



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

TYPE ABSTRACTION AND FACETED TYPES FOR DECLASSIFICATION

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

RAIMIL CRUZ CONCEPCIÓN

PROFESOR GUÍA:  
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:  
FEDERICO OLMEDO BERÓN  
ISMAEL FIGUEROA PALET  
AMAL AHMED

Este trabajo ha sido financiado por CONICYT-PCHA/Doctorado Nacional/2014-63140148

SANTIAGO DE CHILE  
2020

# Resumen

El tipado de seguridad para control de flujos de información previene que información confidencial sea liberada hacia canales públicos. La propiedad fundamental de control de flujos de información, conocida como *no-interferencia*, define que un observador público no puede aprender nada sobre datos privados. No-interferencia es teóricamente muy atractiva, sin embargo, en la práctica: sistemas reales necesitan intencionalmente desclasificar alguna información, de forma selectiva. Entre los distintos enfoques de desclasificación de flujos de información, uno particularmente expresivo es *etiquetas-como-funciones* que asegura una noción de *no-interferencia relajada*. Dicho enfoque permite a los programadores especificar políticas de desclasificación, funciones, que capturan la forma intencional en que información pública puede ser calculada a partir de información privada.

Motivado por el enfoque *etiquetas-como-funciones*, en esta tesis, nosotros proponemos un enfoque *etiquetas-como-tipos* para capturar políticas de desclasificación de una manera simple y expresiva. En lugar de usar funciones como etiquetas, nosotros usamos tipos como etiquetas. Es decir, nosotros usamos tipos para indicar cuando un valor es público, privado o cuando es parcialmente desclasificado. En consecuencia los tipos de seguridad son *tipos con facetas*, dígase un par de tipos que definen las vistas de una valor para un observador privilegiado y un observador público respectivamente. Nosotros desarrollamos el enfoque de etiquetas-como-tipos usando mecanismos estándares de *abstracción de tipos*.

Primero exploramos la forma de abstracción de tipos provista por subtipos en el contexto de programación orientada a objetos, donde los tipos de los objetos son interfaces, dígase un conjunto de métodos disponibles para el cliente de un objeto. Segundo, exploramos desclasificación basada en tipos para lenguajes funcionales, usando el mecanismo de abstracción de tipos provisto por los tipos existenciales. Finalmente, extendemos el enfoque de desclasificación para el contexto de orientación a objetos con *polimorfismo de etiquetas*, formalizando una noción de *polimorfismo para desclasificación*.

Las distintas realizaciones del enfoque etiquetas-como-tipos disfrutan una fuerte propiedad llamada *no-interferencia relajada basada en tipos*, la cual provee un principio de razonamiento modular para desclasificación basada en tipos. El enfoque de etiquetas-como-tipos tiene el beneficio práctico de definirse sobre conceptos que son bien conocidos para los desarrolladores (dígame tipos, subtipo, etc.) para construir sistemas con seguridad de flujos de información que limpiamente tengan en cuenta desclasificación controlada y expresiva. Por tanto, esta tesis provee un fundamento necesario y sólido para integrar desclasificación basada en tipos en lenguajes de programación existentes.



# Abstract

Information-flow security typing statically prevents confidential information from leaking to public channels. The fundamental information flow property, known as *noninterference*, states that a public observer cannot learn anything from secret data. As attractive as it is from a theoretical viewpoint, noninterference is impractical: real systems need to intentionally declassify some information, selectively. Among the different information flow approaches to declassification, a particularly expressive *labels-as-functions* approach was proposed, enforcing a notion of *relaxed noninterference* by allowing programmers to specify *declassification policies*, functions, that capture the intended manner in which public information can be computed from secret data.

Motivated by the *labels-as-functions* approach, in this thesis, we propose a *labels-as-types* approach to capture declassification policies in a simple and expressive manner. Instead of using functions as labels, we use types as labels. That is, we use types to indicate when a value is public, secret or when it is partially declassified. Therefore security types are *faceted types*, *i.e.* a pair of types that defines the views of a value to the privileged and public observers respectively. We develop the labels-as-types approach using standard *type abstraction* mechanisms such as subtyping, parametric types and existential types.

We first explore the form of type abstraction provided by subtyping in the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object. Second, we explore type-based declassification for functional languages, using the type abstraction mechanism provided by existential types. Finally, we extend the object-oriented type-based declassification approach to handle *label polymorphism*. As labels are types, label polymorphism reduces to type polymorphism; and as types are used to express declassification, we formalize a notion of *declassification polymorphism*.

The different realizations of the labels-as-types approach enjoy a strong property called *type-based relaxed noninterference*, which provides a modular reasoning principle for type-based declassification. The labels-as-types approach has the practical benefits of relying on concepts that are well-known to developers (*i.e.* types, subtyping, etc.) in order to build systems with information flow security that cleanly account for controlled and expressive declassification. Therefore, this thesis provides a necessary and solid basis to integrate type-based declassification in existing programming languages.

# Contents

<b>Introduction</b>	<b>1</b>
<b>1 Background</b>	<b>9</b>
1.1 A type system for noninterference . . . . .	9
1.2 Jif: Decentralized label model, declassification and label polymorphism . . . . .	14
1.2.1 Decentralized label model . . . . .	14
1.2.2 Declassification . . . . .	16
1.2.3 Label polymorphism . . . . .	16
<b>2 Type-based declassification for objects</b>	<b>18</b>
2.1 Introduction . . . . .	18
2.2 Overview . . . . .	20
2.2.1 Type abstraction and confidentiality . . . . .	20
2.2.2 Facets of computation . . . . .	21
2.2.3 Faceted types . . . . .	22
2.3 Type-based relaxed noninterference, informally . . . . .	23
2.4 Formal semantics . . . . .	24
2.4.1 Subtyping . . . . .	24
2.4.2 Type system . . . . .	26
2.4.3 Dynamic semantics . . . . .	26
2.4.4 Type safety . . . . .	26
2.4.5 A note on well-formedness . . . . .	29
2.5 Type-based relaxed noninterference . . . . .	29
2.5.1 Type-based equivalence . . . . .	29
2.5.2 Defining type-based relaxed noninterference . . . . .	30
2.5.3 Security type soundness . . . . .	31
2.5.4 A note about casts. . . . .	34
2.6 Related work . . . . .	34
2.7 Conclusion . . . . .	37
<b>3 Existential relaxed noninterference</b>	<b>38</b>
3.1 Introduction . . . . .	38
3.2 Overview . . . . .	39
3.2.1 Existential types . . . . .	39
3.2.2 Type-based declassification policies with existential types . . . . .	40
3.2.3 Computing with secrets . . . . .	41

3.2.4	Public data as (declassifiable) secret . . . . .	42
3.3	Relaxed noninterference with existential types . . . . .	43
3.4	Formal semantics . . . . .	44
3.4.1	Syntax . . . . .	45
3.4.2	Static semantics . . . . .	46
3.4.3	Dynamic semantics and type safety . . . . .	47
3.5	Existential relaxed noninterference, formally . . . . .	48
3.5.1	Logical relation for type-based equivalence . . . . .	48
3.5.2	Existential relaxed noninterference . . . . .	50
3.5.3	Security type soundness . . . . .	51
3.6	Illustration . . . . .	51
3.7	Related work . . . . .	52
3.8	Conclusion . . . . .	53
<b>4</b>	<b>Polymorphic relaxed noninterference</b>	<b>54</b>
4.1	Introduction . . . . .	54
4.2	Overview . . . . .	55
4.2.1	Polymorphic declassification . . . . .	55
4.2.2	Bounded polymorphic declassification . . . . .	56
4.2.3	Polymorphic relaxed noninterference, informally . . . . .	58
4.3	Formal semantics . . . . .	59
4.3.1	Syntax . . . . .	59
4.3.2	Subtyping . . . . .	59
4.3.3	Type system . . . . .	61
4.3.4	Dynamic semantics . . . . .	63
4.3.5	Safety . . . . .	63
4.4	Polymorphic relaxed noninterference, formally . . . . .	64
4.4.1	Logical relation . . . . .	64
4.4.2	Defining polymorphic relaxed noninterference . . . . .	65
4.4.3	Security type soundness . . . . .	66
4.5	Ad-hoc polymorphism for primitive types . . . . .	67
4.5.1	The need and challenge of primitive types . . . . .	67
4.5.2	Sound signatures for primitive types . . . . .	68
4.5.3	Polymorphic primitive signatures . . . . .	68
4.5.4	Formal semantics . . . . .	69
4.5.5	Logical relation for primitive types . . . . .	70
4.5.6	Illustration . . . . .	72
4.6	Declassification polymorphism and existentials . . . . .	73
4.7	Related work . . . . .	76
4.8	Conclusion . . . . .	77
<b>5</b>	<b>Conclusion and perspectives</b>	<b>78</b>
	<b>Bibliography</b>	<b>80</b>
	<b>Appendices</b>	<b>86</b>
	<b>Appendix A Type-based relaxed noninterference</b>	<b>87</b>

A.1	Full Syntax and semantics . . . . .	87
A.1.1	Syntax . . . . .	87
A.1.2	Well-formedness of environments and types . . . . .	88
A.1.3	Subtyping . . . . .	90
A.1.4	Type equivalence . . . . .	91
A.1.5	Simple type system . . . . .	91
A.2	Type safety: proofs . . . . .	92
A.2.1	Auxiliary Lemmas . . . . .	93
A.2.2	Proof: Safety . . . . .	95
A.3	Type-based relaxed noninterference: proofs . . . . .	100
A.3.1	Logical relation . . . . .	100
A.3.2	Auxiliary lemmas: Simple type system . . . . .	101
A.3.3	Auxiliary lemmas . . . . .	103
A.3.4	Proof of the Fundamental Property . . . . .	107
<b>Appendix B Existential relaxed noninterference</b>		<b>113</b>
B.1	Full syntax and semantics . . . . .	113
B.1.1	Syntax . . . . .	113
B.1.2	Subtyping . . . . .	114
B.1.3	Auxiliary definitions . . . . .	115
B.1.4	Well-formedness of types: closed types . . . . .	116
B.1.5	Facet-wise well-formedness of security types . . . . .	117
B.1.6	Well-formedness of environments . . . . .	117
B.1.7	Type system . . . . .	119
B.1.8	Dynamic semantics . . . . .	120
B.1.9	Simple type system . . . . .	121
B.2	Type safety . . . . .	122
B.3	Existential relaxed noninterference: proofs . . . . .	122
B.3.1	Logical relation . . . . .	122
B.3.2	Auxiliary lemmas: Simple typing . . . . .	124
B.3.3	Strong normalization . . . . .	125
B.3.4	Auxiliary lemmas: logical relation . . . . .	126
B.3.5	Auxiliary definitions . . . . .	126
B.3.6	Proof of the Fundamental Property . . . . .	132
<b>Appendix C Polymorphic relaxed noninterference</b>		<b>145</b>
C.1	Full syntax and semantics . . . . .	145
C.1.1	Syntax . . . . .	145
C.1.2	Type equivalence . . . . .	146
C.1.3	Subtyping . . . . .	146
C.1.4	Standard well formedness of types and environments . . . . .	148
C.1.5	Well-formedness of security types facet-wise . . . . .	149
C.1.6	Auxiliary definitions . . . . .	151
C.1.7	Typing . . . . .	152
C.1.8	Dynamic semantics . . . . .	153
C.1.9	Simple type System . . . . .	154
C.2	Type safety: proofs . . . . .	154

C.3	Unary model . . . . .	154
C.4	Type safety . . . . .	156
C.5	Polymorphic relaxed noninterference: proofs . . . . .	156
C.5.1	Logical relation . . . . .	156
C.5.2	Auxiliary lemmas: simple typing . . . . .	158
C.5.3	Auxiliary lemmas: logical relation . . . . .	159
C.5.4	Proof of the Fundamental Property . . . . .	167





# Introduction

The number of applications managing confidential information grow daily, from mobile and *Internet of things* (IoT) applications to complex web-based applications. It is necessary to keep secret data secret. To achieve this goal there are different layers of security that an application and its execution environment need to provide, which range from the system architecture to the application itself. In particular, application security is challenging because real applications manage complex models with complex security policies. One of these challenges is information flow control. For instance, an application that deals with confidential credit cards to make purchases must ensure the confidential credit card data is not leaked to undesired parties during a purchase.

Information flow control can be satisfactorily implemented at the programming language level [50]. *Language-based security* has explored complementary approaches for controlling the flow of sensitive data in programs: statically [67, 28] (*e.g.* with a type system), dynamically [7] (*e.g.* with a runtime monitor) as well as hybrid approaches [49, 58]. The implemented security mechanism protects confidentiality by means of classifying entities with their confidentiality levels, expressed via a lattice of security labels [24]. For instance, a two-level lattice  $L \preceq H$  allows distinguishing public or low-confidentiality data (*e.g.*  $L$ ) from secret or high-confidentiality data (*e.g.*  $H$ ).

A *sound* information-flow control mechanism ensures *noninterference* (NI): that high-confidentiality data may not flow directly or indirectly to lower-confidentiality channels ([63]). To do so, the implemented mechanism (*e.g.* the type system) tracks the confidentiality level of computation based on the confidentiality of the data involved. For instance, given a function `postInTwitter:StringL → UnitL` that posts a public string (`StringL`) to a Twitter timeline, and a variable `creditCard:StringH` that stores a secret credit card, the invocation `postInTwitter("Hello:" + creditCard)` is rejected because it applies the function `postInTwitter` with a secret value. In particular, a sound information-flow type system can provide a *modular reasoning principle* about noninterference, which is a desired property. For instance, a function  $f : \text{String}_H \rightarrow \text{String}_L$  does not reveal any information about its argument; and in fact in a pure language, it is necessarily a constant function.

As attractive as it is, noninterference is too strict to be useful in practice, as it prevents confidential data to have *any* influence whatsoever on observable, public output. Indeed, even a simple password checker function violates noninterference. Consider the following:

```
StringL login(StringL guess, StringH password)
  if(password == guess)
    return "Login Successful"
```

```

else
    return "Login failed"
}

```

By definition, a *public observer* that tries to log in *should* be able to learn something about the confidential input (here, `password`), thereby violating the confidentiality restriction imposed by noninterference.

This problem with noninterference has long been recognized. Supporting such intentional downward information flows is called *declassification*, which can be supported in many different ways [52]. For example, Jif [35]([www.cs.cornell.edu/jif/](http://www.cs.cornell.edu/jif/)) supports an explicit `declassify` operator to allow downward flows to be accepted by the security type system. In the above example, one can use `declassify(password == guess)` to state that the returned value is public knowledge. However, arbitrary uses of a `declassify` operator may lead to serious information flow leaks; for instance `declassify(password)` simply makes the password publicly available. One solution adopted by Jif is to control declassification using principals with privileges, as in the Decentralized Label Model (DLM) [37]. Trusted declassification [29] restricts Jif’s mechanism to specify authorization in a global policy file and formulates *noninterference modulo trusted methods*. Robust declassification [69] relies on integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified.

Sabelfeld and Sands [52] categorize declassification approaches in four dimensions: *what* information is declassified ([31, 51, 27]), *when* the information is declassified ([16]), *who* performs the declassification ([69]) and *where* in the code the declassification happens [32]. Many applications need different declassification approaches of different dimensions in order to capture their declassification policies. For instance, in a login functionality (like the example above), we need to selectively declassify the comparison of the secret password with a public value. In a paper review system, the reviews of a paper are secret *until* the author notification date. Furthermore, in some situations, different approaches of different dimensions can interact at once. For instance, at that moment of releasing the reviews of a paper, not all the review data is necessarily declassified; in many cases the reviewers still remain secret to the author.

To capture the essence of *selective* declassification without appealing to additional mechanisms like integrity or authority, Li and Zdancewic [31] proposed an expressive *labels-as-functions* approach for declassification policies that supports the extensional specification of secrets and their intended declassification. In this approach, instead of having security labels such as H for secret and L for public information that are drawn from a fixed lattice of symbols, security labels are the very functions that describe how a given secret can be manipulated in order to produce a public value. A declassification policy is a function that captures *what* information on a confidential value can be *declassified* to eventually produce a public value. For the password checker example, if the declassification policy for `password` is  $\lambda x.\lambda y.x=y$ , then an equality comparison with `password` can be declassified (and thus be made public). However, this declassification policy for `password` disallows arbitrary declassifications such as revealing the password. With this approach one can express the standard labels H and L as a constant and identity function respectively. Li and Zdancewic also provide a formal security property, called *relaxed noninterference*, that states that a secure program can

be rewritten into an equivalent program without any variable containing confidential data but whose inputs are confidential and declassified.

We found the *labels-as-functions* approach of Li and Zdancewic elegant as it formally captures the essence of flexible declassification while retaining a way to state a clear and extensional security property of interest. However, it suffers from a number of limitations that jeopardize its practical adoption. For instance, security labels are sets of functions that form a security lattice whose ordering, based on a semantic interpretation of these sets of functions, is far from trivial [31]: it relies on a general notion of program equivalences that would be both hard to implement and to comprehend.

Motivated by the elegant approach for declassification of Li and Zdancewic and from long-standing goals for information flow security of integrating information flow mechanisms with existing infrastructures [68] we explore the following research questions: Is it possible to reuse typing features of mainstream languages to capture expressive declassification policies? Can we do it in a simple manner, easy to understand and enforce? What modular reasoning principle accounting for declassification can we provide?

**Contributions and document outline.** In this thesis, we propose a *labels-as-types approach* to capture declassification policies in a simple and expressive manner. Instead of modeling labels with functions, we use the same types of the language as labels. That is, we use types to indicate when a value is public, secret or it is selectively declassified. We develop this *type-based declassification* approach using standard *type abstraction* mechanisms such as subtyping, parametric types and existential types ([41]).

We first explore the form of type abstraction provided by subtyping in the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object. Second, we explore type-based declassification for functional languages, using the type abstraction mechanism provided by existential types. Finally, we extended the object-oriented type-based declassification approach to handle *label polymorphism* [36]. As labels are types, label polymorphism reduces to type polymorphism; and as types are used to express declassification, we formalize a notion of *declassification polymorphism*.

Our approach uses standard types as labels and therefore security types are *faceted types*, *i.e.* pairs of types  $T$  and  $U$ . We develop two different forms of faceted types, depending on the labels-as-types realization. However, in both cases, the first facet  $T$ —called the *safety type*—represents the implementation type and defines the view to the privileged observer (which are authorized to see secret values), and the second facet  $U$ —called the *declassification type*—used for confidentiality, defines the view of the public observer.

The two realizations of the labels-as-types approach enjoy of a notion of *type-based relaxed noninterference* that provides a modular reasoning principle for type-based declassification.

Now, we provide a summary of each chapter of this thesis, highlighting the use of a type abstraction mechanism, the role of faceted types and the respective notion of type-based relaxed noninterference.

**Background** Chapter 1 provides background information on language-based security, more specifically, security typing. First, we use a simple and established information flow type system to illustrate information flow control features and the definition of noninterference through logical relations. Furthermore, we explain the Jif’s mechanisms for declassification and label polymorphism.

**Type-based declassification with object types** Chapter 2 develops *labels-as-types* with object types where selective declassification is expressed with subtyping. In this setting, security types are *faceted types* of the form  $T \triangleleft U$  and the declassification type  $U$  represents the declassification policy as an object interface exposed to the public observer. For instance, the type  $\mathbf{String} \triangleleft \top$ , where  $\top$  is the empty object interface, denotes secret values (no method is declassified) and the type  $\mathbf{String} \triangleleft \mathbf{String}$  represents public values (all methods are declassified). These security types are abbreviated as  $\mathbf{String}_H$  and  $\mathbf{String}_L$ , respectively. Interesting declassification policies stand in between these two extremes: for instance, given the interface  $\mathbf{StringLen} \triangleq [\text{length} : \mathbf{Unit}_L \rightarrow \mathbf{Int}_L]$ , the faceted type  $\mathbf{String} \triangleleft \mathbf{StringLen}$  exposes the method `length` to declassify the length of a string as a public integer, but not its content.

To give an idea of how this type-based declassification approach works we revisit the `login` function. For instance, if the `password` secret is made available at the type  $\mathbf{String} \triangleleft \mathbf{StringEq}$ , where  $\mathbf{StringEq}$  is the interface type  $[\text{eq} : \mathbf{String}_L \rightarrow \mathbf{Bool}_L]$ , the `login` function can be rewritten as follows:

```
StringL login(StringL guess, String < StringEq password){
  if(password.eq(guess)) ...
}
```

Because `password` has type  $\mathbf{String} \triangleleft \mathbf{StringEq}$ , the `login` function cannot accidentally leak information about the password. In particular, note that the function cannot even return the password because the declassification type  $\mathbf{StringEq}$  is a *supertype* of  $\mathbf{String}$ , not a subtype. Therefore, the standard substitutability expressed by subtyping seems to align well with the valid information flows permitted in a confidentiality type system: a (public) string value at type  $\mathbf{String}_L$  can be used freely, and passed as argument expecting a (mostly) secret  $\mathbf{String} \triangleleft \mathbf{StringEq}$ , which only exposes equality comparison. That is, label ordering is simply subtyping.

Furthermore, this approach extends the modular reasoning principle of noninterference to account for declassification, a property named *type-based relaxed noninterference* (TRNI). For instance, with TRNI one can prove that a function of type  $(\mathbf{String} \triangleleft \mathbf{StringLen}) \rightarrow \mathbf{Bool}_L$  *must* produce equal results for strings of equal lengths.

To summarize, Chapter 2 makes the following contributions:

- We develop the novel type-based approach to declassification policies in the setting of object-oriented languages, which supports interesting scenarios while appealing to standard programming concepts such as interface types and subtyping.
- We capture the essence of object-oriented type-based declassification in a core object-oriented language,  $\mathbf{Ob}_{SEC}$ , in which a security type is a pair of (recursive) object types. We describe the static and dynamic semantics of  $\mathbf{Ob}_{SEC}$  and prove type *safety*.

