *type systems* in Section 2.1.3 as a mechanism to statically prevent the occurrence of some errors at runtime.

2.1.1 The Lambda Calculus

Most modern programming languages feature primitive values, operators and functions. In order to reason about these common key features it is customary to define an abstraction that captures these ideas independently.

The λ-calculus was introduced by Church as a clean and elegant formalization of the notion of computation [10, 9], where the only constructs are functions and function application. The λ-calculus captures the notion of computation in the sense that it has been proven to be equivalent to the Turing machine, but it features a more comfortable level of abstraction closer to the way we understand high level programming languages.

In the λ-calculus, an abstraction or a lambda, denoted $(\lambda x.\ t)$, consists in a parameter

x and a body expression t. The parameter x is an identifier that can be used as a variable in the body of the abstraction. Applications are denoted $t_1\ t_2$ and their semantics is given by replacing any free occurrence of x by t_2 inside the body expression of t_1. Formally, the syntax of valid expressions[1] in the lambda calculus can be defined in Backus-Naur Form as follows:

$$\begin{array}{lcll} v & := & (\lambda x.\ t) & \text{(Values)} \\ t & := & v \mid x \mid t\ t & \text{(Terms)} \end{array}$$

Different kinds of values (booleans, integers, etc.) and operators (addition, multiplication, etc.) can be represented by encoding them with abstractions. In fact, any computable function can be expressed just using abstractions and applications. However, as such, the lambda calculus is too low-level to match the characteristics of real programming languages, therefore basic values are often introduced explicitly. For example, we extend the lambda calculus with natural numbers and boolean values.

$$\begin{array}{lcll} n & := & 0 \mid 1 \mid 2 \mid \ldots & \text{(Numbers)} \\ b & := & \mathsf{true} \mid \mathsf{false} & \text{(Boolean Values)} \end{array}$$

Similarly, operators over these base values can be introduced. For example, for natural numbers it is interesting to introduce operations like addition or multiplication. Finally, programming languages often features control flow statements like **`if`** expressions, which can also be introduced explicitly in the language.

$$\begin{array}{lcll} v & := & (\lambda x.\ t) \mid n \mid b & \text{(Values)} \\ t & := & v \mid x \mid t\ t \mid t + t \mid t \times t & \text{(Terms)} \\ & & \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t & \end{array}$$

2.1.2 Operational Semantics

We have seen what counts as a valid expression of the lambda calculus. It now remains to see how expressions evaluate. A small-step operational semantics specifies how a language performs steps of evaluation. We introduce this operational semantic by presenting a set of evaluation rules. Each rule indicates which expression can make a step of evaluation and which expression it evaluates to.

Rules defining an operational semantics specify a transition relation, here denoted $\hookrightarrow$, which associates expressions with the expressions they evaluate to. The relation $\hookrightarrow$ specifies one step of evaluation, the transitive and reflexive closure of this relation ($\hookrightarrow^*$) is used to define the whole evaluation of a program.

Perhaps the simplest reduction rule is when we have an expression of the form $n_1 + n_2$ where n_1 and n_2 are already integers. In this case, we just need to compute the addition between n_1 and n_2. For instance, the expression $1 + 3$ reduces to 4. This can be expressed

[1] We use expression or term interchangeably to denote valid syntactic elements of a language.

with the following rule where we denote the usual math addition with the symbol $\oplus$ to differentiate it from the language symbol $+$.

$$\text{(E-add)}\frac{n_1 \in \mathbb{Z} \quad n_2 \in \mathbb{Z} \quad n_3 = n_1 \oplus n_2}{n_1 + n_2 \hookrightarrow n_3}$$

Suppose now we have an expression $t_1 + t_2$ where neither t_1 nor t_2 is a number. Before computing the addition we have to evaluate t_1 and t_2 to some values. If we assume expressions are evaluated from left to right, we must first evaluate t_1 until a value is reached. Then, once we have already reached a value in the evaluation of t_1, we can proceed to the evaluation of t_2.

$$\text{(E-add-1)}\frac{t_1 \hookrightarrow t_1'}{t_1 + t_2 \hookrightarrow t_1' + t_2} \qquad \text{(E-add-2)}\frac{t_2 \hookrightarrow t_2'}{v_1 + t_2 \hookrightarrow v_1 + t_2'}$$

Note that the rule (E-add-2) applies only when the first expression is already a value.

An important evaluation step in the lambda calculus is application, defined for expressions of the form $t_1\ t_2$. If t_1 is a λ abstraction $(\lambda x.\ t')$ the expression evaluates to an expression where every free occurrence of x in t' has been replaced by t_2. We use the notation $t_1[{}^{t_2}/_{x}]$ to refer to the expression where every free occurrence of x in t_1 has been replaced by t_2.

Programming languages adhere to a specific evaluation strategy which specifies the order in which expressions are evaluated. Previously, we have assumed a left-to-right evaluation regime for the evaluation of expressions of the form $t_1 + t_2$. Another important aspect of an evaluation strategy is when to perform reductions for application expressions. A call-by-value strategy means that, in an application, both expressions should be evaluated to a value before replacing into the body expression. Thus, we first try to evaluate t_1 to a value and then proceed to evaluate the expression t_2.

$$\text{(E-app-1)}\frac{t_1 \hookrightarrow t_1'}{t_1\ t_2 \hookrightarrow t_1'\ t_2} \qquad \text{(E-app-2)}\frac{t_2 \hookrightarrow t_2'}{v_1\ t_2 \hookrightarrow v_1'\ t_2} \qquad \text{(E-app)}\frac{}{(\lambda x.\ t_1)\ v_2 \hookrightarrow t_1[{}^{v_2}/_{x}]}$$

Note that many syntactically valid expressions do not evaluate to any expression. For example, the expression $\mathsf{true} + 3$ is not related to any other expression by the reduction relation. Another example is when we reach an application where the value at function position is not a λ abstraction, *e.g.*, $(5\ \mathsf{true})$. Reaching such an expression means that the evaluation gets stuck and in essence corresponds to an error in the original program.

2.1.3 Type Systems

A type system is a method to statically analyze programs before they are executed. They are used to prevent the occurrence of some kinds of errors at runtime. The kind of errors a type system prevents varies depending on the intent and the complexity of the type system, but in general it (at least) includes the use of an operator on arguments for which it is not defined, and trying to use a non-function value as if it were a function.

Type systems are syntactically defined by a set of inference rules. An inference rule associates a *type* to a group of language expressions related by their syntax. Intuitively, a type associated with an expression represents the range of values the expression can evaluate to. Simple examples of types are Int, for values that are integers, and Bool for booleans.

An inference rule is composed of a set of premises and a conclusion associating a type to a given expression. The conclusion expresses this association using a typing judgment $e : T$, which can be read as "e has type T". To illustrate consider the set of expressions that are literal integers. As they are already integer we associate to them the type Int. We can write this with the following inference rule:

$$\frac{n \in \mathbb{Z}}{n : \mathsf{Int}}$$

Going one step further we could also give a type to expressions using the addition operator:

$$\frac{t_1 : \mathsf{Int} \qquad t_2 : \mathsf{Int}}{t_1 + t_2 : \mathsf{Int}}$$

The above inference rule just states that given two expressions of type Int the expression formed by applying the addition operator to both is also of type Int.

Function Types

Typically, type systems are also able to track and reason about function types, which allows, for example, to ensure that functions are always called with suitable arguments. The simply typed lambda calculus (STLC) [10] is a restriction of the lambda calculus that allows tracking the type of functions. In STLC, the parameter of a λ abstraction is annotated with a type, which restricts the valid arguments:

$$v \;\; := \;\; (\lambda x{:}T.\; t) \mid n \mid b \quad \text{(Values)}$$

The type of a function corresponds to the domain of values it expects plus the range of values it returns. Function types are denoted $T_1 \to T_2$, where T_1 corresponds to the domain and T_2 to the range. For instance, a function expecting integers and returning booleans has type $\mathsf{Int} \to \mathsf{Bool}$. Formally, we define the syntax of types including base types and function types recursively as follows:

$$T \;\; := \;\; \mathsf{Int} \mid \mathsf{Bool} \mid T \to T$$

Then, we need to define a rule associating an appropriate type to functions. Consider the function $(\lambda x{:}\mathsf{Int}.\; x + 1)$ which expects an integer and adds 1 to it. Intuitively, this function returns an Int because the expression $x + 1$ has type Int if x is replaced by an Int. More generally, a function $(\lambda x : T.\; t)$ returns a value with the type of t if we assume that x has type T. In order to track types assumed for variables we use *type environments*, denoted Γ. Type environments are partial functions from identifiers to types and they serve as context for the typing relation. The typing relation is extended to accept this context and it is now denoted $\Gamma \vdash t : T$. This notation can be read as "t has type T under Γ".

Having a context we are now able to give type to identifiers by just extracting the type information from the context. This is expressed by the following inference rule where $\Gamma(x)$ denotes the type associated to x in the environment Γ:

$$(\text{Tvar})\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

We are now able to define the inference rule that gives types to λ abstractions.

$$(\text{T}\lambda)\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x{:}T_1.\ t) : T_1 \to T_2}$$

This rule states that an abstraction $(\lambda x{:}T_1.\ t)$ has type $T_1 \to T_2$ in an environment Γ if the body t has type T_2 in an extended environment where x is bound to T_1.

Finally, we define an inference rule for applications, which checks that the expression at argument position has the appropriate type.

$$(\text{Tapp})\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash (t_1\ t_2) : T_2}$$

In words, given an expression t_1 of type $T_1 \to T_2$ and an expression t_2 of type T_1, the application $(t_1\ t_2)$ is well typed and the result of evaluating it has type T_2 (the return type of the function).

Type Safety

In Section 2.1.2 we have seen some examples in which evaluation of programs gets stuck. The goal of type systems is to statically identify programs that can exhibit this kind of invalid behavior and discard them before they are executed. The crucial property that a type system must satisfy is that programs accepted by it (*i.e.*, well-typed programs) do not get stuck. This property relates the guarantees provided by a type system with the evaluation of programs and it is often called type safety.

As mentioned by Pierce [28] the type safety property can be captured by the combination of two properties, progress and preservation:

- **Progress:** If an expression is well-typed, then it is either a value, or it can perform a step of evaluation.
- **Preservation:** If a well-typed expression performs a step of evaluation, then the resulting expression is also well-typed, and has the same type.

If a well-typed term can always evaluate to some term (progress) and the resulting term preserves the same type (preservation), it means that it will continue to evaluate without getting stuck until a value is reached.

2.2 Refinement Types

Traditional type systems present in mainstream programming languages like Java or C# effectively rule out some simple errors. These errors include adding values that are not numbers or applying values that are not functions. While useful, these type systems are not able to reason about more complex errors often encountered by programmers.

Consider for example an integer division operator, which in traditional type systems could have the type $\mathsf{Int} \to \mathsf{Int} \to \mathsf{Int}$. This type declares that the operator expects two integers as arguments and it returns also an integer. Thus, the type system is able to ensure that a program will never try to divide some values that are not integers. However, a program using this operator could still misbehave if at runtime the actual value provided as second argument is equal to zero. Traditional type systems cannot guarantee that programs will not try to divide by zero and it is the programmer responsibility to verify the absence of this error. Nonetheless, programming languages usually verify at runtime that the division operator is never used with a denominator equal to zero. This allows programming languages like Java or C# to raise an exception when a programs try to divide by zero.

Consider now an array access operator. Traditional type systems could easily guarantee that the operator is always used with indices that are actually integers. However, not every integer is a valid index and quite often programmers have to deal with "index-out-of-range" errors. In programming languages like Java or C# reaching this error will raise an exception, which can then be handled or debugged. This behavior, however, conforms an extra check for every array access imposing a runtime overhead. On the contrary programming languages like C or C++ do not insert runtime checks for array accesses, and hence array accesses do not impose any runtime overhead. Nonetheless, if present, the error will just end up with a *segmentation fault*, if we are lucky, or going silently unnoticed in the worst scenario.

Refinement types are a lightweight form of language-based verification that can guarantee the absence of errors like the ones described above. This allows, for example, to guarantee the correctness of every array access, eliminating the need of any runtime check. The core idea of refinement types is to enrich types with logical predicates. For example, the refinement type $\{\, \nu : \mathsf{Int} \mid \nu > 0 \,\}$ decorates the type Int with the logical predicate $\nu > 0$. This type intuitively corresponds to the set of values of type Int which also satisfies the predicate $\nu > 0$, *i.e.*, the set of positive integers. More generally, a refinement type is denoted with the syntax $\{\, \nu : B \mid p \,\}$ where ν is a special variable, called the *refinement variable*, and p is a logical predicate constraining ν.

Refinement types allow the specification of fine-grained preconditions. For instance, one can assign a refined type to the division operator, requiring that its second argument be non-zero:

$$\mathsf{Int} \to \{\, \nu : \mathsf{Int} \mid \nu \neq 0 \,\} \to \mathsf{Int}$$

Any program that type checks using this operator is guaranteed to be free from division-by-

zero errors at runtime. Consider the following code[2]:

$$\textbf{let } x = 3 \textbf{ in } 1/x$$

At the call site of the division operator the type system must prove that the denominator is non-zero. The type system is able to track that $x = 3$ and consequently prove that the second argument is non-zero. The code is then statically accepted because is guaranteed that the division will always be safe. Consider now the following function definition:

$$\textbf{let } f\ (x\colon \mathsf{Int})\ (y\colon \mathsf{Int}) = 1/(x - y)$$

For this code to be accepted statically, at the use site of the division operator the system must be able to prove $x - y \neq 0$. In the definition of f the only information about x and y is that they are Int, which is insufficient to prove $x - y \neq 0$. Therefore, the code is rejected statically.

2.2.1 Dependent Function Types

Refinement type systems also support *dependent* function types, allowing refinement predicates to depend on prior arguments. For instance, we can recast the definition of f to give it a more expressive type:

$$\textbf{let } f\ (x\colon \mathsf{Int})\ (y\colon \{\,\nu\colon\mathsf{Int} \mid \nu \neq x\,\}) = 1/(x - y)$$

This definition explicitly states that the second argument must be different from the first one. At the use site of the division operator the system is able to use $x \neq y$ as a "premise" that implies $x - y \neq 0$. With this definition the function f is then accepted statically.

Dependent types can also be used to express specific postconditions for functions. We could for example define a function type, that guarantees that it only returns values greater than its argument:

$$x\colon\mathsf{Int} \to \{\,\nu\colon\mathsf{Int} \mid \nu > x\,\}$$

2.2.2 Flow Sensitivity

Another important feature of refinement type systems is that they are flow-sensitive. That is, they can reason about branches in a conditional statement. Suppose that we have two integers x and y and we want to pass them to the function f defined with the dependent signature above. Before passing x and y as arguments for f we have to convince the type system that they are different. A common idiom is to test for equality between x and y before calling f and falling back to a default behavior in case they are equal.

$$\textbf{if } x \neq y \textbf{ then } f\ x\ y \textbf{ else } 1$$

The type system is able to track that $x \neq y$ in the **then** branch and prove the call to f safe. Of course, the type system can also exploit the fact that x is equal to y in the **else** branch, though this is unnecessary in this example.

[2]Through the document we use snippets of an stylished ML like language for illustration, which we found more readable than lambdas.

2.2.3 Subtyping and Verification Conditions

We have seen that refinement type systems can use information in the context to prove properties about values. For example, given a declaration ensuring $x \neq y$ the system is able to prove that $x - y$ is non-zero and consequently prove that it is safe to use it as the second argument for the division operator. To understand it formally it is necessary to recall the Rule (Tapp) of Section 2.1.3.

$$\text{(Tapp)}\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

This rule uses as a premise that the type of the argument should be equal to the domain of the function. However, plain equality is not sufficient in refinement type systems. Consider the following definitions:

$$\textbf{let}\ g\ (x{:}\ \{\,\nu{:}\mathsf{Int} \mid \nu > 0\,\}) = x$$
$$\textbf{let}\ h\ (x{:}\ \{\,\nu{:}\mathsf{Int} \mid \nu > 0\,\})\ (y{:}\ \{\,\nu{:}\mathsf{Int} \mid \nu > x\,\}) = x + g\ y$$

In the body of h, the system must prove that y is a valid argument for g, which should be the case because $y > x > 0$. The type system is able to infer the exact type $\{\,\nu{:}\mathsf{Int} \mid \nu = y\,\}$ for the variable y. Of course $\{\,\nu{:}\mathsf{Int} \mid \nu = y\,\}$ is different from $\{\,\nu{:}\mathsf{Int} \mid \nu > 0\,\}$, thus, the Rule (Tapp) rejects the application.

We must modify the rule (Tapp) to account for information in the context. Formally, instead of using equality, refinement type systems use a contextual *subtyping* relation, denoted $\Gamma \vdash T_1 <: T_2$.

$$\text{(Tapp)}\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash T_2 <: T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}}$$

The subtyping relation uses the logical information in the context to check if a type can be safely used in a place expecting a different type. This subtyping relation crystallize the logical semantics used in the system and it is at the heart of the expressiveness of refinement type systems.

For the previous example the subtyping constraint that must be satisfied is:

$$x{:}\{\,\nu{:}\mathsf{Int} \mid \nu > 0\,\}, y{:}\{\,\nu{:}\mathsf{Int} \mid \nu > x\,\} \vdash \{\,\nu{:}\mathsf{Int} \mid \nu = y\,\} <: \{\,\nu{:}\mathsf{Int} \mid \nu > 0\,\}$$

Intuitively, this boils down to verifying that the set of premises $\{\,x > 0, y > x, \nu = y\,\}$ entails $\nu > 0$. In this case, this can be easily verified by inspection. In general, if we allow arbitrary formulas in refinements, verification conditions can not be resolved automatically since the satisfiability problem is undecidable for arbitrary first-order formulas. However, if formulas are drawn from fragments for which there exist decidable decision procedures, the type system will be able to carry out typechecking automatically. Modern technology on automatic verification has proved powerful enough to be useful in practice [12, 6, 3].

2.3 Satisfiability Modulo Theory

Satisfiability, the problem of checking whether a formula expressing a constraint has a solution, has been fundamental in computer science since its origins. Constraint satisfaction

problems arise in many applications including software and hardware verification, artificial intelligence, test-case generation, among others [5].

Even in their simplest form, constraint satisfaction problems have proven challenging for computer scientists. Well known is the theoretical complexity bound for *propositional satisfiability* SAT, namely the problem of checking if a given formula over boolean values can be made true by choosing true or false values for its variables. Many years of research, and some collaborative engineering effort, have ended up in the creation of many practical tools for solving SAT constraints [22, 20].

Some problems, however, require to be described in more expressive logics such as first-order logic (FOL). We know that the problem of checking the satisfiability of an arbitrary first-order formula is undecidable. Nevertheless, this does not preclude the existence of satisfiability checking procedures that are practical in many use cases, even if there are not complete or are only complete for some fragment of FOL. In fact, much progress has been made in general-purpose (automatic) first-order theorem provers, such as provers based on the resolution calculus [25].

Sadly, formulas required by many applications cannot be typically solved by these provers. The main reason is that these applications are often not concerned about general first-order satisfiability, but satisfiability with respect to some *background theory*. For instance, applications working with integer arithmetic are not interested in the existence of nonstandard interpretations for the arithmetic symbols $<$, $+$, and 0 that makes the formula $x < y \wedge \neg(x < y+0)$ true. Instead, they are only concerned with knowing if the formula is satisfiable in an interpretation where $<$ refers to the usual ordering for integers, $+$ to integer addition and 0 to the additive identity. While general-purpose reasoning can be forced to consider only interpretations consistent with a given background theory $\mathcal{T}$ by explicitly adding the axioms for $\mathcal{T}$, the performance of such provers is often unacceptable. For some background theories, a more viable approach is using satisfiability procedures specialized for those theories.

Satisfiability modulo theories (SMT) [13] is the research field concerned with the satisfiability of formulas with respect to some background theory. Most SMT problems have a high computational complexity, but being tailored for some particular theory, SMT solvers can take advantage of specialized algorithms and data structures suitable for the specific task. Another important aspect in the design of modern SMT decision procedures is that, while problems can in general be complex, they can focus on the more realistic goal of efficiently attacking the problems arising in practice, which may turn out to be considerably less complex than in the general case.

In this section we review the basic formal definitions and conventions related to SMT. Then, we conclude by connecting some useful background theories with the particular application of refinement types.

$$
\begin{array}{rrcl}
\textbf{Terms} & t & ::= & x \mid f(t_1, \ldots, t_n) \\
\textbf{Formulas} & p, q & ::= & \forall x, p \mid \exists x, p \mid p \wedge q \mid p \vee q \mid \neg p \\
 & & & t_1 = t_2 \mid P(t_1, \ldots, t_n)
\end{array}
$$

Figure 2.1: Syntax of first-order formulas.

2.3.1 Formal Definition and Conventions

We consider the usual notions and terminology of first-order logic and model theory. Let Σ be a *signature* consisting of a set of *function* and *predicate* symbols. Each function symbol f and predicate symbol P is associated with a non-negative integer, called its *arity*. We use f, g and h to denote function symbols, P and Q to denote predicate symbols and $x_1, x_2, x_3, \ldots$ to denote variables. We also use pervasively the refinement variable ν, which has a special meaning in our formalization. We use letters p and q to refer to formulas instead of the more commons ϕ and ψ, to stick to the notation used in refinement types [30, 34]. The specific syntax of formulas is shown in Figure 2.1 We write $p(x_1, \ldots, x_n)$ for a formula that may contain variables $x_1, \ldots, x_n$. When there is no confusion we abbreviate $p(x_1, \ldots, x_n)$ as $p(\vec{x})$. When a formula contains the special refinement variable ν we always annotate it explicitly as $p(\vec{x}, \nu)$.