- We specify the formal semantic notion of *type-based relaxed noninterference*, which accounts for type-based declassification policies, independently of any enforcement mechanism. We then prove type *soundness* of  $\text{Ob}_{\text{SEC}}$ : a well-typed program satisfies type-based relaxed noninterference.

**Type-based declassification with existential types** In Chapter 3, we develop type-based declassification for functional languages, which allows us to express advanced declassification policies, such as extrinsic policies, based on a different form of type abstraction: existential types [34]. An existential type, the essence of an *abstract data type* and modules, exposes abstract types and operations on these; we leverage this abstraction mechanism to express secrets that can be declassified using the provided operations. That is, we can establish an analogy between existential types and selective declassification of secrets: an existential type  $\exists X.T$  exposes operations to obtain secret values, at the abstract type  $X$ , and the operations of  $T$  can be used to declassify these secrets.

We observe that developing type-based relaxed noninterference in an object-oriented setting, exploiting subtyping as the type abstraction mechanism, imposes some restrictions on the declassification policies that can be expressed. In particular, because security types are of the form  $T \triangleleft U$  where the declassification type  $U$  is a supertype of the safety type  $T$ —a necessary constraint to ensure type safety—means that one cannot declassify properties that are *extrinsic* to (*i.e.* computed externally from) the secret value. For instance, because a typical `String` interface does not feature an `encrypt` method, it is not possible to express the declassification policy that “the encrypted representation of the password is public”.

Here, we explain how to express the `login` example, encoding the password declassification policy with an existential type. The type `AccountStore` below models a simplified user repository. It provides the password of a user at type  $X$  with the function `userPass` and a function `verifyPass` to check (observe) whether an arbitrary string value is equal to the password.

$$\text{AccountStore} \triangleq \exists X.[ \text{userPass} : \text{String} \rightarrow X \\ \text{verifyPass} : \text{String} \rightarrow X \rightarrow \text{Bool} ]$$

Values of an existential type  $\exists X.T$  take the form of a *package* that packs together the *representation* type for the abstract type  $X$  with an implementation  $v$  of the operations provided by  $T$ . One can think of packages as *modules* with signatures. For instance, the package  $p \triangleq \text{pack}(\text{String}, v)$  as `AccountStore` is a value of type `AccountStore`, where `String` is the representation type and  $v$  is a record implementing the functions `userPass` and `verifyPass` that use the implementation type `String` instead of abstract type  $X$ .

To use an existential type, we have to *open* the package (*i.e.* import the module) to get access to the implementation  $v$ , along with the abstract type that hides the actual representation type. The expression `open(X, x) = p in e'` opens the package  $p$  above, exposing the representation type abstractly as a type variable  $X$ , and the implementation as term variable  $x$ , within the scope of the body  $e'$ . Crucially, the expression  $e'$  has no access to the representation type `String`, therefore nothing can be done with a value of type  $X$ , beyond using it with the operations provided by `AccountStore`.

We can use the declassification policy modeled with `AccountStore` to implement a valid well-typed, secure login functionality. The `login` function below is defined in a scope where the package `p` of type `AccountStore` is opened, providing the type name `X` for the abstract type and the variable `store` for the package implementation.

```
open(X, store) = p in
...
String login(String guess, String username){
    if(store.verifyPass(guess, store.userPass(username)))
        ...
}
```

The `login` function first obtains the user secret password of type `X` with `store.userPass(username)`, and then passes the secret password (of type `X`) to the function `verifyPass` with the public `guess` to obtain the public boolean result. The above code makes a valid use of `AccountStore` and therefore is well-typed.

The type abstraction provided by `AccountStore` avoids leaking information accidentally. For instance, directly returning the secret password of type `X` is a type error, even if internally it is a string. Likewise, the expression `length(store.userPass(username))` is ill-typed.

To support computations with secret values, a standard feature of information flow type systems, we also develop a form of faceted types  $T@U$  for this labels-as-types approach. For instance, with faceted types, the `userPass` operation of the `AccountStore` type could return `String@X`. This enables a privileged observer to see the result of a computation with secret values, such as `length(store.userPass(username))`.

To summarize, Chapter 3 makes the following contributions:

- We explore an alternative type abstraction mechanism to realize the labels-as-types approach to expressive declassification, retaining the practical aspect of using an existing language mechanism (here, existential types), while supporting more expressive declassification policies.
- We define a new version of type-based relaxed noninterference, called *existential relaxed noninterference*, which accounts for extrinsic declassification using existential types.
- We capture the essence of the use of existential types for relaxed noninterference in a core functional language  $\lambda_{\text{SEC}}^{\exists}$ , and prove that its type system soundly enforces existential relaxed noninterference.

**Polymorphic type-base declassification** In Chapter 4 we extend the object-oriented labels-as-types approach (Chapter 2) in order to support *polymorphic* declassification. Security label polymorphism is a very useful feature of practical security-typed languages such as Jif [36] and FlowCaml [45], which has only been explored in the context of standard security labels (symbols from a lattice). First, we identify the need for bounded polymorphism through concrete examples. We then formalize polymorphic relaxed noninterference in a typed object-oriented calculus  $\text{Ob}_{\text{SEC}}^{\diamond}$  to prove that all well-typed terms are secure. After that, we address the case of primitive types, which requires a form of ad-hoc polymorphism. Finally, we close the chapter explaining how the labels-as-types approach with existential types enjoys a form of declassification polymorphism.

Here, we illustrate the benefits of bounded polymorphic declassification by giving the example of a list of strings that is polymorphic with respect to the declassification type of its elements:

$$\text{ListEqStr}\langle X \text{ <: StringEq} \rangle \triangleq [\text{isEmpty} : \text{Unit}_L \rightarrow \text{Bool}_L, \\ \text{head} : \text{Unit}_L \rightarrow \text{String} \triangleleft X, \\ \text{tail} : \text{Unit}_L \rightarrow \text{ListEqStr} \langle X \rangle_L]$$

The type `ListEqStr` denotes a recursive polymorphic declassification policy that allows a public observer to traverse the list and compare its elements for equality with a given public element. This restriction is visible in the signature of the `head` method, which returns a value of type `String  $\triangleleft$  X`, where `X <: StringEq` and `StringEq  $\triangleq$  [eq : StringL → BoolL]`.

With this policy we can implement a generic `contains` function with publicly observable result:

```
BoolL contains <X <: StringEq> (ListEqStr <X>L l, StringL s){
  if(l.isEmpty()) return false
  if(l.head().eq(s)) return true
  return contains(l.tail(), s)
}
```

The key here is that `l.head().eq(s)` is guaranteed to be publicly observable, because the actual declassification policy with which `X` will be instantiated necessarily includes (at least) the `eq` method.

To summarize, Chapter 4 makes the following contributions:

- We extend the object-oriented type-based declassification approach of Chapter 2 with bounded polymorphic declassification. This brings benefits in the expressiveness and design of declassification interfaces. Additionally, we address the necessary support for primitive types, through a form of ad-hoc polymorphism.
- We capture bounded polymorphic declassification in  $\text{Ob}_{\text{SEC}}^{\diamond}$ , an extension to  $\text{Ob}_{\text{SEC}}$  with bounded polymorphism.
- We develop the theory of bounded polymorphic declassification as an extension of TRNI, called *polymorphic relaxed noninterference* (PRNI for short) and show that all well-typed  $\text{Ob}_{\text{SEC}}^{\diamond}$  terms satisfy PRNI.

**Conclusions and Appendices** Finally, in Chapter 5 we develop the conclusions of this thesis and discuss future research directions. Full definitions, as well as proofs of the results mentioned in the thesis, are found in Appendix.

Summarizing, in this thesis we show that type abstraction with faceted types provides a sound and expressive theory for declassification in security-typed languages. This labels-as-types approach has the practical benefits of relying on concepts that are well-known to developers in order to build systems with information flow security that cleanly account for controlled and expressive declassification.



**Publications** The results presented in Chapter 2 were published at the European Conference on Object-Oriented Programming (ECOOP 2017) [22]. Chapter 4 is based on a publication accepted at IEEE Secure Development Conference (SecDev 2019) [21], while Chapter 3 is based on an article accepted at the Asian Symposium on Programming Languages and Systems (APLAS 2019) [20].

# Chapter 1

## Background

In this chapter we provide a brief background on language-based security, more specifically, security typing. For that goal, we first introduce a simple and standard information flow type system  $\lambda_{\text{SEC}}$  [67] to illustrate common information flow control concepts. Then, we describe Jif’s mechanisms for declassification and label polymorphism.

### 1.1 A type system for noninterference

Figure 1.1 shows a simplified version of the  $\lambda_{\text{SEC}}$  syntax and static semantics [67]. The core of the syntax of  $\lambda_{\text{SEC}}$  is that of the simply-typed lambda calculus. It includes booleans, functions, binary operations over booleans, function applications and conditional expressions. The security-specific syntactic notions are security values  $v$  and security types  $S$ : standard values and types constructors labeled with a label  $\ell$  respectively. Labels  $\ell$  are taken from a fixed lattice, where the operations  $\sqsubseteq$  and  $\sqcup$  define the ordering relation and the join operator respectively; and  $\perp$  and  $\top$  are the bottom and top elements of the lattice, respectively.

The lattice of labels induces a subtyping relation  $S_1 <: S_2$  between security types that defines valid (and invalid) information flows. For instance, for a lattice  $L \sqsubseteq H$ , we have that  $\text{Bool}_L <: \text{Bool}_H$ . It means that it is secure to pass a value of type  $\text{Bool}_L$  where a value of type  $\text{Bool}_H$  is expected, but not the other way around.  $\text{Bool}_H$  to  $\text{Bool}_L$  represents a violation of confidentiality.

The typing judgment  $\Gamma \vdash e : S$  holds if the expression  $e$  has security type  $S$  under the type environment  $\Gamma$  (a mapping from variables to security types, *i.e.*  $\Gamma ::= \bullet \mid \Gamma, x : S$ ). The rules (Tx), (Ts), (Tb) and (Tf) are mostly standard with respect to the simply typed lambda calculus with subtyping [41]. Here, we focus on the special features for security typing. Rule (Tx) gives a security type to variable, from the type environment. Rule (Ts) is the standard subsumption rule: if an expression  $e$  has type  $S'$ , subtype of  $S$ , it also has type  $S$ . Rules (Tb) and (Tf) give types to boolean and function values, by labeling the type constructor with the label of the value.

The most interesting rules are (Tbop), (TIf) and (Tapp) for binary operations, conditional expressions and function applications, respectively— elimination rules for booleans

$e ::= v \mid e e \mid x \mid e \oplus e \mid \text{if } e \text{ then } e \text{ else } e$	(terms)	
$r ::= \mathbf{b} \mid \lambda x : S.e$	(raw values)	
$\mathbf{b} ::= \text{true} \mid \text{false}$	(booleans)	
$v ::= r_\ell$	(security values)	$x, y, z$ (variables)
$T ::= \text{bool} \mid (S \rightarrow S)$	(types)	$\ell$ (labels)
$S ::= T_\ell$	(security types)	
$\oplus ::= \vee \mid \wedge$	(operations)	

$\Gamma \vdash e : S$

$$\begin{array}{c}
\text{(Tx)} \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(Ts)} \frac{\Gamma \vdash e : S' \quad S' <: S}{\Gamma \vdash e : S} \\
\\
\text{(Tb)} \frac{}{\Gamma \vdash \mathbf{b}_\ell : \text{Bool}_\ell} \quad \text{(Tbop)} \frac{\Gamma \vdash e_1 : \text{Bool}_{\ell_1} \quad \Gamma \vdash e_2 : \text{Bool}_{\ell_2}}{\Gamma \vdash e_1 \oplus e_2 : \text{Bool}_{\ell_1 \sqcup \ell_2}} \\
\\
\text{(Tf)} \frac{\Gamma, x : S_1 \vdash e : S_2}{\Gamma \vdash (\lambda x : S_1.e)_\ell : (S_1 \rightarrow S_2)_\ell} \quad \text{(Tapp)} \frac{\Gamma \vdash e_1 : (S_1 \rightarrow S_2)_\ell \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1 e_2 : S_2 \sqcup \ell} \\
\\
\text{(Tif)} \frac{\Gamma \vdash e : \text{Bool}_\ell \quad \Gamma \vdash e_1 : S \quad \Gamma \vdash e_2 : S}{\Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : S \sqcup \ell}
\end{array}$$

$S <: S$

$$\frac{\ell \sqsubseteq \ell'}{\text{Bool}_\ell <: \text{Bool}_{\ell'}}$$

$$\frac{S'_1 <: S_1 \quad S_2 <: S'_2 \quad \ell \sqsubseteq \ell'}{(S_1 \rightarrow S_2)_\ell <: (S'_1 \rightarrow S'_2)_{\ell'}}$$

$S \sqcup \ell$

$$T_\ell \sqcup \ell' = T_{(\ell \sqcup \ell')}$$

Figure 1.1:  $\lambda_{\text{SEC}}$ : Syntax and static semantics

and functions. They share a common pattern for information flow type system: the resulting label of an expression is an upper bound of the labels of its sub-expressions. This allows computations over secret values to proceed. In details, rule (Tbop) ensures that the security label  $\ell_1 \sqcup \ell_2$  of the result of the binary operation be an upper bound of the labels of both arguments by means of the join  $\sqcup$  operator of lattice. Rule (Tapp) protects the result of the function application by reflecting the function label  $\ell$  in the resulting type. For that mean, the rule uses the auxiliary *label stamping* function  $T_\ell \sqcup \ell' = T_{\ell \sqcup \ell'}$ , with the function return security type  $S_2$  and the function label  $\ell$ . Finally, Rule (Tif) ensures that the final label of the conditional expression reflects the label  $\ell$  of the condition expression and the labels of both branch expressions. It also uses the label stamping function to achieve that.

A final subtle thing to note about rule (Tbop) is that it encodes label polymorphism for binary operations. That is, we can use a binary boolean operator with two booleans at any security labels.

**Dynamic semantics** Evaluation of  $\lambda_{\text{SEC}}$  terms is formalized using a *labeled* big-step dynamic semantics. The labeled dynamic semantics propagates security labels at runtime, which is a necessary artifact to define and prove noninterference for  $\lambda_{\text{SEC}}$ . In this thesis, we define small-step dynamic semantics, and then we present a labeled small-step dynamic semantics for  $\lambda_{\text{SEC}}$  instead of the original big-step formulation (Figure 1.2).

$$\begin{array}{ll}
e ::= \dots \mid [e]_\ell & \text{(runtime terms)} \\
E ::= [] \mid E e \mid v E \mid E \oplus e \mid v \oplus E \mid \text{if } E \text{ then } e \text{ else } e \mid [E]_\ell & \text{(evaluation contexts)}
\end{array}$$
  

$$\begin{array}{l}
(\lambda x : S. e)_\ell v \mapsto [e[v/x]]_\ell \\
\mathbf{b}_{1\ell_1} \oplus \mathbf{b}_{2\ell_2} \mapsto [\theta(\oplus, \mathbf{b}_1, \mathbf{b}_2)]_{\ell_1 \sqcup \ell_2} \\
\text{if true}_\ell \text{ then } e_1 \text{ else } e_2 \mapsto [e_1]_\ell \\
\text{if false}_\ell \text{ then } e_1 \text{ else } e_2 \mapsto [e_2]_\ell \\
[r_{\ell_1}]_{\ell_2} \mapsto r_{(\ell_1 \sqcup \ell_2)}
\end{array}
\quad \frac{e \mapsto e'}{E[e] \mapsto E[e']} \quad (\text{Tp}) \frac{\Gamma \vdash e : S}{\Gamma \vdash [e]_\ell : S \sqcup \ell}$$

Figure 1.2:  $\lambda_{\text{SEC}}$ : Labeled small-step dynamic semantics

The dynamic semantics of Figure 1.2 are basically the dynamic semantics of the simply-typed lambda calculus with booleans. The function  $\theta$  provides computational meaning to binary operators  $\oplus$ . The security-specific feature is label propagation, *i.e.* every time a value is eliminated (as in the first four reduction rules), its label must be reflected on the resulting computation. To keep track of the reflected label of a computation, there is a new protected expression  $[e]_\ell$ . Rule (Tp) gives type to an expression  $[e]_\ell$ , ensuring the resulting type reflects both the label  $\ell$  and the label of the expression  $e$ .

**Defining noninterference** Noninterference is defined over observer-sensitive value and expression equivalences. These equivalences characterize the view that different observers have of values and computations. For instance, for an observer that is allowed to see only values and computations at the security label  $L$  (but not at  $H$ ), the values  $\text{true}_H$  and  $\text{false}_H$  are equivalent. However, she can distinguish between the values  $\text{true}_L$  and  $\text{false}_L$ .

The observer-sensitive equivalences can be defined using different techniques. The  $\lambda_{\text{SEC}}$  equivalence as well as all the equivalences in this thesis are defined using logical relations [42]. Figure 1.3 presents the logical relation for  $\lambda_{\text{SEC}}$ , defined inductively on the structure of types  $S$  and indexed by an observation level  $\zeta$ . We now explain the specific elements of this logical relation, but also general principles to define observer-sensitive equivalences.

- For the  $\lambda_{\text{SEC}}$  logical relation, the observation level  $\zeta$  is itself a label from the lattice. It means that there are as many observation levels (*i.e.* observers) as labels in the lattice. However, in general observation levels and security labels do not need to coincide. In particular, the observation level does not need to be a label from the lattice.
- Equivalences for values share two common things. First, and in any case, for two values  $v_1$  and  $v_2$  to be equivalent at the security type  $T_\ell$  and observational level  $\zeta$ , noted  $v_1 \approx_\zeta v_2 : T_\ell$ , they need to have type  $T_\ell$ . Second, if the security type is not observable at security level  $\zeta$ , then the values are directly equivalent. A security type  $T_\ell$  is not observable at level  $\zeta$  if  $\ell \not\sqsubseteq \zeta$ . For instance, for the lattice  $L \sqsubseteq H$ ,  $\text{Bool}_H$  is not observable at  $L$ , because  $H \not\sqsubseteq L$ .
- We now comment on the specific details for each value equivalence:
  - Two boolean security values can be equivalent at type  $\text{Bool}_\ell$  and observational level  $\zeta$ , noted  $v_1 \approx_\zeta v_2 : \text{Bool}_\ell$ , if the type  $\text{Bool}_\ell$  is observable at  $\zeta$  and the raw boolean values are syntactically equivalent. To obtain the raw value from a security value

$$\begin{aligned}
v_1 \approx_\zeta v_2 : \text{Bool}_\ell &\iff \vdash v_i : \text{Bool}_\ell \wedge \ell \sqsubseteq \zeta \Rightarrow \text{rval}(v_1) = \text{rval}(v_2) \\
v_1 \approx_\zeta v_2 : (S_1 \rightarrow S_2)_\ell &\iff \vdash v_i : (S_1 \rightarrow S_2)_\ell \wedge \ell \sqsubseteq \zeta \Rightarrow (\forall v'_1 \approx_\zeta v'_2 : S_1 \Rightarrow v_1 v'_1 \approx_\zeta v_2 v'_2 : \mathcal{C}(S_2 \sqcup \ell)) \\
e_1 \approx_\zeta e_2 : \mathcal{C}(S) &\iff \vdash e_i : S \wedge e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge v_1 \approx_\zeta v_2 : S
\end{aligned}$$

Figure 1.3:  $\lambda_{\text{SEC}}$ : Logical relation

with use the auxiliary function  $\text{rval}(r_\ell) = r$ .

- Two function security values can be equivalent at type  $(S_1 \rightarrow S_2)_\ell$  and observational level  $\zeta$ , noted  $v_1 \approx_\zeta v_2 : (S_1 \rightarrow S_2)_\ell$ , if the type  $(S_1 \rightarrow S_2)_\ell$  is observable at  $\zeta$  and the functions are semantically equivalent. Two functions  $v_1$  and  $v_2$  are semantically equivalent at type  $(S_1 \rightarrow S_2)_\ell$  if given equivalent arguments  $v'_1$  and  $v'_2$  at type  $S_1$ , their invocations produce equivalent expressions at type  $S_2 \sqcup \ell$ , noted  $v_1 v'_1 \approx_\zeta v_2 v'_2 : \mathcal{C}(S_2 \sqcup \ell)$ .

- Finally, two expressions  $e_1$  and  $e_2$  are equivalent at type  $S$  and observational level  $\zeta$ , noted  $e_1 \approx_\zeta e_2 : \mathcal{C}(S)$ , if they have type  $S$ , and both expressions reduce to equivalent values at type  $S$ .

Concerning the equivalence of expressions, note that because all expressions of  $\lambda_{\text{SEC}}$  terminate, the expression equivalence is termination-sensitive. We will come back to the subject of termination-(in)sensitive equivalences in Chapter 2.

With the observer-sensitive relation we can define noninterference. Noninterference is stated as a modular reasoning principle for open expressions: an expression  $e$  satisfies noninterference if given equivalent (secret) inputs for an observer, the executions of  $e$  with each input produce equivalent results for the observer. It characterizes that  $e$  is secure if an observer does not learn anything new about the (secret) inputs.

To account for open expressions, there are notions of satisfactory value substitution and equivalent value substitutions.

**Definition 1** (Satisfactory value substitution). *A substitution  $\gamma$  satisfies type environment  $\Gamma$ , noted  $\gamma \models \Gamma$ , iff  $\text{dom}(\gamma) = \text{dom}(\Gamma) \wedge \forall x \in \text{dom}(\Gamma). \vdash \gamma(x) : \Gamma(x)$*

**Definition 2** (Equivalent value substitutions). *Two substitutions  $\gamma_1$  and  $\gamma_2$  are related with respect to a type environment  $\Gamma$ , noted  $\gamma_1 \approx_\zeta \gamma_2 : \Gamma$ , if  $\gamma_i \models \Gamma$  and*

$$\forall x \in \text{dom}(\Gamma). \gamma_1(x) \approx_\zeta \gamma_2(x) : \Gamma(x)$$

With the notion of equivalent value substitutions, we can define noninterference. Instead of following exactly the notation and definitions of  $\lambda_{\text{SEC}}$  for noninterference, we slightly adapt them to match the style that we follow in the rest of the technical chapters of this thesis. In particular, we introduce the notation  $\text{NI}(\Gamma, e, S)_\zeta$ :

**Definition 3** (Noninterference). *An expression  $e$  satisfies noninterference under type environment  $\Gamma$  at type  $S$  and observation level  $\zeta$ , noted  $\text{NI}(\Gamma, e, S)_\zeta$  if and only if:*

$$\Gamma \vdash e : S \wedge (\forall \gamma_1, \gamma_2. \gamma_1 \approx_\zeta \gamma_2 : \Gamma \implies \gamma_1(e) \approx_\zeta \gamma_2(e) : \mathcal{C}(S))$$

A typical property about noninterference is that if it holds for an observation level  $\zeta_1$ , then it also holds for any observation level  $\zeta_2$  where  $\zeta_1 \sqsubseteq \zeta_2$ . In other words, if a program is secure for a less privileged observer, then it is also secure for a more privileged observer. In particular, all well-typed programs are secure at  $\zeta = \top$ .

**Soundness** The ultimate goal for an information flow type system is to prove that it is sound with respect to its security property, here noninterference, *i.e.* all well-typed programs are secure. For  $\lambda_{\text{SEC}}$ , this is captured in the following theorem:

**Theorem 1** (Soundness).  $\Gamma \vdash e : S \implies \forall \zeta. \text{NI}(\Gamma, e, S)_\zeta$

Theorem 1 states the if a  $\lambda_{\text{SEC}}$  program is well-typed, then it satisfies noninterference for any observer. In particular, if observation levels adhere to a lattice structure with a bottom element defined (as is the case in  $\lambda_{\text{SEC}}$ ), it suffices to show that noninterference holds for the bottom observation level, *i.e.* the public observer. Furthermore, Theorem 1 provides a modular reasoning principle for  $\lambda_{\text{SEC}}$  programs. For instance, we can prove that a function  $f : \text{Bool}_H \rightarrow \text{Bool}_L$  is necessarily a constant function—its public result cannot possibly depend on its (secret) input.

**The approach of this thesis for noninterference** The approach that we follow in this thesis to prove the different notions of type-based relaxed noninterference uses a few intermediate steps. Here, we illustrate it for the case of noninterference.

First, we generalize the definition of noninterference with a more general definition that relates two different open terms. Definition 4 provides such relation. It essentially differs from Definition 3 in that it is stated for two expressions  $e_1$  and  $e_2$  instead of only one expression  $e$ . The notion of equivalent open expressions of Definition 4 is standard for the proof technique of logical relations. Relying on different expressions, instead of the same expression, provides a more powerful statement to exploit in the proofs.

**Definition 4** (Equivalent open expressions).

$$\begin{aligned} \Gamma \vdash e_1 \approx_\zeta e_2 : S &\iff \Gamma \vdash e_1 : S \wedge \\ \forall \gamma_1, \gamma_2. \gamma_1 \approx_\zeta \gamma_2 : \Gamma &\implies \gamma_1(e_1) \approx_\zeta \gamma_2(e_2) : S \end{aligned}$$

Second, we show that noninterference is a special case of this open expressions equivalence through a simple lemma:

**Lemma 2** (Self-equivalence implies noninterference).  $\Gamma \vdash e \approx_\zeta e : S \implies \text{NI}(\Gamma, e, S)_\zeta$

Third, we define the Fundamental Property of the logical relation, which state that any well-typed open term is related to itself:

**Theorem 3** (Fundamental property).  $\Gamma \vdash e : S \implies \forall \zeta. \Gamma \vdash e \approx_\zeta e : S$

Fourth, we prove noninterference as a simple composition of the Fundamental Property and Lemma 2. Therefore, the important (and challenging) part of the proof is the Fundamental Property of the logical relation.

## 1.2 Jif: Decentralized label model, declassification and label polymorphism

Jif is an extension of Java for information flow control and it is the most mature practical information flow type system to date. The Decentralized Label Model (DLM) [37] is the label model used in Jif and in many other declassification approaches for information flow type systems. Here, we highlight the main features needed to understand later discussions, mainly in the discussions of related work (§ 2.6, § 3.7, § 4.7), in Chapter 4 when discussing label polymorphism, and in the conclusion of this thesis.

### 1.2.1 Decentralized label model

Labels in the DLM have two components, expressing a confidentiality policy and an integrity policy, respectively. However, as this thesis focuses on confidentiality, we omit the integrity policy and present labels with just confidentiality policies. Given that policies are expressed in terms of *principals*, we first introduce principals and after that, confidentiality policies and labels.

**Principals.** A principal in the DLM can be used to model users, groups, processes, etc. Methods in Jif execute in the context of an *authority*, *i.e.* a principal. Each principal can delegate its authority to another principal, which technically results in an *acts-for* relation. The notation  $p \succeq q$  means “the principal  $p$  acts for the principal  $q$ ”. The acts-for relation is reflexive and transitive. In particular, Jif models a  $\top$  principal that can act for any other principal and a  $\perp$  principal that any other principal can act for.

**Confidentiality policies** A confidentiality policy defines the *owner* of the data, used for declassification, and a list of *readers*<sup>1</sup>. For instance, the policy  $p \rightarrow q, r$  specifies that  $p$  is the owner of the data; and  $q$  and  $r$  are the readers. The owner of the data plays an important role in the declassification mechanism of Jif, since principals can only declassify their data. Regarding the terminology “readers”, it should be considered just as that: a terminology. There is no special rule to allow the running principal to see data based on the readers list. Instead, the readers list is a defining artifact. We will see now its role in characterizing valid information flows.

Confidentiality policies have an order relation  $c_1 \sqsubseteq_C c_2$  that defines when a policy  $c_1$  is at most as restrictive as the policy  $c_2$  or, equivalently, when the policy  $c_2$  is at least as restrictive as the policy  $c_1$ . Since a confidentiality policy is composed of an owner and a list of readers, all of them principals, the notion of acts-for plays a role in the order of confidentiality policies.

Intuitively, the order  $c_1 \sqsubseteq_C c_2$  holds, if the owner of  $c_2$  can act for the owner of  $c_1$  and if the readers of  $c_2$  is a subset of the readers of  $c_1$  (*i.e.*  $c_2$  has less readers). The set of syntactic readers of a confidentiality policy may represent more principals when it is expanded using the acts-for relation. To get the expanded set of readers, the DLM uses a function `readers` that takes a set of readers and returns its transitive closure with respect to the acts-for relation:

$$\begin{aligned} \text{readers} &: \bar{p} \rightarrow \bar{p} \\ \text{readers}(\bar{p}) &= \{p' \mid \exists p'' \in \bar{p}. p' \succeq p''\} \end{aligned}$$

$$\boxed{c_1 \sqsubseteq_C c_2}$$

$$\frac{r' \succeq r \quad \text{readers}(\bar{p}') \subseteq \text{readers}(\bar{p})}{r \rightarrow \bar{p} \sqsubseteq_C r' \rightarrow \bar{p}'}$$

We now illustrate the ordering of confidentiality policies with the following examples:

- $Alice \rightarrow Bob \sqsubseteq_C Alice \rightarrow \top$ . Justification:  $\text{readers}(\{\top\}) = \{\top\} \subseteq \text{readers}(\{Bob\}) = \{Bob, \top\}$
- Assuming  $Bob \succeq Dave$ ,  $Alice \rightarrow Dave \sqsubseteq_C Alice \rightarrow Bob$  holds. Justification:  $\text{readers}(\{Bob\}) = \{Bob, \top\} \subseteq \text{readers}(\{Dave\}) = \{Dave, Bob, \top\}$
- $\top \rightarrow \top$  is the most restrictive policy because not principal can act for  $\top$  and  $\text{readers}(\{\top\}) = \{\top\}$ , *i.e.* the smallest set of principals.
- $\perp \rightarrow \perp$  is the least restrictive policy because every principal can act for  $\perp$  and  $\text{readers}(\perp)$  returns all principals, *i.e.* the biggest set of principals.

**Labels** As in this presentation labels are composed only of a confidentiality policy, the label format is  $\{c\}$ , where  $c$  is a confidentiality policy. Therefore, the label ordering relation  $\sqsubseteq$  that characterizes valid information flows reduces to the confidentiality relation  $\sqsubseteq_C$ , where the labels  $\{\top \rightarrow \top\}$  and  $\{\perp \rightarrow \perp\}$  are the top and bottom elements of the lattice of confidentiality, respectively.

<sup>1</sup>In this background, we present a simplified version of the confidentiality policies of Jif. For instance, we omit conjunction and disjunction of confidentiality policies.



## 1.2.2 Declassification

To illustrate declassification in Jif, we re-implement the login functionality of the introduction. The program below declares a `login` method that takes a public argument `guess` and secret `password` that Alice owns. The label of the comparison expression `password == guess` is  $\{\text{Alice} \rightarrow \top\}$ , which reflects the most restrictive label between the labels of the arguments `password` and `guess`. Since the method return label is  $\{\perp \rightarrow \perp\}$ , and  $\{\text{Alice} \rightarrow \top\} \not\sqsubseteq \{\perp \rightarrow \perp\}$ , the Jif type system rejects the program because of the invalid flow.

```
bool{\perp \to \perp} login(String{\perp \to \perp} guess, String{\text{Alice} \to \top} password)
  return password == guess;
}
```

For the type system to accept the program above, we must use declassification. We can use the `declassify` operator to declassify the comparison expression, explicitly specifying the label after the declassification:

```
bool{\perp \to \perp} login(String{\perp \to \perp} guess, String{\text{Alice} \to \top} password)
  return declassify(password == guess, {\perp \to \perp});
}
```

However, Jif implements *selective declassification* where declassification is mediated by the authorization mechanism. That is, declassification can be performed for confidentiality policies owned by the running principal. Therefore, to know if the above code is accepted by the Jif type system, we need to take into account the running authority for the `login` method. If the running authority is `Alice` or another principal that can act for `Alice`, then the program is accepted, otherwise it is rejected.

Jif also enforces *robust declassification* [69, 38], which expresses that a principal cannot influence the declassification mechanism to increase the observation about secrets. To account for the influence of a principal in values and computations, Jif uses the integrity policy (that is part of a label). We do not present the Jif's machinery to enforce robust declassification. Instead, we give an example to understand the concept of robust declassification, using a modified version of the above `login` method.

The `login` method below receives an extra public argument `condition`, which is used to trigger the declassification mechanism. Therefore, if `condition` is provided by an attacker, then the declassification is controlled by the attacker. It gives her the possibility to observe the comparison expression, that she otherwise could not observe.

```
bool{\perp \to \perp} login(String{\perp \to \perp} guess, String{\text{Alice} \to \top} password,
                        bool{\perp \to \perp} condition)
  if(condition)
    return declassify(password == guess, {\perp \to \perp});
  return false{\perp \to \perp};
}
```

## 1.2.3 Label polymorphism

Label polymorphism allows programs to be generic with respect to security policies, therefore, it is a very practical feature for information flow type systems. Jif supports label polymor-

phism in different flavors. Here, we highlight explicit label polymorphism, since it is the one we develop in Chapter 4.

We illustrate the need for label polymorphism with the following method `square`, which computes the square of a number.

```
int square(int value){ return value * value; }
```

This is a common functionality, and it is natural to be able to apply it to values of any security label. However, in an information flow type system without label polymorphism, we must write one version of the method `square` for each security label, which is clearly impractical.

With explicit label polymorphism, in Jif, the method `square` can introduce a generic label `L`, providing a label-polymorphic definition.

```
int{L} square[label L](int{L} value){ return value * value; }
```

Then, we can instantiate `square` with any security label. For instance, `square[Alice  $\rightarrow$   $\top$ ]` gives a specialized version that takes an integer at label `Alice  $\rightarrow$   $\top$`  and returns its square at the same label.

Jif supports implicit label polymorphism for methods. That is, if the arguments and the return type of a method do not have explicit label annotations, then the method is implicitly polymorphic.

Jif additionally supports parametrized classes, *i.e.* explicit label polymorphism at the class level, which is useful to write label-agnostic data structures. To illustrate the concept, we present the definition of a label-polymorphic linked list of strings:

```
class ListNode[label L]{
    public ListNode[L]{L} next;
    public String{L} value;

    public ListNode(String{L} value){
        this.value = value;
    }
}
```

The class `ListNode` above introduces a generic label `L` that is available in the scope of the class. Both fields `next` and `value` have the label `L`. In particular, for the field `ListNode[L]{L} next`, the label `L` is also used to instantiate the recursive use of the type `ListNode`. With the class `ListNode`, we can define any kind of label-polymorphic operations over lists, such as reversing a list.

# Chapter 2

## Type-based declassification for objects

In this chapter, we develop the object-oriented labels-as-types approach. As mentioned in the introduction, real systems need to selectively declassify some information. The labels-as-functions approach of Li and Zdancewic [31] is particularly interesting, allowing programmers to specify *declassification policies* that capture the intended manner in which public information can be computed from private data.

Here, we show how we can exploit the familiar notion of type abstraction to support expressive declassification policies in a simpler, yet more expressive manner. In particular, the type-based approach to declassification—here developed in an object-oriented setting—addresses several issues and challenges with respect to prior work, including a simple notion of label ordering based on subtyping, support for recursive declassification policies, and a local, modular reasoning principle for relaxed noninterference.

This chapter is based on the work of Cruz et al. [22]

### 2.1 Introduction

Recall the login example from the introduction, which does not satisfy noninterference, and requires selective declassification to allow the public observer to observe the result of the comparison of a secret password with a public value.

```
String login(String guess, String password)
  if(password == guess)
    return "Login Successful"
  else
    return "Login failed"
}
```

Li and Zdancewic [31] proposed an expressive mechanism for declassification policies that supports the extensional specification of secrets and their intended declassification. A declassification policy is a function that captures *what* information on a confidential value can be declassified to eventually produce a public value. For the password checker example, if the declassification policy for password is  $\lambda x.\lambda y.x==y$ , then an equality comparison with

`password` can be declassified (and thus be made public). However, this declassification policy for `password` disallows arbitrary declassifications such as revealing the password. Furthermore, declassification can be *progressive*, requiring several operations to be performed in order to obtain public data: *e.g.*  $\lambda x.\lambda y.\text{hash}(x)=y$  specifies that only the result of comparing the hash of the password for equality can be made public.

The formal security property, called *relaxed noninterference*, states that a secure program can be rewritten into an equivalent program without any variable containing confidential data but whose inputs are confidential and declassified. For the password checker example with  $p \triangleq \lambda x.\lambda y.x=y$  as the declassification policy for `password`, the program `login(guess,password)` can be rewritten to the equivalent program `login'(guess,p(password))` where `login'` is:

```
String login'(String guess, String→Bool eq){
  if(eq(guess)) ...
}
```

Note that `p(password)` is a closure that strongly encapsulates the secret value (on its environment), and only allows equality comparisons.

The proposal of Li and Zdancewic elegantly and formally captures the essence of flexible declassification but it suffers from a number of limitations that jeopardize its practical adoption. First, security labels are sets of functions that form a security lattice whose ordering, based on a semantic interpretation of these sets of functions, is far from trivial [31]: it relies on a general notion of program equivalences that would be both hard to implement and to comprehend. Second, Li and Zdancewic explicitly rule out *recursive* declassification policies, which are however natural when expressing declassification of recursive data structures. Finally, the rewriting-based definition of relaxed noninterference is unsatisfying for practical software development, as it rigidly requires all secrets to be both *global* and *external*, thereby losing modular reasoning; as recognized by the authors, local language constructs for introducing secrets and their policies are lacking [31].

In this chapter, we exploit the familiar notion of *type abstraction* to capture declassification policies in a simpler, yet more expressive manner. Here, we specifically adopt the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object, and type abstraction is driven by subtyping. For instance, the empty interface type—the root of the subtyping hierarchy—denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data. Our initial observation is that any interface in between these two extremes denotes *declassification* opportunities. Additionally, choosing objects, as opposed to records, allows us to explore recursive declassification policies from the start, given that the essence of data abstraction in OOP are recursive types [18].

The object-oriented type-based approach to confidentiality is very intuitive as it only relies on concepts that are readily available in object-oriented languages: a declassification policy is simply a *method signature*, a security label is an *object interface*, and label ordering boils down to *subtyping*. Progressive declassification occurs through chaining of *method invocations*. In fact, the only extension to the standard programming model is that a security type has two facets, each representing the view available to a private and public observer, respectively. In

addition to being intuitive, the type-based approach addresses the issues and challenges of the downgrading policies of Li and Zdancewic: *a)* there is no need to rely on general program equivalences to define and decide label ordering, which is just standard, syntactic subtyping; *b)* declassification naturally scales to recursive policies over recursive data structures; and *c)* type-based relaxed noninterference is formulated as a *modular* reasoning principle, and local secrets can be introduced with standard type annotations.

This chapter makes the following contributions:

- We develop a novel type-based approach to declassification policies, which supports interesting scenarios while appealing to standard programming concepts such as interface types and subtyping (Section 2.2).
- We capture the essence of type-based declassification in a core object-oriented language, `ObSEC`, in which a security type is a pair of (recursive) object types (Section 2.4). We describe the static and dynamic semantics of `ObSEC` and prove type *safety*.
- We specify the formal semantic notion of *type-based relaxed noninterference*, which accounts for type-based declassification policies, independently of any enforcement mechanism (Section 2.5). We then prove type *soundness* of `ObSEC`: a well-typed program satisfies type-based relaxed noninterference.

Section 2.6 discusses related work and Section 2.7 gives conclusions related to the work developed in the chapter. We have implemented an interactive prototype of `ObSEC` available at <https://pleiad.cl/obsec/>.

## 2.2 Overview

We now progressively and informally introduce the object-oriented type-based approach to declassification policies, appealing first to a simple intuitive connection with type abstraction. We then explain why this first intuition is insufficient, and refine it in order to support the key features of a security-typed language with expressive declassification. We end by discussing the security guarantee supported by the approach.

### 2.2.1 Type abstraction and confidentiality

It is well-known that type abstraction can capture the need to expose only a subset of the operations of an object. For instance, if the `password` secret is made available using the interface type `StringEq`  $\triangleq$  `[eq : String → Bool]`, the `login` function from Section 2.1 can be rewritten as follows:

```
String login(String guess, StringEq password){
  if(password.eq(guess)) ...
}
```

Because `password` has type `StringEq`, the `login` function cannot accidentally leak information about the password. In particular, note that the function cannot even return the password because `StringEq` is a *supertype* of `String`, not a subtype. Therefore, the standard substitutability expressed by subtyping seems to align well with the valid information flows permitted

in a confidentiality type system: a (public) string value at type `String` can be used freely, and passed as argument expecting a (mostly) private `StringEq`, which only exposes equality comparison. Similarly, any value can flow to a secret variable, characterized by the empty interface type,  $\top \triangleq []$ .<sup>1</sup>

Progressive declassification policies can be expressive with *nested* interface types. For instance, assume that `String` objects have a `hash` method, of type `Unit → Int`. To specify that only the hash of the password can be compared for equality, it suffices to expose the password at type `StringHashEq`  $\triangleq$  `[hash : Unit → IntEq]`, where `IntEq`  $\triangleq$  `[eq : Int → Bool]`:

```
String login(Int guess, StringHashEq password){
  if(password.hash().eq(guess)) ...
}
```

In the code above, the only available operation on `password` is `hash()`, which in turn returns an integer that only exposes an equality comparison. Note that here again, `StringHashEq >: String` and `IntEq >: Int`.

**Recursive declassification.** The informal presentation of type-based declassification so far has exemplified two of the main advantages of our approach: security label ordering is syntactic subtyping, and secrets and their declassification policies can be declared locally, by standard type annotations. We now illustrate recursive declassification policies.

Recursive declassification policies are desirable to express interesting declassification of either inductive data structures or object interfaces (whose essence are recursive types [18]). Consider for instance a secret list of strings, for which we want to allow traversal of the structure and comparison of its elements with a given string. This can be captured by the recursive type `StrEqList` defined as:

$$\text{StrEqList} \triangleq [\text{isEmpty} : \text{Unit} \rightarrow \text{Bool}, \text{head} : \text{Unit} \rightarrow \text{StringEq}, \text{tail} : \text{Unit} \rightarrow \text{StrEqList}]$$

To allow traversal, the declassification policy exposes the methods `isEmpty`, `head` and `tail`, with the specific constraints that *a*) accessing an element through `head` yields a `StringEq`, not a full `String`, and *b*) the `tail` method returns the tail of the list *with the same* declassification policy. Type-based declassification policies can therefore naturally be recursive, as long as the underlying type language allows (some form of) recursive types.

## 2.2.2 Facets of computation

With the standard programming approach described so far, a program that attempts to violate the declassification protocol of an object is rejected by the (standard) type system because it is ill-typed. For instance:

```
String login(Int guess, StringEq password){
  if(password.length().eq(guess)) ...
}
```

---

<sup>1</sup>The reader might wonder at this point about the effect of using arbitrary downcasts, as supported in Java. Indeed, downcasts are a way to violate type abstraction, and therefore to violate the type-based security guarantees. For instance, the `login` function could return `(String)password`, thereby returning the password for public consumption. Fortunately, there is a simple solution to this issue, which we discuss in Section 2.5.4.

is rejected because `length` is not part of the exposed interface of `password`.

However, security-typed languages typically are more flexible than this: they allow computation to proceed with private information, but ensure the result of such computation is itself private [67]. For instance, adding a public integer and a private integer yields a private result (as we illustrated in 1.1). Li and Zdancewic [31] follow the same approach with declassification policies: using a secret in a way that does not follow its declassification policy yields a private result. The justification of these approaches is that computation with private data *is* relevant, but only visible to a high security, privileged observer; noninterference only dictates that a low security, public observer should not be able to deduce information about private data by observing public outputs.

This means that security-typed languages inherently adopt a multi-faceted view of computation, where each observation level corresponds to a different facet. Sticking to a two-facet, private/public model, the definition of `login` above is well-typed if one “knows” that `password` is in fact a `String` object. In this case using `length` is valid: it just yields a private result. Flow-sensitivity then ensures that the result of `login`, which follows from a conditional branching computed based on a secret value, is also private.