A Σ-structure or model $\mathcal{M}$ consists of a non-empty universe $|\mathcal{M}|$ and a mapping $\mathcal{M}(\cdot)$, assigning to each function symbol f of arity n a function $\mathcal{M}(f) : |\mathcal{M}|^n \to |\mathcal{M}|$, to each predicate symbol P of arity n a function $\mathcal{M}(P) : |\mathcal{M}|^n \to \{\, \mathsf{true}, \mathsf{false} \,\}$ and to each variable x an element $\mathcal{M}(x) \in |\mathcal{M}|$. We omit Σ when it is clear from the context and talk just about a model. Given a model $\mathcal{M}$ we use the standard definition for the interpretation of terms given by $\mathcal{M}[\![x]\!] = \mathcal{M}(x)$ and $\mathcal{M}[\![f(t_1, \ldots, t_n)]\!] = \mathcal{M}(f)(\mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!])$. As is standard, we assume that formulas do not contain free variables. If this is not the case, we extend the signature with a fresh constant for every free variable.

We use $\mathcal{M}[x \mapsto v]$ to denote a structure where the variable x is interpreted as v, and all other variables, function and predicate symbols remain with the same interpretation as in $\mathcal{M}$. Given a formula p and a model $\mathcal{M}$, satisfaction $\mathcal{M} \models p$ is defined as follows:

$$
\begin{array}{lcl}
\mathcal{M} \models p \wedge q & \iff & \mathcal{M} \models p \text{ and } \mathcal{M} \models q \\
\mathcal{M} \models p \vee q & \iff & \mathcal{M} \models p \text{ or } \mathcal{M} \models q \\
\mathcal{M} \models \neg p & \iff & \mathcal{M} \not\models p \\
\mathcal{M} \models \forall x, p & \iff & \mathcal{M}[x \mapsto v] \models p \text{ for all } v \in |\mathcal{M}| \\
\mathcal{M} \models \exists x, p & \iff & \mathcal{M}[x \mapsto v] \models p \text{ for some } v \in |\mathcal{M}| \\
\mathcal{M} \models t_1 = t_2 & \iff & \mathcal{M}[\![t_1]\!] = \mathcal{M}[\![t_2]\!] \\
\mathcal{M} \models P(t_1, \ldots, t_n) & \iff & \mathcal{M}(P)(\mathcal{M}[\![t_1]\!], \ldots, \mathcal{M}[\![t_n]\!]) = \mathsf{true}
\end{array}
$$

We extend satisfaction to sets of formulas: $\mathcal{M} \models \Delta$ if for all $p \in \Delta$, $\mathcal{M} \models p$. A formula p is said to be *satisfiable* if there exists a model $\mathcal{M}$ such that $\mathcal{M} \models p$. A formula p is said to be *valid* if $\mathcal{M} \models p$ for every $\mathcal{M}$. A set of formulas Δ *entails* a formula q if for every model

$\mathcal{M}$ such that $\mathcal{M} \models \Delta$ then $\mathcal{M} \models q$.

We define a theory $\mathcal{T}$ as a collection of models. A formula p is said to be *satisfiable modulo* $\mathcal{T}$ if there exists a model $\mathcal{M} \in \mathcal{T}$ such that $\mathcal{M} \models p$. A formula p is said to be *valid modulo* $\mathcal{T}$ if $\mathcal{M} \models p$ for every $\mathcal{M} \in \mathcal{T}$. A set of formulas Δ *entails a formula* q *modulo* $\mathcal{T}$, notation $\Delta \models_{\mathcal{T}} q$, if for all model $\mathcal{M} \in \mathcal{T}$ such that $\mathcal{M} \models \Delta$ then $\mathcal{M} \models q$.

For conciseness, in the remainder of this document we relax the formal treatment of satisfiability. In particular, we avoid explicit references to the background theory and let it be inferred from the context. The reader should keep in mind, however, that whenever we talk about satisfiability, validity or entailment we are implicitly fixing a background theory.

2.3.2 Background Theories

Perhaps the most known theory is linear arithmetic over integers, also known as Presburger arithmetic. The theory of linear arithmetic $\mathcal{T}_{\mathbb{Z}}$ is defined for the signature $\Sigma_{\mathbb{Z}} = (0, 1, +, -, \leq)$ and the only model in the theory interprets the symbols with their usual meaning for integers. The satisfiability problem for linear arithmetic is full first-order (including quantifiers) decidable. Linear arithmetic has been studied in the first place by Presburger, who proved its decidability by showing that its admits quantifier elimination, that is, for every formula an equivalent one without quantifiers can be found [29].

In the context of refinement types linear arithmetic allows us to encode expressive constrains as discussed in Section 2.2. For instance, we could give the addition operator a type expressing its exact behavior or express that a function only takes even numbers as arguments:

$$\begin{array}{lll} \mathsf{(+)} & :: & x{:}\mathsf{Int} \to y{:}\mathsf{Int} \to \{\, \nu{:}\mathsf{Int} \mid \nu = x + y \,\} \\ \mathsf{apa} & :: & \{\, \nu{:}\mathsf{Int} \mid \nu \,\%\, 2 = 0 \,\} \to \mathsf{Int} \end{array}$$

An immediate question is whether we can extend the theory to reason about arithmetic including multiplication and division. It is well known that the problem becomes undecidable in this case. However, if we interpret symbols in $\Sigma_{\mathbb{Z}}$ over the reals we recover decidablity for the quantifier free fragment. The problem becomes double exponential, though. Nevertheless, modern SMT solvers often turn out to be efficient in practice. They even provide procedures to solve constraints in the integer case, even though these procedures cannot be complete.

Another theory, particularly useful in the context of refinement types, is the theory of uninterpreted functions and linear arithmetic. In this theory, we extend the signature $\Sigma_{\mathbb{Z}}$ with an arbitrary number of uninterpreted function symbols. Different models in the theory give a different interpretation for those uninterpreted function symbols. The satisfiability problem for the quantifier free fragment of this theory is decidable and it has an efficient decision procedure based on computing the congruence closure [27].

The quantifier free theory of uninterpreted functions and linear arithmetic allows us to express constraints about data structures. For example, we can give an array access operator

a type stating that only valid indices can be accessed:

$$a{:}\mathsf{IntArray} \to i{:}\{\,\nu{:}\mathsf{Int} \mid 0 \leq \nu \wedge \nu < \mathit{size}(a)\,\} \to \mathsf{Int}$$

Here, the *size* function in the refinement of the second argument is just an uninterpreted function and it is not connected in any way with the actual meaning of the length of the array. We can introduce this connection by defining a language function computing the size of an array and give it a type connecting it with the logic function *size*:

$$\texttt{size} :: a{:}\mathsf{IntArray} \to \{\,\nu{:}\mathsf{Int} \mid \nu = \mathit{size}(a)\,\}$$

Note that the language function `size` and the (uninterpreted) logical function *size* do not share anything in common, except for the connection established in the previous definition.

Other common theories supported by modern SMT solvers, but not so useful in the context of refinement types, are the theory of bitvectors, the theory of arrays and the theory of data types [12].

2.4 Gradual Typing

Programming languages like Java, C# or Scala have type systems that statically ensure the inexistence of some errors. These type systems force programmers to correct errors before the code is actually executed. Static type checking in these languages provides both documentation and guarantees about the correct execution of programs. However, some of the effort imposing these guarantees is transferred to programmers who have to deal with the strict requirements imposed by types. Fulfilling these requirements can be too restrictive for software developers, especially in early stages of development where a lot of prototyping happens. Dealing with the restriction imposed by types that are not yet final may become an unnecessary overhead.

Many popular programming languages, notably Python and Javascript, do not perform any kind of static verification. These programming languages emancipate programmers from the troubles of dealing with type restrictions, granting flexibility and agility to developers. However, flexibility is given at the cost of more fragile software, consequence of the lack of static checks. Moreover, to ensure safety, these static checks must be delayed to runtime introducing some performance overhead.

Gradual typing [32] is an attempt to combine the flexibility of dynamic typing with the strong guarantees provided by static typing. With gradual typing, programmers can define as dynamic some portions of a program, for example, at an early stage of development, and let these portions be checked dynamically. As the code becomes more stable the programmer can choose to annotate more types in the program, gaining the benefits of static typing at points where types are provided.

2.4.1 Gradual Types

A central insight of gradual typing is the introduction of an unknown type (denoted ?), associated to dynamic portions of programs. Intuitively, the unknown type represents the absence of static knowledge about some type, and the intent of gradual typing is to reason about programs with imprecise type information. Consider a variable x with gradual type $\mathsf{Int} \to ?$. This type conveys *some* information about x —a function from Int to some type— that can be statically exploited. For example, the program $x + 1$ could never be correct because x is already known to be a function, thus, it cannot be added to 1 and the gradual type system definitively rejects this program. Additionally, besides knowing that x is a function, we also known that it expects an integer. Thus, the application $x(1)$ is valid, and the type system definitively accepts it, but $x(\mathsf{true})$ is not, and the type system definitively rejects it. When checking the program $x(1) + 1$, the type system sees that the application $x(1)$ has type ?, thus, it *may be* safe to add 1 to it. However, this is done only optimistically and the type system must insert a runtime check to verify that the presumption is actually correct when the code is executed.

Concretely, gradual types extend the syntax of static types by introducing the unknown type ?.

$$\widetilde{T} \quad := \quad \mathsf{Int} \mid \mathsf{Bool} \mid ? \mid \widetilde{T} \to \widetilde{T} \quad \text{(Gradual Types)}$$

2.4.2 Type Consistency

We have seen that intuitively a variable x of gradual type $\mathsf{Int} \to ?$ cannot be used in the program $x + 1$, but it can be used as a function expecting an integer. But, how can we know formally if an expression is being used correctly? In a statically-typed language the discrimination for the correct use of expressions is mostly driven by equality between types. For instance, an expression of type T can be used as an argument for the addition operator only if T is *equal* to Int. In the presence of gradual types equality is not enough because we expect, for example, to be able to use a variable of type ? as if it were an Int. To accommodate this behavior gradual typing introduces the notion of *type consistency* [32].

Intuitively, two gradual types are consistent if it may be the case that they can be used interchangeably. Take for example our variable x of type $\mathsf{Int} \to ?$. The application $x(1)$ has type ? and consequently it may be possible to use it safely as an Int. Therefore, the program $x(1)+1$ is accepted statically because ? is consistent with Int. On the other hand the program $x + 1$ is always incorrect because it cannot be the case that x can be used safely as an Int. Formally, $\mathsf{Int} \to ?$ is not consistent with Int.

Type consistency is made precise by defining a binary relation denoted $\sim$. An important requirement for a gradual system is that it should be a conservative extension of the underlying static type system: fully annotated terms should be still be typeable in the gradual system. A gradual system should also be able to encode the flexible idioms allowed by dynamic languages. The type consistency relation accommodates these requirements. First, every static type T should be consistent with itself. Second, the unknown type ? should

be consistent with every type. And finally, two function types are consistent if both the domains and ranges are point-wise consistent. This can by concisely captured with the following inductive definition:

$$\frac{}{T \sim T} \qquad \frac{}{T \sim ?} \qquad \frac{}{? \sim T} \qquad \frac{\widetilde{T}_{11} \sim \widetilde{T}_{21} \quad \widetilde{T}_{12} \sim \widetilde{T}_{22}}{\widetilde{T}_{11} \to \widetilde{T}_{12} \sim \widetilde{T}_{21} \to \widetilde{T}_{22}}$$

An important property of this relation is that it is both reflexive and symmetric, but *not* transitive. If transitive, it would be possible to relate any two types through the unknown type ($\widetilde{T}_1 \sim ?$ and $? \sim \widetilde{T}_2$ would imply $\widetilde{T}_1 \sim \widetilde{T}_2$ for all $\widetilde{T}_1$ and $\widetilde{T}_2$).

To define a gradual type system, inference rules for the static system are modified accordingly to use type consistency instead of type equality. For example, the rule for function application is as follows:

$$(\widetilde{\text{T}}\text{app})\frac{\Gamma \vdash t_1 : \widetilde{T}_1 \quad \Gamma \vdash t_2 : \widetilde{T}_2 \quad \widetilde{T}_1 \sim \widetilde{T}_2 \to \widetilde{T}_3}{\Gamma \vdash t_1\ t_2 : \widetilde{T}_3}$$

In words, given an expression t_1 of type $\widetilde{T}_1$ and an expression t_2 of type $\widetilde{T}_2$ the application $t_1\ t_2$ is well-typed if $\widetilde{T}_1$ is consistent with the type of a function expecting $\widetilde{T}_2$.

2.4.3 Casts as Runtime Checks

As presented so far the gradual type system accepts programs that could not be proven definitively safe. These programs are accepted only optimistically and to ensure safety explicit runtime checks must be inserted. These checks are performed as type casts which delay type verification until runtime. If a cast fails the program must throw an error. A cast is denoted $(\langle \widetilde{T}_2 \Leftarrow \widetilde{T}_1 \rangle\ t)$ and it means that even though t statically has type $\widetilde{T}_1$, we can assume that it also has type $\widetilde{T}_2$. This assumption is, however, checked dynamically.

Consider the following program:

$$\begin{array}{l} \textbf{let}\ f\ (x\colon \mathsf{Int}) = x + 1 \\ \textbf{let}\ g\ (x\colon ?) = f\ x \end{array}$$

The type system accepts this program statically because it may be the case that x is an Int in the body of g. However, this is done only optimistically and at runtime it must be verified that the actual argument for f is an Int. To achieve this, an explicit cast is inserted, resulting in the following program:

$$\begin{array}{l} \textbf{let}\ f\ (x\colon \mathsf{Int}) = x + 1 \\ \textbf{let}\ g\ (x\colon ?) = f\ (\langle \mathsf{Int} \Leftarrow ? \rangle x) \end{array}$$

If we call g with an integer the program will run without problems but if g is called with a boolean the program will fail to cast it to an Int and the program will end up with a *cast error*.

2.4.4 Type Precision

An important notion in gradual typing is *type precision*, which intuitively captures how much static information a gradual type contains. For instance, the type $\mathsf{Int} \to \mathsf{Int}$ is more precise than $\mathsf{Int} \to ?$, which in turn is more precise than $?$. Formally, type precision is captured as a binary relation, denoted $\sqsubseteq$, which forms a partial order and can be used to compare the amount of static information between gradual types. The type precision relation is defined inductively as follows:

$$\frac{}{T \sqsubseteq T} \qquad \frac{}{? \sqsubseteq \widetilde{T}} \qquad \frac{\widetilde{T}_{11} \sqsubseteq \widetilde{T}_{12} \quad \widetilde{T}_{21} \sqsubseteq \widetilde{T}_{22}}{\widetilde{T}_{11} \to \widetilde{T}_{12} \sqsubseteq \widetilde{T}_{21} \to \widetilde{T}_{22}}$$

The notion of type precision can be raised to terms and it allows to distinguish gradual typing from other forms of static-dynamic integration: gradual typing allows a smooth transition between programs with different precision. For example, the program $(\lambda x\!:\!?.\ x(1)+1)$ has less precise information than $(\lambda x\!:\!\mathsf{Int} \to ?.\ x(1)+1)$ and gradual typing guarantees that we can smoothly transit from the first program to the second one.

Siek et al. [33] captured what this smooth transition means with two important properties referred to as the static and dynamic gradual guarantees. The first property relates the static behavior of programs which differs only by the precision on their type annotations. Precisely, the *static gradual guarantee* states that weakening the precision of a term preserves typeability at a less precise type:

Proposition 1 (Static gradual guarantee)**.** *If* $\cdot \vdash t_1 : \widetilde{T}_1$ *and* $t_1 \sqsubseteq t_2$*, then* $\cdot \vdash t_2 : \widetilde{T}_2$ *and* $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$*.*

The second property relates the dynamic behavior of programs with different type precision. Specifically, the *dynamic gradual guarantee* states that a well-typed gradual program that runs without errors still does with less precise type annotations:

Proposition 2 (Dynamic gradual guarantee)**.** *Suppose* $\cdot \vdash t_1 : \widetilde{T}_1$ *and* $t_1 \sqsubseteq t_2$*. If* $t_1 \hookrightarrow t_1'$ *then* $t_2 \hookrightarrow t_2'$ *where* $t_1' \sqsubseteq t_2'$*.*

2.5 Abstracting Gradual Typing

For many typing disciplines the notion of type consistency is not enough to accommodate richer features. For instance, object-oriented programming languages pervasively use the notion of subtyping. In these languages the correct use of objects depends on this notion: a function can receive any argument that is a subtype of the declared argument type. Then, the question is how a programming language with subtyping should be extended to support gradual typing. Type consistency, as the counterpart for equality, is arguably easy to follow, however, it seems not so easy to come up with a relation associated with subtyping.

In order to define a gradual language with subtyping Siek and Taha [31] developed the notion of *consistent subtyping* (denoted $\lesssim$). Their definition is conceived in terms of a

restriction operator $T_1|_{T_2}$ that masks off the parts of a type T_1 that are unknown in a type T_2. By using this operator in ad hoc ways Siek and Taha [31] devise the notion of consistent subtyping. There have been many attempts to extend the idea of gradual typing to other richer typing disciplines like security typing [14, 15]. These attempts have, however, been guided primarily by the intuition and supported by ad hoc justifications.

In an attempt to establish steadier foundations for gradual typing, Garcia et al. [18] developed Abstracting Gradual Typing (AGT). AGT is a methodology to systematically derive the gradual counterpart of a static typing discipline by viewing gradual types through the lens of abstract interpretation [11]. The key insight of AGT is seeing a gradual type as the abstraction of a set of static types. For instance, the unknown type ? represents the set of all types, the function type $\mathsf{Int} \to ?$ the set of all function with Int as the domain, and the gradual type Int just represents the singleton set containing Int. Formally, being TYPE the set of static types and GTYPE the set of gradual types, we can define a *concretization function* $\gamma : \text{GTYPE} \to \mathcal{P}(\text{TYPE})$, which relates each gradual type to the set of static types it represents.

$$\begin{aligned} \gamma(T) &= \{\, T \,\} \\ \gamma(?) &= \text{TYPE} \\ \gamma(\widetilde{T_1} \to \widetilde{T_2}) &= \{\, T_1 \to T_2 \mid T_1 \in \gamma(\widetilde{T_1}) \text{ and } T_2 \in \gamma(\widetilde{T_2}) \,\} \end{aligned}$$

Type Precision

The definition of precision in Section 2.4.4 is guided by intuition. AGT, instead, proposes a direct way to define the precision relation based on concretization: a gradual type $\widetilde{T_1}$ is more precise than a type $\widetilde{T_2}$ if it represents a smaller set of static types.

Definition 1 (Type Precision)**.** *$\widetilde{T_1}$ is less imprecise (*i.e.*, more precise) than $\widetilde{T_2}$, notation $\widetilde{T_1} \sqsubseteq \widetilde{T_2}$, if and only if $\gamma(\widetilde{T_1}) \subseteq \gamma(\widetilde{T_2})$.*

Interestingly, this definition is equivalent to the definition in Section 2.4.4, but has the advantage to be derived directly from the definition of concretization. And moreover, the same idea can be used to directly define precision in systems with more complex features.

2.5.1 Lifting Predicates to Gradual Types

With the interpretation given for gradual types, a natural definition of type consistency follows. Adhering to the plausibility interpretation of gradual typing two types can be said to be consistent if there are some types in their concretizations that are equal. Formally:

$$\widetilde{T_1} \sim \widetilde{T_2} \text{ if and only if } T_1 = T_2 \text{ for some } T_1 \in \gamma(\widetilde{T_1}) \text{ and } T_2 \in \gamma(\widetilde{T_2})$$

Garcia et al. [18] proved this definition coincides with the inductive definition of consistency in Section 2.4.2. Nonetheless, this definition has the advantage of being formulated inside a clear framework instead of being justified by intuition only. More compelling, this idea can

be used to lift any other predicate on static types to a *consistent predicate* on gradual types. Thus, a natural definition for consistent subtyping follows:

$$\widetilde{T_1} \lesssim \widetilde{T_2} \text{ if and only if } T_1 <: T_2 \text{ for some } T_1 \in \gamma(\widetilde{T_1}) \text{ and } T_2 \in \gamma(\widetilde{T_2})$$

Interestingly, Garcia et al. [18] also proved that this definition is equivalent to the one of Siek and Taha [31].

2.5.2 Lifting Functions on Gradual Types

Besides using type predicates, type systems also rely on (partial) functions on static types. For instance, although somewhat hidden, the rule for application has to extract the domain of a function type to check if it is compatible with the type of the argument. We can recast the rule for application in Section 2.1.3 to make this explicit.

$$\text{(Tapp)}\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad dom(T_1) = T_2}{\Gamma \vdash t_1\ t_2 : T_2}$$

This definition uses the partial function $dom : \textsc{Type} \rightharpoonup \textsc{Type}$, which returns the domain of a function and it is undefined for non-function types.