### 2.2.3 Faceted types

To accommodate the possibility of computing with private data, we extend standard types to *faceted types*. A security type  $S$ , noted  $T \triangleleft U$ , consists of two standard types: type  $T$  for the private interface, and type  $U$  for the public interface<sup>2</sup>. In this chapter, we often use the notation  $T_L$  as a shortcut for the lowest-confidentiality security type  $T \triangleleft T$ , in which the public facet exposes the same interface as the private facet, and  $T_H$  for the fully-confidential security type  $T \triangleleft \top$  in which the public facet is empty.

To express that `password` is a private string that can only be declassified through equality comparison, we can use the following signature for `login`:

```
StringL login(IntL guess, String $\triangleleft$ StringEq password)
```

With this signature the previous definition of `login`, which invokes `length`, is still ill-typed. Indeed, the body of the function now has type `StringH`, capturing the fact that the resulting string is private, but the signature pretends that the result of `login` is public, which violates noninterference. For `login` to be well-typed, either the declared return type should be changed to `StringH`, or the conditional should adhere to the public facet `StringEq`.

Note that subtyping naturally extends *covariantly* to faceted types, *i.e.*  $T_1 \triangleleft U_1 <: T_2 \triangleleft U_2$  iff both  $T_1 <: T_2$  and  $U_1 <: U_2$ . Therefore, it is invalid to pass a private string of type `String $\triangleleft$  $\top$`  to a function expecting a declassifiable string of type `String $\triangleleft$ StringEq`, because  $\top$  is not a subtype of `StringEq`. Subtyping on the public facet corresponds to security label ordering; compared to the semantic, equivalence-based interpretation of labels of Li and Zdancewic, here label ordering is just standard syntactic subtyping.

---

<sup>2</sup>Similarly to multi-faceted execution [8], one can extend the model to support  $n$  levels of observations, by introducing security types with  $n$  facets.

Object types directly support the possibility to offer different declassification paths for the same secret. For instance, the security type  $\text{String} \triangleleft [\text{hash} : \text{Unit}_L \rightarrow \text{Int}_L, \text{length} : \text{Unit}_L \rightarrow \text{Int}_L]$  allows a client to obtain a public integer from a string by using either its hash or its length. Naturally, by breadth subtyping, such a secret with two possible declassification paths can also be used as a more restricted secret, *e.g.* one that only exposes its hash publicly.

## 2.3 Type-based relaxed noninterference, informally

The security property we establish in this chapter is a particular form of termination insensitive noninterference, called *typed-based relaxed noninterference* (TRNI for short). Like the relaxed noninterference result of Li and Zdancewic [31], TRNI accounts for declassification policies.

To understand the intuition behind TRNI, we must first establish a notion of type-based observational equivalence between objects. The starting point of the notion of equivalence is that an object is defined by the observations that can be made on it, that is, by invoking its methods [18]. More precisely, two objects  $o_1$  and  $o_2$  are said to be observationally equivalent at type  $S$ , with  $S \triangleq T \triangleleft U$ , if for each method  $m : S_1 \rightarrow S_2$  of the *public* facet  $U$ , invoking  $m$  on  $o_1$  and  $o_2$  with equivalent arguments at type  $S_1$ , yields equivalent results at type  $S_2$ . Crucially, the definition of equivalence uses the *public* facet of the type, thereby accounting for observational equivalence only up to declassified information.

For example, the strings "john" and "mary" are not equivalent at type  $\text{String} \triangleleft \text{String}$ , because a public observer can observe the first character of each string and realize they are different. However, these strings are equivalent when observed at  $\text{String} \triangleleft \text{StringLen}$ , where  $\text{StringLen} \triangleq [\text{length} : \text{Unit}_L \rightarrow \text{Int}_L]$ , because the only declassified information about the strings is their length, which is here equal. This also means that "john" and "james" are equivalent when are observed at type  $\text{String}_H$  (*i.e.*  $\text{String} \triangleleft \top$ ) since there are no observations available to distinguish them. In fact, any two objects of type  $T$  are equivalent at type  $T_H$ .

Given this notion of equivalence, a program satisfies TRNI at type  $S_{out}$ , if given two inputs that are equivalent at type  $S_{in}$ , it produces two results that are equivalent at type  $S_{out}$ . Intuitively, the types  $S_{in}$  and  $S_{out}$  capture the *knowledge* of public observers. Another way to understand TRNI is that, if the initial knowledge *implies* the final knowledge, then the program is secure for the public observer.

For instance, consider a program with an input  $x$  of type  $\text{String} \triangleleft \text{StringLen}$ . The program  $x.\text{length}$  satisfies TRNI at type  $\text{Int} \triangleleft \text{Int}$ : two executions of the program with related inputs at  $\text{String} \triangleleft \text{StringLen}$ , such as "john" and "mary", yields two identical results at type  $\text{Int} \triangleleft \text{Int}$  (*i.e.* 4 in both cases). However, the program  $\text{if}(x.\text{eq}(\text{"mary"})) \text{return } 1 \text{ else } 2$  does not satisfy TRNI at type  $\text{Int} \triangleleft \text{Int}$  because there are equivalent inputs at type  $\text{String} \triangleleft \text{StringLen}$  ("john" and "mary") that yield different outputs at type  $\text{Int} \triangleleft \text{Int}$  (1 and 2). For this program, the only secure observation level is  $\text{Int} \triangleleft \top$ .

We formally define these notions, and prove that the type system we propose enforces TRNI, in Section 2.5.



$e ::= v \mid e.m(e) \mid x$	(terms)	$x, y, z$	(variables)
$v ::= [z : S \Rightarrow \overline{m(x)} e]$	(values)	$\alpha, \beta$	(type variables)
$T, U ::= O \mid \alpha$	(types)	$m$	(method labels)
$O ::= \mathbf{Obj}(\alpha). [\overline{m : S \rightarrow S}]$	(object type)		
$S ::= T \triangleleft U$	(security type)	$T_{\mathbb{L}} \triangleq T \triangleleft T$	$T_{\mathbb{H}} \triangleq T \triangleleft \top$

Figure 2.1:  $\mathbf{Ob}_{\text{SEC}}$ : Syntax

## 2.4 Formal semantics

We develop type-based declassification and relaxed noninterference using a core object-oriented language,  $\mathbf{Ob}_{\text{SEC}}$ , whose syntax is presented in Figure 2.1. The syntax of  $\mathbf{Ob}_{\text{SEC}}$  is similar to that of the object calculi of Abadi and Cardelli [1]. It includes three kinds of expressions: variables, objects and method invocations. Note that we do not include method updates or classes, both unnecessary to formulate our proposal. An object  $[z : S \Rightarrow \overline{m(x)} e]$  is a collection of method definitions, where method names are unique. The object definition explicitly binds the self variable  $z$  in method bodies, with ascribed security type  $S$ . The distinguishing feature of  $\mathbf{Ob}_{\text{SEC}}$  are security types: as introduced in Section 2.2, a security type  $S$  is a two-faceted type  $T \triangleleft U$ , where  $T$  (resp.  $U$ ) is the private (resp. public) facet. The public facet corresponds to the declassification policy of an object. A fully opaque secret has type  $T \triangleleft \top$  (also noted  $T_{\mathbb{H}}$ ), exposing no method at all, while a low-confidentiality object has type  $T \triangleleft T$  (also noted  $T_{\mathbb{L}}$ ), publicly exposing its full interface. A type  $T$  or  $U$  is either a (recursive) object type  $\mathbf{Obj}(\alpha). [\overline{m : S \rightarrow S}]$ , where method types can use the self type variable  $\alpha$ , or a type variable. Note that we do not model parametric polymorphism in this core calculus, so type variables are only used for self types. Following the tradition of Abadi and Cardelli,  $\mathbf{Ob}_{\text{SEC}}$  does not include base (non-object) types, however they can be easily added or encoded.

### 2.4.1 Subtyping

The  $\mathbf{Ob}_{\text{SEC}}$  subtyping judgment  $\Phi \vdash T <: U$  is presented in Figure 2.2. The subtyping environment  $\Phi$  is a set of subtyping assumptions between type variables, *i.e.*  $\Phi ::= \cdot \mid \Phi, \alpha <: \beta$ .<sup>3</sup> For all judgments in this thesis, we often omit the empty environment, *e.g.* we write  $\vdash T <: U$  for  $\cdot \vdash T <: U$ .

Rule (SObj) accounts for subtyping between object types. Object type  $T_1$  is a subtype of object type  $T_2$  if  $T_1$  has at least the same methods as  $T_2$ , possibly more specialized. For this, the rule checks subtyping between method types under a subtyping assumption between the self type variable of  $T_1$  and that of  $T_2$ . For instance, consider the following object types:

$$\begin{aligned} \text{Counter} &\triangleq \mathbf{Obj}(\alpha). [\text{get} : \text{Unit}_{\mathbb{L}} \rightarrow \text{Int}_{\mathbb{L}}, \text{inc} : \text{Unit}_{\mathbb{L}} \rightarrow \alpha_{\mathbb{L}}, \text{dec} : \text{Unit}_{\mathbb{L}} \rightarrow \alpha_{\mathbb{L}}] \\ \text{IncCounter} &\triangleq \mathbf{Obj}(\beta). [\text{get} : \text{Unit}_{\mathbb{L}} \rightarrow \text{Int}_{\mathbb{L}}, \text{inc} : \text{Unit}_{\mathbb{L}} \rightarrow \beta_{\mathbb{L}}]. \end{aligned}$$

To establish that  $\text{Counter}$  is a subtype of  $\text{IncCounter}$ , the covariance between the return types of the  $\text{inc}$  method requires a subtyping assumption between type variables, here  $\alpha <: \beta$ .

<sup>3</sup>Type variables must appear at most once in the subtyping environment.

$$\boxed{\Phi \vdash T <: T}$$

$$\text{(SObj)} \frac{O_1 \triangleq \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \quad O_2 \triangleq \mathbf{Obj}(\beta). \overline{[m' : S'_1 \rightarrow S'_2]} \quad \overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j})}{\Phi \vdash O_1 <: O_2}$$

$$\text{(SVar)} \frac{\alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \quad \text{(SSubEq)} \frac{O_1 \equiv O_2}{\Phi \vdash O_1 <: O_2} \quad \text{(STrans)} \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$$\boxed{\Phi \vdash S <: S}$$

$$\text{(TSubST)} \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 2.2:  $\mathbf{Ob}_{\text{SEC}}$ : Subtyping rules

$$\boxed{\text{msig}(O, m) = S \rightarrow S}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \quad S \triangleq S_{1i}[O/\alpha] \quad S' \triangleq S_{2i}[O/\alpha]}{\text{msig}(O, m_i) = S \rightarrow S'}$$

$$\boxed{m \in O}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]}}{m_i \in O}$$

$$\boxed{\text{methimpl}(o, m) = x.e}$$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m(x)}e]}{\text{methimpl}(o, m_i) = x.e_i}$$

Figure 2.3:  $\mathbf{Ob}_{\text{SEC}}$ : Some auxiliary definitions

Rule (SVar) specifies subtyping between type variables, which only holds if the relation is in the subtyping environment. Rule (SSubEq) justifies subtyping between *equivalent types*. We consider type equivalence up to renaming and folding/unfolding of self type variables; for instance:

$$\begin{aligned} \mathbf{Obj}(\alpha). [m : \alpha_L \rightarrow \alpha_L] &\equiv \mathbf{Obj}(\beta). [m : \beta_L \rightarrow \beta_L] && \text{(alpha equivalence)} \\ \mathbf{Obj}(\alpha). [m : S \rightarrow \alpha_L] &\equiv \mathbf{Obj}(\alpha). [m : S \rightarrow \mathbf{Obj}(\beta). [m : S \rightarrow \beta_L]_L] && \text{(fold/unfold equivalence)} \end{aligned}$$

(Appendix A.1.4 provides the complete definition of type equivalence.)

Rule (STrans) is standard. Rule (TSubST) justifies subtyping between security types, which is covariant in both facets.

Figure 2.3 presents auxiliary functions used to test method membership in a type ( $m \in T$ ), to get the type of a method in an object type ( $\text{msig}$ ) and to get the implementation of a method ( $\text{methimpl}$ ). These operations are standard; the only interesting thing to note is that in  $\text{msig}$  we close the types in the method signature, by replacing type variables with their object types.

## 2.4.2 Type system

Figure 2.4 shows the typing rules of  $\text{Ob}_{\text{SEC}}$ . The type judgment  $\Gamma \vdash e : S$  gives a security type to an expression under a type environment  $\Gamma$  that binds variables to types ( $\Gamma ::= \cdot \mid \Gamma, x : S$ ). In what follows, we assume well-formedness of types and environments: informally, an environment is well-formed if all security types are closed and well-formed; a well-formed security type satisfies the requirement that the safety type is a subtype of the declassification type. We further discuss well-formedness at the end of this section.

Rules (TVar) and (TSub) are standard. The (TObj) rule accounts for objects. It requires each method body to be well-typed with respect to the private facet of the object. In particular, the method body must match the return type of the method signature in the private facet of the self type  $S$ .

From a security point of view, the interesting rules are the ones for method invocation. Rule (TmD) applies when the invoked method is part of the *public* facet of the receiver. In this case, because the method invocation respects the declassification policy, the overall type of the invocation is the return type of the method in the public facet. This expresses that the invocation advances a step in the progressive declassification of the object. For instance, if the expression  $e_1$  has the declassification type  $\text{StringHashEq} \triangleq [\text{hash} : \text{Unit}_L \rightarrow \text{Int} \triangleleft \text{IntEq}]$ , the invocation  $e_1.\text{hash}()$  has type  $\text{Int} \triangleleft \text{IntEq}$ , expressing that the returned value is a secret that can further be declassified by calling the method  $\text{eq}$  from  $\text{IntEq}$ .

Rule (TmH) applies when the method is not in the declassification type  $U$ , but only in the safety type  $T$  (if the method is not in  $T$ , the expression is ill typed). In this case, the method call is accessing the “secret” part of the object: the result of the method invocation must therefore be protected by changing its public facet to  $\top$ . This rule captures the design decision that using a secret beyond its declassification policy is allowed, but the result must be secret. In other words, only a privileged observer can use objects beyond their declassification policies; to a public observer, the results of these interactions are unobservable.<sup>4</sup>

## 2.4.3 Dynamic semantics

We define a standard call-by-value small-step semantics for  $\text{Ob}_{\text{SEC}}$ , based on evaluation contexts  $E ::= [ ] \mid E.m(e) \mid v.m(E)$ .

The language includes a single reduction rule, for method invocation, which is standard:

$$(\text{EMInv}) \frac{o \triangleq [z : \_ \Rightarrow \_] \quad \text{methimpl}(o, m) = x.e}{E[o.m(v)] \mapsto E[e[o/z][v/x] ]}$$

## 2.4.4 Type safety

We now establish that well-typed  $\text{Ob}_{\text{SEC}}$  programs are safe. Note that type safety does not provide any *security guarantees* for  $\text{Ob}_{\text{SEC}}$ . (Security guarantees will be addressed in

---

<sup>4</sup>Access modifiers in object-oriented languages, such as `private` and `public` in Java are a really different mechanism. Such modifiers are about *encapsulation*, not about *information flow*. The essential difference can be observed in rule (TmH), which propagates privacy on return values.

$$\boxed{\Gamma \vdash e : S}$$

$$\begin{array}{c}
\text{(TVar)} \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(TSub)} \frac{\Gamma \vdash e : S' \quad \vdash S' <: S}{\Gamma \vdash e : S} \\
\text{(TObj)} \frac{S \triangleq T \triangleleft U \quad \text{msig}(T, m_i) = S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}{\Gamma \vdash [z : S \Rightarrow \overline{m(x)} e] : S} \\
\text{(TmD)} \frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \in U \quad \text{msig}(U, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2} \\
\text{(TmH)} \frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \notin U \quad \text{msig}(T, m) = S_1 \rightarrow T_2 \triangleleft U_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \top}
\end{array}$$

Figure 2.4:  $\text{Ob}_{\text{SEC}}$ : Static semantics

$$\begin{aligned}
\mathcal{V}_k[S] &= \{v = [z : S_1 \Rightarrow \_ ] \mid S \triangleq T \triangleleft U \quad \vdash S_1 <: S \wedge \\
&\quad (\forall j < k. v \in \mathcal{V}_j[S_1]) \wedge \\
&\quad (\forall m \in T, v'. \text{msig}(T, m) = S' \rightarrow S'' \quad \text{methimpl}(v, m) = x.e \\
&\quad \quad v' \in \mathcal{V}_j[S'] \implies e[v/z][v'/x] \in \mathcal{C}_j[S''])\} \\
\mathcal{C}_k[S] &= \{e \mid \forall j < k. \forall e'. (e \mapsto^j e' \wedge \text{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[S]\}
\end{aligned}$$

Figure 2.5:  $\text{Ob}_{\text{SEC}}$ : Unary logical relation for safety

Section 2.5.) A program  $e$  is *safe*, noted  $\text{safe}(e)$ , if it does not get stuck, *i.e.* if it either reduces to a value or diverges.

**Definition 5** (Safety).  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$

We prove type safety for  $\text{Ob}_{\text{SEC}}$  using a semantic interpretation of types as a unary logical relation [3]. We cannot however define the logical relation based on a direct induction over the structure of types, because of recursive types, which would make such a definition ill-founded. Therefore, we use a step-indexed logical relation [4, 5]. We establish an intermediary result for a fixed number  $k$  of steps, meaning that a term is safe for  $k$  evaluation steps, and then quantify  $\forall k \geq 0$  to obtain the general result. Step indexing ensures the well-foundedness of the logical relation.

Figure 2.5 defines the unary logical relation that captures the safety interpretation of types as values and computations, in a mutually recursive manner. The set  $\mathcal{V}_k[S]$  denotes the safe value interpretation of type  $S$  for  $k$  steps; it contains all the *values* (*i.e.* objects) for which it is safe (for any  $j < k$  number of steps) to invoke methods of the safety type  $T$  of the security type  $S \triangleq T \triangleleft U$ . Note that the definition needs to assume that the self object is in the value interpretation of  $S$ , for  $j < k$  steps; without step-indexing, this relation would be ill-founded due to the recursive nature of objects through their self variables. The set  $\mathcal{C}_k[S]$  contains all the *expressions* that can be safely executed for  $k$  steps at the security type  $S$ . In the

definition, the  $\text{irred}(e)$  predicate denotes irreducible expressions, *i.e.* expressions  $e$  such that  $\nexists e'. e \mapsto e'$ .

We define *semantic typing*, written  $\models e : S$ , to denote that a closed expression  $e$  executes safely for any fixed number of steps:

**Definition 6** (Semantic typing).  $\models e : S \iff \forall k \geq 0. e \in \mathcal{C}_k[S]$ .

We then first prove that semantic typing does imply safety as per Definition 5.

**Lemma 4** (Semantic type safety).  $\models e : S \implies \text{safe}(e)$ .

*Proof.* To show  $\text{safe}(e)$  we need to consider an arbitrary  $e'$  such that  $e \mapsto^* e'$  and then show that either  $e' = v$  or  $\exists e''. e' \mapsto e''$

Let us consider an arbitrary  $j_1$  to count the step that takes  $e \mapsto^* e'$ . Let us denote  $l = j_1 + 1$ . By expanding the definition of  $\models e : S$  we have  $\forall k \geq 0. e \in \mathcal{C}_k[S]$ . We instantiate this with  $k = l$  to obtain  $e \in \mathcal{C}_l[S]$ . By expanding this we have:

$\forall j < l. \forall e_1. (e \mapsto^j e_1 \wedge \text{irred}(e_1)) \implies e_1 \in \mathcal{V}_{k-j}[S]$ . We instantiate  $e \in \mathcal{C}_l[S]$  with  $j_1$  and  $e'$  and we obtain:  $(e \mapsto^{j_1} e' \wedge \text{irred}(e')) \implies e' \in \mathcal{V}_{k-j_1}[S]$ .

There are two cases to consider:  $\neg \text{irred}(e')$  and  $\text{irred}(e')$ . If  $\neg \text{irred}(e')$ , then by definition  $\exists e''. e' \mapsto e''$ . If  $\text{irred}(e')$ , we have that  $e' \in \mathcal{V}_{k-j_1}[S]$ , so  $e'$  is a value.  $\square$

Second, we prove that syntactic typing (Figure 2.4) implies semantic typing.

**Lemma 5** (Syntactic typing implies semantic typing).  $\vdash e : S \implies \models e : S$

*Proof.* The result follows from a similar lemma on open terms:  $\Gamma \vdash e : S \implies \Gamma \models e : S$ . We define a standard notion of safe value substitutions [3], *i.e.* partial maps from variables to safe values,  $\gamma \in \mathcal{G}_k[\Gamma]$  and  $\Gamma \models e : S$  as follows:

$\gamma \in \mathcal{G}_k[\Gamma] \iff \text{dom}(\gamma) = \text{dom}(\Gamma) \text{ and } \forall x \in \text{dom}(\Gamma). \gamma(x) \in \mathcal{V}_k[\Gamma(x)]$   
 $\Gamma \models e : S \iff \forall k \geq 0, \forall \gamma. \gamma \in \mathcal{G}_k[\Gamma] \implies \gamma(e) \in \mathcal{C}_k[S]$ .

Then we prove that  $\Gamma \vdash e : S \implies \Gamma \models e : S$  by induction on the typing derivation of  $e$ . The case (TVar) is direct from the definition of  $\gamma \in \mathcal{G}_k[\Gamma]$ . The case (TSub) follows directly from a subsumption lemma ( $e \in \mathcal{C}_k[S] \wedge \vdash S <: S' \implies e \in \mathcal{C}_k[S']$ ). Cases (TObj), (TmD) and (TmH) are proven by unfolding the definitions of  $\mathcal{C}_k[S]$  and  $\mathcal{V}_k[S]$ , and applying the induction hypotheses for smaller indexes. For these cases, we use mainly a monotonicity lemma for the value interpretation of a type regarding the index, *i.e.*  $e \in \mathcal{V}_k[S] \wedge j \leq k \implies v \in \mathcal{V}_j[S]$ .  $\square$

Together, Lemmas 4 and 5 imply that well-typed programs are safe.