As predicates on static types are lifted to consistent predicates on gradual types, AGT also proposes to lift functions on static types to corresponding functions on gradual types. In previous work on gradual typing this has been done intuitively to justify for example that ? can be seen as a function type with domain ?. AGT provides a systematic way to do this lifting. The lifting begins by defining a collecting semantics for functions. Given a function on static types $F : \textsc{Type}^2 \rightharpoonup \textsc{Type}$, its collecting lifting is defined as a function $\widehat{F} : \mathcal{P}(\textsc{Type})^2 \to \mathcal{P}(\textsc{Type})$ that operates on sets of static types:[3]

$$\widehat{F}(\widehat{T_1}, \widehat{T_2}) = \{\, F(T_1, T_2) \mid \langle T_1, T_2 \rangle \in \widehat{T_1} \times \widehat{T_2} \,\}$$

This definition can be generalized to any finite arity function $F : \textsc{Type}^n \rightharpoonup \textsc{Type}$. For example, the collecting lifting of the domain function is defined as:

$$\widehat{dom}(\widehat{T}) = \{\, dom(T) \mid T \in \widehat{T} \,\}$$

The collecting lifting applies the original partial function pointwise to the set of static types, ignoring cases where the function is not defined. If the function is undefined for the entire collection the result is just the empty set. For example, $\widehat{dom}(\textsc{Type}) = \textsc{Type}$ because every type appears in the domain of some type.

From here we want to produce a corresponding function for gradual types. We could compose the concretization function γ with the collecting lifting of a function, $\widehat{F} \circ \gamma$. Applying this composition yields a set of static types. To get from here back to a gradual type we define an *abstraction function* $\alpha : \mathcal{P}(\textsc{Type}) \rightharpoonup \textsc{GType}$, which represents the collection of static types as precisely as possible with a gradual type.

[3] By convention, given a static entity X, $\widetilde{X}$ denotes a gradual entity, and $\widehat{X}$ a set of Xs

Given a set of static types, Garcia et al. [18] found a natural function representing the set of static types.

$$\begin{aligned}
\alpha(\{\,\mathsf{Int}\,\}) &= \mathsf{Int} \\
\alpha(\{\,\mathsf{Bool}\,\}) &= \mathsf{Bool} \\
\alpha(\{\,\overline{T_{i1} \to T_{i2}}\,\}) &= \alpha(\{\,\overline{T_{i1}}\,\}) \to \alpha(\{\,\overline{T_{i2}}\,\}) \\
\alpha(\emptyset) &\text{ is undefined} \\
\alpha(\widehat{T}) &= \mathsf{?} \text{ otherwise}
\end{aligned}$$

It may seem arbitrary, but this function actually shares a tight connection with the concretization function previously defined. This connection is expressed in two properties that capture that α is the *best abstraction* possible. These properties are called *soundness* and *optimality*. Soundness ensures that α retains at least all of the types in the collection $\widehat{T}$. Optimality ensures that α always retains as much precision as possible, given the definition of gradual types and their meaning via γ.

Proposition 3 (α is sound)**.** *If $\alpha(\widehat{T})$ is non empty, then $\widehat{T} \subseteq \gamma(\alpha(\widehat{T}))$.*

Proposition 4 (α is optimal)**.** *If $\alpha(\widehat{T})$ is non empty, then $\widehat{T} \subseteq \gamma(\widetilde{T})$ implies $\alpha(\widehat{T}) \sqsubseteq \widetilde{T}$.*

In abstract interpretation terminology [11], a pair of functions satisfying soundness and optimality is called a Galois connection. Galois connections are the basis for defining abstract operators in the context of abstract interpretation, on which AGT relies to define the corresponding lifting for functions. The lifting of a function F in the abstract interpretation framework is just defined as $\alpha \circ \widehat{F} \circ \gamma$.

2.5.3 Deriving the Runtime Semantics

Traditionally, the approach for defining the runtime semantics for gradual languages has been given by the introduction of casts as described in Section 2.4.3. This requires to define an intermediate language with casts, together with a translation from source programs to this intermediate language. AGT, instead, proposes a direct manner to define the runtime semantics exploiting the Curry-Howard correspondence between proof normalization and term reduction [21].

As a term reduces, it is necessary to justify new judgments for the typing derivation of the new term, such as equality. In a type safe static language, these new judgments can always be established, as justified in the type preservation proof, which relies on properties of judgments such as transitivity of equality. However, in the case of gradual typing derivations, these properties may not always hold: for instance the two consistent judgments $\mathsf{Int} \sim \mathsf{?}$ and $\mathsf{?} \sim \mathsf{Bool}$ cannot be combined to justify the transitive judgment $\mathsf{Int} \sim \mathsf{Bool}$.

More precisely, Garcia et al. [18] introduce the notion of *evidence* to characterize *why* a consistent judgment holds. A consistent operator, such as *consistent transitivity*, determines when evidences can be combined to produce evidence for a new judgment. The impossibility to combine evidences so as to justify a combined consistent judgment corresponds to a cast

error: the realization, at runtime, that the plausibility based on which the program was considered (gradually) well-typed is not tenable anymore.

The resulting language derived with AGT satisfies a series of desired properties. First, the language is type safe by construction, but more interestingly it satisfies the gradual guarantees discussed in Section 2.4.4. We elaborate over AGT to derive a gradual refinement type system satisfying the aforementioned properties.

Chapter 3

A Static Refinement Type System

Every gradual language is the counterpart of a language with a static type system. This section presents a core calculus with refinement types, which we call simple refinement types. This core calculus is the base for the derivation of our language with gradual refinement types.

Many presentation of refinement types systems have been proposed in the literature [23, 30, 34, 19]. The formalization presented here take these proposals as a basis and focus on specific details relevant for the subsequent gradualization.

Independently, this presentation of simple refinement types makes the following contributions:

- Following Chugh et al. [8], it provides a formalization direct on logically qualified refinements instead of proving soundness by translation to manifest contracts [30, 34].
- All proofs in the formalization are mechanized using the Coq proof assistant [1, 4].

The material presented in this chapter was independently presented as an extended abstract in CoqPL [24]. The formalization in Coq can be found in the following link: `https://github.com/pleiad/refinements`[1]

3.1 Syntax and Operational Semantics

We begin with the syntax and runtime semantics of simple refinement types. Figures 3.1 shows the syntax of values, expressions, and types.

[1]The formalization in Coq have some inessential differences. First, the Coq development does not make use of the logical environment. Second, the system presented here uses subtyping in an algorithmic style, while the Coq development uses a separate subsumption rule.

$$T \in \textsc{Type}, \quad x \in \textsc{Var}, \quad c \in \textsc{Const}, \quad t \in \textsc{Term},$$
$$f \in \textsc{LogicFun}, \quad p \in \textsc{Formula}$$

$$\begin{array}{rrcl}
\textbf{Values} & v & ::= & \lambda x{:}T.\ t \mid x \mid c \\
\textbf{Terms} & t & ::= & v \mid t\ v \mid \mathsf{let}\ x = t\ \mathsf{in}\ t \\
 & & & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t :: T \\
\textbf{Base Types} & B & ::= & \mathsf{Int} \mid \mathsf{Bool} \\
\textbf{Types} & T & ::= & \{\, \nu{:}B \mid p \,\} \mid x{:}T \to T \\
\textbf{Formulas} & p & ::= & p = p \mid p < p \mid p + p \mid v \mid \nu \mid p\ \mathtt{mod}\ n \\
 & & & p \wedge p \mid p \vee p \mid \neg p \mid \top \mid \bot
\end{array}$$

Figure 3.1: Syntax of simple refinement types.

Values. Values v include functions and constants. We assume that constants contain at least integers, booleans and typical arithmetic operators, like addition, subtraction and multiplication. We also assume an equality operator to test for equality between base values. Additionally, and for convenience, variables are defined as values, which is important for the syntactic restrictions for application and conditionals described below.

Expressions. Expressions contain values, applications, let expressions, ascriptions and if expressions. For reasons that will become clear when we explain the typing rules, arguments for application and conditionals are restricted to be syntactic values. This includes variables as discussed above. Ascriptions do not play an important role in the static systems and they only serve as statically verified checks. Nevertheless, they are extremely relevant for gradualization.

Types. A type T is either a refinement type of the form $\{\, \nu{:}B \mid p \,\}$, where p is a refinement formula and B a base type, or a dependent function type $x{:}T_1 \to T_2$. For convenience we pose the notational shortcut $B \stackrel{\mathsf{def}}{=} \{\, \nu{:}B \mid \top \,\}$. Additionally, if the variable x of a function $x{:}T_1 \to T_2$ does not appear free in T_2 we just write $T_1 \to T_2$.

Refinement formulas. The language of refinement formulas is independent from the formalization as long as it corresponds to a decidable theory. Here, we consider the simple quantifier free logic of linear arithmetic (QF-LIA), which suffices to captures the key issues arising by the subsequent gradualization of the language. We discuss other theories in Chapter 7. Refinement formulas and expressions belong to completely separate syntactic categories in opposition to other formalizations [16, 30, 34]. Formulas and expressions are related by types given to constants.

$$\text{(E-let1)}\ \frac{t_1 \hookrightarrow t_1'}{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \hookrightarrow \mathsf{let}\ x = t_1'\ \mathsf{in}\ t_2} \qquad \text{(E-let)}\ \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ t \hookrightarrow t[v/x]}$$

$$\text{(E-app1)}\ \frac{t \hookrightarrow t'}{t\ v \hookrightarrow t'\ v} \qquad \text{(E-app)}\ \frac{}{(\lambda x{:}T.\ t)\ v \hookrightarrow t[v/x]}$$

$$\text{(E-true)}\ \frac{}{\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 \hookrightarrow t_2} \qquad \text{(E-false)}\ \frac{}{\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 \hookrightarrow t_3}$$

$$\text{(E-delta)}\ \frac{}{c\ v \hookrightarrow \delta_c(v)}$$

Figure 3.2: Small-step operational semantics.

Type Environments	Γ	$::=$	$\cdot \mid \Gamma, x{:}T$
Logical Environments	Φ	$::=$	$\cdot \mid \Phi, x{:}p$

Typing	$\Gamma\ ;\ \Phi \vdash t : T$
Subtyping	$\Phi \vdash T_1 <: T_2$
Well Formedness	$\vdash \Phi \qquad \Phi \vdash p \qquad \Phi \vdash T$

Figure 3.3: Syntax of judgments.

Operational Semantics. The small-step operational semantics is standard for a call-by-value λ-calculus. Reduction rules are shown in Figure 3.2. By the syntactic restrictions of expressions there are only two congruence rules (E-let1) and (E-app1). As standard the rule (E-delta) uses a function δ_c which gives the semantic for constants. For instance, the constant `not` which negates a boolean is such that $\delta_{\mathsf{not}}(\mathsf{true}) = \mathsf{false}$ and $\delta_{\mathsf{not}}(\mathsf{false}) = \mathsf{true}$.

3.2 Type System

In this section we present the type system for simple refinement types. Besides using type environments to track variables in scope we introduce *logical environments*, denoted Φ, to track logical information. Other presentations of refinement types do not use logical environments, since all the relevant information is already present in type environments. Logical environments are just introduced to ease gradualization. Figure 3.3 summarizes the syntax of type environments, logical environments and typing judgments.

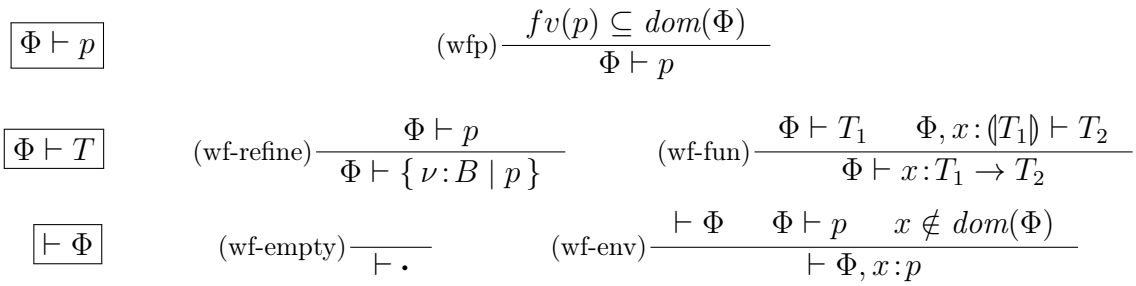

Figure 3.4: Well-formedness judgments.

3.2.1 Well-formedness

We require formulas, types and logical environments to be *well-formed*, which means that all variables mentioned are currently in scope. Well-formedness is checked against a logical environment. Figure 3.4 shows the well-formed relation for formulas, types and environments. A formula p is well-formed in an environment Φ if all free variables in p are in the domain of Φ. A refinement type $\{\, \nu{:}B \mid p \,\}$ is well-formed in an environment Φ if p is well-formed in Φ. Well-formedness for function types uses the extraction function $(\!|\cdot|\!)$ which extracts the *logical content* from a type and it is defined as follows:

$$(\!|\{\, \nu{:}B \mid p \,\}|\!) = p$$
$$(\!|x{:}T_1 \to T_2|\!) = \top$$

For refinement types we just extract the refinement formula. Function types are extracted to the trivial formula $\top$ because functions do not have a logical interpretation per se. A type $x{:}T_1 \to T_2$ is well-formed in an environment Φ if T_1 is well-formed in Φ and T_2 is well-formed in an environment extended with the binding $x : (\!|T_1|\!)$. For the well-formed judgment only the bound variable is relevant and the formula extracted from T_1 is not used. Finally, a logical environment is well-formed if there are no duplicated bindings and every formula is well-formed in the environment considering bindings to its left.

3.2.2 Typing

The expression typing judgment $\Gamma\ ;\ \Phi \vdash t : T$ uses a type environment Γ to track variables in scope and a separate logical environment Φ to track logical information. This separation is motivated by our desire to gradualize only the logical parts of types. We highlight important aspects of the typing rules.

Variables. There are two rules for typing variables (Tx-fun) and (Tx-refine). A variable x bound to a refinement type is given the exact type $\{\, \nu{:}B \mid \nu = x \,\}$ regardless of the refinement formula present in the environment [30]. Rule (Tx-fun) gives to variables the type bound in the environment when it is bound to a function type.

Constants. Each primitive constant c has a predefined type, denoted by $ty(c)$, used by Rule (Tc). Basic values like integers, booleans, *etc* are assumed to have a singleton type with a refinement stating that it is equal to the corresponding constant in the language. For instance:

$$\begin{aligned} 1 &:: \{\, \nu : \mathsf{Int} \mid \nu = 1 \,\} \\ \mathsf{true} &:: \{\, \nu : \mathsf{Bool} \mid \nu = \mathsf{true} \,\} \\ \mathsf{false} &:: \{\, \nu : \mathsf{Bool} \mid \nu = \mathsf{false} \,\} \end{aligned}$$

Arithmetic and booleans operations have types reflecting their semantics and connecting them to the logic.

$$\begin{aligned} (+) &:: x : \mathsf{Int} \to y : \mathsf{Int} \to \{\, \nu : \mathsf{Int} \mid \nu = x + y \,\} \\ (-) &:: x : \mathsf{Int} \to y : \mathsf{Int} \to \{\, \nu : \mathsf{Int} \mid \nu = x - y \,\} \\ \mathtt{not} &:: b : \mathsf{Bool} \to \{\, \nu : \mathsf{Bool} \mid \nu = \mathsf{false} \leftrightarrow b = \mathsf{true} \,\} \\ (=) &:: x : \mathsf{Int} \to y : \mathsf{Int} \to \{\, \nu : \mathsf{Bool} \mid \nu = \mathsf{true} \leftrightarrow x = y \,\} \end{aligned}$$

Abstractions. The rule for λ abstractions is fairly standard safe for the fact that the body expression is typed in a logical environment extended with the logical information extracted from the domain. This differs from other presentations missing the logical environment. We also require the type annotated for the argument to be well-formed.

Application. Since the system features dependent types, Rule (Tapp) must replace the argument into the resulting type of the application. Since we are using a separate logical language we cannot introduce arbitrary expressions into the logic. Allowing arbitrary expression in the logic will of course render type checking undecidable. However, by the syntactic restrictions of the language, an expression in argument position will always be a value. Thus, formulas always stay within the original decidable logical language. There is one more small little detail, though. We allow arbitrary values to appear in argument position which also includes lambdas. Introducing arbitrary lambda expressions will also produce formulas outside the original logical language. To circumvent this issue, when replacing by lambdas we introduce a fresh constant instead [7].

If expression. The rule for if expressions is standard, safe for the fact it allows flow sensitivity by using as assumption that the conditional is either true or false respectively in each branch. Here we also make use of the syntactic restriction which only allows values to appear as conditionals.

Let expression. The body of a let expression is typed in an environment extended with the new binding. However, we must ensure that the type of the whole expression is well-formed in the original environment. Therefore, we pick any super type of the resulting type that is well-formed in the original environment. This rule is non algorithmic, but in practice the type can be inferred [30] or an explicit annotation may be necessary.

$$\boxed{\Gamma \,;\, \Phi \vdash t : T}$$

$$\text{(Tx-refine)}\frac{\Gamma(x) = \{\, \nu{:}B \mid p \,\}}{\Gamma \,;\, \Phi \vdash x : \{\, \nu{:}B \mid \nu = x \,\}}$$

$$\text{(Tx-fun)}\frac{\Gamma(x) = y{:}T_1 \to T_2}{\Gamma \,;\, \Phi \vdash x : (y{:}T_1 \to T_2)} \qquad \text{(Tc)}\frac{}{\Gamma \,;\, \Phi \vdash c : ty(c)}$$

$$\text{(T}\lambda\text{)}\frac{\Phi \vdash T_1 \qquad \Gamma, x{:}T_1 \,;\, \Phi, x{:}(\!|T_1|\!) \vdash t : T_2}{\Gamma \,;\, \Phi \vdash \lambda x{:}T_1.\ t : (x{:}T_1 \to T_2)}$$

$$\text{(Tapp)}\frac{\Gamma \,;\, \Phi \vdash t : (x{:}T_1 \to T_2) \qquad \Gamma \,;\, \Phi \vdash v : T \qquad \Phi \vdash T <: T_1}{\Gamma \,;\, \Phi \vdash t\ v : T_2[v/x]}$$

$$\text{(Tif)}\frac{\begin{array}{c}\Gamma \,;\, \Phi \vdash v : \{\, \nu{:}\mathsf{Bool} \mid p \,\} \qquad \Phi \vdash T_1 <: T \qquad \Phi \vdash T_2 <: T \\ \Gamma \,;\, \Phi, x{:}(v{=}\mathsf{true}) \vdash t_1 : T_1 \qquad \Gamma \,;\, \Phi, x{:}(v{=}\mathsf{false}) \vdash t_2 : T_2\end{array}}{\Gamma \,;\, \Phi \vdash \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : T}$$

$$\text{(Tlet)}\frac{\begin{array}{c}\Gamma \,;\, \Phi \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \,;\, \Phi, x{:}(\!|T_1|\!) \vdash t_2 : T_2 \\ \Phi, x{:}(\!|T_1|\!) \vdash T_2 <: T \qquad\qquad \Phi \vdash T\end{array}}{\Gamma \,;\, \Phi \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : T}$$

$$\text{(T::)}\frac{\Gamma \,;\, \Phi \vdash t : T_1 \qquad \Phi \vdash T_1 <: T_2}{\Gamma \,;\, \Phi \vdash (t :: T_2) : T_2}$$

Figure 3.5: Term Typing for simple refinement types.

$$\boxed{\Phi \vdash T_1 <: T_2}$$

$$(<:\text{-refine})\frac{(\!|\Phi|\!) \cup \{\, p \,\} \models q \quad \vdash \Phi \quad \Phi \vdash p \quad \Phi \vdash q}{\Phi \vdash \{\, \nu{:}B \mid p \,\} <: \{\, \nu{:}B \mid q \,\}}$$

$$(<:\text{-fun})\frac{\Phi \vdash T_{21} <: T_{11} \quad \Phi, x{:}(\!|T_{21}|\!) \vdash T_{12} <: T_{22}}{\Phi \vdash x{:}T_{11} \to T_{12} <: x{:}T_{21} \to T_{22}}$$

$$(\!|x{:}p|\!) = p[x/\nu]$$
$$(\!|x_1{:}p_1 \ldots, x_n{:}p_n|\!) = (\!|x_1{:}p_1|\!) \cup \cdots \cup (\!|x_n{:}p_n|\!)$$

Figure 3.6: Subtyping judgment.

3.2.3 Subtyping

Subtyping is essential for the expressiveness of refinement types systems since it is the bridge between the language and the logic semantics. The subtyping judgment, shown in Figure 3.6, syntactically reduces subtyping obligations to entailment in the logic which can be discharged using an SMT solver.

Rule (<:-refine) specifies that $\{\, \nu{:}B \mid p \,\}$ is a subtype of $\{\, \nu{:}B \mid q \,\}$ in an environment Φ if p, in conjunction with the information in Φ, entails q. Extraction of logical information from an environment denoted $(\!|\Phi|\!)$, substitutes actual variables for refinement variables. The judgment $\Delta \models p$ states that the set of formulas Δ entails p modulo the theory from which formulas are drawn. A judgment $\Delta \models p$ can be checked issuing the query $\texttt{VALID}(\Delta \to p)$ to an SMT solver. Note that this is the only point where the system is connected to the semantics of the logic. As is standard, subtyping between function types, Rule (<:-fun), is contravariant in the argument type and covariant in the return type. The only particularity is that subtyping between return types is checked in an environment extended with the strongest logical information from the argument types.

3.3 Soundness

Typical formalizations of refinement types prove soundness for a system allowing full language expressions in refinements [30, 34, 23]. Typechecking for the resulting system is obviously undecidable, but then a decidable restriction of the system, like the one presented here, is introduced for which soundness follows as a corollary [30, 34]. We instead prove soundness directly on the restricted system in the spirit of Chugh et al. [8]. This distinction is important because the AGT methodology, on which we base the subsequent gradualization, relies on the syntactic type safety proof of the static system.

Precisely, the system presented here is type safe, which we prove as usual by progress and preservation.

Proposition 5 (Type preservation). *If* $\cdot\,;\,\cdot \vdash t : T$ *and* $t \hookrightarrow t'$ *then* $\cdot\,;\,\cdot \vdash t' : T'$ *and* $\cdot \vdash T' <: T$.