**Theorem 6** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

Now that we have established that  $\text{Ob}_{\text{SEC}}$  is a well-defined, type-safe language, Section 2.5 will develop its security guarantees.

## 2.4.5 A note on well-formedness

Before we proceed, however, we need to mention a technical yet important issue that we overlooked so far. For the main results of Section 2.5 to hold, we need to ensure that we work with *well-formed* security types, *i.e.* that the private facet type is a subtype of the public facet type. In a language with simple, non-recursive types, defining such subtyping constraints is straightforward. However, in the presence of recursive (object) types, defining the rules for the subtyping constraint of security types is rather subtle and involved. The subtlety with type variables is that, at some point, we might have to check well-formedness of a security type with a type variable in one of its facets, *e.g.*  $\alpha \triangleleft T$ , without knowing any relation between  $\alpha$  and  $T$ . To address this, we need *to remember* the surrounding recursive object type  $O$  that binds  $\alpha$ , and to transform the check  $\vdash \alpha <: T$  to  $\vdash O <: T$ . For conciseness, we leave out the well-formedness rules from the main body of the chapter; they are fully described in Appendix A.1.2. In what follows, we systematically assume that security types (and by extension, type environments) are well-formed.

## 2.5 Type-based relaxed noninterference

Faceted security types support information-flow security with declassification. The security property that type-based declassification supports is a form of relaxed noninterference [31], which we informally explained in Section 2.2. This section formally defines the notion of type-based relaxed noninterference (TRNI) *independently of any enforcement mechanism*. Then, we prove that the type system of  $\text{Ob}_{\text{SEC}}$  is sound with respect to this property.

### 2.5.1 Type-based equivalence

As introduced in Section 2.2, TRNI is defined in terms of a notion of type-based equivalence between objects: a program satisfies TRNI at type  $S_{out}$ , if given two equivalent inputs at type  $S_{in}$ , it produces two equivalent results at type  $S_{out}$ . Equivalence at a type accounts for the possible observations (*i.e.* method invocations) that one is allowed to make on an object. We define this equivalence as a step-indexed logical relation [4], in Figure 2.6. We define how to relate values (*i.e.* objects) as well as computations (*i.e.* expressions). Step indexing is required due to the recursive nature of object types, as explained below.

Note that the definitions use a simple typing judgment that does not account for security typing at all; its sole purpose is to ensure safety. This is crucial: the public facets of security types only play the role of *specifications* of declassification policies, and the logical relation specifies the *meaning* of these specifications, without any consideration for an enforcement mechanism. In particular, observe that the definitions in Figure 2.6 do *not* appeal to security type judgments ( $\vdash$ ), but only to simple type judgments ( $\vdash_1$ ).

**Definition 7** (Simple typing judgment). *Based on the security typing judgment  $\Gamma \vdash e : S$ , we define the simple typing judgment  $\Gamma \vdash_1 e : T$  by focusing only on the private facet of security types. Formally:  $\Gamma \vdash_1 e : T \iff \Gamma \vdash e : T \triangleleft U$  for some  $U$ . (The inductive definition of simple typing is in Appendix A.1.5.)*

Intuitively, two objects  $v_1$  and  $v_2$  are equivalent at type  $S \triangleq T \triangleleft U$  for  $k$  steps, noted

$$\begin{aligned}
v_1 \approx_k v_2 : \mathcal{V}[[S]] &\iff S \triangleq T \triangleleft U \quad v_i \triangleq [z : \_ \Rightarrow \_] \\
&\vdash_1 v_i : T \wedge (\forall m \in U. \text{msig}(U, m) = S' \rightarrow S'' \quad \text{methimpl}(v_i, m) = x.e_i \\
&\forall j < k, v'_1, v'_2. v_1 \approx_j v_2 : \mathcal{V}[[S]] \wedge \\
&\quad (v'_1 \approx_j v'_2 : \mathcal{V}[[S']] \implies e_1 [v_1/z] [v'_1/x] \approx_j e_2 [v_2/z] [v'_2/x] : \mathcal{C}[[S'']])) \\
e_1 \approx_k e_2 : \mathcal{C}[[S]] &\iff S \triangleq T \triangleleft U \\
&\vdash_1 e_i : T \wedge (\forall j < k. (e_1 \mapsto^{\leq j} v_1 \wedge e_2 \mapsto^{\leq j} v_2) \implies v_1 \approx_{k-j} v_2 : \mathcal{V}[[S]])
\end{aligned}$$

Figure 2.6: Step-indexed logical relation for type-based equivalence

$v_1 \approx_k v_2 : \mathcal{V}[[S]]$ , when one cannot distinguish them by invoking any method  $m$  of  $U$ . More precisely, to ensure safety, we first demand that both values are well-typed at  $T$  with the simple type system. Then, for each method  $m \in U$  and every  $j < k$ , the invocations of  $m$  on  $v_1$  and  $v_2$  with related arguments at the argument type  $S'$  of  $m$  must be equivalent computations at the return type  $S''$  for  $j$  steps, as defined below. Finally, note that the definition also requires that  $v_1$  and  $v_2$  are related self objects, for  $j < k$  steps; this is necessary for the relation to be well-founded. (Observe that two simply well-typed objects are vacuously equivalent for zero steps.)

Two expressions  $e_1$  and  $e_2$  are equivalent at security type  $S \triangleq T \triangleleft U$  for  $k$  steps, noted  $e_1 \approx_k e_2 : \mathcal{C}[[S]]$ , if they are both (simply) well-typed at  $T$  and, provided that they both reduce to values in *at most*  $j < k$  steps (noted  $e \mapsto^{\leq j} v$ ), then both values are equivalent at type  $S$  for the remaining  $k - j$  steps. Note that this definition is termination insensitive: if one expression does not terminate in less than  $k$  steps, then both expressions are deemed equivalent.

## 2.5.2 Defining type-based relaxed noninterference

The type-based approach to declassification policies allows us to formulate the corresponding relaxed noninterference property as a *modular* reasoning principle, similarly to the common formulation of noninterference in languages without declassification [67], thereby avoiding the global and external formulation of the transformation approach [31].

Standard noninterference is usually stated as a modular reasoning principle on open terms [67]: given a well-typed open term, which depends on some secret variables, closing the term with private inputs yields equivalent programs when observed by a low-confidentiality observer. This statement can be generalized using the notion of *value substitutions*, *i.e.* partial maps from variables to values: given an open term that typechecks in a given environment  $\Gamma$ , applying two *related* substitutions yields equivalent computations. Applying a substitution, noted  $\gamma(e)$ , substitutes the free variables of  $e$  with their values in  $\gamma$ .

**Definition 8** (Satisfactory substitution). *A substitution  $\gamma$  satisfies type environment  $\Gamma$ , noted  $\gamma \models \Gamma$ , iff  $\text{dom}(\gamma) = \text{dom}(\Gamma) \wedge \forall x \in \text{dom}(\Gamma). \vdash_1 \gamma(x) : T$  where  $\Gamma(x) \triangleq T \triangleleft U$*

**Definition 9** (Related substitutions). *Two substitutions  $\gamma_1$  and  $\gamma_2$  are equivalent for  $k$  steps*

with respect to a type environment  $\Gamma$ , noted  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$ , if  $\gamma_i \models \Gamma$  and

$$\forall x \in \text{dom}(\Gamma). \gamma_1(x) \approx_k \gamma_2(x) : \mathcal{V}[\Gamma(x)]$$

The statement of type-based relaxed noninterference is a direct generalization of standard noninterference: an open term  $e$ , simply well-typed in environment  $\Gamma$ , satisfies type-based relaxed noninterference at security type  $S$ , noted  $\text{TRNI}(\Gamma, e, S)$ , if two executions of  $e$  with related substitutions with respect to  $\Gamma$  produce equivalent computational expressions at type  $S$ , for any number of steps.

**Definition 10** (Type-based relaxed noninterference).

$$\text{TRNI}(\Gamma, e, S) \iff S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \wedge \\ \forall k \geq 0. \forall \gamma_1, \gamma_2. \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma] \implies \gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[S]$$

This definition captures the semantic characterization of TRNI-secure expressions, independently of any enforcement mechanism (recall that, in Figure 2.6, the public facets of security types only play the role of *specifications* of declassification policies). The  $\text{Ob}_{\text{SEC}}$  type system is a sound, conservative enforcement mechanism for TRNI.

### 2.5.3 Security type soundness

To establish that well-typed  $\text{Ob}_{\text{SEC}}$  programs satisfy TRNI, we first introduce a general notion of type-based equivalence between open expressions. Two open expressions, well-typed under a type environment  $\Gamma$ , are equivalent at a security type  $S \triangleq T \triangleleft U$ , if both expressions have simple type  $T$ , and given two related value substitutions for  $\Gamma$ , closing each expression with a satisfactory substitution yields equivalent expressions at type  $S$ .

**Definition 11** (Equivalence of open terms).

$$\Gamma \vdash e_1 \approx e_2 : S \iff S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e_i : T \wedge \\ \forall k \geq 0. \forall \gamma_1, \gamma_2. \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma] \implies \gamma_1(e_1) \approx_k \gamma_2(e_2) : \mathcal{C}[S]$$

As is clear from the definitions, if a term is equivalent to itself at type  $S$ , then it satisfies TRNI at  $S$ .

**Lemma 7** (Self-equivalence).  $\Gamma \vdash e \approx e : S \implies \text{TRNI}(\Gamma, e, S)$

Type soundness of  $\text{Ob}_{\text{SEC}}$  follows from the fact that the  $\text{Ob}_{\text{SEC}}$  type system enforces such a self-equivalence.

**Lemma 8** (Fundamental property).  $\Gamma \vdash e : S \implies \Gamma \vdash e \approx e : S$

*Proof.* The proof is by induction on the typing derivation of  $e$ . The (TVar) case follows directly from Definition 9 and the (TSub) case follows from a subtyping lemma: if  $e_1 \approx_k e_2 : \mathcal{C}[S]$  and  $\vdash S <: S'$  then  $e_1 \approx_k e_2 : \mathcal{C}[S']$ . The (TObj) case applies the induction hypothesis (IH) on method bodies. To use the IH results, we need to show that the value



substitutions that result from extending the current substitutions with both self and actual arguments are also related. This step requires auxiliary lemmas of monotonicity of the logical relations regarding smaller indexes. The (TmD) case follows from applying the IH over both subexpressions, selecting adequate indexes. The (TmH) case is simpler because there is no method to invoke in the declassification type  $\top$ .  $\square$

Finally, type soundness for  $\text{Ob}_{\text{SEC}}$  follows directly from Lemmas 7 and 8.

**Theorem 9** (Security type soundness).  $\Gamma \vdash e : S \implies \text{TRNI}(\Gamma, e, S)$

**Illustration.** We now illustrate the relation between the security typing and the definition of TRNI. In the examples we use some standard constructs like conditionals, not included in  $\text{Ob}_{\text{SEC}}$ , but easily encodable.

As introduced in Section 2.2, the property  $\text{TRNI}(\Gamma, e, T \triangleleft U)$  can be intuitively understood as: the initial knowledge of a public observer in  $\Gamma$  (*i.e.* the declassification policies) implies the final knowledge (*i.e.* the resulting declassification type  $U$ ) that the observer has at hand to distinguish the results of two arbitrary executions of the *secure* program  $e$  of simple type  $T$ .

Let us recall the type  $\text{StringLen} \triangleq [\text{length} : \text{Unit}_{\text{L}} \rightarrow \text{Int}_{\text{L}}]$  from the end of Section 2.2. Consider the open term  $e \triangleq x.\text{length}$  under the type environment  $\Gamma \triangleq x : \text{String} \triangleleft \text{StringLen}$ . The judgment  $\Gamma \vdash e : \text{Int}_{\text{L}}$  ensures that  $\text{TRNI}(\Gamma, e, \text{Int}_{\text{L}})$  holds. It says that executing  $e$ , with two different strings  $v_1$  and  $v_2$  of *the same length* is secure because the observer *does not learn anything new* by exploiting the knowledge of distinguishing the resulting integers with any method of  $\text{Int}$ . In fact, if we use the definition of TRNI, for any equivalent substitutions  $\gamma_1$  and  $\gamma_2$  such that  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$ , such as  $\gamma_i \triangleq x \mapsto v_i$ , we need to show  $\gamma_1(x).\text{length}() \approx_k \gamma_2(x).\text{length}() : \mathcal{C}[\text{Int}_{\text{L}}]$ . It is easy to see that this result follows from the assumption that  $v_1$  and  $v_2$  have the same length (*i.e.* are equivalent at  $\text{String} \triangleleft \text{StringLen}$ ).

We have a different situation if we consider  $e' \triangleq \text{if}(x.\text{eq}(\text{"mary"})) \text{ return } 1 \text{ else } 2$ , with the same type environment  $\Gamma$ . We cannot prove that  $\text{TRNI}(\Gamma, e', \text{Int}_{\text{L}})$  holds, meaning this program is *not* secure at type  $\text{Int}_{\text{L}}$ . Indeed, take  $\gamma_1 \triangleq x \mapsto \text{"mary"}$  and  $\gamma_2 \triangleq x \mapsto \text{"john"}$ . Because both strings have the same length, we have  $\text{"mary"} \approx_k \text{"john"} : \mathcal{V}[\text{String} \triangleleft \text{StringLen}]$ , so the two substitutions are equivalent. However, we cannot show that  $\gamma_1(e') \approx_k \gamma_2(e') : \mathcal{C}[\text{Int}_{\text{L}}]$ , because this requires to show that  $1 \approx_k 2 : \mathcal{V}[\text{Int}_{\text{L}}]$ , which is obviously false.

The type system of  $\text{Ob}_{\text{SEC}}$  indeed rejects the judgment  $\Gamma \vdash e : \text{Int}_{\text{L}}$ . It does accept the judgment  $\Gamma \vdash e : \text{Int}_{\text{H}}$ , meaning that  $e'$  is secure at type  $\text{Int}_{\text{H}}$ . This is correct because then the public observer has no ability to compare the resulting values of  $e'$ . Note in fact that any simply well-typed expression of type  $T$  is secure at type  $T_{\text{H}}$ . Such expressions are opaque to a public observer, but are observable by a privileged observer.

**Principles of declassification.** Our approach to type-based declassification satisfies the declassification principles stated by Sabelfeld and Sands [52].<sup>5</sup> We now briefly introduce

<sup>5</sup>Sabelfeld and Sands mention a fourth principle, *non-occlusion*, which addresses the interaction between declassification and covert channels, such as heap assignments, exceptions or termination behavior.  $\text{Ob}_{\text{SEC}}$

each principle and informally argue why it is respected.

- *Conservativity*—*i.e.* “*Security for programs with no declassification is equivalent to non-interference*”. It is easy to see that if a program satisfies  $\text{TRNI}(\Gamma, e, T_L)$ , for some  $T$ , and all security types in both  $\Gamma$  and  $e$  are either highly confidential ( $T_H$ ) or not confidential at all ( $T_L$ ), then the definition of TRNI coincides exactly with the definition of pure noninterference [67]. Therefore type-based relaxed noninterference is a generalization of pure noninterference.
- *Monotonicity of Release*—*i.e.* “*Adding further declassifications to a secure program cannot render it insecure*”. This lemma follows from subtyping naturally. Recall that in our approach, in the judgment  $\text{TRNI}(\Gamma, e, S)$ , declassification policies come from types ascribed in both  $\Gamma$  and  $e$ . “Adding further declassification” in the inputs means in our context replacing security types in  $\Gamma$  with subtypes, more precisely, where the public facets are subtypes of the original types. The security typing judgment also holds in this scenario of additional declassification in the inputs. Similarly, adding declassification in the expression  $e$  means specializing the public facets of types in object type declarations. Again, this does not affect the semantic TRNI judgment. Note, however, that if argument types are specialized, the program might not be typable anymore with the security type system, as such a change breaks the contravariance of subtyping for argument method types.
- *Semantic Consistency*—*i.e.* “*The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms*.”. The principle says that it is possible to replace an expression that does not use declassification with another semantically-equivalent expression, without affecting security. As observed by Sabelfeld and Sands, the approach to declassification policies of Li and Zdancewic [31] violates this principle, because they rely on a restricted, mostly-syntactic form of program equivalence to decide label ordering. Therefore, many semantically-equivalent programs are not deemed equivalent, hence affecting their (in)security. In contrast, our notion of type-based equivalence (Figure 2.6) is semantic, not syntactic.

**Limitations of security typing.** The  $\text{Ob}_{\text{SEC}}$  type system is a *static* enforcement mechanism for type-based relaxed noninterference. As such, it is inherently conservative. This has two implications regarding Theorem 9.

First, the type system can reject some programs that are in fact secure. For example, consider the following definitions:

$$\begin{aligned} T &\triangleq \mathbf{Obj}(\alpha). [n : \text{String}_L \rightarrow \text{String}_L] \\ T' &\triangleq \mathbf{Obj}(\alpha). [m : \text{String}_H \rightarrow \text{String}_H] \\ v &\triangleq [z : T_L \Rightarrow n(x) \text{"hello"}] \\ v' &\triangleq [z : T'_L \Rightarrow m(x) v.n(x)] \end{aligned}$$

Here,  $v'$  is not well-typed using the security type system, because of the call  $v.n(x)$  ( $\vdash \text{String}_H \not\prec: \text{String}_L$ ). However, we can show that  $v'$  does satisfy  $\text{TRNI}(\cdot, v', T'_L)$ , because a public observer always obtains the same result (*i.e.* “hello”) for any two secrets passed to

---

has neither mutation nor control operators, and termination is not considered a covert channel because we only deal with termination-insensitive noninterference.

method  $m$ ; the program is not leaking any information.

Second, the type system can assign the security type  $T \triangleleft \top$  to an expression, despite the fact that  $\top$  is not the tighter secure type for TRNI to hold. For instance, let us assume that  $\text{Int}$  has built-in methods  $\text{mod2}$  and  $\text{mod4}$  with the standard mathematical meaning, and we define the type  $\text{IntMod4} \triangleq [\text{mod4} : \text{Unit}_L \rightarrow \text{Int}_L]$ . Consider  $\Gamma \triangleq v : \text{Int} \triangleleft \text{IntMod4}$  and  $e \triangleq v.\text{mod2}()$ . The type system admits  $\Gamma \vdash e : \text{Int}_H$ , which implies  $\text{TRNI}(\Gamma, e, \text{Int}_H)$ , but it does not admit  $\Gamma \vdash e : \text{Int}_L$ ; despite the fact that  $\text{TRNI}(\Gamma, e, \text{Int}_L)$  also holds—because if  $a$  and  $b$  are equivalent modulo 4, then they are also equivalent modulo 2.

### 2.5.4 A note about casts.

In Section 2.2 we alluded to the challenge of integrating explicit downcasts in a language that adopts type-based declassification policies. Casts can be soundly incorporated in such a language provided that we only allow casting values from a security type to another one that has the *same declassification type*, *i.e.* casts cannot affect the declassification policy. Therefore the interesting typing rule for a cast expression  $\langle T \rangle e$  is:

$$(\text{TCast}) \frac{\Gamma \vdash e : T' \triangleleft U \quad \vdash T <: T'}{\Gamma \vdash \langle T \rangle e : T \triangleleft U}$$

As usual in security languages with casts, cast errors are seen as a non-termination channel, hence not affecting the security definitions.

## 2.6 Related work

Information flow security in general, and declassification in particular, are very active areas of research. We now discuss the most salient proposals related to the work developed in this chapter.

**Secure information flow and type abstraction.** Our work shows a connection between type abstraction and declassification policies for secure information flow. Previous works also attempt to connect type abstraction and secure information flow.

Tse and Zdancewic [59] encode the Dependency Core Calculus (DCC) [2] in System F. The correctness theorem of their translation aims at showing that the parametricity theorem of System F implies the noninterference property. Unfortunately, Shikuma and Igarashi [53] identify a mistake in the proof of their main result; they also gave a noninterference-preserving translation for a version of DCC to the simply-typed lambda calculus. However, this translation left open the connection between parametricity and noninterference, initially aimed by Tse and Zdancewic.

Recently, Bowman and Ahmed [14] provide a translation from DCC to System  $F_\omega$ , successfully demonstrating that noninterference can be encoded via parametricity. Our work

generalizes this by showing that type abstraction implies *relaxed* noninterference. Information flow analyses have been proposed to generalize parametricity in the presence of runtime type analysis [65]. Using security labels, a programmer can specify data structures that should remain confidential in order to hide implementation details and rely on type abstraction for abstract datatypes.

An interesting research direction is to investigate whether our proposal of solving information flow problems via type abstraction, here through subtyping, can be used to generalize parametricity as proposed by Washburn and Weirich [65].

**Declassification.** As extensively discussed, our policies and security property are based on the work of Li and Zdancewic [31], which proposes two kinds of downgrading policies (which we call here declassification policies, since they only relate to confidentiality): local and global policies. The declassification policies in this chapter directly correspond to local policies, as discussed in the introduction. Global policies refer to declassifications that involve more than one secret simultaneously. It is unclear if and how global policies can be supported using our type-driven approach; further exploration is necessary to settle this issue. Additionally, in contrast to the definition of relaxed noninterference of Li and Zdancewic, our definition is independent from the security enforcement mechanism. This allows us to distinguish programs that are not secure from programs that are not typable due to a necessarily conservative static security mechanism (see Section 2.5). Also, our definition of relaxed noninterference is formulated as a generalization of the semantic characterization of pure noninterference [67], providing a modular reasoning principle, as opposed to the global translation approach of Li and Zdancewic.

In the following, we focus on the closest related work on declassification policies starting from 2005 and refer the reader to [52] for a survey prior to 2005.

**Typing declassification in object-oriented languages.** Since 2005, several works have studied static enforcement of declassification in object-oriented languages [9, 29, 11, 17].