Proposition 6 (Progress). *If* $\cdot\,;\,\cdot \vdash t : T$ *then* t *is a value or* $t \hookrightarrow t'$.

In addition we also prove the soundness of refinements, namely that values of a refined type comply with the sated formula. Note that this result is a direct consequence of type preservation.

Proposition 7 (Refinement soundness). *If* $\cdot\,;\,\cdot \vdash t : \{\, \nu{:}B \mid p \,\}$ *and* $t \hookrightarrow^* v$ *then* $p[^v/_\nu]$ *is valid.*

Proofs of type safety and refinement soundness can be found in the Coq mechanization.

Chapter 4

A Gradual Refinement Types System

In this chapter we present the static semantic of a gradual refinement type system derived from the simple refinement types system presented in Chapter 3. We systematically derive the static semantic following the Abstracting Gradual Typing (AGT) methodology [18]. In the path, we identify a crucial design point related to the interpretation of imprecise logical information upon which the whole system is built. In order to isolate this crucial aspect we *parametrize* the design presented in this chapter respect to it and defer the specific discussion to Chapter 5.

4.1 Gradual Formulas, Types and Environments

We begin by defining the syntax and meaning of the gradual *entities* involved in the system. Specifically, we review the notion of gradual formulas, gradual types and gradual logical environments. Terms t and type environments Γ also have occurrences of gradual types in them, but we do not change their notation for readability. In particular, functions that operate over the type environment Γ are unaffected by gradualization, which only affects the meaning of relations over the logical environment Φ, most notably subtyping.

4.1.1 Gradual Formulas

Our intent is to develop a gradual language that ranges from simple types to logically-refined types. As such the system is based in the fundamental idea of imprecise logical information, crystallized in the notion of gradual formula. Therefore we need to specify a concrete syntax of gradual formulas, $\widetilde{p} \in \text{GFormula}$, the meaning of which can be defined following the AGT methodology through a concretization function $\gamma_p : \text{GFormula} \to \mathcal{P}(\text{Formula})$. Once γ_p is defined, we need to specify the corresponding best abstraction α_p such that $\langle \gamma_p, \alpha_p \rangle$ is a Galois connection. We discovered that capturing a proper definition of gradual formulas and their interpretation to yield a *practical* gradual refinement type system—*i.e.* one that does not degenerate and accept almost any program—is rather subtle. In order to isolate this

crucial design point, we defer the exact definition of GFormula, γ_p and α_p to Chapter 5. In the remainder of this chapter, we define the gradual refinement type system and establish its properties independently of the specific interpretation of gradual formulas.

4.1.2 Gradual Types

As we intend the imprecision of gradual types to just reflect the imprecision of formulas, the syntax of gradual types $\widetilde{T} \in \text{GType}$ simply contains gradual formulas.

$$\widetilde{T} \;::=\; \{\, \nu{:}B \mid \widetilde{p} \,\} \mid x{:}\widetilde{T} \to \widetilde{T} \quad \text{(Gradual Types)}$$

Then, the (assumed) concretization function for gradual formulas γ_p can be compatibly lifted to gradual types.

Definition 2 (Concretization of gradual types)**.** *Let the concretization function for gradual types* $\gamma_{\scriptscriptstyle T} : \text{GType} \to \mathcal{P}(\text{Type})$ *be defined as follows:*

$$\gamma_{\scriptscriptstyle T}(\{\, \nu{:}B \mid \widetilde{p} \,\}) = \{\, \{\, \nu{:}B \mid p \,\} \mid p \in \gamma_p(\widetilde{p}) \,\}$$
$$\gamma_{\scriptscriptstyle T}(x{:}\widetilde{T}_1 \to \widetilde{T}_2) = \{\, x{:}T_1 \to T_2 \mid T_1 \in \gamma_{\scriptscriptstyle T}(\widetilde{T}_1) \wedge T_2 \in \gamma_{\scriptscriptstyle T}(\widetilde{T}_2) \,\}$$

The notion of *precision* can be directly defined for gradual refinement types based on this concretization function.

Definition 3 (Precision of gradual refinement types)**.** $\widetilde{T}_1$ *is less imprecise than* $\widetilde{T}_2$*, notation* $\widetilde{T}_1 \sqsubseteq \widetilde{T}_2$*, if and only if* $\gamma_{\scriptscriptstyle T}(\widetilde{T}_1) \subseteq \gamma_{\scriptscriptstyle T}(\widetilde{T}_2)$*.*

4.1.3 Gradual Logical Environments

Logical environments, which play an important role in the static systems as they serve as context for the subtyping judgment, can now contain bindings to gradual formulas. We introduce gradual logical environments $\widetilde{\Phi} \in \text{GLEnv}$ accordingly to serve this purpose.

$$\widetilde{\Phi} \;::=\; \cdot \mid \widetilde{\Phi}, x{:}\widetilde{p} \quad \text{(Gradual Logical Environments)}$$

The interpretation of a gradual environment is directly obtained by pointwise lifting of the concretization of gradual formulas.

Definition 4 (Concretization of gradual logical environments)**.** *Let* $\gamma_{\scriptscriptstyle \Phi} : \text{GLEnv} \to \mathcal{P}(\text{LEnv})$ *be defined as:*

$$\gamma_{\scriptscriptstyle \Phi}(\widetilde{\Phi}) = \{\, \Phi \mid \forall x.\Phi(x) \in \gamma_p(\widetilde{\Phi}(x)) \,\}$$

4.2 Consistent Relations and Functions

With the meaning of gradual types and logical environments, we can lift static subtyping to its consistent counterpart following AGT: consistent subtyping holds between two gradual types, in a given logical environment, if and only if static subtyping holds *for some* static types and logical environment in the respective concretizations.

Definition 5 (Consistent subtyping)**.** $\widetilde{\Phi} \vdash \widetilde{T_1} \mathrel{\widetilde{<:}} \widetilde{T_2}$ *if and only if* $\Phi \vdash T_1 <: T_2$ *for some* $\Phi \in \gamma_{\Phi}(\widetilde{\Phi}), T_1 \in \gamma_{T}(\widetilde{T_1})$ *and* $T_2 \in \gamma_{T}(\widetilde{T_2})$.

Solving a consistent subtyping judgment means finding suitable formulas in the interpretation of gradual formulas justifying the underlying implication. For instance, consider the following consistent subtyping judgment:

$$x{:}?, y{:}(\nu > x) \vdash \{\, \nu{:}\mathsf{Int} \mid \nu = y \,\} \mathrel{\widetilde{<:}} \{\, \nu{:}\mathsf{Int} \mid \nu > 0 \,\}$$

Intuitively this judgment should hold because the ? in the refinement for x may be interpreted as $x > 0$ which together with $y > x$ and $\nu = y$ suffices to prove $\nu > 0$. However, the *search space* for suitable formulas cannot be established yet, for which we have to wait until we fix the concretization function for gradual formulas.

As may be noted, this definition of consistent subtyping does not suggest an algorithm, nor is even clear if it is decidable for some interesting concretization function. Later in Chapter 7, after fixing the interpretation for gradual formulas, we present an algorithmic characterization of consistent subtyping, $\cdot \vdash \cdot \lesssim \cdot$, based on solving *consistent entailment constraints*, and prove that it coincides with the definition above.

The static type system also relies on a type substitution function. Following AGT, lifting type functions to operate on gradual types requires an abstraction function from sets of types to gradual types: the lifted function is defined by abstracting over all the possible results of the static function applied to all the represented static types. Instantiating this principle for type substitution:

Definition 6 (Consistent type substitution)**.**

$$\widetilde{\widetilde{T}[v/x]} = \alpha_{T}(\{\, T[{}^{v}/_{x}] \mid T \in \gamma_{T}(\widetilde{T}) \,\})$$

where α_{T} is the natural lifting of the abstraction for formulas α_p:

Definition 7 (Abstraction for gradual refinement types)**.** *Let* $\alpha_{T} : \mathcal{P}(\textsc{Type}) \rightharpoonup \textsc{GType}$ *be defined as:*

$$\alpha_{T}(\{\, \overline{\{\, \nu{:}B \mid p_i \,\}} \,\}) = \{\, \nu{:}B \mid \alpha_p(\{\, \overline{p_i} \,\}) \,\}$$
$$\alpha_{T}(\{\, \overline{x{:}T_{i1} \to T_{i2}} \,\}) = x{:}\alpha_{T}(\{\, \overline{T_{i1}} \,\}) \to \alpha_{T}(\{\, \overline{T_{i2}} \,\})$$

The algorithmic version of consistent type substitution, noted $\cdot[\![\cdot/\cdot]\!]$, substitutes in the known parts of formulas (Section 7.1).

$\widetilde{T} \in \text{GType},\ t \in \text{GTerm},\ \widetilde{p} \in \text{GFormula},\ \Gamma \in \text{GEnv},\ \widetilde{\Phi} \in \text{GLEnv}$

$\boxed{\Gamma\ ;\ \Phi \vdash t : \widetilde{T}}$

$$(\widetilde{\text{T}}\text{x-refine})\frac{\Gamma(x) = \{\, \nu{:}B \mid \widetilde{p} \,\}}{\Gamma\ ;\ \Phi \vdash x : \{\, \nu{:}B \mid \nu = x \,\}}$$

$$(\widetilde{\text{T}}\text{x-fun})\frac{\Gamma(x) = y{:}\widetilde{T}_1 \to \widetilde{T}_2}{\Gamma\ ;\ \Phi \vdash x : (y{:}\widetilde{T}_1 \to \widetilde{T}_2)} \qquad (\widetilde{\text{T}}\text{c})\frac{}{\Gamma\ ;\ \Phi \vdash c : ty(c)}$$

$$(\widetilde{\text{T}}\lambda)\frac{\widetilde{\Phi} \vdash \widetilde{T}_1 \qquad \Gamma, x{:}\widetilde{T}_1\ ;\ \widetilde{\Phi}, x{:}(\!|\widetilde{T}_1|\!) \vdash t : \widetilde{T}_2}{\Gamma\ ;\ \Phi \vdash \lambda x{:}\widetilde{T}_1.\ t : (x{:}\widetilde{T}_1 \to \widetilde{T}_2)}$$

$$(\widetilde{\text{T}}\text{app})\frac{\Gamma\ ;\ \Phi \vdash t : x{:}\widetilde{T}_1 \to \widetilde{T}_2 \qquad \Gamma\ ;\ \Phi \vdash v : \widetilde{T} \qquad \widetilde{\Phi} \vdash \widetilde{T} \lesssim \widetilde{T}_1}{\Gamma\ ;\ \Phi \vdash t\ v : \widetilde{T}_2[\![v/x]\!]}$$

$$(\widetilde{\text{T}}\text{if})\frac{\begin{array}{c}\Gamma\ ;\ \Phi \vdash v : \{\, \nu{:}\mathsf{Bool} \mid \widetilde{p} \,\} \qquad \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T} \qquad \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T} \\ \Gamma\ ;\ \widetilde{\Phi}, x{:}(v{=}\mathsf{true}) \vdash t_1 : \widetilde{T}_1 \qquad \Gamma\ ;\ \widetilde{\Phi}, x{:}(v{=}\mathsf{false}) \vdash t_2 : \widetilde{T}_2\end{array}}{\Gamma\ ;\ \Phi \vdash \mathsf{if}\ v\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : \widetilde{T}}$$

$$(\widetilde{\text{T}}\text{let})\frac{\begin{array}{c}\Gamma, x{:}\widetilde{T}_1\ ;\ \widetilde{\Phi}, x{:}(\!|\widetilde{T}_1|\!) \vdash t_2 : \widetilde{T}_2 \qquad \Gamma\ ;\ \Phi \vdash t_1 : \widetilde{T}_1 \\ \widetilde{\Phi}, x{:}(\!|\widetilde{T}_1|\!) \vdash \widetilde{T}_2 \lesssim \widetilde{T} \qquad \widetilde{\Phi} \vdash \widetilde{T}\end{array}}{\Gamma\ ;\ \Phi \vdash \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 : \widetilde{T}}$$

$$(\widetilde{\text{T}}\text{::})\frac{\Gamma\ ;\ \Phi \vdash t : \widetilde{T}_1 \qquad \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2}{\Gamma\ ;\ \Phi \vdash t :: \widetilde{T}_2 : \widetilde{T}_2}$$

Figure 4.1: Typing rules of the gradual refinement types language.

4.3 The System and its Properties

The typing rules of the gradual language (Figure 4.1) directly mimic the static language typing rules, save for the fact that they use gradual refinement types $\widetilde{T}$, built from gradual formulas $\widetilde{p}$, and gradual environments $\widetilde{\Phi}$. Also, the rules use the consistent lifting of the predicates and functions on the static system. Notably, Rule ($\widetilde{\text{T}}$app) uses *consistent subtyping* to test for the correct use of the argument and *consistent type substitution* to substitute in the result type. We use the notation for the algorithmic versions for readability, but this definitions cannot actually be given until the definitions of γ_p and α_p are fixed.

The gradual refinement type system satisfies a number of desirable properties. First, the system is a conservative extension of the underlying static system: for every fully-annotated term both systems coincide (we use $\vdash_S$ to denote the static system).

Proposition 8 (Equivalence for fully-annotated terms)**.** *For any* $t \in \text{Term}$, $\Gamma\ ;\ \Phi \vdash_S t : T$

if and only if $\Gamma\,;\,\Phi \vdash t : T$

More interestingly, the system satisfies the static gradual guarantee of Siek et al. [33]: weakening the precision of a term preserves typeability, at a less precise type.

Proposition 9 (Static gradual guarantee)**.** *If* $\cdot \vdash \cdot : t_1\widetilde{T_1}$ *and* $t_1 \sqsubseteq t_2$*, then* $\cdot \vdash \cdot : t_2\widetilde{T_2}$ *and* $\widetilde{T_1} \sqsubseteq \widetilde{T_2}$.

We prove both properties parametrically with respect to the actual definitions of γ_p, α_p and GFormula. The proof of Prop 8 only requires that static type information is preserved exactly, *i.e.* $\gamma_{_T}(T) = \{\, T \,\}$ and $\alpha_{_T}(\{\, T \,\}) = T$, which follows directly from the same properties for γ_p and α_p. These hold trivially for the different interpretations of gradual formulas we consider in the next chapter. The proof of Prop 9 relies on the fact that $\langle \gamma_{_T}, \alpha_{_T} \rangle$ is a Galois connection. Again, this follows from $\langle \gamma_p, \alpha_p \rangle$ being a Galois connection—a result we will establish in due course.

Chapter 5

Defining the Logical Imprecision

The definition of the gradual type system of the previous chapter is parametric over the interpretation of gradual formulas. Starting from a naive interpretation, in this chapter we progressively build a practical interpretation of gradual formulas. More precisely, we start in Section 5.1 with a definition of the syntax of gradual formulas, GFormula, and an associated concretization function γ_p, and then successively *redefine* both until reaching a satisfactory definition in Section 5.4. We then define the corresponding abstraction function α_p in Section 5.5.

We insist on the fact that any interpretation of gradual formulas that respects the conditions stated in Section 4.3 would yield a "coherent" gradual type system. Discriminating between these different possible interpretations is eventually a *design decision*, motivated by the expected behavior of a gradual refinement type system, and is hence driven by considering specific examples.

5.1 Naive Interpretation

Following the abstract interpretation viewpoint on gradual typing, a *gradual logical formula* denotes a set of possible logical formulas. As such, it can contain some statically-known logical information, as well as some additional, unknown assumptions. Syntactically, we can denote a gradual formula as either a *precise* formula (equivalent to a fully-static formula), or as an *imprecise* formula, $p \wedge ?$, where p is called its *known part*.

$$
\begin{array}{l}
\widetilde{p} \in \text{GFormula},\ p \in \text{Formula} \\
\begin{array}{lcll}
\widetilde{p} & ::= & p & \text{(Precise Formulas)} \\
 & \mid & p \wedge ? & \text{(Imprecise Formulas)}
\end{array}
\end{array}
$$

We use a conjunction in the syntax to reflect the intuition of a formula that can be made more precise by adding logical information. Note however that the symbol $?$ can only appear once and in a conjunction at the top level. That is, $p \vee ?$ and $p \vee (q \wedge ?)$ are not syntactically valid gradual formulas. We also pose $? \stackrel{\text{def}}{=} \top \wedge ?$ for convenience.

Having defined the syntax of gradual formulas, we must turn to their semantics. Following AGT, we give gradual formulas meaning by concretization to sets of static formulas. Here, the ? in a gradual formula $p \wedge ?$ can be understood as a placeholder for additional logical information that strengthens the known part p. A natural, but naive, definition of concretization follows.

Definition 8 (Naive concretization of gradual formulas)**.** *Let the concretization function* $\gamma_p : \text{GFormula} \to \mathcal{P}(\text{Formula})$ *be defined as follows:*

$$\gamma_p(p) = \{\, p \,\} \qquad\qquad \gamma_p(p \wedge ?) = \{\, p \wedge q \mid q \in \text{Formula} \,\}$$

This definition is problematic, however. Consider a value v refined with the gradual formula $\nu \geq 2 \wedge ?$. With the above definition, we would accept passing v as argument to a function that expects a negative argument! Indeed, a possible interpretation of the gradual formula would be $\nu \geq 2 \wedge \nu = 1$, which is unsatisfiable[1] and hence trivially entails $\nu < 0$. Therefore, accepting that the unknown part of a formula denotes any arbitrary formula—including ones that contradict the known part of the gradual formula—annihilates one of the benefits of gradual typing, which is to reject such blatant inconsistencies between pieces of static information.

5.2 Non-Contradicting Interpretation

To avoid this extremely permissive behavior, we must develop a *non-contradicting interpretation* of gradual formulas. The key requirement is that when the known part of a gradual formula is satisfiable, the interpretation of the gradual formula should remain satisfiable, as captured by the following definition (we write $\texttt{SAT}(p)$ for a formula p that is satisfiable):

Definition 9 (Non-contradicting concretization of gradual formulas)**.** *Let the concretization function* $\gamma_p : \text{GFormula} \to \mathcal{P}(\text{Formula})$ *be defined as:*

$$\gamma_p(p) = \{\, p \,\} \qquad\qquad \gamma_p(p \wedge ?) = \{\, p \wedge q \mid \texttt{SAT}(p) \Rightarrow \texttt{SAT}(p \wedge q) \,\}$$

This new definition of concretization is however still problematic. Recall that a given concretization induces a natural notion of precision by relating the concrete sets [18]. Precision of gradual formulas is the key notion on top of which precision of gradual types and precision of gradual terms are built.

Definition 10 (Precision of gradual formulas)**.** $\widetilde{p}$ *is less imprecise (more precise) than* $\widetilde{q}$*, noted* $\widetilde{p} \sqsubseteq \widetilde{q}$*, if and only if* $\gamma_p(\widetilde{p}) \subseteq \gamma_p(\widetilde{q})$*.*

The non-contradicting interpretation of gradual formulas is *purely syntactic*. As such, the induced notion of precision fails to capture intuitively useful connections between programs.

[1] We prefer the term "(un)satisfiable" over "(in)consistent" to avoid confusion with the term "consistency" from the gradual typing literature.

For instance, the sets of static formulas represented by the gradual formulas $x \geq 0 \wedge ?$ and $x > 0 \wedge ?$ are incomparable, because they are *syntactically* different. However, the gradual formula $x > 0 \wedge ?$ should intuitively refer to a more restrictive set of formulas, because the static information $x > 0$ is more *specific* than $x \geq 0$.

5.3 Semantic Interpretation

To obtain a meaningful notion of precision between gradual formulas, we appeal to the notion of *specificity* of logical formulas, which is related to the actual semantics of formulas, not just their syntax.

Formally, a formula p is more specific than a formula q if $\{ p \} \models q$. Technically, this relation only defines a pre-order, because formulas that differ syntactically can be logically equivalent. As usual we work over the equivalence classes and consider equality up to logical equivalence. Thus, when we write p we actually refer to the equivalence class of p. In particular, the equivalence class of unsatisfiable formulas is represented by $\bot$, which is the bottom element of the specificity pre-order.

In order to preserve non-contradiction in our semantic interpretation of gradual formulas, it suffices to remove (the equivalence class of) $\bot$ from the concretization. Formally, we isolate $\bot$ from the specificity order, and define the order only for the satisfiable fragment of formulas, denoted SFormula:

Definition 11 (Specificity of satisfiable formulas)**.** *Given two formulas $p, q \in$ SFormula, we say that p is more specific than q in the satisfiable fragment, notation $p \preceq q$, if $\{ p \} \models q$.*

Then, we define gradual formulas such that the known part of an imprecise formula is required to be satisfiable:

$$\widetilde{p} \in \text{GFormula},\ p \in \text{Formula},\ p^{\checkmark} \in \text{SFormula}$$
$$\begin{array}{rcll} \widetilde{p} & ::= & p & \text{(Precise Formulas)} \\ & | & p^{\checkmark} \wedge ? & \text{(Imprecise Formulas)} \end{array}$$

Note that the imprecise formula $x > 0 \wedge x = 0 \wedge ?$, for example, is syntactically rejected because its known part is not satisfiable. However, $x > 0 \wedge x = 0$ is a syntactically valid formula because precise formulas are not required to be satisfiable.

The semantic definition of concretization of gradual formulas captures the fact that an imprecise formula stands for any satisfiable strengthening of its known part:

Definition 12 (Semantic concretization of gradual formulas)**.** *Let the concretization function $\gamma_p : \text{GFormula} \to \mathcal{P}(\text{Formula})$ be defined as follows:*

$$\gamma_p(p) = \{ p \} \qquad\qquad \gamma_p(p^{\checkmark} \wedge ?) = \{ q^{\checkmark} \mid q^{\checkmark} \preceq\ p^{\checkmark} \}$$

This semantic interpretation yields a practical notion of precision that admits the judgment $x > 0 \wedge ? \sqsubseteq x \geq 0 \wedge ?$, as wanted.

Unfortunately, despite the fact that, taken in isolation, gradual formulas cannot introduce contradictions, the above definition does not yield an interesting gradual type system yet, because it allows other kinds of contradictions to sneak in. Consider the following:

$$\textbf{let}\ \ g\ (x\colon \{\,\nu\!:\!\mathsf{Int} \mid \nu > 0\,\})\ (y\colon \{\,\nu\!:\!\mathsf{Int} \mid \nu = 0 \wedge ?\,\}) = x/y$$

The static information $y = 0$ should suffice to statically reject this definition. But, at the use site of the division operator, the consistent subtyping judgment that must be proven is:

$$x\!:\!(\nu > 0), y\!:\!(\nu = 0 \wedge ?) \vdash \{\,\nu\!:\!\mathsf{Int} \mid \nu = y\,\} \lesssim \{\,\nu\!:\!\mathsf{Int} \mid \nu \neq 0\,\}$$

While the interpretation of the imprecise refinement of y cannot contradict $y = 0$, it can stand for $\nu = 0 \wedge x \leq 0$, which contradicts $x > 0$. Hence the definition is statically accepted.

The introduction of contradictions in the presence of gradual formulas can be even more subtle. Consider the following program:

$$\begin{array}{l}\textbf{let}\ \ h\ (x\colon \{\,\nu\!:\!\mathsf{Int} \mid ?\,\})\ (y\colon \{\,\nu\!:\!\mathsf{Int} \mid ?\,\})\ (z\colon \{\,\nu\!:\!\mathsf{Int} \mid \nu = 0\,\}) \\ \quad = (x + y)/z\end{array}$$

One would expect this program to be rejected statically, because it is clear that $z = 0$. But, again, one can find an environment that makes consistent subtyping hold:

$$x\!:\!(\nu > 0), y\!:\!(\nu = x \wedge \nu < 0), z : (\nu = 0).$$

This interpretation is not contradicting any assumed static information—*i.e.*, $z = 0$—but it does introduces a contradiction between the separate interpretations of *different* gradual formulas.

5.4 Local Interpretation

We need to restrict the space of possible static formulas represented by gradual formulas, in order to avoid contradicting already-established static assumptions, and to avoid introducing contradictions between the interpretation of different gradual formulas involved in the same consistent subtyping judgment.

Stepping back: what do refinements refine? Intuitively, the refinement type $\{\,\nu\!:\!B \mid p\,\}$ refers to all values of type B that satisfy the formula p. Note that apart from ν, the formula p can refer to other variables in scope. In the following, we use the more explicit syntax $p(\vec{x}; \nu)$ to denote a formula p that constrains the refinement variable ν based on the variables in scope $\vec{x}$.

The well-formedness condition in the static system ensures that variables $\vec{x}$ on which a formula depends are in scope, but does not restrict in any way how a formula *uses* these variables. This permissiveness of traditional static refinement type systems admits curious

definitions. For example, the first argument of a function can be constrained to be positive by annotating the second argument:

$$x : \mathsf{Int} \to y : \{\, \nu : \mathsf{Int} \mid x > 0 \,\} \to \mathsf{Int}$$

Applying this function to some negative value is perfectly valid but yields a function that expects false. A formula can even contradict information already assumed about a prior argument:

$$x : \{\, \nu : \mathsf{Int} \mid \nu > 0 \,\} \to y : \{\, \nu : \mathsf{Int} \mid x < 0 \,\} \to \mathsf{Int}$$

We observe that this unrestricted freedom of refinement formulas is the root cause of the (non-local) contradiction issues that can manifest in the interpretation of gradual formulas.

Local formulas. The problem with contradictions arises from the fact that a formula $p(\vec{x}; \nu)$ is allowed to express *restrictions* not just on the refinement variable ν but also on the variables in scope $\vec{x}$. In essence, we want unknown formulas to stand for any *local* restriction on the refinement variable, without allowing for contradictions with prior information on variables in scope.

Intuitively, we say that a formula is *local* if it only restricts the refinement variable ν. Capturing when a formula is local goes beyond a simple syntactic check because formulas should be able to mention variables in scope. For example, the formula $\nu > x$ is local: it restricts ν based on x without further restricting x. Consider also the following dependent function type:

$$x : \mathsf{Int} \to y : \{\, \nu : \mathsf{Int} \mid x \geq \nu \wedge \nu \geq 0 \,\} \to \mathsf{Int}$$

It may seem that the type signature is innocuously restricting the second argument to be between zero and the first argument. However, a closer looks reveals that the refinement is not just restricting ν, but also implicitly restricting x to be greater than or equal to zero. A local annotation of this function would explicitly restricts x in its refinements and states the restriction for y only when $x \geq 0$.

$$x : \{\, \nu : \mathsf{Int} \mid \nu \geq 0 \,\} \to y : \{\, \nu : \mathsf{Int} \mid x \geq 0 \to x \geq \nu \wedge \nu \geq 0 \,\} \to \mathsf{Int}$$

The key to identify a formula as local is that, for every assignation for variables other than ν, there exists a value for ν for which the formula holds.

Definition 13 (Local formula)**.** *A formula $p(\vec{x}; \nu)$ is* local *if the formula $\exists \nu . p(\vec{x}; \nu)$ is valid.*

We call LFormula the set of local formulas. Note that the definition above implies that a local formula is satisfiable, because there must exist some ν for which the formula holds. Hence, LFormula $\subset$ SFormula $\subset$ Formula.

The crucial property that allows us to use local formula to get rid of contradictions is that a local formula always produces satisfiable assumptions when combined with a satisfiable logical environment:

Proposition 10. *Let Φ be a logical environment, $\vec{x} = dom(\Phi)$ the vector of variables bound in Φ, and $q(\vec{x}, \nu) \in$ LFormula. If $(\!|\Phi|\!)$ is satisfiable then $(\!|\Phi|\!) \cup \{\, q(\vec{x}, \nu) \,\}$ is satisfiable.*

Moreover, we note that local formulas have the same expressiveness than non-local formulas when taken as a conjunction (we use $\equiv$ to denote logical equivalence).

Proposition 11. *Let Φ be a logical environment. If $(\!|\Phi|\!)$ is satisfiable then there exists an environment Φ' with the same domain such that $(\!|\Phi|\!) \equiv (\!|\Phi'|\!)$ and for all x the formula $\Phi'(x)$ is local.*

Similarly to what we did for the semantic interpretation, we redefine the syntax of gradual formulas such that the known part of an imprecise formula is required to be local:

$$\widetilde{p} \in \text{GFormula},\ p \in \text{Formula},\ p^\circ \in \text{LFormula}$$
$$\begin{array}{lcll} \widetilde{p} & ::= & p & \text{(Precise Formulas)} \\ & | & p^\circ \wedge ? & \text{(Imprecise Formulas)} \end{array}$$

The local concretization of gradual formulas allows imprecise formulas to denote any *local* formula strengthening the known part:

Definition 14 (Local concretization of gradual formulas)**.** *Let the concretization function $\gamma_p : \text{GFormula} \to \mathcal{P}(\text{Formula})$ be defined as follows:*

$$\gamma_p(p) = \{\, p \,\} \qquad\qquad \gamma_p(p^\circ \wedge ?) = \{\, q^\circ \mid q^\circ \preceq p^\circ \,\}$$

From now on, we simply write $p \wedge ?$ for imprecise formulas, leaving implicit the fact that p is a local formula.

Examples. The local interpretation of imprecise formulas forbids the restriction of previously-defined variables. To illustrate, consider the following definition:

$$\textbf{let}\ \ f\ (x\colon \mathsf{Int})\ (y\colon \{\, \nu\colon \mathsf{Int} \mid ? \,\}) = y/x$$

The static information on x is not sufficient to prove the code safe. Because any interpretation of the unknown formula restricting y must be local, x cannot possibly be restricted to be non-zero, and the definition is rejected statically. This behavior matches the intuition because we do not expect that a tighter restriction on y makes code only involving x safe.

The impossibility to restrict previously-defined variables avoids generating contradictions and hence accepting too many programs. Recall the example of contradictions between different interpretations of imprecise formulas:

$$\begin{array}{l} \textbf{let}\ \ h\ (x\colon \{\, \nu\colon \mathsf{Int} \mid ? \,\})\ (y\colon \{\, \nu\colon \mathsf{Int} \mid ? \,\})\ (z\colon \{\, \nu\colon \mathsf{Int} \mid \nu = 0 \,\}) \\ \quad = (x + y)/z \end{array}$$

This definition is now rejected statically because accepting it would mean finding well-formed local formulas p and q such that the following static subtyping judgment holds:

$$x\colon p, y\colon q, z\colon (\nu = 0) \vdash \{\, \nu\colon \mathsf{Int} \mid \nu = z \,\} <: \{\, \nu\colon \mathsf{Int} \mid \nu \neq 0 \,\}$$

However, by well-formedness, p and q cannot restrict z; and by locality, p and q cannot contradict each other.

5.5 Abstracting Formulas

Having reached a satisfactory definition of the syntax and concretization function γ_p for gradual formulas, we must now find the corresponding best abstraction α_p in order to construct the required Galois connection. We observe that, due to the definition of γ_p, specificity $\preceq$ is central to the definition of precision $\sqsubseteq$. We exploit this connection to derive a framework for abstract interpretation based on the structure of the specificity order.

The specificity partial order for the satisfiable fragment of formulas forms a join-semilattice with disjunction as the finite join[2]. However, it does not contain a join for arbitrary (possible infinite) non-empty sets. The existence of a join for an arbitrary set depends on the expressiveness of the logical language.

The expressiveness of a logic can be characterized by the sets it can define. For example, in Presburger arithmetic the formula $\nu > 0$ defines the set of positive numbers and the formula $\nu = 3$ the singleton set $\{\,3\,\}$[3]. Consider now the set of formulas $\Delta = \{\,(\nu = n) \mid n \text{ is even}\,\}$. The union of all (singleton) sets defined by formulas in Δ is equal to the set of all even numbers. Even numbers can be defined in Presburger arithmetic by the formula $(\exists n.\nu = n + n)$, and this formula exactly corresponds to the join of Δ. On the contrary, consider the set $\Delta' = \{\,(\nu = n) \mid n \text{ is prime}\,\}$. All formulas in Δ' are in Presburger arithmetic because they define singleton sets. However, intuitively speaking Δ' does not have a join because its union corresponds to all prime numbers, which is not definable in Presburger arithmetic. More concretely, the set Δ' does not have a least upper bound because there is an infinite decreasing chain of sets definable in Presburger arithmetic containing all prime numbers. Take for example the chain $(A_i)_{i\in\mathbb{N}}$ where A_0 contains all natural numbers and each set in the chain is formed by removing a non-prime number from the previous one. All sets formed by this (infinite) process are definable in Presburger arithmetic and all of them contains the set of prime numbers, but there is no least one.

The lack of a join for arbitrary sets means that it is not always possible to have a best abstraction. We can however define a *partial* abstraction function, defined whenever it is possible to define a best one.

Definition 15. *Let* $\alpha_p : \mathcal{P}(\textsc{Formula}) \rightharpoonup \textsc{GFormula}$ *be the partial abstraction function defined as follows.*

$$\begin{aligned}
&\alpha_p(\{\,p\,\}) = p \\
&\quad \alpha_p(\,\widehat{p}\,) = \left(\bigvee \widehat{p}\right) \wedge ? \text{ if } \widehat{p} \subseteq \textsc{LFormula} \text{ and } \bigvee \widehat{p} \text{ is defined} \\
&\quad \alpha_p(\,\widehat{p}\,) \text{ is undefined otherwise}
\end{aligned}$$

($\bigvee$ is the join for the specificity order in the satisfiable fragment)

The function α_p is well defined because the join of a set of local formulas is necessarily a local formula. In fact, an even stronger property holds: any upper bound of a local formula is

[2]The partial order also has conjunction as the meet operator, but it is not defined for every finite set since we are strictly referring to the satisfiable fragment which has no bottom element.

[3]Presburger arithmetic defines the class of semi-linear sets.

local. In other words the local fragment of formulas can be understood as a join-semilattice of the satisfiable fragment.

Proposition 12. *If* $p \in \text{LFormula}$ *and* $p \preceq q$ *then* $q \in \text{LFormula}$.

We establish that, whenever α_p is defined, it is the *best* possible abstraction that corresponds to γ_p. This characterization validates the use of specificity instead of precision in the definition of α_p.

Proposition 13 (α_p is sound)**.** *If* $\alpha_p(\widehat{p})$ *is defined, then* $\widehat{p} \subseteq \gamma_p(\alpha_p(\widehat{p}))$.

Proposition 14 (α_p is optimal)**.** *If* $\alpha_p(\widehat{p})$ *is defined, then* $\widehat{p} \subseteq \gamma_p(\widetilde{p})$ *implies* $\alpha_p(\widehat{p}) \sqsubseteq \widetilde{p}$.

A pair $\langle \alpha, \gamma \rangle$ that satisfies soundness and optimality is a Galois connection. However, Galois connections relate *total* functions. Here α_p is undefined whenever: (1) $\widehat{p}$ is the empty set (the join is undefined since there is no least element), (2) $\widehat{p}$ is non-empty, but contains both local and non-local formulas, or (3) $\widehat{p}$ is non-empty, and only contains local formulas, but $\bigvee \widehat{p}$ does not exist.

Garcia et al. [18] also define a partial abstraction function for gradual types, but the only source of partiality is the empty set. Technically, it would be possible to abstract over the empty set by adding a least element. But they justify the decision of leaving abstraction undefined based on the observation that, just as static type functions are partial, consistent functions (which are defined using abstraction) must be too. In essence, statically, abstracting the empty set corresponds to a *type error*, and dynamically, it corresponds to a *cast error*, as we will revisit in Chapter 6.

The two other sources of partiality of α_p cannot however be justified similarly. Fortunately, both are benign in a very precise sense: whenever we operate on sets of formulas obtained from the concretization of gradual formulas, we *never* obtain a non-empty set that cannot be abstracted. Miné [26] generalized Galois connections to allow for *partial* abstraction functions that are always defined whenever applying some operator of interest. More precisely, given a set $\mathcal{F}$ of concrete operators, Miné defines the notion of $\langle \alpha, \gamma \rangle$ being an $\mathcal{F}$-partial Galois connection, by requiring, in addition to soundness and optimality, that the composition $\alpha \circ F \circ \gamma$ be defined for every operator $F \in \mathcal{F}$.

Definition 16 (Partial Galois connection)**.** *Let* $(C, \sqsubseteq_C)$ *and* $(A, \sqsubseteq_A)$ *be two posets,* $\mathcal{F}$ *a set of operators on* C, $\alpha : C \rightharpoonup A$ *a partial function and* $\gamma : A \to C$ *a total function. The pair* $\langle \alpha, \gamma \rangle$ *is an* $\mathcal{F}$*-partial Galois connection if and only if:*

1. *If* $\alpha(c)$ *is defined, then* $c \sqsubseteq_C \gamma(\alpha(c))$*, and*
2. *If* $\alpha(c)$ *is defined, then* $c \sqsubseteq_C \gamma(a)$ *implies* $\alpha(c) \sqsubseteq_A a$*, and*
3. *For all* $F \in \mathcal{F}$ *and* $c \in C$, $\alpha(F(\gamma(c)))$ *is defined.*

This definition can be generalized for a set $\mathcal{F}$ *of arbitrary* n*-ary operators.*

Abstraction for gradual types α_T is the natural extension of abstraction for gradual formu-

las α_p, and hence inherits its partiality. Observe that, in the static semantics of the gradual language, abstraction is only used to define the consistent type substitution operator $\cdot[\cdot/\cdot]$ (Section 4.2). We establish that, despite the partiality of α_p, the pair $\langle \alpha_{_T}, \gamma_{_T} \rangle$ is a partial Galois connection:

Proposition 15 (Partial Galois connection for gradual types)**.** *The pair $\langle \alpha_{_T}, \gamma_{_T} \rangle$ is a $\{ \widehat{tsubst} \}$-partial Galois connection, where $\widehat{tsubst}$ is the collecting lifting of type substitution,* i.e.

$$\widehat{tsubst}(\widehat{T}, v, x) = \{ T[{}^{v}/_{x}] \mid T \in \widehat{T} \}$$

The runtime semantics described in Chapter. 6 rely on another notion of abstraction built over α_p, hence also partial, for which a similar result will be established, considering the relevant operators.

Chapter 6

Abstracting the Dynamic Semantics

In this chapter we present the dynamic semantics for our gradual refinement types system. Following Garcia et al. [18] we derive the dynamic semantics of the gradual system by reduction of gradual typing derivations exploiting the correspondence between proof normalization and term reduction [21]. This approach provides the *direct* runtime semantics of gradual programs, instead of the usual approach by translation to some intermediate cast calculus [32].

As a term (*i.e.* and its typing derivation) reduces, it is necessary to justify new judgments for the typing derivation of the new term, such as subtyping. In a type safe static language, these new judgments can always be established, as justified in the type preservation proof, which relies on properties of judgments such as transitivity of subtyping. However, in the case of gradual typing derivations, these properties may not always hold: for instance the two *consistent* subtyping judgments $\mathsf{Int} \lesssim ?$ and $? \lesssim \mathsf{Bool}$ cannot be combined to justify the transitive judgment $\mathsf{Int} \lesssim \mathsf{Bool}$.

More precisely, Garcia et al. [18] introduce the notion of *evidence* to characterize *why* a consistent judgment holds. A consistent operator, such as *consistent transitivity*, determines when evidences can be combined to produce evidence for a new judgment. The impossibility to combine evidences so as to justify a combined consistent judgment corresponds to a cast error: the realization, at runtime, that the plausibility based on which the program was considered (gradually) well-typed is not tenable anymore.

Deriving the runtime semantics of gradual refinements presents a number of challenges compared to the simple typing discipline considered by Garcia et al. [18]. First, evidence of consistent subtyping has to account for the logical environment in the judgment (Sect. 6.1), yielding a more involved definition of the consistent subtyping transitivity operator (Sect. 6.2). Second, dependent types introduce the need for two additional consistent operators: one corresponding to the subtyping substitution lemma, accounting for substitution in types (Sect. 6.3), and one corresponding to the lemma that substitution in terms preserves typing (Sect. 6.4).

Section 6.5 presents the resulting runtime semantics and the properties of the gradual refinement language.

6.1 Evidence for Consistent Subtyping

Evidence represents the plausible static types that support some consistent judgment. Consider the valid consistent subtyping judgment $x{:}? \vdash \{\, \nu{:}\mathsf{Int} \mid \nu = x \,\} \lesssim \{\, \nu{:}\mathsf{Int} \mid \nu > 0 \,\}$. In addition to knowing *that* it holds, we know *why* it holds: for any satisfying interpretation of the gradual environment, x should be refined with a formula ensuring that it is positive. That is, we can deduce precise bounds on the set of static entities that supports why the consistent judgment holds. The abstraction of these static entities is what Garcia et al. [18] call *evidence*.

Because a consistent subtyping judgment involves a gradual environment and two gradual types, we extend the abstract interpretation framework coordinate-wise to *subtyping tuples*:[1]

Definition 17 (Subtyping tuple concretization)**.** *Let* $\gamma_\tau : \text{GTuple}^{<:} \to \mathcal{P}(\text{Tuple}^{<:})$ *be defined as:*

$$\gamma_\tau(\widetilde{\Phi}, \widetilde{T_1}, \widetilde{T_2}) = \gamma_{\Phi}(\widetilde{\Phi}) \times \gamma_T(\widetilde{T_1}) \times \gamma_T(\widetilde{T_2})$$

Definition 18 (Subtypting tuple abstraction)**.** *Let* $\alpha_\tau : \mathcal{P}(\text{Tuple}^{<:}) \rightharpoonup \text{GTuple}^{<:}$ *be defined as:*

$$\alpha_\tau(\{\, \overline{\Phi_i, T_{i1}, T_{i2}} \,\}) = \langle \alpha_{\Phi}(\{\, \overline{\Phi_i} \,\}), \alpha_T(\{\, \overline{T_{i1}} \,\}), \alpha_T(\{\, \overline{T_{i2}} \,\}) \rangle$$

This definition uses abstraction of gradual logical environments which is naturally defined as the pointwise lifting of the abstraction function for formulas.

Definition 19 (Abstraction for gradual logical environments)**.** *Let* $\alpha_{\Phi} : \mathcal{P}(\text{Env}) \rightharpoonup \text{GEnv}$ *be defined as:*

$$\alpha_{\Phi}(\widehat{\Phi})(x) = \alpha_p(\{\, \Phi(x) \mid \Phi \in \widehat{\Phi} \,\})$$

We can now define the *interior* of a consistent subtyping judgment, which captures the best coordinate-wise information that can be deduced from knowing that such a judgment holds.