Banerjee and Naumann [9] study the interaction between security typing for noninterference and access control in a Java-like language. Security levels are not fixed but rather depend on access permissions. In contrast to our work, security levels are independent of method signatures or types and thus their typing does not relate to type abstraction.

Hicks et al. [29] propose trusted declassification for an object calculus. Principals in a program have access to specified trusted *declassifier* functions or methods. Typeable programs are secure for noninterference modulo trusted methods, in the same spirit as typing of noninterference of programs with cryptographic functions [26]. In contrast to relaxed noninterference, trusted declassification does not consider declassifiers as part of security levels. Instead, declassifiers need to be associated by a policy to different principals (security labels in our setting) in the lattice.

Barthe et al. [11] propose a modular method to extend type systems and proofs for noninterference to declassification and discuss how the method extends to object-oriented languages. The declassification property called delimited non-disclosure [32] does not support

fine-grained specification of how to declassify a given secret, as supported by relaxed noninterference.

Tse and Zdancewic [60] propose a security-typed language for robust declassification: declassification cannot be triggered unless there is a digital certificate to assert the proper authority. Their language inherits many features from System  $F_{<}$  and uses monadic labels as in DCC [2]. The monadic style allows them to integrate computational effects, which we do not support. In contrast to our work, security labels are based on the Decentralized Label Model (DLM) [37], and are not semantically unified with the standard safety types of the language.

Chong and Myers [17] propose hybrid typing to enforce declassification and erasure policies and implement it in Jif [36]. Their language features a special declassification function that takes as input the expression and levels to declassify and also the conditions under which declassification can occur. Security policies are specified by means of security levels and conditions to downgrade them. This resembles our declassification policies, which specify the methods that can be applied in order to (partially) declassify; at a more abstract level, the interface types of the public facet can be seen as “conditions” for declassifying. The type system developed by Chong and Myers statically checks that conditions in declassification commands comply with the specified security policies. A dynamic mechanism enforces this, or returns a dummy value (instead of the declassified value) at runtime. In contrast to our work, their type system significantly departs from standard typing rules, and dynamic checks are required for guaranteeing security.

**Extensional specification of declassification policies.** The language Air [57] expresses declassification policies as security automata. The policies, seen as automata, transition when a release obligation is satisfied. When an accepting state is reached, declassification is performed. These policies resemble relaxed noninterference and our own declassification policies but they require very specific typing rules.

Banerjee et al. [10] study declassification properties using ideas from epistemic logic can capture global policies (as in the original work of relaxed noninterference) with an extensional property. Their policies are not expressed using standard types as in our work.

The language Paralocks [15] supports declassification policies represented as Horn clauses, whose antecedents are conditions that should be satisfied for a flow to occur. There is a natural order between declassification policies that correspond to the logical entailment when viewing policies as Horn clauses. The policies together with the logical entailment order define a lattice that supports an extensional specification of secrets and their intended declassification, as in our work. However, declassification policies in Paralocks are not specified by using the standard types of the language, and thus their enforcement requires specific typing rules.

**Multiple facets for dynamic enforcement of declassification** Austin and Flanagan introduce Multiple Facets [8] as a dynamic mechanism to enforce secure information flow. The main idea behind multiple facets is to execute a program using multiple values, one value or facet for each security level of observation. A value considered confidential will only

flow to a public facet by facet declassification, based on robust declassification [69]. Robust declassification requires the decision to declassify to be trusted according to integrity labels used to model trust. In our work, we do not consider integrity labels or robust declassification. However, the idea of multiple facets (having a facet for each observer at a given security level) is similar to our faceted types. Just as Austin and Flanagan can run a program for different facets simultaneously, we type check programs providing different views to observers with different security clearances.

Multiple facets are also inspired by Secure Multi Execution (SME) [25, 12], a dynamic mechanism that roughly executes a program multiple times in order to enforce noninterference. Hence, observers with different security clearances will potentially observe different values during the execution of a program. Several works have studied declassification in the context of SME [46, 61, 13]. Rafnsson and Sabelfeld propose declassification in SME based on the gradual release property [6]. This property differs from the property we consider in our work in that it is not possible to extensionally specify what is being released or declassified. The latest works on SME declassification [61, 13] generalize security levels as declassifier functions, resembling declassification policies of both Li and Zdancewic and ours. Since SME is a dynamic enforcement mechanism, these declassification policies are not used for relating declassification and type abstraction.

## 2.7 Conclusion

One of the open challenges in the area of information flow security is integrating information flow mechanisms with existing infrastructures [68]. The work of this chapter partially addresses this challenge by showing a connection between type abstraction, more precisely that induced by the the subtyping relation in an object-oriented language, and the order relation in security lattices. In particular, we exploit an intuitive connection between object interfaces and declassification policies: an object interface already gives a way to control the exposed behavior of an object. These connections imply that standard type systems can be used as a direct means to enforce secure information flow, when types express security policies. It is left to explore how this connection scales in practice, but we expect the economy of concepts to be an important asset for adoption.

# Chapter 3

## Existential relaxed noninterference

The object-oriented labels-as-types approach developed in Chapter 2 suffers from limitations due to its receiver-centric paradigm. Here, we consider an alternative approach to labels-as-types, applicable in non-object-oriented languages, which allows us to express advanced declassification policies, such as extrinsic policies, based on a different form of type abstraction: existential types. An existential type exposes abstract types and operations on these; we leverage this abstraction mechanism to express secrets that can be declassified using the provided operations. We formalize the approach in a core functional calculus  $\lambda_{\text{SEC}}^{\exists}$  with existential types, define existential relaxed noninterference, and prove that well-typed  $\lambda_{\text{SEC}}^{\exists}$  programs satisfy this form of type-based relaxed noninterference.

This chapter is based on the work of Cruz and Tanter [20]

### 3.1 Introduction

Developing type-based relaxed noninterference in an object-oriented setting, exploiting subtyping as the type abstraction mechanism, imposes some restrictions on the declassification policies that can be expressed. In particular, because security types are of the form  $T \triangleleft U$  where the declassification type  $U$  is a supertype of the safety type  $T$ —a necessary constraint to ensure type safety—means that one cannot declassify properties that are *extrinsic* to (*i.e.* computed externally from) the secret value. For instance, because a typical `String` interface does not feature an `encrypt` method, it is not possible to express the declassification policy that “the encrypted representation of the password is public”.

In this chapter, we explore an alternative approach to labels-as-types and relaxed noninterference, exploiting another well-known type abstraction mechanism: existential types. An existential type  $\exists X.T$  provides an abstract type  $X$  and an interface  $T$  to operate with values of the abstract type  $X$ . Then values of the abstract type  $X$  are akin to secrets that can be declassified using the operations described by  $T$ . For instance, the existential type  $\exists X.[\text{get} : X, \text{length} : X \rightarrow \text{Int}]$  makes it possible to obtain a (secret) value of type  $X$  with `get`, that can only be “declassified” with the `length` function to obtain a (public) integer.

Because existential types are the essence of abstraction mechanisms like abstract data

types and modules [34], this chapter shows how the labels-as-types approach can be applied in non-object-oriented languages. Once again, the only required extension is the notion of faceted types, which are necessary to capture the natural separation between *privileged observers* (allowed to observe secret results) and *public observers* (*i.e.* the attacker, which can only observe public values)<sup>1</sup>. Additionally, the existential approach is more expressive than the object-oriented one in that extrinsic declassification policies can naturally be encoded with existential types.

The contributions of this chapter are:

- We explore an alternative type abstraction mechanism to realize the labels-as-types approach to expressive declassification, retaining the practical aspect of using an existing language mechanism (here, existential types), while supporting more expressive declassification policies (Section 3.2).
- We define a new version of type-based relaxed noninterference, called existential relaxed noninterference, which accounts for extrinsic declassification using existential types (Section 3.3).
- We capture the essence of the use of existential types for relaxed noninterference in a core functional language  $\lambda_{\text{SEC}}^{\exists}$  (Section 3.4), and prove that its type system soundly enforces existential relaxed noninterference (Section 3.5).

Section 3.6 explains how the formal definitions apply by revisiting an example from Section 3.3. Section 3.7 discusses related work and Section 3.8 presents conclusions for the results of this chapter.

## 3.2 Overview

We now explain how to use the type abstraction mechanism of existential types to denote secrets that can be selectively declassified. First, we give a quick overview of existential types, with their introduction and elimination forms. Next, we develop the intuitive connection between the type abstraction of standard existential types and security typing. Then, we show that to support computing with secrets, which is natural for information-flow control languages, we need to introduce faceted types.

### 3.2.1 Existential types

An existential type  $\exists X.T$  is a pair of an (abstract) type variable  $X$  and a type  $T$  where  $X$  is bound; typically  $T$  provides operations to create, transform and observe values of the abstract type  $X$  [34].

For instance, the type `AccountStore` below models a simplified user repository. It provides the password of a user at type  $X$  with the function `userPass` and a function `verifyPass` to check (observe) whether an arbitrary string value is equal to the password.

$$\text{AccountStore} \triangleq \exists X.[ \text{userPass} : \text{String} \rightarrow X \\ \text{verifyPass} : \text{String} \rightarrow X \rightarrow \text{Bool} ]$$


---

<sup>1</sup>To account for  $n > 2$  observation levels, faceted types can be extended to have  $n$  facets.



Values of an existential type  $\exists X.T$  take the form of a *package* that packs together the *representation* type for the abstract type  $X$  with an implementation  $v$  of the operations provided by  $T$ . One can think of packages as *modules* with signatures.

For instance, the package  $p \triangleq \text{pack}(\text{String}, v)$  as `AccountStore` is a value of type `AccountStore`, where `String` is the representation type and  $v$ , defined below, is a record implementing functions `userPass` and `verifyPass`:

$$v \triangleq [\text{userPass} = \lambda x : \text{String}. \text{userPassFromDb}(x) \\ \text{verifyPass} = \lambda x : \text{String}. \lambda y : \text{String}. \text{equal}(x, y)]$$

Note that the implementation,  $v$ , directly uses the representation type `String`, *e.g.* `userPass` has type `String`  $\rightarrow$  `String` and is implemented using a primitive function `userPassFromDb` : `String`  $\rightarrow$  `String` to retrieve the user password from a database. Likewise, the implementation of `verifyPass` uses equality between its arguments of type `String`.

To use an existential type, we have to *open* the package (*i.e.* import the module) to get access to the implementation  $v$ , along with the abstract type that hides the actual representation type. The expression `open(X, x) = p` in  $e'$  opens the package  $p$  above, exposing the representation type abstractly as a type variable  $X$ , and the implementation as term variable  $x$ , within the scope of the body  $e'$ . Crucially, the expression  $e'$  has no access to the representation type `String`, therefore nothing can be done with a value of type  $X$ , beyond using it with the operations provided by `AccountStore`.

### 3.2.2 Type-based declassification policies with existential types

We can establish an analogy between existential types and selective declassification of secrets: an existential type  $\exists X.T$  exposes operations to obtain secret values, at the abstract type  $X$ , and the operations of  $T$  can be used to declassify these secrets.

For instance, `AccountStore` provides a secret string password with the function `userPass`, and the function `verifyPass` expresses the declassification policy: “the comparison of a secret password with a public string can be made public”. With this point of view, concrete types such as `Bool` and `String` represent public values. A fully-secret value, *i.e.* a secret that is not declassified, can be modeled by an existential type without any observation function for the abstract type.

We can use the declassification policy modeled with `AccountStore` to implement a valid well-typed login functionality. The `login` function below is defined in a scope where the package  $p$  of type `AccountStore` is opened, providing the type name  $X$  for the abstract type and the variable `store` for the package implementation.

```
open(X, store) = p in
  ...
  String login(String guess, String username){
    if(store.verifyPass(guess, store.userPass(username)))
      ...
  }
```

The `login` function first obtains the user secret password of type  $X$  with `store.userPass(username)`, and then passes the secret password (of type  $X$ ) to the function `verifyPass` with the `guess` public password to obtain the public boolean result. The above code makes a valid use of `AccountStore` and therefore is well-typed.

The type abstraction provided by `AccountStore` avoids leaking information accidentally. For instance, directly returning the secret password of type  $X$  is a type error, even though internally it is a string. Likewise, the expression `length(store.userPass(username))` is ill-typed.

**Progressive declassification.** The analogy of existential types as a mechanism to express declassification holds when one considers *progressive declassification* (Section 2.2 and [31]), which refers to the possibility of only declassifying information after a *sequence* of operations is performed. With existential types, we can express progressive declassification by constraining the creation of secrets based on other secrets.

Consider the following refinement of `AccountStore`, which supports the declassification policy “whether an *authenticated* user’s salary is above \$100,000”:

$$\text{AccountStore} \triangleq \exists X, Y, Z. [ \begin{array}{l} \text{userPass} : \text{String} \rightarrow X \\ \text{verifyPass} : \text{String} \rightarrow X \rightarrow \text{Option } [Y] \\ \text{userSalary} : Y \rightarrow Z \\ \text{isSixDigit} : Z \rightarrow \text{Bool} \end{array} ]$$

`AccountStore` provides extra abstract types  $Y$  and  $Z$ , denoting an authentication token (for a specific user) and a user salary, respectively. The type signatures enforce that, to obtain the user salary, the user must be authenticated: a value of type  $Y$  is needed to apply `userSalary`. Such a value can be obtained only after successful authentication: `verifyPass` now returns an `Option [Y]` value, instead of a `Bool` value. Note that the salary itself is secret, since it has the abstract type  $Z$ . Finally, `isSixDigit` function reveals whether a salary is above \$100,000 by returning a public boolean result.

Observe how the use of abstract type variables allows the existential type to enforce sequencing among operations. Also, we can provide more declassification policies for a user salary  $Z$ , and can use the authentication token  $Y$  with more operations. An existential type is therefore an expressive means to capture rich declassification policies, including sequencing and alternation.

### 3.2.3 Computing with secrets

As we have seen, with standard existential types, values of an abstract type  $X$  must be eliminated with operations provided by the existential type. While so far the analogy between type abstraction with existential types and expressive declassification holds nicely, there are some obstacles.

First, with standard existential types, it is simply forbidden to compute with secrets. For instance, applying the function `length: String → Int` with a (secret) value of type  $X$  is a type error. However, information-flow type systems are more flexible: they support computing

$$\text{AccountStore} \triangleq \exists X, Y, Z. [ \text{userPass} : \text{String}_L \rightarrow \text{String}@X \\ \text{verifyPass} : \text{String}_L \rightarrow \text{String}@X \rightarrow \text{Option} [Y_L]_L \\ \text{userSalary} : Y_L \rightarrow \text{Int}@Z \\ \text{isSixDigit} : \text{Int}@Z \rightarrow \text{Bool}_L ]$$

Figure 3.1: Account store with faceted types

with secret values, as long as the computation itself is henceforth considered secret, *e.g.* the value it produces is itself secret [67]. Allowing secret computations is useful for privileged observers, which are authorized to see secret values.

We introduced *faceted types* in Chapter 2 to support this “dual mode” of information-flow type systems in the labels-as-types approach. While that chapter is based on objects and subtyping, here we develop the notion of *existential faceted types*: faceted types of the form  $T@U$ , where  $T$  indicates the safety type used for the implementation and  $U$  the declassification type used for confidentiality.

Figure 3.1 shows `AccountStore` with existential faceted types. Given a public string ( $T_L$  denotes  $T@T$ ), `userPass` returns a value that is a string *for the privileged observer*, and a secret of type  $X$  *for the public observer* (*i.e.* the attacker).

When computing with a value of type  $\text{String}@X$ , there are now two options: either we use a function that expects a  $\text{String}@X$  as argument, such as `verifyPass`, or we use a function that goes beyond declassification, such as `length`, and should therefore produce a *fully* private result. What type should such private results have? In order to avoid having to introduce a fresh type variable, we assume a fixed (unusable) type  $\top$ , and write  $\text{Int}_H$  to denote  $\text{Int}@\top$ .

This supports computing with secrets as follows:

```
StringH login(StringL guess, StringL username){
  if(length(store.userPass(username)) == length(guess))...;
}
```

Instead of being ill-typed, `length(store.userPass(username))` is well-typed at type  $\text{Int}_H$ , so the function `login` can return a private result, *e.g.* a private string at type  $\text{String}_H$ .

### 3.2.4 Public data as (declassifiable) secret

Information-flow type systems allow any value to be considered private. With existential faceted types, this feature is captured by a subtyping relation such that for any  $T$ ,  $T@T <: T@\top$ , and for any  $X$ ,  $T@X <: T@\top$ . Value flows that are justified by subtyping are safe from a confidentiality point of view. In particular, if a (declassifiable) value of type  $\text{String}@X$  is passed at type  $\text{String}_H$ , it is henceforth fully private, disallowing any further declassification.

Additionally, in the presence of declassifiable secrets, of type  $T@X$ , one would also expect public values to be “upgraded” to declassifiable secrets. This requires the security subtyping relation to admit that, for any type  $T$  and type variable  $X$ , we have  $T_L <: T@X$ .

Note that admitting such flows means that type variables in a declassification type position are more permissive than when they occur in a safety type position. For instance, `isSixDigit` can be applied to any public integer (of type `IntL`), and not only to ones returned by `userSalary`. In contrast, `userSalary` can only be applied to a value opaquely obtained as a result of `verifyPass`. In effect, the representation of authentication tokens is still kept abstract, at type `YL` (*i.e.* `Y@Y`). This prevents clients from actually knowing how these tokens are implemented, preserving the benefits of standard existential types. Conversely, the salary and password expose their representation types (`Int` and `String` respectively), thereby enabling secret computation by clients.

### 3.3 Relaxed noninterference with existential types

Existential faceted types support a novel notion of type-based relaxed noninterference called *existential relaxed noninterference* (ERNI) that defines if a program with existential faceted types is secure. ERNI is based on type-based *equivalences* between values at existential faceted types. We formally define the notions of type-based equivalence and ERNI in Section 3.5, but here we provide an intuition for this security criterion and the associated reasoning. Let us first consider simple types, before looking at existential types.

**Type-based relaxed noninterference.** Two integers are equivalent at type `Int@Int = IntL` if they are syntactically equal, meaning that a public observer can distinguish between two integers at type `IntL`. We can characterize the meaning of the faceted type `IntL` with the partial equivalence relation  $Eq_{Int} = \{(n, n) \in Int \times Int\}$ . Using this, two integers  $v_1$  and  $v_2$  are equivalent at type `IntL` if they are in the relation  $Eq_{Int}$ —meaning they are syntactically equal.

Dually, the type `Int@T = IntH` characterizes integer values that are indistinguishable for a public observer, therefore *any* two integers are equivalent at type `IntH`. Consequently, the meaning of the faceted type `IntH` is the total relation  $All_{Int} = Int \times Int$  that relates any two integers  $v_1$  and  $v_2$ .

With these base type-based equivalences, one can express the security property of functions, open terms, and programs with inputs as follows: a program  $p$  satisfies ERNI at an observation type  $S_{out}$  if, given two input values that are equivalent at type  $S_{in}$ , the executions of  $p$  with each value produce results that are equivalent at type  $S_{out}$ . This modular reasoning principle is akin to standard noninterference [67] and type-based relaxed noninterference (Chapter 2).

Intuitively,  $S_{in}$  models the *initial knowledge* of the public observer about the (potentially-secret) input, and  $S_{out}$  denotes the *final knowledge* that the public observer has to distinguish results of the executions of the program  $p$ . The program  $p$  is secure if, given inputs from the same equivalence class of  $S_{in}$ , it produces results in the same equivalence class of  $S_{out}$ . Consider the program  $e = \text{length}(x)$  where  $x$  has type `StringH`. The program  $e$  does not satisfy ERNI at type `IntL`, because given two strings “a” and “aa” that are equivalent at `StringH`, *i.e.*  $(\text{“a”}, \text{“aa”}) \in All_{String}$ , we obtain the results 1 and 2, which are not equivalent at type `IntL`, *i.e.*  $(1, 2) \notin Eq_{Int}$ . However,  $e$  is secure at type `IntH`.

**Relaxed noninterference and existentials.** When we introduce faceted types with type variables such as  $\text{Int}@X$ , we need to answer: what values are equivalent at type  $\text{Int}@X$ ? Without stepping into technical details yet, let us say that the meaning of a type  $\text{Int}@X$  is an *arbitrary* partial equivalence relation  $R_X \subseteq \text{Int} \times \text{Int}$ , and two values  $v_1$  and  $v_2$  are equivalent at  $\text{Int}@X$  if they are in  $R_X$ . Because  $X$  is an existentially-quantified variable, inside the package implementation that exports the type variable  $X$ ,  $R_X$  is known, but outside the package, *i.e.* for clients of a type  $\text{Int}@X$ ,  $R_X$  is completely abstract: a public observer that opens a package exporting the type variable  $X$  does not know anything about values of type  $\text{Int}@X$ .

For instance, consider again the program  $e = \text{length}(x)$  but assume that  $x$  now has type  $\text{String}@Y$ . Does  $e$  satisfy ERNI at type  $\text{Int}_L$ ? Here, we need to know what is the relation  $R_Y$  that gives meaning to  $\text{String}@Y$ . Instead of picking only one relation  $R_Y$ , ERNI quantifies over *all* possible relations  $R_Y$ . That is, the program  $e$  satisfies ERNI at type  $\text{Int}@\text{Int}$ , if it is secure for all relations  $R_Y \subseteq \text{String} \times \text{String}$ . Then, to show that ERNI at type  $\text{Int}@\text{Int}$  does not hold for  $e$  it suffices to exhibit a specific relation for which ERNI is violated. Take the relation  $R_Y = \{("a", "aa")\}$ , and observe that  $\text{length}("a") \neq \text{length}("aa")$ .

**Illustration.** Finally, we give an intuition of how ERNI accounts for extrinsic declassification policies. We reuse the salary operations from `AccountStore`, simplifying the retrieval of the secret salary. The type `SalaryPolicy` provides a secret `salary` and a function `isSixDigit` to declassify the salary as before.

$$\text{SalaryPolicy} \triangleq \exists Z. [\text{salary} : \text{Int}@Z, \text{isSixDigit} : \text{Int}@Z \rightarrow \text{Bool}_L]$$

Intuitively two values  $v_1$  and  $v_2$  are equivalent at  $\text{SalaryPolicy}_L$  if  $v_1.\text{salary}$  and  $v_2.\text{salary}$  are equivalent at  $\text{Int}@Z$  and  $v_1.\text{isSixDigit}$  and  $v_2.\text{isSixDigit}$  are equivalent at  $\text{Int}@Z \rightarrow \text{Bool}_L$ , *i.e.* the functions `isSixDigit` of  $v_1$  and  $v_2$  produce equivalent results at  $\text{Bool}_L$  when given equivalent arguments at  $\text{Int}@Z$ .

Consider now the program  $e' = x.\text{isSixDigit}(x.\text{salary})$  with input  $x$  of type  $\text{SalaryPolicy}_L$ . Taking into account the above equivalence for `SalaryPolicy`, the program  $e'$  satisfies ERNI at type  $\text{Bool}_L$  because it adheres to the salary policy. Indeed, given *any* two equivalent packages  $v_1$  and  $v_2$ , the expressions  $v_1.\text{isSixDigit}(v_1.\text{salary})$  and  $v_2.\text{isSixDigit}(v_2.\text{salary})$  are *necessarily* equivalent at type  $\text{Bool}_L$ . However, the program  $(x.\text{salary})\%2$  does not satisfy ERNI at  $\text{Int}_L$ , because given equivalent packages  $v_1 \triangleq [\text{salary} = 100001, \dots]$  and  $v_2 \triangleq [\text{salary} = 100002, \dots]$ , it yields 1 for  $v_1$  and 0 for  $v_2$ , and both values are not equivalent at  $\text{Int}_L$ , *i.e.*  $(1, 0) \notin Eq_{\text{Int}}$ .

### 3.4 Formal semantics

We model existential faceted types in  $\lambda_{\text{SEC}}^{\exists}$ , which is essentially the simply-typed lambda calculus augmented with the unit type, pair types, sum types, existential types, and faceted types. All the examples presented in Section 3.2 can thus be encoded in  $\lambda_{\text{SEC}}^{\exists}$  using standard techniques. This section covers the syntax, static and dynamic semantics of  $\lambda_{\text{SEC}}^{\exists}$ . The formalization of existential relaxed noninterference and the security type soundness of  $\lambda_{\text{SEC}}^{\exists}$  are presented in Section 3.5.

$e ::=$	$\lambda x : S.e \mid e \ e \mid x \mid \mathbf{b} \mid e \oplus e \mid \langle \rangle \mid \langle e, e \rangle$	(terms)
	$\mid \mathbf{fst} \ e \mid \mathbf{snd} \ e \mid \mathbf{inl} \ e \mid \mathbf{inr} \ e \mid \mathbf{case} \ t \ \mathbf{of} \ \mathbf{inl} \ x.e \mid \mathbf{inr} \ x.e$	
	$\mid \mathbf{pack}(T, e) \ \mathbf{as} \ T \mid \mathbf{open}(X, x) = e \ \mathbf{in} \ e$	
$v ::=$	$\lambda x : S.e \mid \mathbf{b} \mid \langle \rangle \mid \langle v, v \rangle \mid \mathbf{inl} \ v \mid \mathbf{inr} \ v \mid \mathbf{pack}(T, v) \ \mathbf{as} \ T \mid$	(values)
$T, U ::=$	$S \rightarrow S \mid P \mid \mathbf{1} \mid S + S \mid S \times S \mid \exists X.T \mid X \mid \top$	(types)
$P ::=$	$(e.g. \ \mathbf{Int}, \ \mathbf{String})$	(primitive types)
$S ::=$	$T@U$	(security types)
$T_L \triangleq$	$T@T$	$T_H \triangleq T@\top$

Figure 3.2:  $\lambda_{\text{SEC}}^{\exists}$ : Syntax

### 3.4.1 Syntax

Figure 3.2 presents the syntax of  $\lambda_{\text{SEC}}^{\exists}$ . Expressions  $e$  are completely standard [41], including functions, applications, variables, primitive values, binary operations on primitive values, the unit value, pairs with their first and second projections, injections  $\mathbf{inl} \ e$  and  $\mathbf{inr} \ e$  to introduce sum types, as well as a **case** construct to eliminate sums; finally, **pack** and **open** introduce and eliminate existential packages, respectively. Types  $T$  include function types  $S \rightarrow S$ , primitive types  $P$ , the unit type  $\mathbf{1}$ , sum types  $S + S$ , pair types  $S \times S$ , existential types  $\exists X.T$ , type variables  $X$  and the top type  $\top$ . A security type  $S$  is a faceted type  $T@U$  where  $T$  is the safety type and  $U$  is the declassification type.

**Well-formedness of security types.** We now comment on the rules for valid security types, *i.e.* *facet-wise* well-formed types. We have three general form of security types  $T_L$ ,  $T_H$  and  $T@X$ . While there is no constraint to form types  $T_L$  and  $T_H$ , such as  $\mathbf{Int}_L$ ,  $X_L$  and  $\mathbf{Int}_H$ , we need two considerations for types such as  $\mathbf{Int}@X$ .

The first consideration is that *inside* an existential type  $\exists X.T$  the type variable  $X$ , when used as a declassification type, must be uniquely associated to a concrete safety type. For instance, the existential type  $\exists X.(\mathbf{String}@X \rightarrow \mathbf{Int}@X)$  is ill-formed, while  $\exists X.(\mathbf{Int}@X \rightarrow \mathbf{Int}@X)$  and  $\exists X.(X@X \rightarrow \mathbf{Int}@X)$  are well-formed. For such well-formed types, we use the auxiliary function  $\mathbf{sftype}(\exists X.T)$ :

$$\mathbf{sftype}(\exists X.T) = \begin{cases} T' & \mathbf{sftypes}(X, T) \setminus X = \{T'\} \setminus X \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function  $\mathbf{sftype}(\exists X.T)$  uses the auxiliary function  $\mathbf{sftypes}(X, T)$  that collects the safety types associated to  $X$ . The function  $\mathbf{sftype}(\exists X.T)$  is defined in the two following scenarios:

- There is only a safety type  $T'$  associated to  $X$ . For instance,  $\mathbf{sftype}(\exists X.(\mathbf{Int}@X \rightarrow \mathbf{Int}@X)) = \mathbf{Int}$  and  $\mathbf{sftype}(\exists X.(X@X \rightarrow X@X)) = X$ .
- The type variable  $X$  and some type  $T'$  are safety types associated to  $X$ . In that case, the function  $\mathbf{sftype}(\exists X.T)$  returns the concret type  $T'$ . For instance  $\mathbf{sftype}(\exists X.(X@X \rightarrow \mathbf{Int}@X)) = \mathbf{Int}$ .

For ill-formed types, the function  $\mathbf{sftype}(\exists X.T)$  is undefined.

The second consideration is when a client opens a package. The expression  $\text{open}(X, x) = e$  in  $e'$  binds the type variable  $X$  in  $e'$ , therefore the expression  $e'$  can declare security types of the form  $T'@X$ . However, for the declaration of the type  $T'@X$  to be valid, the safety type of the declassification type variable  $X$  must be  $T'$ . For instance, if the safety type of  $X$  is  $\text{Int}$ , the expression  $e'$  cannot declare security types such as  $\text{String}@X$ , otherwise computations over secrets could get stuck. The question is how to determine the safety type  $T'$  of  $X$  in  $e'$ . Crucially, the expression  $e$  necessarily has to be of type  $(\exists X.T)_L$ , therefore we can obtain the safety type for  $X$  with  $\text{sftype}(\exists X.T)$ . To keep track of the safety type for each type variable  $X$ , we use a type variable environment  $\Delta$  that maps type variables to types  $T$  (*i.e.*  $\Delta ::= \bullet \mid \Delta, X : T$ )

With the previous considerations in mind, the rules for well-formed security types are straightforward (Appendix B.1.5). In the rest of the chapter, we use the judgment  $\Delta \models S$  to mean *well-formed* security types  $S$  under type environment  $\Delta$ . A well formed security type  $S$  is both facet-wise well-formed and closed with respect to type variables. We also use  $\Delta \models \Gamma$  to indicate that a type environment is well-formed, *i.e.* all types in  $\Gamma$  are well-formed (Appendix B.1.6). In the following, we assume well-formed security types and environments.

### 3.4.2 Static semantics

Figure 3.3 presents the static semantics of  $\lambda_{\text{SEC}}^{\exists}$ . Security typing relies on a subtyping judgment that validates secure information flows. The left-most rule justifies subtyping by reflexivity. The middle rule justifies subtyping for two security types with the same safety type, when the declassification type of right security type is  $\top$ . Finally, the right-most rule justifies subtyping between a public type  $T_L$  and  $T@X$ .

As usual, the typing judgment  $\Delta; \Gamma \vdash e : S$  denotes that “the expression  $e$  has type  $S$  under the type variable environment  $\Delta$  and the type environment  $\Gamma$ ”. The typing rules are mostly standard [41]. Omitted rules for variables, pairs, pair projections, unit, and sum injections can be found in Appendix B.1.7. Here, we only discuss the special treatment of security types.

Rule (TS) is the standard subtyping subsumption rule, and rules (TFun) and (TPack) introduce function and existential types. In particular, rule (TPack) requires the representation type of the package to be *more precise* than the safety type associated to  $X$  in the existential type, *i.e.*  $T' \sqsubseteq \text{sftype}(\exists X.T)$ . The precision judgment has only two rules: reflexivity  $T \sqsubseteq T$ , and any type is more precise than a type variable  $T \sqsubseteq X$ .

Rules (TApp), (TCase) and (TOpen) are elimination rules for function, sum and existential types, respectively. When a secret is eliminated, the resulting computation must protect that secret. This is done with  $\lceil S' \rceil_S$ , which changes the declassification type of  $S'$  to  $\top$  if the type  $S$  is not public:

$$\lceil T_1@U_1 \rceil_{T_2@U_2} = T_1@U_1 \text{ if } T_2 = U_2, \text{ otherwise } T_1@\top$$

Let us illustrate the use of rule (TApp). On the one hand, if the type of the function expression  $e_1$  is  $(S_1 \rightarrow S_2)_L$ , *i.e.* it represents a public function, then the type of the function application is  $S_2$ . On the other hand, if the function expression  $e_1$  has type  $(S_1 \rightarrow T_2@U_2)@X$

$S <: S$

$T@U <: T@U \quad T@U <: T@T \quad T_L <: T@X$

$\Delta; \Gamma \vdash e : S$

$$\begin{array}{c}
\text{(TS)} \frac{\Delta; \Gamma \vdash e : S' \quad S' <: S}{\Delta; \Gamma \vdash e : S} \quad \text{(TFun)} \frac{\Delta; \Gamma, x : S \vdash e : S'}{\Delta; \Gamma \vdash \lambda x : S. e : (S \rightarrow S')_L} \\
\\
\text{(TPack)} \frac{\Delta; \Gamma \vdash e : (T[T'/X])_L \quad T' \sqsubseteq \text{sftype}(\exists X.T)}{\Delta; \Gamma \vdash \text{pack}(T', e) \text{ as } \exists X.T : (\exists X.T)_L} \quad \text{(TApp)} \frac{\Delta; \Gamma \vdash e_1 : S \quad S = (S_1 \rightarrow S_2)@U \quad \Delta; \Gamma \vdash e_2 : S_1}{\Delta; \Gamma \vdash e_1 e_2 : [S_2]_S} \\
\\
\text{(TCase)} \frac{\Delta; \Gamma \vdash e : S \quad S = (S_1 + S_2)@U \quad \Delta; \Gamma, x_1 : S_1 \vdash e_1 : S' \quad \Delta; \Gamma, x_2 : S_2 \vdash e_2 : S'}{\Delta; \Gamma \vdash \text{case } e \text{ of } \text{inl } x_1.e_1 \mid \text{inr } x_2.e_2 : [S']_S} \\
\\
\text{(TOpen)} \frac{\Delta; \Gamma \vdash e : S \quad S = (\exists X.T)@U \quad \Delta, X : T'; \Gamma, x : T_L \vdash e' : S' \quad T' \triangleq \text{sftype}(\exists X.T) \quad \Delta \models S'}{\Delta; \Gamma \vdash \text{open}(X, x) = e \text{ in } e' : [S']_S}
\end{array}$$

Figure 3.3:  $\lambda_{\text{SEC}}^{\exists}$ : Static semantics (selected rules)

or  $(S_1 \rightarrow T_2@U_2)_H$ , *i.e.* it represents a secret, then the function application has type  $T_2@T$ .

Rule (TCase) requires the discriminée to be of type  $(S_1 + S_2)@U$ , and both branches must have the same type  $S'$ . Likewise, it protects the resulting computation with  $[S']_S$ .

Finally, rule (TOpen) applies to expressions of the form  $\text{open}(X, x) = e \text{ in } e'$ , by typing the body expression  $e'$  in an extended type variable environment  $\Delta, X : T'$  and a type environment  $\Gamma, x : T_L$ . Two points are worth noticing. First, the association  $X : T'$  allows us to verify that security types of the form  $T'@X$  defined in the body expression  $e'$  are well-formed. Second, we make the well-formedness requirement explicit for the result type  $\Delta \models S'$ , which implies that  $S'$  is facet-wise well-formed and closed under  $\Delta$ —*i.e.* the type variable  $X$  cannot appear in  $S'$ .

### 3.4.3 Dynamic semantics and type safety

The execution of  $\lambda_{\text{SEC}}^{\exists}$  expressions is defined with a standard call-by-value small-step dynamic semantics based on evaluation contexts (Figure 3.4). To execute primitive operators, we use a *function*  $\theta$  to abstract over an internal runtime for primitive values.

We define the predicate  $\text{safe}(e)$  to indicate that the evaluation of the expression  $e$  does not get stuck.

**Definition 12** (Safety).  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$



$$E ::= [] \mid \text{fst } E \mid \text{snd } E \mid \text{case } E \text{ of } \text{inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mid E \text{ e} \mid v \ E \quad (\text{evaluation contexts})$$

$$\mid E \oplus e \mid v \oplus E \mid \text{open}(X, x) = E \text{ in } e'$$

$$\begin{array}{l} \text{fst } \langle v_1, v_2 \rangle \mapsto v_1 \\ \text{snd } \langle v_1, v_2 \rangle \mapsto v_2 \\ \text{case inl } v \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mapsto e_1 [v/x_1] \\ \text{case inr } v \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mapsto e_2 [v/x_2] \\ (\lambda x : S. e) v \mapsto e [v/x] \\ \mathbf{b}_1 \oplus \mathbf{b}_2 \mapsto \theta(\oplus, \mathbf{b}_1, \mathbf{b}_2) \\ \text{open}(X, x) = (\text{pack}(T', v) \text{ as } \exists X.T) \text{ in } e' \mapsto e' [v/x] [T'/X] \end{array} \quad \frac{e \mapsto e'}{E[e] \mapsto E[e']}$$

Figure 3.4:  $\lambda_{\text{SEC}}^{\exists}$ : Dynamic semantics

Well-typed  $\lambda_{\text{SEC}}^{\exists}$  closed terms are safe.

**Theorem 10** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

Having formally defined the language  $\lambda_{\text{SEC}}^{\exists}$ , we move to the main result of this chapter, which is to show that the  $\lambda_{\text{SEC}}^{\exists}$  is sound from a security standpoint, *i.e.* its type system enforces existential relaxed noninterference.

## 3.5 Existential relaxed noninterference, formally

In Section 3.3 we gave an overview of existential relaxed noninterference (ERNI), explaining how it depends on type-based equivalences. To formally capture these type-based equivalences, we define a logical relation, defined by induction on the structure of types. To account for type variables, we build upon prior work on logical relations for parametricity [4, 47, 64]. Then, we formally define ERNI on top of this logical relation. Finally, we prove that the type system of  $\lambda_{\text{SEC}}^{\exists}$  enforces existential relaxed noninterference.

### 3.5.1 Logical relation for type-based equivalence

As explained in Section 3.3, two values  $v_1$  and  $v_2$  are equivalent at type  $S$ , if they are in the partial equivalence relation denoted by  $S$ . To capture this, the logical relation (Figure 3.5) interprets types as set of *atoms*, *i.e.* pairs of closed expressions. We use  $\text{Atom}[T_1, T_2]$  to characterize the set of atoms with expressions of type  $T_1$  and  $T_2$  respectively. This definition appeals to a simply-typed judgment  $\Delta; \Gamma \vdash_1 e : T$  that does not consider the declassification type and is therefore completely standard (Appendix B.1.9). The use of this simple type system clearly separates the *definition* of secure programs from the *enforcement* mechanism, *i.e.* the security type system of Figure 3.3.

In Section 3.3 we explained what it means to be equivalent at type  $\text{Int}@X$  appealing to a relation on integers  $R_X \subseteq \text{Int} \times \text{Int}$ . To formally characterize the set of valid relations  $R_X$  for types  $T_1$  and  $T_2$  we use the definition  $\text{Rel}[T_1, T_2]$ . To keep track of the relation associated to a type variable, most definitions are indexed by an environment  $\rho$  that maps type variables  $X$

to triplets  $(T_1, T_2, R)$ , where  $T_1$  and  $T_2$  are two representation types of  $X$  and  $R$  is a relation on closed values of type  $T_1$  and  $T_2$  (*i.e.*  $\rho ::= \emptyset \mid \rho [X \mapsto (T_1, T_2, R)]$ ). We will explain later where these types  $T_1$  and  $T_2$  come from. We write  $\rho_1(U)$  (resp.  $\rho_2(U)$ ) to replace all type variables of  $\rho$  in types with the associated type  $T_1$  (resp.  $T_2$ ), and  $\rho_R(X)$  to retrieve the relation  $R$  of a type variable  $X$  in  $\rho$ .

Figure 3.5 defines the *value interpretation* of a type  $T$ , noted  $\mathcal{V}[[T]]\rho$ , then the value interpretation of a security type  $S$ , noted  $\mathcal{V}[[S]]\rho$ , and finally the *expression interpretation* of a type  $S$ , noted  $\mathcal{C}[[S]]\rho$ .

**Interpreting concrete types.** We first explain the definitions that do not involve types variables.  $\mathcal{V}[[T@T]]\rho$  (resp.  $\mathcal{V}[[T@T]]\rho$ ) characterizes when values of  $T$  are indistinguishable (resp. distinguishable) for the public observer.  $\mathcal{V}[[T@T]]\rho$  is defined as  $\text{Atom}[\rho_1(T), \rho_2(T)]$  indicating that any two values of type  $T$  are equivalent at type  $T@T$ . Note that this also includes values of type  $X@T$ .

Two *public* values are equivalent at a security type  $T@T$  if they are equivalent at their safety type, *i.e.*  $\mathcal{V}[[T_1]]\rho = \mathcal{V}[[T]]\rho$ . The definition  $\mathcal{V}[[P]]\rho$  relates syntactically-equal primitive values at type  $P$ . Two functions are equivalent at type  $S_1 \rightarrow S_2$ , noted  $\mathcal{V}[[S_1 \rightarrow S_2]]\rho$ , if given equivalent arguments at type  $S_1$ , their applications are equivalent expressions at type  $S_2$ . Two pairs are equivalent at type  $S_1 \times S_2$  if they are component-wise equivalent. Two values are equivalent at  $S_1 + S_2$  if they are either both left-injected values  $\text{inl } v_1$  and  $\text{inl } v_2$  such as  $v_1$  and  $v_2$  are equivalent at  $S_1$ , or both right-injected values  $\text{inr } v_1$  and  $\text{inr } v_2$  such as  $v_1$  and  $v_2$  are equivalent at  $S_2$ .

Finally, two expressions  $e_1$  and  $e_2$  are equivalent at type  $T@U$ , noted  $\mathcal{C}[[T@U]]\rho$ , if they both reduce to values  $v_1$  and  $v_2$  respectively and these values are related at type  $T@U$ . (Note that all well-typed  $\lambda_{\text{SEC}}^{\exists}$  expressions terminate.)

**Interpreting existential types.** We now explain the value interpretation of existential types  $\exists X.T$ , type variables  $X$  and security types of the form  $T@X$ , which all involve type variables.

Two public package expressions  $\text{pack}(T_1, v_1)$  as  $\exists X.T$  and  $\text{pack}(T_2, v_2)$  as  $\exists X.T$  are equivalent at type  $\exists X.T$ , noted  $\mathcal{V}[[\exists X.T]]\rho$ , if there exists a relation  $R$  on the representation types  $T_1$  and  $T_2$  that makes the package implementations  $v_1$  and  $v_2$  equivalent at type  $T$ , noted  $(v_1, v_2) \in \mathcal{V}[[T]]\rho [X \mapsto (T_1, T_2, R)]$ . Note that if the existential type  $\exists X.T$  has a concrete safety type  $T'$  (not a type variable) for  $X$ , then both  $T_1$  and  $T_2$  necessarily have to be equal to  $T'$ . Otherwise,  $T_1$  and  $T_2$  are arbitrary types. Two values are related at type  $X$ , noted  $\mathcal{V}[[X]]\rho$ , if they are in the relational interpretation  $R$  associated to  $X$  (retrieved with  $\rho_R(X)$ ). Two values are related at type  $T@X$ , noted  $\mathcal{V}[[T@X]]\rho$ , if they are in  $\rho_R(X)$ , or if they are publicly-equivalent values of type  $T$  (*i.e.* a package can accept public values of type  $T$  where values of  $T@X$  are expected).