Definition 20 (Interior)**.** *The interior of the judgment* $\widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2}$*, notation* $\mathcal{I}_{<:}(\widetilde{\Phi}, \widetilde{T_1}, \widetilde{T_2})$ *is defined by the function* $\mathcal{I}_{<:} : \text{GTuple}^{<:} \to \text{GTuple}^{<:}$*:*

$$\mathcal{I}_{<:}(\widetilde{\tau}) = \alpha_\tau(F_{\mathcal{I}_{<:}}(\gamma_\tau(\widetilde{\tau})))$$

where $F_{\mathcal{I}_{<:}} : \mathcal{P}(\text{Tuple}^{<:}) \to \mathcal{P}(\text{Tuple}^{<:})$

$$F_{\mathcal{I}_{<:}}(\widehat{\tau}) = \{\, \langle \Phi, T_1, T_2 \rangle \in \widehat{\tau} \mid \Phi \vdash T_1 <: T_2 \,\}$$

Based on interior, we define what counts as *evidence* for consistent subtyping. Evidence is represented as a tuple in $\text{GTuple}^{<:}$ that abstracts the possible satisfying static tuples. The tuple is *self-interior* to reflect the most precise information available:

[1] We pose $\tau \in \text{Tuple}^{<:} = \text{LEnv} \times \text{Type} \times \text{Type}$ for subtyping tuples, and $\text{GTuple}^{<:} = \text{GLEnv} \times \text{GType} \times \text{GType}$ for their gradual lifting.

Definition 21 (Evidence for consistent subtyping)**.**

$$\text{Ev}^{<:} = \{\, \langle \widetilde{\Phi}, \widetilde{T_1}, \widetilde{T_2} \rangle \in \text{GTuple}^{<:} \mid \mathcal{I}_{<:}(\widetilde{\Phi}, \widetilde{T_1}, \widetilde{T_2}) = \langle \widetilde{\Phi}, \widetilde{T_1}, \widetilde{T_2} \rangle \,\}$$

We use the metavariable ε to range over $\text{Ev}^{<:}$, and introduce the extended judgment $\varepsilon \triangleright \widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2}$, which associates particular runtime evidence to some consistent subtyping judgment. Initially, before a program executes, evidence ε corresponds to the interior of the judgment, also called the *initial evidence* [18].

The abstraction function α_τ inherits the partiality of α_p. As we did for the pair $\langle \alpha_{_T}, \gamma_{_T} \rangle$ in Section 5.5, we prove that $\langle \alpha_\tau, \gamma_\tau \rangle$ is a partial Galois connection for every operator of interest, starting with $F_{\mathcal{I}_{<:}}$, used in the definition of interior:

Proposition 16 (Partial Galois connection for interior)**.** *The pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a $\{\, F_{\mathcal{I}_{<:}} \,\}$-partial Galois connection.*

6.2 Consistent Subtyping Transitivity

The initial gradual typing derivation of a program uses initial evidence for each consistent judgment involved. As the program executes, evidence can be combined to exhibit evidence for other judgments. The way evidence evolves to provide evidence for further judgments mirrors the type safety proof, and justifications supported by properties about the relationship between static entities.

As noted by Garcia et al. [18], a crucial property used in the proof of preservation is transitivity of subtyping, which, because of the imprecision of gradual types, may or may not hold in the case of consistent subtyping judgments. For instance, both $\cdot \vdash \{\, \nu\!:\!\mathsf{Int} \mid \nu > 10 \,\} \lesssim \{\, \nu\!:\!\mathsf{Int} \mid ? \,\}$ and $\cdot \vdash \{\, \nu\!:\!\mathsf{Int} \mid ? \,\} \lesssim \{\, \nu\!:\!\mathsf{Int} \mid \nu < 10 \,\}$ hold, but $\cdot \vdash \{\, \nu\!:\!\mathsf{Int} \mid \nu > 10 \,\} \lesssim \{\, \nu\!:\!\mathsf{Int} \mid \nu < 10 \,\}$ does not. In some cases, however, it may be possible to provide justification that transitivity of consistent subtyping holds. Following AGT, we can formally define how to combine evidences to provide justification for *consistent subtyping*.

Definition 22 (Consistent subtyping transitivity)**.** *Suppose:*

$$\varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2} \quad \varepsilon_2 \triangleright \widetilde{\Phi} \vdash \widetilde{T_2} \lesssim \widetilde{T_3}$$

We deduce evidence for consistent subtyping transitivity *as*

$$(\varepsilon_1 \circ^{<:} \varepsilon_2) \triangleright \widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_3}$$

where $\circ^{<:} : \text{Ev}^{<:} \to \text{Ev}^{<:} \rightharpoonup \text{Ev}^{<:}$ *is defined by:*

$$\varepsilon_1 \circ^{<:} \varepsilon_2 = \alpha_\tau(F_{\circ<:}(\gamma_\tau(\varepsilon_1), \gamma_\tau(\varepsilon_2)))$$

and $F_{\circ<:} : \mathcal{P}(\text{Tuple}^{<:}) \to \mathcal{P}(\text{Tuple}^{<:}) \to \mathcal{P}(\text{Tuple}^{<:})$ *is:*

$$\begin{aligned} F_{\circ<:}(\hat{\tau}_1, \hat{\tau}_2) = \{ \langle \Phi, T_1, T_3 \rangle \mid \exists T_2.\ \langle \Phi, T_1, T_2 \rangle \in \hat{\tau}_1 \wedge \\ \langle \Phi, T_2, T_3 \rangle \in \hat{\tau}_2 \wedge \Phi \vdash T_1 <: T_2 \wedge \Phi \vdash T_2 <: T_3 \} \end{aligned}$$

The consistent transitivity operator collects and abstracts all available justifications that transitivity might hold in a particular instance. Consistent transitivity is a partial function: if $F_{\circ<:}$ produces an empty set, α_τ is undefined, and the transitive claim has been refuted. Intuitively this corresponds to a runtime cast error.

Consider, for example, the following evidence judgments:

$$\varepsilon_1 \triangleright \bullet \vdash \{\, \nu : \mathsf{Int} \mid \nu > 0 \wedge ? \,\} \lesssim \{\, \nu : \mathsf{Int} \mid ? \,\}$$
$$\varepsilon_2 \triangleright \bullet \vdash \{\, \nu : \mathsf{Int} \mid ? \,\} \lesssim \{\, \nu : \mathsf{Int} \mid \nu < 10 \,\}$$

where

$$\varepsilon_1 = \langle \bullet, \{\, \nu : \mathsf{Int} \mid \nu > 0 \wedge ? \,\}, \{\, \nu : \mathsf{Int} \mid ? \,\} \rangle$$
$$\varepsilon_2 = \langle \bullet, \{\, \nu : \mathsf{Int} \mid \nu < 10 \wedge ? \,\}, \{\, \nu : \mathsf{Int} \mid \nu < 10 \,\} \rangle$$

Using consistent subtyping transitivity we can deduce evidence for the judgment:

$$(\varepsilon_1 \circ^{<:} \varepsilon_2) \triangleright \bullet \vdash \{\, \nu : \mathsf{Int} \mid \nu > 0 \wedge ? \,\} \lesssim \{\, \nu : \mathsf{Int} \mid \nu < 10 \,\}$$

where

$$\varepsilon_1 \circ^{<:} \varepsilon_2 = \langle \bullet, \{\, \nu : \mathsf{Int} \mid \nu > 0 \wedge \nu < 10 \wedge ? \,\}, \{\, \nu : \mathsf{Int} \mid \nu < 10 \,\} \rangle$$

As required, $\langle \alpha_\tau, \gamma_\tau \rangle$ is a partial Galois connection for the operator used to define consistent subtyping transitivity.

Proposition 17 (Partial Galois connection for transitivity)**.** *The pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a $\{ F_{\circ<:} \}$-partial Galois connection.*

6.3 Consistent Subtyping Substitution

The proof of type preservation for refinement types also relies on a subtyping substitution lemma, stating that a subtyping judgment is preserved after a value is substituted for some variable x, and the binding for x is removed from the logical environment:

$$\frac{\Gamma \vdash \Phi_1 : v T_{11} \qquad \Phi_1 \vdash T_{11} <: T_{12} \qquad \Phi_1, x : (\!|T_{12}|\!), \Phi_2 \vdash T_{21} <: T_{22}}{\Phi_1, \Phi_2[v/x] \vdash T_{21}[v/x] <: T_{22}[v/x]}$$

In order to justify reductions of gradual typing derivations, we need to define an operator of *consistent subtyping substitution* that combines evidences from consistent subtyping judgments in order to derive evidence for the consistent subtyping judgment between types after substitution of v for x.

Definition 23 (Consistent subtyping substitution)**.** *Suppose:*

$$\Gamma \,;\, \widetilde{\Phi}_1 \vdash v : \widetilde{T}_{11} \quad \varepsilon_1 \triangleright \widetilde{\Phi}_1 \vdash \widetilde{T}_{11} \lesssim \widetilde{T}_{12}$$
$$\varepsilon_2 \triangleright \widetilde{\Phi}_1, x : (\!|\widetilde{T}_{12}|\!), \widetilde{\Phi}_2 \vdash \widetilde{T}_{21} \lesssim \widetilde{T}_{22}$$

Then we deduce evidence for consistent subtyping substitution *as*

$$(\varepsilon_1 \circ_{<:}^{[v/x]} \varepsilon_2) \triangleright \widetilde{\Phi}_1, \widetilde{\Phi}_2[\![v/x]\!] \vdash \widetilde{T}_{21}[\![v/x]\!] \lesssim \widetilde{T}_{22}[\![v/x]\!]$$

where $\circ_{<:}^{[v/x]} \colon \textsc{Ev}^{<:} \to \textsc{Ev}^{<:} \rightharpoonup \textsc{Ev}^{<:}$ *is defined by:*

$$\varepsilon_1 \circ_{<:}^{[v/x]} \varepsilon_2 = \alpha_\tau(F_{\circ_{<:}^{[v/x]}}(\gamma_\tau(\varepsilon_1), \gamma_\tau(\varepsilon_2)))$$

and $F_{\circ_{<:}^{[v/x]}} : \mathcal{P}(\textsc{Tuple}^{<:}) \to \mathcal{P}(\textsc{Tuple}^{<:}) \to \mathcal{P}(\textsc{Tuple}^{<:})$ *is:*

$$\begin{aligned} F_{\circ_{<:}^{[v/x]}}(\widehat{\tau}_1, \widehat{\tau}_2) = \{ \langle \Phi_1 \cdot \Phi_2[{}^{v}/_{x}], T_{21}[{}^{v}/_{x}], T_{22}[{}^{v}/_{x}] \rangle \mid \\ \exists T_{11}, T_{12}.\ \langle \Phi_1, T_{11}, T_{12} \rangle \in \widehat{\tau}_1 \wedge \\ \langle \Phi_1 \cdot x : (\!|T_{12}|\!) \cdot \Phi_2, T_{21}, T_{22} \rangle \in \widehat{\tau}_2 \wedge \\ \Phi_1 \vdash T_{11} <: T_{12} \wedge \Phi_1 \cdot x : (\!|T_{12}|\!) \cdot \Phi_2 \vdash T_{21} <: T_{22} \} \end{aligned}$$

The consistent subtyping substitution operator collects and abstracts all justifications that some consistent subtyping judgment holds after substituting in types with a value, and produces the most precise evidence, if possible. Note that this new operator introduces a new category of runtime errors, made necessary by dependent types, and hence not considered in the simply-typed setting of Garcia et al. [18].

To illustrate consistent subtyping substitution consider:

$$\begin{gathered} \cdot \vdash \cdot : 3\{\, \nu : \mathsf{Int} \mid \nu = 3 \,\} \\ \varepsilon_1 \triangleright \cdot \vdash \{\, \nu : \mathsf{Int} \mid \nu = 3 \,\} \lesssim \{\, \nu : \mathsf{Int} \mid ? \,\} \\ \varepsilon_2 \triangleright x : ?, y : ? \vdash \{\, \nu : \mathsf{Int} \mid \nu = x + y \,\} \lesssim \{\, \nu : \mathsf{Int} \mid \nu \geq 0 \,\} \end{gathered}$$

where

$$\begin{aligned} \varepsilon_1 &= \langle \cdot, \{\, \nu : \mathsf{Int} \mid \nu = 3 \,\}, \{\, \nu : \mathsf{Int} \mid ? \,\} \rangle \\ \varepsilon_2 &= \langle x : ? \cdot y : ?, \{\, \nu : \mathsf{Int} \mid \nu = x + y \,\}, \{\, \nu : \mathsf{Int} \mid \nu \geq 0 \,\} \rangle \end{aligned}$$

We can combine ε_1 and ε_2 with the consistent subtyping substitution operator to justify the judgment after substituting 3 for x:

$$(\varepsilon_1 \circ_{<:}^{[3/x]} \varepsilon_2) \triangleright y : ? \vdash \{\, \nu : \mathsf{Int} \mid \nu = 3 + y \,\} \lesssim \{\, \nu : \mathsf{Int} \mid \nu \geq 0 \,\}$$

where

$$\varepsilon_1 \circ_{<:}^{[3/x]} \varepsilon_2 = \langle (y : \nu \geq -3 \wedge ?), \{\, \nu : \mathsf{Int} \mid \nu = 3 + y \,\}, \{\, \nu : \mathsf{Int} \mid \nu \geq 0 \,\} \rangle$$

Once again the pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a partial Galois connection for the operator of interest.

Proposition 18 (Partial Galois connection for subtyping substitution)**.** *The pair* $\langle \alpha_\tau, \gamma_\tau \rangle$ *is a* $\{\, F_{\circ_{<:}^{[v/x]}} \,\}$*-partial Galois connection.*

6.4 Consistent Term Substitution

Another important aspect of the proof of preservation is the use of a term substitution lemma, *i.e.* substituting in an open term preserves typing. Even in the simply-typed setting considered by Garcia et al. [18], the term substitution lemma does not hold for the gradual language because it relies on subtyping transitivity. Without further discussion, they adopt a simple technique: instead of substituting a plain value v for the variable x, they substitute an *ascribed* value $v :: \widetilde{T}$, where $\widetilde{T}$ is the expected type of x. This technique ensures that the substitution lemma always holds.

With dependent types, the term substitution lemma is more challenging. A subtyping judgment can rely on the plausibility that a gradually-type variable is replaced with the right value, which may not be the case at runtime. Consider the following example:

$$\begin{array}{l}\textbf{let}\ \ f\ (x\colon \{\,\nu\colon \mathsf{Int} \mid \nu > 0\,\}) = x\\ \textbf{let}\ \ g\ (x\colon \{\,\nu\colon \mathsf{Int} \mid ?\,\})\ (y\colon \{\,\nu\colon \mathsf{Int} \mid \nu \geq x\,\}) =\\ \quad \textbf{let}\ \ z = f\ y\ \ \textbf{in}\ \ z + x\end{array}$$

This code is accepted statically due to the possibility of x being positive inside the body of g. If we call g with -1 the application $f\ y$ can no longer be proven possibly safe. Precisely, the application $f\ y$ relies on the consistent subtyping judgment

$$x\colon ?\cdot y\colon \nu \geq x \vdash \{\,\nu\colon \mathsf{Int} \mid \nu = y\,\} \lesssim \{\,\nu\colon \mathsf{Int} \mid \nu > 0\,\}$$

supported by the evidence

$$\langle x\colon \nu > 0 \wedge ?\cdot y\colon \nu \geq x, \{\,\nu\colon \mathsf{Int} \mid \nu = y\,\}, \{\,\nu\colon \mathsf{Int} \mid \nu > 0\,\}\rangle$$

After substituting by -1 the following judgment must be justified:

$$y\colon \nu \geq -1 \vdash \{\,\nu\colon \mathsf{Int} \mid \nu = y\,\} \lesssim \{\,\nu\colon \mathsf{Int} \mid \nu > 0\,\}$$

This (fully precise) judgment cannot however be supported by any evidence.

Note that replacing by an ascribed value does not help in the dependently-typed setting because, as illustrated by the previous example, judgments that must be proven after substitution may not even correspond to syntactic occurrences of the replaced variable. Moreover, substitution also pervades types, and consequently formulas, but the logical language has no notion of ascription.

Stepping back, the key characteristic of the ascription technique used by Garcia et al. [18] is that the resulting substitution operator on gradual terms preserves exact types. Noting that after substitution some consistent subtyping judgments may fail, we define a *consistent term substitution* operator that preserves typeability, but is undefined if it cannot produce evidence for some judgment. This yields yet another category of runtime failure, occurring at substitution time. In the above example, the error manifests as soon as the application $g\ {-1}$ beta-reduces, before reducing the body of g.

Consistent term substitution relies on the consistent subtyping substitution operator defined in Section 6.2 to produce evidence for consistent subtyping judgments that result from substitution. We defer its exact definition to Section 6.5.3 below.

6.5 Dynamic Semantics and Properties

We now turn to the actual reduction rules of the gradual language with refinement types. Following AGT, reduction is expressed over gradual *typing derivations*, using the consistent operators mentioned in the previous section. Because writing down reduction rules over (bidimensional) derivation trees is unwieldy, we use *instrincally-typed terms* [10] as a convenient unidimensional notation for derivation trees [18].

We expose this notational device in Section 6.5.1, and then use it to present the reduction rules (Section 6.5.2) and the definition of the consistent term substitution operator (Section 6.5.3). Finally, we state the meta-theoretic properties of the resulting language: type safety, gradual guarantee, and refinement soundness (Section 6.5.4).

6.5.1 Intrinsic Terms

We first develop gradual intrinsically-typed terms, or gradual intrinsic terms for short. Intrinsic terms are isomorphic to typing derivation trees, so their structure corresponds to the gradual typing judgment $\Gamma \,;\, \widetilde{\Phi} \vdash t : \widetilde{T}$—a term is given a type in a specific type environment and gradual logical environment. Intrinsic terms are built up from disjoint families of intrinsically-typed variables $x^{\widetilde{T}} \in \textsc{Var}_{\widetilde{T}}$. Because these variables carry type information, type environments Γ are not needed in intrinsic terms. Because typeability of a term depends on its logical context, we define a family $\textsc{Term}^{\widetilde{\Phi}}_{\widetilde{T}}$ of sets indexed by both types and gradual logical environments. For readability, we use the notation $\widetilde{\Phi} \,;\, t^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}$, allowing us to view an intrinsic term as made up of a logical environment and a term (when $\widetilde{\Phi}$ is empty we stick to $\textsc{Term}_{\widetilde{T}}$).

Figure 6.1 presents formation rules of intrinsic terms. Rules (In), (Ib), (Ix-refine) and (Ix-fun) are straightforward. Rule (Iλ) requires the body of the lambda to be typed in an extended logical environment. Note that because gradual typing derivations include evidence for consistent judgments, gradual intrinsic terms carry over evidences as well, which can be seen in rule (Iapp), (I::), (Iif) and (Ilet). The rule for application additionally features a type annotation with the @ notation. As observed by Garcia et al. [18], this annotation is necessary because intrinsic terms represent typing derivations *at different steps of reduction*. Therefore, they must account for the fact that runtime values can have more precise types than the ones determined statically. For example, a term t in function position of an application may reduce to some term whose type is a subtype of the type given to t statically. An intrinsic application term hence carries the type given statically to the subterm in function position. This notation can also be seen in rules (Iif) and (Ilet).

6.5.2 Reduction

Figure 6.2 presents the syntax of the intrinsic term language and its evaluation frames, in the style of Garcia et al. [18]. Values v are either raw values u or ascribed values $\varepsilon u :: \widetilde{T}$, where ε

$$\text{(In)}\ \frac{}{\widetilde{\Phi}\ ;\ n \in \textsc{Term}_{\{\,\nu:\mathsf{Int}\,\mid\,\nu=n\,\}}} \qquad \text{(Ib)}\ \frac{}{\widetilde{\Phi}\ ;\ b \in \textsc{Term}_{\{\,\nu:\mathsf{Bool}\,\mid\,\nu=b\,\}}}$$

$$\text{(Ix-refine)}\ \frac{}{\widetilde{\Phi}\ ;\ x^{\{\,\nu:B\,\mid\,\widetilde{p}\,\}} \in \textsc{Term}_{\{\,\nu:B\,\mid\,\nu=x\,\}}} \qquad \text{(Ix-fun)}\ \frac{}{\widetilde{\Phi}\ ;\ x^{y:\widetilde{T}_1\to\widetilde{T}_2} \in \textsc{Term}_{y:\widetilde{T}_1\to\widetilde{T}_2}}$$

$$\text{(I}\lambda\text{)}\ \frac{\widetilde{\Phi}, x:(\!|\widetilde{T}_1|\!)\ ;\ t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2}}{\widetilde{\Phi}\ ;\ \lambda x^{\widetilde{T}_1}.t^{\widetilde{T}_2} \in \textsc{Term}_{x:\widetilde{T}_1\to\widetilde{T}_2}} \qquad \text{(I::)}\ \frac{\begin{array}{c}\widetilde{\Phi}\ ;\ t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \\ \varepsilon \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2\end{array}}{\widetilde{\Phi}\ ;\ \varepsilon t^{\widetilde{T}_1} :: \widetilde{T}_2 \in \textsc{Term}_{\widetilde{T}_2}}$$

$$\text{(Iapp)}\ \frac{\begin{array}{cc}\widetilde{\Phi}\ ;\ ^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} & \varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim (x:\widetilde{T}_{11} \to y:\widetilde{T}_{12} \to) \\ \widetilde{\Phi}\ ;\ v \in \textsc{Term}_{\widetilde{T}_2} & \varepsilon_2 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T}_{11}\end{array}}{\widetilde{\Phi}\ ;\ (\varepsilon_1 t^{\widetilde{T}_1})@^{x\widetilde{T}_{11}\to\widetilde{T}_{12}}\ (\varepsilon_2 v) \in \textsc{Term}_{\widetilde{T}_{12}[\![v/x]\!]}}$$

$$\text{(Iif)}\ \frac{\begin{array}{c}\widetilde{\Phi}\ ;\ u \in \textsc{Term}_{\{\,\nu:\mathsf{Bool}\,\mid\,\widetilde{p}\,\}} \\ \widetilde{\Phi}, x:(v{=}\mathsf{true})\ ;\ t^{\widetilde{T}_1} \in \textsc{Term}_{\widetilde{T}_1} \qquad \varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T} \\ \widetilde{\Phi}, x:(v{=}\mathsf{false})\ ;\ t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2} \qquad \varepsilon_2 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_2 \lesssim \widetilde{T}\end{array}}{\widetilde{\Phi}\ ;\ (\mathsf{if}\ u\ \mathsf{then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}}$$

$$\text{(Ilet)}\ \frac{\begin{array}{cc}\widetilde{\Phi}\ ;\ t^{\widetilde{T}_{11}} \in \textsc{Term}_{\widetilde{T}_{11}} & \varepsilon_1 \triangleright \widetilde{\Phi} \vdash \widetilde{T}_{11} \lesssim \widetilde{T}_{12} \\ \widetilde{\Phi}, x:\widetilde{T}_{12}\ ;\ t^{\widetilde{T}_2} \in \textsc{Term}_{\widetilde{T}_2} & \varepsilon_2 \triangleright \widetilde{\Phi}, x:(\!|\widetilde{T}_{12}|\!) \vdash \widetilde{T}_2 \lesssim \widetilde{T}\end{array}}{\widetilde{\Phi}\ ;\ (\mathsf{let}\ x^{\widetilde{T}_{12}} = \varepsilon_1 t^{\widetilde{T}_{11}}\ \mathsf{in}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}}$$

Figure 6.1: Gradual intrinsic terms.

is the evidence that u is of a subtype of $\widetilde{T}$. Such a pair $\varepsilon u \in \textsc{EvValue}$ is called an *evidence value*. Similarly, an *evidence term* $\varepsilon t \in \textsc{EvTerm}$ is a term augmented with evidence. We use $\textsc{Var}_*$ (resp. $\textsc{Term}^*_*$) to denote the set of all intrinsic variables (resp. terms).

Figure 6.2 presents the reduction relation $\hookrightarrow$ and the two notions of reductions $\longrightarrow$ and $\longrightarrow_c$. Reduction rules preserve the exact type of a term and explicit ascriptions are used whenever a reduction may implicitly affect type precision. The rules handle evidences, combining them with consistent operators to derive new evidence to form new intrinsic terms. Whenever combining evidences fails, the program ends with an **error**. An application may produce an error because it cannot produce evidence using consistent transitivity to justify that the actual argument is subtype of the formal argument. Additionally, the rules for application and let expression use consistent term substitution, which fails whenever consistent subtyping substitution cannot combine evidences to justify all substituted occurrences.

$et \in \textsc{EvTerm}, \quad ev \in \textsc{EvValue}, \quad u \in \textsc{SimpleValue}, \quad x^* \in \textsc{Var}_*$
$t \in \textsc{Term}^*_*, \quad v \in \textsc{Value}, \quad g \in \textsc{EvFrame}, \quad f \in \textsc{TmFrame}$

$$
\begin{array}{rcl}
et & ::= & \varepsilon t \\
ev & ::= & \varepsilon u \\
u & ::= & x^* \mid n \mid b \mid \lambda x^*.t^* \\
v & ::= & u \mid \varepsilon u :: \widetilde{T} \\
g & ::= & \square \ @^{\widetilde{T}} et \mid ev \ @^{\widetilde{T}} \square \mid \square :: \widetilde{T} \mid (\mathsf{let}\ x = \square \ @^{\widetilde{T}}\ \mathsf{in}\ et)@^{\widetilde{T}} \\
f & ::= & g[\varepsilon\square]
\end{array}
$$

$\boxed{\longrightarrow : \textsc{Term}^{\cdot}_{\widetilde{T}} \times (\textsc{Term}^{\cdot}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}$

$$
\varepsilon_1(\lambda x^{\widetilde{T}_{11}}.t)@^{x\widetilde{T}_1 \to \widetilde{T}_2}\varepsilon_2 u \longrightarrow \begin{cases} icod_u(\varepsilon_2, \varepsilon_1)t[(\varepsilon_2 \circ^{<:} idom(\varepsilon_1))u/x^{\widetilde{T}_{11}}] :: \widetilde{T}_2[\![u/x]\!] \\ \mathbf{error} \qquad \text{if } (\varepsilon_2 \circ^{<:} idom(\varepsilon_1)), icod_u(\varepsilon_2, \varepsilon_1) \text{ or} \\ \qquad\qquad t[\varepsilon_u u/x^{\widetilde{T}_{11}}] \text{ is not defined} \end{cases}
$$

$$
(\mathsf{let}\ x^{\widetilde{T}_1} = \varepsilon_1 u\ \mathsf{in}\ \varepsilon_2 t)@^{\widetilde{T}} \longrightarrow \begin{cases} (\varepsilon_1 \circ_{<:}^{[v/x]} \varepsilon_2)t[\varepsilon_1 u/x^{\widetilde{T}_1}] :: \widetilde{T} \\ \mathbf{error} \quad \text{if } t[\varepsilon_1 u/x^{\widetilde{T}_1}] \text{ or} \\ \quad (\varepsilon_1 \circ_{<:}^{[v/x]} \varepsilon_2) \text{ is not defined} \end{cases}
$$

$$
(\mathsf{if\ true\ then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}} \longrightarrow \varepsilon_1 t^{\widetilde{T}_1} :: \widetilde{T}
$$

$$
(\mathsf{if\ false\ then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}} \longrightarrow \varepsilon_2 t^{\widetilde{T}_2} :: \widetilde{T}
$$

$\boxed{\longrightarrow_c : \textsc{EvTerm} \times (\textsc{EvTerm} \cup \{\,\mathbf{error}\,\})}$

$$
\varepsilon_1(\varepsilon_2 u :: \widetilde{T}) \longrightarrow_c \begin{cases} (\varepsilon_2 \circ^{<:} \varepsilon_1)u \\ \mathbf{error} \qquad \text{if } (\varepsilon_2 \circ^{<:} \varepsilon_1) \text{ is not defined} \end{cases}
$$

$\boxed{\hookrightarrow : \textsc{Term}^{\cdot}_{\widetilde{T}} \times (\textsc{Term}^{\cdot}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}$

$$
(\mathrm{R}\hookrightarrow)\frac{t^{\widetilde{T}} \longrightarrow r \qquad r \in (\textsc{Term}^{\cdot}_{\widetilde{T}} \cup \{\,\mathbf{error}\,\})}{t^{\widetilde{T}} \hookrightarrow r} \qquad (\mathrm{R}g)\frac{et \longrightarrow_c et'}{g[et] \hookrightarrow g[et']}
$$

$$
(\mathrm{R}g\mathrm{err})\frac{et \longrightarrow_c \mathbf{error}}{g[et] \hookrightarrow \mathbf{error}} \qquad (\mathrm{R}f)\frac{t_1^{\widetilde{T}} \hookrightarrow t_2^{\widetilde{T}}}{f[t_1^{\widetilde{T}}] \hookrightarrow f[t_2^{\widetilde{T}}]} \qquad (\mathrm{R}f\mathrm{err})\frac{t^{\widetilde{T}} \hookrightarrow \mathbf{error}}{f[t^{\widetilde{T}}] \hookrightarrow \mathbf{error}}
$$

Figure 6.2: Intrinsic reduction

$\boxed{(\cdot)[\cdot/\cdot] : \textsc{Term}^*_* \to \textsc{EvValue} \to \textsc{Var}_* \rightharpoonup \textsc{Term}^*_*}$

$n[\varepsilon u/x^{\widetilde{T}}] = n \qquad\qquad b[\varepsilon u/x^{\widetilde{T}}] = b$

$x^{\{\,\nu:B\,|\,\tilde{p}\,\}}[\varepsilon u/x^{\{\,\nu:B\,|\,\tilde{p}\,\}}] = u \qquad y^{\widetilde{T}_2}[\varepsilon u/x^{\widetilde{T}_1}] = y^{\widetilde{T}[\![u/x]\!]} \quad \text{if } x^{\widetilde{T}_1} \neq y^{\widetilde{T}_2}$

$(\varepsilon_1 t :: \widetilde{T}_2)[\varepsilon u/x^{\widetilde{T}}] = (\varepsilon_1 t)[\varepsilon u/x^{\widetilde{T}}] :: \widetilde{T}_2[\![u/x]\!]$

$x^{y:\widetilde{T}_1 \to \widetilde{T}_2}[\varepsilon u/x^{y:\widetilde{T}_1 \to \widetilde{T}_2}] = \varepsilon u :: (y{:}\widetilde{T}_1 \to \widetilde{T}_2)$

$(\lambda y^{\widetilde{T}}.t)[\varepsilon u/x^{\widetilde{T}}] = \lambda y^{\widetilde{T}[\![u/x]\!]}.t[\varepsilon u/x^{\widetilde{T}}]$

$((\varepsilon_1 t^{\widetilde{T}_1})@^{x\widetilde{T}_{11}\to\widetilde{T}_{12}}\ (\varepsilon_2 v))[\varepsilon u/x^{\widetilde{T}}] = (\varepsilon_1 t^{\widetilde{T}_1})[\varepsilon u/x^{\widetilde{T}}]@^{(x\widetilde{T}_{11}\to\widetilde{T}_{12})[\![u/x]\!]}(\varepsilon_2 v)[\varepsilon u/x^{\widetilde{T}}]$

$((\mathsf{if}\ v\ \mathsf{then}\ \varepsilon_1 t^{\widetilde{T}_1}\ \mathsf{else}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}})[\varepsilon u/x^{\widetilde{T}}] =$
$\quad (\mathsf{if}\ v[\varepsilon u/x^{\widetilde{T}}]\ \mathsf{then}\ (\varepsilon_1 t^{\widetilde{T}_1})[\varepsilon u/x^{\widetilde{T}}]\ \mathsf{else}\ (\varepsilon_2 t^{\widetilde{T}_2})[\varepsilon u/x^{\widetilde{T}}])@^{\widetilde{T}[\![u/x]\!]}$

$((\mathsf{let}\ y^{\widetilde{T}_{12}} = \varepsilon_1 t^{\widetilde{T}_{11}}\ \mathsf{in}\ \varepsilon_2 t^{\widetilde{T}_2})@^{\widetilde{T}})[\varepsilon u/x^{\widetilde{T}}] =$
$\quad (\mathsf{let}\ y^{\widetilde{T}_{12}[\![u/x]\!]} = (\varepsilon_1 t^{\widetilde{T}_{11}})[\varepsilon u/x^{\widetilde{T}}]\ \mathsf{in}\ (\varepsilon_2 t^{\widetilde{T}_2})[\varepsilon u/x^{\widetilde{T}}])@^{\widetilde{T}[\![u/x]\!]}$

$\boxed{(\cdot)[\cdot/\cdot] : \textsc{EvTerm} \to \textsc{EvValue} \to \textsc{Var}^* \rightharpoonup \textsc{EvTerm}}$

$(\varepsilon_1 t^{\widetilde{T}_2})[\varepsilon u/x^{\widetilde{T}_1}] = (\varepsilon \circ^{[u/x]}_{<:} \varepsilon_1)t^{\widetilde{T}_2}[\varepsilon u/x^{\widetilde{T}_1}]$

Figure 6.3: Consistent term substitution.

6.5.3 Consistent Term Substitution

The consistent term substitution operator described in Section 6.4 is defined on intrinsic terms (Figure 6.3). To substitute a variable $x^{\widetilde{T}}$ by a value u we must have evidence justifying that the type of u is a subtype of $\widetilde{T}$, supporting that substituting by u may be safe. Therefore, consistent term substitution is defined for evidence values. The consistent term substitution operator recursively traverses the structure of an intrinsic term applying consistent subtyping substitution to every evidence, using an auxiliary definition for substitution into evidence terms. When substituting by an evidence value $\varepsilon_1 u$ in an evidence term $\varepsilon_2 t$, we first combine ε_1 and ε_2 using consistent subtyping substitution and then substitute recursively into t. Note that substitution is undefined whenever consistent subtyping substitution is undefined.

When reaching a variable, there is a subtle difference between substituting by a lambda and a base constant. Because variables with base types are given the exact type $\{\,\nu{:}B \mid \nu = x\,\}$, after substituting x by a value u the type becomes $\{\,\nu{:}B \mid \nu = u\,\}$, which exactly corresponds to the type for a base constant. For higher order variables an explicit ascription is needed to preserve the same type. Another subtlety is that types appearing in annotations above @ must be replaced by the same type, but substituting for the variable x being replaced. This is necessary for the resulting term to be well-typed in an environment where the binding for the substituted variable has been removed from the logical environment. Similarly an intrinsic variable $y^{\widetilde{T}}$ other than the one being replaced must be replaced by a variable $y^{\widetilde{T}[\![u/x]\!]}$.

The key property is that consistent term substitution preserves typeability whenever it is defined.

Proposition 19 (Consistent substitution preserves types)**.** *Suppose* $\widetilde{\Phi}_1 \,;\, u \in \textsc{Term}_{\widetilde{T}_u}$, $\varepsilon \triangleright \widetilde{\Phi}_1 \vdash \widetilde{T}_u \lesssim \widetilde{T}_x$, *and* $\widetilde{\Phi}_1 \cdot x{:}(\!|\widetilde{T}_x|\!) \cdot \widetilde{\Phi}_2 \,;\, t \in \textsc{Term}_{\widetilde{T}}$ *then* $\widetilde{\Phi}_1 \cdot \widetilde{\Phi}_2[\![u/x]\!] \,;\, t[\varepsilon u/x^{\widetilde{T}_x}] \in \textsc{Term}_{\widetilde{T}[\![u/x]\!]}$ *or* $t[\varepsilon u/x^{\widetilde{T}_x}]$ *is undefined.*

6.5.4 Properties of the Gradual Refinement Types Language

We establish three fundamental properties based on the dynamic semantics. First, the gradual language is type safe by construction.

Proposition 20 (Type Safety)**.** *If* $t_1^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}^{\cdot}$ *then either* $t_1^{\widetilde{T}}$ *is a value* v, $t_1^{\widetilde{T}} \hookrightarrow t_2^{\widetilde{T}}$ *for some term* $t_2^{\widetilde{T}} \in \textsc{Term}_{\widetilde{T}}^{\cdot}$, *or* $t_1^{\widetilde{T}} \hookrightarrow \mathbf{error}$.

More interestingly, the language satisfies the dynamic gradual guarantee of Siek et al. [33]: a well-typed gradual program that runs without errors still does with less precise type annotations.

Proposition 21 (Dynamic gradual guarantee)**.** *Suppose* $t_1^{\widetilde{T}_1} \sqsubseteq t_1^{\widetilde{T}_2}$. *If* $t_1^{\widetilde{T}_1} \hookrightarrow t_2^{\widetilde{T}_1}$ *then* $t_1^{\widetilde{T}_2} \hookrightarrow t_2^{\widetilde{T}_2}$ *where* $t_2^{\widetilde{T}_1} \sqsubseteq t_2^{\widetilde{T}_2}$.

We also establish refinement soundness: the result of evaluating a term yields a value that

complies with its refinement. This property is a direct consequence of type preservation.

Proposition 22 (Refinement soundness)**.** *If* $t^{\{\,\nu:B\,|\,\widetilde{p}\,\}} \in \textsc{Term}^{\cdot}_{\{\,\nu:B\,|\,\widetilde{p}\,\}}$ *and* $t^{\{\,\nu:B\,|\,\widetilde{p}\,\}} \hookrightarrow^{*} v$ *then:*

1. *If* $v = u$ *then* $(\!|\widetilde{p}|\!)_!{[^u/_\nu]}$ *is valid*
2. *If* $v = \varepsilon u :: \{\,\nu:B \mid \widetilde{p}\,\}$ *then* $(\!|\widetilde{p}|\!)_!{[^u/_\nu]}$ *is valid*

where $(\!|\widetilde{p}|\!)_!$ extracts the static part of $\widetilde{p}$.

Chapter 7

Towards a Practical Implementation

In previous chapters we established the foundations for the gradual refinement types system and proved its crucial methatheoretical properties. While the system satisfies many interesting properties it is far from being practical yet. In this chapter we step into the pragmatics of the system.

First, we analyze the algorithmics of gradual refinement types and devise procedures for the consistent operators used in the static semantics. Most notably, we design a syntax directed characterization of consistent subtyping, which is a decision procedure for the theory of linear arithmetic. Second, we discus how the system could be extended to support a more expressive languages of refinements. In particular, we show how to extend the system to support *measures* [34], which allow reasoning about algebraic data types. Finally, we end up by further discussing the dynamic semantics proposed in Chapter 6, but from a practical point of view. We identify some issues and shed some light into the direction that may be taken to target a practical implementation. Overall, this raises interesting venues for future work.

7.1 Algorithmic Consistent Type Substitution

In Section 4.2 we gave a characterization of consistent type substitution based on the abstract interpretation framework. We now give an algorithmic characterization considering the already fixed local interpretation. We note that this version simply substitutes in the known parts of formulas.

Definition 24 (Algorithmic consistent type substitution)**.**

$$\begin{aligned}
\{\, \nu{:}B \mid p \,\}\, [\![v/x]\!] &= \{\, \nu{:}B \mid p[^{v}/_{x}] \,\} \\
\{\, \nu{:}B \mid p \wedge ? \,\}\, [\![v/x]\!] &= \{\, \nu{:}B \mid p[^{v}/_{x}] \wedge ? \,\} \\
(y{:}\widetilde{T}_1 \to \widetilde{T}_2)[\![v/x]\!] &= y{:}\widetilde{T}_1[\![v/x]\!] \to \widetilde{T}_2[\![v/x]\!]
\end{aligned}$$

We prove that this characterization is equivalent to Definition 6 when considering the local

interpretation of formulas.

Proposition 23. $\widetilde{\widetilde{T}[v/x]} = \widetilde{T}[\![v/x]\!]$

To prove this equivalence we only need to establish that substitution preserves upper-bounds and locality.

Lemma 24. *If* $p \preceq q$ *then* $p[^{v}/_{x}] \preceq p[^{v}/_{x}]$.

Lemma 25. *If* $p \in \textsc{Local}$ *then* $p[^{v}/_{x}] \in \textsc{Local}$.

7.2 Algorithmic Consistent Subtyping

While some definitions can be easily characterized algorithmically, consistent subtyping (Sect. 4.2) is both central and particularly challenging. We now present a syntax-directed characterization of consistent subtyping, which is a decision procedure when refinements are drawn from the theory of linear arithmetic.

The algorithmic characterization is based on solving *consistent entailment constraints* of the form $\widetilde{\Phi} \;\mathrel{|\!\approx}\; \widetilde{q}$. Solving such a constraint consists in finding a well-formed environment $\Phi \in \gamma_{\Phi}(\widetilde{\Phi})$ and a formula $q \in \gamma_{p}(\widetilde{q})$ such that $(\!|\Phi|\!) \models q$. We use the notation $\mathrel{|\!\approx}$ to mirror $\models$ in a consistent fashion. However, note that $\mathrel{|\!\approx}$ does not correspond to the consistent lifting of $\models$, because entailment is defined for sets of formulas while consistent entailment is defined for (ordered) gradual logical environments. This is important to ensure well-formedness of logical environments.

As an example consider the consistent entailment constraint:

$$x{:}?, y{:}?, z{:}(\nu \geq 0) \;\mathrel{|\!\approx}\; x + y + z \geq 0 \wedge x \geq 0 \wedge ? \tag{7.1}$$

First, note that the unknown on the right hand side can be obviated, because anything implying something more specific than $x+y+z \geq 0 \wedge x \geq 0$ can also imply $x+y+z \geq 0 \wedge x \geq 0$ and it suffices to imply $x+y+z \geq 0 \wedge x \geq 0$. So, to solve the constraint we must find formulas that restrict the possible values of x and y such that $x + y + z \geq 0 \wedge x \geq 0$ is always true. There are many ways to achieve this; we are only concerned about the *existence* of such an environment.

We describe a canonical approach to determine whether a consistent entailment is valid, by reducing it to a fully static judgment.[1] Let us illustrate how to reduce constraint (7.1) above. We first focus on the rightmost gradual formula in the environment, for y, and consider a static formula that guarantees the goal, using the static information further right. Here, this means binding y to $\forall z.z \geq 0 \rightarrow (x + \nu + z \geq 0 \wedge x \geq 0)$. After quantifier elimination, this formula is equivalent to $x + \nu \geq 0 \wedge x \geq 0$. Because this formula is not local, we retain the strongest possible local formula that corresponds to it. In general, given a formula $q(\nu)$, the

[1] Our approach relies on the theory of linear arithmetic being full first order (including quantifiers) decidable—see discussion at the end of Section 7.3.

formula $\exists\nu.q(\nu)$ captures the non-local part of $q(\nu)$, so the formula $(\exists\nu.q(\nu)) \to q(\nu)$ is local. Here, the non-local information is $\exists\nu.x+\nu \geq 0 \wedge x \geq 0$, which is equivalent to $x \geq 0$, so the local formula for y is $x \geq 0 \to x+\nu \geq 0$. Constraint (7.1) is reduced to:

$$x{:}?, y{:}x \geq 0 \to x+\nu \geq 0, z{:}\nu \geq 0 \;\not\approx\; x+y+z \geq 0 \wedge x \geq 0$$

Applying the same reduction approach focusing on x, we obtain (after extraction) the following *static* entailment, which is valid:

$$\{\, x \geq 0, x \geq 0 \to x+y \geq 0, z \geq 0 \,\} \models x+y+z \geq 0 \wedge x \geq 0$$

Thus the consistent entailment constraint (7.1) can be satisfied.

With function types, subtyping conveys many consistent entailment constraints that must be handled *together*, because the same interpretation for an unknown formula must be maintained between different constraints. The reduction approach above can be extended to the higher-order case noting that constraints involved in subtyping form a tree structure, sharing common prefixes.

Proposition 26 (Constraint reduction)**.** *Consider a set of consistent entailment constrains sharing a common prefix* $(\widetilde{\Phi}_1, y{:}(p(\vec{x},\nu) \wedge ?))$*:*

$$\{\, \overline{\widetilde{\Phi}_1, y{:}(p(\vec{x},\nu) \wedge ?), \Phi_2^i \not\approx r_i(\vec{x}, y, \vec{z_i})} \,\}$$

Where $\vec{x} = dom(\widetilde{\Phi}_1)$ *(resp.* $\vec{z_i} = dom(\Phi_2^i)$*) is the set of variables bound in* $\widetilde{\Phi}_1$ *(resp.* Φ_2^i*). Let* $\vec{z} = \bigcup_i \vec{z_i}$ *and define the canonical formula* $q(\vec{x},\nu)$ *and its local restriction* $q'(\vec{x},\nu)$ *as follows:*

$$q(\vec{x},\nu) = (\forall \vec{z}, \bigwedge_i (\llparenthesis \Phi_2 \rrparenthesis \to r_i(\vec{x},\nu,\vec{z_i}))) \wedge p(\vec{x},\nu)$$
$$q'(\vec{x},\nu) = (\exists\nu, q(\vec{x},\nu)) \to q(\vec{x},\nu) \wedge \neg(\exists\nu, q(\vec{x},\nu)) \to p(\vec{x},\nu)$$

Let $\Phi_1 \in \gamma_{_\Phi}(\widetilde{\Phi}_1)$ *be any logical environment in the concretization of* $\widetilde{\Phi}_1$*. The following proposition holds: there exists* $p'(\vec{x},\nu) \in \gamma_p(p(\vec{x},\nu) \wedge ?)$ *such that* $\llparenthesis \Phi_1, y{:}p'(\vec{x},\nu), \Phi_2^i \rrparenthesis \models r_i(\vec{x}, y, \vec{z_i})$ *for every* i *if and only if* $\llparenthesis \Phi_1, y{:}q'(\vec{x},\nu), \Phi_2^i \rrparenthesis \models r_i(\vec{x}, y, \vec{z_i})$ *for every* i*.*

In words, when a set of consistent entailment constraints share the same prefix $\widetilde{\Phi}_1, y : p(\vec{x},\nu) \wedge ?$, we can replace the rightmost gradual formula $p(\vec{x},\nu) \wedge ?$ by a canonical local *static* formula $q'(\vec{x},\nu)$ that justifies the satisfiability of the constraints. This reduction preserves the set of interpretations of the prefix $\widetilde{\Phi}_1$ that justify the satisfaction of the constraints. As discussed in the example above we first consider a formula $q(\vec{x},\nu)$ that guarantees the goal using the static information further right. In this case, however, all constraints must be taken into account at once. Additionally, we must ensure this formula is more specific than the bound $p(\vec{x},\nu)$ of the gradual formula being considered, thus we take the conjunction. Then, we massage the formula to obtain a local formula $q'(\vec{x},\nu)$, with care in remaining more specific than $p(\vec{x},\nu)$.

The algorithmic subtyping judgment $\widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2}$ is calculated in two steps. First, we recursively traverse the structure of types to collect a set of constraints C^* with the judgment $\widetilde{T_1} \diamond \widetilde{T_2} \mid C^*$. This judgment makes constraints static by reduction. Second, we check that

these constraints, prepended with $\widetilde{\Phi}$, again reduced to static constraints, can be satisfied. The full definition of constraint collecting and algorithmic consistent subtyping is shown in Figure 7.1. The algorithmic definition of consistent subtyping coincides with Definition 5 (Sect. 4.2), considering the local interpretation of gradual formulas.

Proposition 27. $\widetilde{\Phi} \vdash \widetilde{T}_1 \lesssim \widetilde{T}_2$ *if and only if* $\widetilde{\Phi} \vdash \widetilde{T}_1 \mathrel{\widetilde{<:}} \widetilde{T}_2$.

7.3 A Boost in Expressiveness: Measures

The derivation of the gradual refinement language is largely independent from the refinement logic. We now explain how to extend our approach to support a more expressive refinement logic, by considering *measures* [34], *i.e.* inductively-defined functions that axiomatize properties of data types.

Suppose for example a data type $\mathsf{IntList}$ of lists of integers. The measure len determines the length of a list.

$$\begin{array}{l} \texttt{measure}\ len : \mathsf{IntList} \to \mathsf{Int} \\ len(\texttt{[]}) \quad\ = 0 \\ len(x :: xs) \ = 1 + len(xs) \end{array}$$

Measures can be encoded in the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA): a fresh uninterpreted function symbol is defined for every measure, and each measure equation is translated into a refined type for the corresponding data constructor [34]. For example, the definition of len yields refined types for the constructors of $\mathsf{IntList}$, namely $\{\, \nu : \mathsf{IntList} \mid len(\nu) = 0 \,\}$ for empty list, and $x : \mathsf{Int} \to l : \mathsf{IntList} \to \{\, \nu : \mathsf{IntList} \mid len(\nu) = 1 + len(l) \,\}$ for cons.

Appropriately extending the syntax and interpretation of gradual formulas with measures requires some care. Suppose a function *get* to obtain the n-th element of a list, with type:

$$l : \mathsf{IntList} \to n : \{\, \nu : \mathsf{Int} \mid 0 \leq \nu < len(l) \,\} \to \mathsf{Int}$$

Consider now a function that checks whether the n-th element of a list is less than a given number:

$$\begin{array}{l} \texttt{let}\ \ f\ (l\colon \{\, \nu : \mathsf{IntList} \mid ? \,\})\ (n\colon \{\, \nu : \mathsf{Int} \mid ? \,\})\ (m\colon \{\, \nu : \mathsf{Int} \mid ? \,\}) = \\ \quad (get\ l\ n) < m \end{array}$$

We expect this code to be accepted statically because n could stand for some valid index. We could naively consider that the unknown refinement of n stands for $0 \leq \nu < len(l)$. This interpretation is however *non-local*, because it restricts $len(l)$ to be strictly greater than zero; a non-local interpretation would then also allow the refinement for m to stand for some formula that contradicts this restriction on l. We must therefore adhere to locality to avoid contradictions (Sect. 5.4). Note that we *can* accept the definition of f based on a local interpretation of gradual formulas: the unknown refinement of l could stand for $len(l) > 0$,

$$\boxed{\widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2}} \qquad \frac{\widetilde{T_1} \diamond \widetilde{T_2} \mid C^* \qquad \vdash \widetilde{\Phi} \cdot C^*}{\widetilde{\Phi} \vdash \widetilde{T_1} \lesssim \widetilde{T_2}}$$

$$\boxed{\widetilde{T_1} \diamond \widetilde{T_2} \mid C^*} \qquad (\diamond\text{refine}) \frac{}{\{\, \nu : B \mid \widetilde{p} \,\} \diamond \{\, \nu : B \mid \widetilde{q} \,\} \mid (x : \widetilde{p}) \circ \{ \cdot \approx (\!|\widetilde{q}[\![x/\nu]\!]|\!)_! \}}$$

$$(\diamond\text{fun1}) \frac{\widetilde{T_1} \diamond \widetilde{T_2} \mid C_1^* \qquad C_2^* = (x : \widetilde{p_2}) \circ (C_1^* \cup \{ \cdot \approx (\!|\widetilde{p_1}[\![x/\nu]\!]|\!)_! \})}{x : \{\, \nu : B \mid \widetilde{p_1} \,\} \to \widetilde{T_1} \diamond x : \{\, \nu : B \mid \widetilde{p_2} \,\} \to \widetilde{T_2} \mid C_2^*}$$

$$(\diamond\text{fun2}) \frac{\begin{array}{c} y : \widetilde{T_{21}} \to \widetilde{T_{22}} \diamond y : \widetilde{T_{11}} \to \widetilde{T_{12}} \mid C_1^* \\ \widetilde{T_{13}} \diamond \widetilde{T_{23}} \mid C_2^* \qquad C_3^* = C_1^* \cup C_2^* \end{array}}{x : (y : \widetilde{T_{11}} \to \widetilde{T_{12}}) \to \widetilde{T_{13}} \diamond x : (y : \widetilde{T_{21}} \to \widetilde{T_{22}}) \to \widetilde{T_{23}} \mid C_3^*}$$

$$\boxed{\vdash C^*} \qquad \frac{(\!|\Phi_i|\!) \models r_i \quad \text{forall } i}{\vdash \{\, \overline{\Phi_i \approx r_i} \,\}}$$

$$\frac{\vec{z} = \bigcup_i dom(\Phi_2^i) \qquad q = \text{canonical}(p, \vec{z}, \{\, \overline{\Phi_2^i} \,\}, \{\, \overline{r_i} \,\}) \qquad \vdash \{\, \overline{(\widetilde{\Phi}_1, y : q, \Phi_2^i) \approx r_i} \,\}}{\vdash \{\, \overline{(\widetilde{\Phi}_1, y : p \wedge ?, \Phi_2^i) \approx r_i} \,\}}$$

$$\widetilde{\Phi} \cdot \{\Phi_1 \approx r_1, \ldots, \Phi_n \approx r_n\} = \{(\widetilde{\Phi} \cdot \Phi_1 \approx r_1), \ldots, (\widetilde{\Phi} \cdot \Phi_n \approx r_n)\}$$

$$(x : p) \circ \{\Phi_1 \approx r_1, \ldots, \Phi_n \approx r_n\} = \{(x : p \cdot \Phi_1 \approx r_1), \ldots, (x : p \cdot \Phi_n \approx r_n)\}$$

$$(x : p \wedge ?) \circ \{\Phi_1 \approx r_1, \ldots, \Phi_n \approx r_n\} = \{(x : q \cdot \Phi_1 \approx r_1), \ldots, (x : q \cdot \Phi_n \approx r_n)\}$$

where $q = \text{canonical}(p, \bigcup_i dom(\Phi_i), \{\, \overline{\Phi_i} \,\}, \{\, \overline{r_i} \,\})$

$$\text{canonical}(p, \vec{z}, \{\, \overline{\Phi_i} \,\}, \{\, \overline{r_i} \,\}) = ((\exists \nu, q) \to q) \wedge (\neg(\exists \nu, q) \to p)$$

where $q = (\forall \vec{z}, \bigwedge_i ((\!|\Phi_i|\!) \to r_i)) \wedge p$

Figure 7.1: Constraints collecting and algorithmic consistent subtyping.

and the refinement of n could stand for a local constraint on n based on the fact that $len(l) > 0$ holds, *i.e.* $len(l) > 0 \to 0 \leq \nu < len(l)$.

To easily capture the notion of locality we leverage the fact that measures can be encoded in a restricted fragment of QF-EUFLIA that contains only unary function symbols, and does not allow for nested uninterpreted function applications. We accordingly extend the syntax of formulas in the static language, with $f \in \textsc{Measure}$:

$$p \;::=\; \ldots | f\ v | f\ \nu$$

For this logic, locality can be defined syntactically, mirroring Definition 13. It suffices to notice that, in addition to restricting the refinement variable ν, formulas are also allowed to restrict a measure applied to ν. To check locality of a formula, we consider each syntactic occurrence of an application $f(\nu)$ as an atomic constant.

Definition 25 (Local formula for measures)**.** *Let p be a formula in the restricted fragment of QF-EUFLIA. Let p' be the formula resulting by substituting every occurrence of $f(\nu)$ for some function f by a fresh symbol $c_{f(\nu)}$. Then, let X be the set of all symbols $c_{f(\nu)}$. We say that p is local if $\exists X.\exists \nu.p'$ is valid.*

The critical property for local formulas is that they always preserves satisfiability (recall Proposition 10).

Proposition 28. *Let Φ be a logical environment with formulas in the restricted fragment of QF-EUFLIA, $\vec{x} = dom(\Phi)$ the vector of variables bound in Φ, and $q(\vec{x}, \nu)$ a local formula. If $(\!|\Phi|\!)$ is satisfiable then $(\!|\Phi|\!) \cup \{\, q(\vec{x}, \nu) \,\}$ is satisfiable.*

The definition of the syntax and interpretation of gradual formulas follows exactly the definition from Section 5.4, using the new definition of locality. Then, the concretization function for formulas is naturally lifted to refinement types, gradual logical environment and subtyping triples, and the gradual language is derived as described in previous sections. Recall that the derived semantics relies on $\langle \alpha_{\scriptscriptstyle T}, \gamma_{\scriptscriptstyle T} \rangle$ and $\langle \alpha_\tau, \gamma_\tau \rangle$ being partial Galois connections. The abstraction function for formulas with measures is again partial, thus $\alpha_{\scriptscriptstyle T}$ and α_τ are also partial. Therefore, we must establish that $\langle \alpha_{\scriptscriptstyle T}, \gamma_{\scriptscriptstyle T} \rangle$ and $\langle \alpha_\tau, \gamma_\tau \rangle$ are still partial Galois connections for the operators used in the static and dynamic semantics.

Proposition 29 (Partial Galois connections for measures)**.** *The pair $\langle \alpha_{\scriptscriptstyle T}, \gamma_{\scriptscriptstyle T} \rangle$ is a $\{\, \widehat{tsubst} \,\}$-partial Galois connection. The pair $\langle \alpha_\tau, \gamma_\tau \rangle$ is a $\{\, F_{\mathcal{I}_{<:}}, F_{\circ_{<:}}, F_{\circ_{<:}^{[v/x]}} \,\}$-partial Galois connection.*

To sum up, adapting our approach to accommodate a given refinement logic requires extending the notion of locality (preserving satisfiability), and establishing the partial Galois connections for the relevant operators. This is enough to derive a gradual language that satisfies the properties of Sections 4.3 and 6.5.4.

Additionally, care must be taken to maintain decidable checking. For example, our algorithmic approach to consistent subtyping (Section 7.2) relies on the theory of linear arithmetic accepting quantifier elimination, which is of course not true in all theories. The syntactic restriction for measures allows us to exploit the same approach for algorithmic consistent subtyping, since we can always see a formula in the restricted fragment of QF-EUFLIA as

an "equivalent" formula in QF-LIA. But extensions to other refinement logics may require devising other techniques, or may turn out to be undecidable; this opens interesting venues for future work.

7.4 Revisiting the Dynamic Semantics

If we want to target an implementation of gradual refinement types special attention must be given to the dynamic semantics. We first note that the semantics derived using AGT is intended to serve as a reference semantics, but not as efficient implementation technique. Therefore, designing an appropriate cast calculus and a translation from gradual source programs to internal programs with casts is an open issue. The resulting translational semantics should be equivalent to the reference semantics. We believe, however, that there are some issues in the reference semantics that will hinder the definition of a satisfactory cast calculus.

One of the major issues is related to the eagerness of runtime failures. As noted in Chapter 6, to preserve soundness in general a novel consistent term substitution operator is needed; this operator eagerly detects inconsistencies at the time of beta reduction. Recall, for instance, the example discussed in the introduction where we have the following library functions:

$$\begin{aligned} \mathit{check} &:: \mathsf{Int} \to \mathsf{Bool} \\ \mathit{get} &:: \{\, \nu{:}\mathsf{Int} \mid \nu > 0 \,\} \to \mathsf{Int} \end{aligned}$$

We mentioned that by carefully importing the function *check* with an unknown refinement the imprecision can be transferred to the branches of a conditional to optimistically accept the following code:

$$\textbf{if}\ \mathit{check}(x)\ \textbf{then}\ \mathit{get}(x)\ \textbf{else}\ \mathit{get}(-x)$$

Instead of worrying about the refinement for *check* we could choose to refine x with the unknown formula and write the following code:

$$\textbf{let}\ \mathit{ris}\ (x{:}\ \{\, \nu{:}\mathsf{Int} \mid ?\,\}) = \textbf{if}\ \mathit{check}(x)\ \textbf{then}\ \mathit{get}(x)\ \textbf{else}\ \mathit{get}(-x)$$

This code is accepted statically because at each branch the unknown refinement is interpreted accordingly to support the possible correct use of the function *get*. At runtime, however, we observe an unexpected behavior. If we apply the function *ris* to any argument the program will eagerly fail before reducing the body, because provided a particular value for x the system will gain full precision and it will not be possible to prove both branches safe after beta reduction.

Besides not allowing idioms like the one in the previous example, the eagerness of the system is also related to performance issues. Detecting inconsistencies at the time of beta reduction requires verifying consistency relations on open terms, hence resorting to some SMT-based reasoning at runtime. However, we believe that there are many cases where justifications can be delayed until closed terms are reached; in this case, a runtime check only implies a direct evaluation of a refinement, because all values are known by then. This raises the need to clearly identify how early inconsistencies must be detected to preserve soundness.

Chapter 8

Conclusions

In this work we have proven that a gradual refinement type system is both possible and sound. Gradual refinement types support a smooth evolution between simple types and logically-refined types. Supporting this continuous slider led us to analyze how to deal with imprecise logical information. We developed a novel *semantic* and *local* interpretation of gradual formulas that is key to practical gradual refinements. This specific interpretation of gradual formulas is the main challenge in extending the refinement logic, as illustrated with measures. We also demonstrate the impact of dependent function types in a gradual language, requiring new notions of term and type substitutions with associated runtime checks. We also believe this work should inform the gradualization of even more complex typing disciplines like full-fledged dependent types.

To address the pragmatics of the system we have defined algorithmic characterizations for consistent predicates and operators in the static system. Most notably, we defined a decision procedure for the consistent subtyping judgment based in solving consistent entailment constraints. We anticipate the need for an specific experimental analysis of the performance of this algorithm, since the theoretical complexity bound is high, but as noted by many other applications it can turn out to be efficient in practice. To address the expressiveness of the system we also explained how to extend the refinement logic with measures, but reconciling locality and decidability in a more expressive setting might be challenging.

While the system satisfies many interesting properties, we have identified some remaining issues in the dynamic semantics that have to be considered, specially if we want to target a practical implementation. These issues are mainly related to the eagerness of runtime failures. As we noted the drawbacks of this eagerness are twofold. First, it precludes some flexible idioms, but most importantly it forces the use of some SMT-based reasoning at runtime, which could impose prohibitive runtime overhead.

Finally, to study the usability of gradual refinement types, we believe that an interesting perspective is to extend LiquidHaskell [34] with gradual refinement types, thereby allowing us to build upon the existing experience with static refinement types in practice. This additionally raises the question of how to combine gradual refinement types with refinement type inference. We believe this integration can be essentially based on the approach of Garcia

and Cimini [17] for combining type inference and gradual typing, but the details need to be worked out.

Bibliography

[1] Coq development team: The Coq proof assistant. Software and documentation available on the web. `http://coq.inria.fr`.

[2] F. Bañados Schwerter, R. Garcia, and É. Tanter. A theory of gradual effect systems. In *19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 283–295, Gothenburg, Sweden, Sept. 2014. ACM Press.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[4] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[5] N. Bjørner and L. De Moura. Z310: Applications, enablers, challenges and directions. *Constraints in Formal Verification, CFV*, 9, 2009.

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 317–333. Springer, 2005.

[7] R. Chugh. *Nested Refinement Types for JavaScript*. PhD thesis, University of California, Sept. 2013.

[8] R. Chugh, P. M. Rondon, A. Bakst, and R. Jhala. Nested refinements: a logic for duck typing. In *39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 231–244, Philadelphia, USA, Jan. 2012. ACM Press.

[9] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 1936.

[10] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

[11] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238–252, Los Angeles, CA, USA, Jan. 1977. ACM Press.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36. Springer, 2009.

[14] T. Disney and C. Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[15] L. Fennell and P. Thiemann. Gradual security typing with references. In *Computer Security Foundations Symposium*, pages 224–239, June 2013.

[16] C. Flanagan. Hybrid type checking. In *33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 245–256, Charleston, SC, USA, Jan. 2006. ACM Press.

[17] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*, pages 303–315. ACM Press, Jan. 2015.

[18] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, Jan. 2016. ACM Press.

[19] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 353–364. ACM Press, Jan. 2010.

[20] S. Hölldobler, N. Manthey, V. Nguyen, J. Stecklina, and P. Steinke. A short overview on modern parallel sat-solvers. In *Proceedings of the International Conference on Advanced Computer Science and Information Systems*, pages 201–206, 2011.

[21] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980. Reprint of 1969 article.

[22] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 343–356. Springer, 2011.

[23] K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010.

[24] N. Lehmann and É. Tanter. Formalizing simple refinement types in Coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL'16)*, St. Petersburg, FL, USA, Jan. 2016.

[25] A. Leitsch. *The Resolution Calculus.* Springer-Verlag New York, Inc., New York, NY, USA, 1997. ISBN 3-540-61882-1.

[26] A. Miné. *Weakly Relational Numerical Abstract Domains.* PhD thesis, L'École Polythechnique, Dec. 2004.

[27] G. Nelson. *Techniques for program verification.* Xerox. Palo Alto Research Center, 1981.

[28] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[29] M. Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt.* publisher not identified, 1931.

[30] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169, Tucson, AZ, USA, June 2008. ACM Press.

[31] J. Siek and W. Taha. Gradual typing for objects. In E. Ernst, editor, *21st European Conference on Object-oriented Programming (ECOOP 2007)*, number 4609 in LNCS, pages 2–27, Berlin, Germany, July 2007. Springer-Verlag.

[32] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, Sept. 2006.

[33] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, pages 274–293, 2015.

[34] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282, Gothenburg, Sweden, Sept. 2014. ACM Press.

[35] N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 48–61, Vancouver, Canada, Sept. 2015. ACM Press.