We illustrate these formal type-based equivalences in Section 3.6, after formally defining existential relaxed noninterference and proving security type soundness.

$$\begin{aligned}
\text{Atom}[T_1, T_2] &= \{(e_1, e_2) \mid \bullet; \bullet \vdash_1 e_1 : T_1 \wedge \bullet; \bullet \vdash_1 e_2 : T_2\} \\
\text{Atom}_\rho[T] &= \text{Atom}[\rho_1(T), \rho_2(T)] \\
\text{Rel}[T_1, T_2] &= \{R \subseteq \text{Atom}[T_1, T_2]\} \\
\mathcal{V}[\mathbf{1}]\rho &= \{(\langle \rangle, \langle \rangle) \in \text{Atom}_\rho[\mathbf{1}]\} \\
\mathcal{V}[P]\rho &= \{(\mathbf{b}, \mathbf{b}) \in \text{Atom}_\rho[P]\} \\
\mathcal{V}[S_1 \rightarrow S_2]\rho &= \{(v_1, v_2) \in \text{Atom}_\rho[S_1 \rightarrow S_2] \mid \\
&\quad \forall v'_1, v'_2. (v'_1, v'_2) \in \mathcal{V}[S_1]\rho \implies (v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2]\rho\} \\
\mathcal{V}[S_1 \times S_2]\rho &= \{(\langle v_1, v'_1 \rangle, \langle v_2, v'_2 \rangle) \in \text{Atom}_\rho[S_1 \times S_2] \mid (v_1, v_2) \in \mathcal{V}[S_1]\rho \wedge (v'_1, v'_2) \in \mathcal{V}[S_2]\rho\} \\
\mathcal{V}[S_1 + S_2]\rho &= \{(\text{inl } v_1, \text{inl } v_2) \in \text{Atom}_\rho[S_1 + S_2] \mid (v_1, v_2) \in \mathcal{V}[S_1]\rho\} \\
&\quad \cup \{(\text{inr } v_1, \text{inr } v_2) \in \text{Atom}_\rho[S_1 + S_2] \mid (v_1, v_2) \in \mathcal{V}[S_2]\rho\} \\
\mathcal{V}[\exists X.T]\rho &= \{(\text{pack}(T_1, v_1) \text{ as } \exists X.T, \text{pack}(T_2, v_2) \text{ as } \exists X.T) \in \text{Atom}_\rho[\exists X.T] \mid \\
&\quad T_1 \sqsubseteq \text{sftype}(\exists X.T) \wedge T_2 \sqsubseteq \text{sftype}(\exists X.T) \wedge \\
&\quad \exists R \in \text{Rel}[T_1, T_2]. (v_1, v_2) \in \mathcal{V}[T]\rho[X \mapsto (T_1, T_2, R)]\} \\
\mathcal{V}[X]\rho &= \rho_{\mathbf{R}}(X) \\
\mathcal{V}[T@X]\rho &= \rho_{\mathbf{R}}(X) \cup \mathcal{V}[T]\rho \\
\mathcal{V}[T@T]\rho &= \text{Atom}[\rho_1(T), \rho_2(T)] \\
\mathcal{V}[T@T]\rho &= \mathcal{V}[T]\rho \\
\mathcal{C}[T@U]\rho &= \{(e_1, e_2) \in \text{Atom}_\rho[T] \mid e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[T@U]\rho\}
\end{aligned}$$

Figure 3.5:  $\lambda_{\text{SEC}}^{\exists}$  Logical relation for type-based equivalence

### 3.5.2 Existential relaxed noninterference

As illustrated in Section 3.3, ERNI is a modular property that accounts for open expressions over both variables and type variables. To account for open expressions, we first need to define the relational interpretation of a type environment  $\Gamma$  and a type variable environment  $\Delta$ :

$$\begin{aligned}
\mathcal{G}[\cdot]\rho &= \{(\emptyset, \emptyset)\} \\
\mathcal{G}[\Gamma; x : S]\rho &= \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \wedge (v_1, v_2) \in \mathcal{V}[S]\rho\} \\
\mathcal{D}[\cdot] &= \{\emptyset\} \\
\mathcal{D}[\Delta; X : T] &= \{\rho[X \mapsto (T_1, T_2, R)] \mid \rho \in \mathcal{D}[\Delta] \wedge T_1 \sqsubseteq T \wedge T_2 \sqsubseteq T \wedge R \in \text{Rel}[T_1, T_2]\}
\end{aligned}$$

The type environment interpretation  $\mathcal{G}[\Gamma]\rho$  is standard; it characterizes when two value substitutions  $\gamma_1$  and  $\gamma_2$  are equivalent. A value substitution  $\gamma$  is a mapping from variables to closed values (*i.e.*  $\gamma ::= \emptyset \mid \gamma[x \mapsto v]$ ). Two value substitutions are equivalent if for all associations  $x : S$  in  $\Gamma$ , the mapped values to  $x$  in  $\gamma_1$  and  $\gamma_2$  are equivalent at  $S$ . Finally, the interpretation of a type variable environment  $\Delta$ , denoted  $\mathcal{D}[\Delta]$ , is a set of type substitutions  $\rho$  with the same domain as  $\Delta$ . For each type variable  $X$  bound to  $T$  in  $\Delta$ , such a  $\rho$  maps  $X$  to triples  $(T_1, T_2, R)$ , where  $T_1$  and  $T_2$  are closed types that are more precise than  $T$ .  $R$  must be a valid relation for the types  $T_1$  and  $T_2$ . We write  $\rho_1(e)$  (resp.  $\rho_2(e)$ ) to replace all type variables of  $\rho$  in terms with their associated type  $T_1$  (resp.  $T_2$ ).

We can now formally define ERNI. An expression  $e$  satisfies existential relaxed noninterference for a type variable environment  $\Delta$  and a type environment  $\Gamma$  at the security type  $S$ , noted  $\text{ERNI}(\Delta, \Gamma, e, S)$  if, given a type substitution  $\rho$  satisfying  $\Delta$  and two values substitutions  $\gamma_1$  and  $\gamma_2$  that are equivalent at  $\Gamma$ , applying the substitutions produces equivalent

expressions at type  $S$ .

**Definition 13** (Existential relaxed noninterference).

$$\begin{aligned} \text{ERNI}(\Delta, \Gamma, e, S) &\iff S \triangleq T@U \quad \Delta; \Gamma \vdash_1 e : T \wedge \Delta \models \Gamma \wedge \Delta \models S \wedge \\ &\forall \rho, \gamma_1, \gamma_2. \rho \in \mathcal{D}[\Delta]. (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \implies (\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e))) \in \mathcal{C}[S]\rho \end{aligned}$$

### 3.5.3 Security type soundness

Instead of directly proving that the type system of Figure 3.3 implies existential relaxed noninterference for all well-typed terms, we prove it through the standard definition of logically-related open terms (as we did in Chapter 2):

**Definition 14** (Logically-related open terms).

$$\begin{aligned} \Delta; \Gamma \vdash e_1 \approx e_2 : S &\iff \Delta; \Gamma \vdash e_i : S \wedge \Delta \models \Gamma \wedge \Delta \models S \wedge \\ &\forall \rho, \gamma_1, \gamma_2. \rho \in \mathcal{D}[\Delta]. (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \implies (\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{C}[S]\rho \end{aligned}$$

The next lemma captures that if an expression is logically related to itself, then it satisfies ERNI.

**Lemma 11** (Self logical relation implies PRNI).

$$\Delta; \Gamma \vdash e \approx e : S \implies \text{ERNI}(\Delta, \Gamma, e, S)$$

The proof of security type soundness relies on the Fundamental Property of the logical relation: a well-typed  $\lambda_{\text{SEC}}^{\exists}$  term is related to itself.

**Theorem 12** (Fundamental property).  $\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \vdash e \approx e : S$

Security type soundness follows from Lemma 11 and Theorem 12.

**Theorem 13** (Security type soundness).  $\Delta; \Gamma \vdash e : S \implies \text{ERNI}(\Delta, \Gamma, e, S)$

## 3.6 Illustration

With all the formal definitions at hand, we end by revisiting the informal example of Section 3.3. `SalaryPolicy` can be encoded in  $\lambda_{\text{SEC}}^{\exists}$  as follow :  $\exists X. \text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L)$ . Let us show that the following two packages are equivalent at type `SalaryPolicyL` (`gte` is a curried comparison function, and we omit the `as`):

$$p_1 \triangleq \text{pack}(\text{Int}, \langle 100001, \text{gte}(100000) \rangle) \quad p_2 \triangleq \text{pack}(\text{Int}, \langle 100002, \text{gte}(100000) \rangle)$$

The definition of  $\mathcal{V}[\text{SalaryPolicy}]\emptyset$  requires picking a relation  $R \in \text{Rel}[\text{Int}, \text{Int}]$ . Pick  $R = \{(100001, 100002)\}$ . Then apply the rest of the definitions to verify that the package implementations are equivalent at  $\mathcal{V}[\text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L)]\emptyset[X \mapsto (\text{Int}, \text{Int}, R)]$ . Use the  $\rho_R(X)$  part of the definition of  $\mathcal{V}[\text{Int}@X]\emptyset[X \mapsto (\text{Int}, \text{Int}, R)]$  to show that the first components 100001 and 100002 are equivalent at `Int@X`, *i.e.*  $(100001, 100002) \in \rho_R(X)$ .

First we illustrate the formal reasoning that we obtain from Theorem 13. Let us pose  $\Delta = X : \text{Int}$  and  $\Gamma = x : (\text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L))_L$ . The program  $e = (\text{snd } x) (\text{fst } x)$  has type  $\text{Bool}_L$ , therefore, by Theorem 13,  $\text{ERNI}(\Delta, \Gamma, e, \text{Bool}_L)$  holds— $e$  is secure at  $\text{Bool}_L$ . We can verify this formally. By Definition 13 we have to assume an arbitrary type substitution  $\rho \in \mathcal{D}[\![X : \text{Int}]\!]$  and two values substitutions  $(\gamma_1, \gamma_2) \in \mathcal{G}[\![x : (\text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L))_L]\!]\rho$  and to show that  $(\rho_1(\gamma_1((\text{snd } x) (\text{fst } x))), \rho_2(\gamma_2((\text{snd } x) (\text{fst } x)))) \in \mathcal{C}[\![\text{Bool}_L]\!]\rho$ . From  $(\gamma_1, \gamma_2) \in \mathcal{G}[\![x : (\text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L))_L]\!]\rho$  we know that  $\gamma_1 = x \mapsto v_1$  and  $\gamma_2 = x \mapsto v_2$ , such as  $(v_1, v_2) \in \mathcal{V}[\![\text{Int}@X \times (\text{Int}@X \rightarrow \text{Bool}_L)]_L]\rho$ . Then  $(\text{snd } v_1) (\text{fst } v_1) \mapsto^* v'_{11} v'_{12}$  and  $(\text{snd } v_2) (\text{fst } v_2) \mapsto^* v'_{21} v'_{22}$ , such that  $(v'_{11}, v'_{21}) \in \mathcal{V}[\![\text{Int}@X \rightarrow \text{Bool}_L]\!]\rho$  and  $(v'_{12}, v'_{22}) \in \mathcal{V}[\![\text{Int}@X]\!]\rho$ . Finally, instantiate  $(v'_{11}, v'_{21}) \in \mathcal{V}[\![\text{Int}@X \rightarrow \text{Bool}_L]\!]\rho$  with  $(v'_{12}, v'_{22}) \in \mathcal{V}[\![\text{Int}@X]\!]\rho$  to obtain  $(v'_{11} v'_{12}, v'_{21} v'_{22}) \in \mathcal{C}[\![\text{Bool}_L]\!]\rho$ .

Second, we formally show why  $\text{ERNI}(\Delta, \Gamma, (\text{fst } x)\%2, \text{Int}_L)$  does not hold. Instantiate Definition 13 with  $\rho = X \mapsto (\text{Int}, \text{Int}, R)$  and  $\gamma_1 = x \mapsto \langle 100001, \text{gte}(100000) \rangle$  and  $\gamma_2 = x \mapsto \langle 100002, \text{gte}(100000) \rangle$ . Note that  $\rho = X \mapsto (\text{Int}, \text{Int}, R) \in \mathcal{D}[\![X : \text{Int}]\!]$  and  $(\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!]\rho$ . To show  $(\rho_1(\gamma_1((\text{fst } x)\%2)), \rho_2(\gamma_2((\text{fst } x)\%2))) \in \mathcal{C}[\![\text{Int}_L]\!]\rho$  requires showing  $(100001\%2, 100002\%2) \in \mathcal{C}[\![\text{Int}_L]\!]\rho$ , which means showing  $(1, 0) \in \mathcal{C}[\![\text{Int}_L]\!]\rho$ —which is false. Similarly, we can verify that  $\text{ERNI}(\Delta, \Gamma, x.\text{salary}, \text{Int}_L)$  does not hold, which means that a declassifiable secret cannot be directly observed by a public observer.

### 3.7 Related work

We have already extensively discussed the relation to the original formulation of the labels-as-types approach in an object-oriented setting, itself inspired by the work on declassification policies (labels-as-functions) of Li and Zdancewic [31]. Formulating type-based declassification with existential types shows how to exploit another type abstraction mechanism that is found in non-object-oriented languages, with abstract data types and modules. Also, existential types support extrinsic declassification policies, which are not expressible in the receiver-centric approach of objects. For instance, the `AccountStore` example of Section 3.2 is not supported by design in the object-oriented approach.

The extrinsic declassification policies supported by our approach are closely related to *trusted declassification* [29], where declassification is globally defined, associating principals that own secrets with trusted (external) methods that can declassify these secrets. In our approach, the relation between secrets and declassifiers is not globally defined, but is local to an existential type and its usage. In both approaches the implementations of declassifiers have a privileged view of the secrets.

Bowman and Ahmed [14] present a translation of noninterference into parametricity with a compiler from the Dependency Core Calculus (DCC) [2] to System  $F\omega$ . In a recent (as yet unpublished) article, Ngo et al. [40] extend this work to support translating declassification policies, inspired by prior work on type-based relaxed noninterference [22]. They first provide a translation into abstract types of the polymorphic lambda calculus [47], and then into signatures of a module calculus [19]. While that work and ours encode declassification policies via existential types (module signatures), we focus on providing a surface language for information flow control with type-based declassification. In particular, their translated programs do not support computing with secrets, which is enabled in both this chapter

and the Chapter 2( Cruz et al. [22]) thanks to faceted types. Additionally, they only model first-order secrets (integers), while our modular reasoning principle seamlessly accommodates higher-order secrets.

In Chapter 4 we extend the object-oriented approach to type-based relaxed noninterference with parametric polymorphism, thereby supporting polymorphic declassification policies. In that Chapter, in Section 4.6 we comments how the approach developed here supports a form of declassification polymorphism in the client code.

The idea of using the abstraction mechanism of modules to express a form of declassification can also be found in the work of Nanevski et al. [39] on Relational Hoare Type Theory (RHTT). RHTT is formulated with a monadic security type constructor  $\text{STsec } A(p, q)$ , where  $p$  is a pre-condition on the heap, and  $q$  is a post-condition relating output values, input heaps and output heaps. Thanks to the expressive power of the underlying dependent type theory, preconditions and postconditions can characterize very precise declassification policies. The price to pay is that proofs of noninterference have to be provided explicitly as proof terms (or discharged via tactics or other means when possible), while our less expressive approach is a simple, non-dependent type system. Finding the right balance between the expressiveness and the complexity of the typing discipline to express security policies is an active subject of research.

## 3.8 Conclusion

We present a novel approach to type-based relaxed noninterference, based on existential types as the underlying type abstraction mechanism. In contrast to the object-oriented, subtyping-based approach, the existential approach naturally supports external declassification policies. This chapter shows that the general approach of faceted security types for expressive declassification can be applied in non-object-oriented languages that support abstract data types or modules. As such, it represents a step towards providing a practical realization of information-flow security typing that accounts for controlled and expressive declassification with a modular reasoning principle about security.

# Chapter 4

## Polymorphic relaxed noninterference

In this chapter we extend the object-oriented type-based declassification (Chapter 2) approach in order to support *polymorphic* declassification. First, we identify the need for bounded polymorphism through concrete examples. We then formalize polymorphic relaxed noninterference in a typed object-oriented calculus  $\text{Ob}_{\text{SEC}}^{\langle \rangle}$ , using a step-indexed logical relation to prove that all well-typed terms are secure. Furthermore, we address the case of primitive types, which requires a form of ad-hoc polymorphism. Finally, we show how the type-based declassification approach with existential types (Chapter 3) enjoys a form of declassification polymorphism for the client code of a package.

This chapter is based on the work of Cruz and Tanter [21], except for Section 4.6.

### 4.1 Introduction

The object-oriented labels-as-types (Chapter 2) approach lacks *security label polymorphism*. Security label polymorphism is a very useful feature of practical security-typed languages such as Jif [36] and FlowCaml [45], which has only been explored in the context of standard security labels (symbols from a lattice). To the best of our knowledge, polymorphism has not been studied for expressive declassification mechanisms, such as labels-as-functions [31].

Furthermore, the development of the object-oriented labels-as-types in Chapter 2 ignores the case of primitive types, such as integers and strings. It means that in  $\text{Ob}_{\text{SEC}}$  the only real observation that can be made on programs is termination. However, type-based relaxed noninterference for  $\text{Ob}_{\text{SEC}}$  is termination-insensitive, and therefore useless without some primitive types with a purely syntactic notion of equality. Indeed, all the examples we presented in Chapter 2 assume a syntactic notion of observation for strings and integers.

This chapter addresses the two major shortcomings of the object-oriented labels-as-type approach in order to bring this approach closer to real-world secure programming. First, we extend the labels-as-types approach with declassification *polymorphism*, specifically *bounded* polymorphism. We reuse the theory of bounded type polymorphism to express bounded declassification polymorphism, which is possible because labels-as-types uses standard types as declassification mechanism. Bounded type polymorphism allows generic types to specify

a subtype and a supertype, the lower and upper bound, respectively. For instance, we can use generic type bounds to capture that a sorting method over a list requires the elements of the list to be comparable, *i.e.* that they have at least a method `compare`. In the case of bounded declassification polymorphism, the generic type bounds are used to selectively declassify information over secrets with generic declassification types. Second, we address the necessary support for primitive types, through a form of ad-hoc polymorphism.

We also explain that the labels-as-types approach with existential types enjoys a form of declassification polymorphism for the client code of a package, for user-defined types and primitive types.

In summary, this chapter makes the following contributions:

- We extend the object-oriented type-based declassification with bounded polymorphic declassification which brings new benefits in the expressiveness and design of declassification (Section 4.2). Additionally, we address the necessary support for primitive types, through a form of ad-hoc polymorphism (4.5).
- We capture bounded polymorphic declassification in  $\mathbf{Ob}_{\text{SEC}}^{\diamond}$ , an extension to  $\mathbf{Ob}_{\text{SEC}}$  with bounded polymorphism (Section 4.3).
- We develop the theory of bounded polymorphic declassification as an extension of TRNI, called *polymorphic relaxed noninterference* (PRNI for short) and we show that all well-typed  $\mathbf{Ob}_{\text{SEC}}^{\diamond}$  terms satisfy PRNI (Section 4.4).

Section 4.6 explains the form of declassification polymorphism that enjoys that existential labels-as-type approach. Section 4.7 discusses related work and Section 4.8 gives conclusions about the work developed in this chapter. We have implemented an interactive prototype of  $\mathbf{Ob}_{\text{SEC}}^{\diamond}$  available at <https://pleiad.cl/gobsec/>.

## 4.2 Overview

We first motivate polymorphic declassification with faceted types, and then we illustrate the role of bounded polymorphism for declassification. Finally, we give an overview of the modular reasoning principle of polymorphic relaxed noninterference.

### 4.2.1 Polymorphic declassification

We illustrate the benefits of polymorphic declassification by giving the example of a list of strings that is polymorphic with respect to the declassification type of its elements:

$$\text{ListStr}\langle X \rangle \triangleq [\text{isEmpty} : \text{Unit}_{\mathbb{L}} \rightarrow \text{Bool}_{\mathbb{L}}, \\ \text{head} : \text{Unit}_{\mathbb{L}} \rightarrow \text{String} \triangleleft X, \\ \text{tail} : \text{Unit}_{\mathbb{L}} \rightarrow \text{ListStr}\langle X \rangle_{\mathbb{L}}]$$

This recursive polymorphic declassification policy allows a public observer to traverse the list, and to observe *up to*  $X$  on each of the elements. This restriction is visible in the signature of the `head` method, which returns a value of type  $\text{String} \triangleleft X$ .



Then, with polymorphic declassification we can implement data structures that are agnostic to the declassification policies of their elements, as well as polymorphic methods over these data structures. For example, we can construct declassification-polymorphic lists of strings with the following `cons` method:

```
ListStr⟨X⟩L cons<X>(String<X> s, ListStr⟨X⟩L l) {
  return new {
    self: ListStr⟨X⟩L
    isEmpty() => false
    head() => s
    tail() => l
  }
}
```

The `cons` method does not even access any method of list `l`, it simply returns a new declassification-polymorphic list of strings as a new object with the expected methods. We can then use this method to define a declassification-polymorphic list concatenation method `concat`:  $\text{ListStr}\langle X\rangle_L \times \text{ListStr}\langle X\rangle_L \rightarrow \text{ListStr}\langle X\rangle_L$  defined below:

```
ListStr⟨X⟩L concat<X>(ListStr⟨X⟩L l1, ListStr⟨X⟩L l2) {
  if(l1.isEmpty()) return l2
  return cons<X>(l1.head(),
                 concat<X>(l1.tail(), l2))
}
```

The `concat` and `cons` methods are standard object-oriented implementations of list concatenation and construction, respectively. The `concat` method respects the declassification type  $\text{ListStr}\langle X\rangle$  of both lists because it uses `l1.isEmpty()` and `l1.tail()` to iterate over `l1`, and it uses `l1.head()` to create a new declassification-polymorphic list of type  $\text{ListStr}\langle X\rangle_L$ . In particular, it uses no string-specific methods.

## 4.2.2 Bounded polymorphic declassification

The declassification interface  $\text{ListStr}\langle X\rangle$  above is fully polymorphic, in that a public observer cannot exploit *a priori* any information about the elements of the list. In particular, it is not possible to implement a polymorphic `contains` method that would yield publicly observable results. Indeed, `contains` needs to invoke `eq` over the elements of the list (obtained with `head`). Because the result of `head` has declassification type  $X$ , for *any*  $X$ , the results of equality comparisons are necessarily private.

In order to support polymorphic declassification more flexibly, we turn to *bounded* parametric polymorphism. Bounded parametric polymorphism supports the specification of both upper and lower bounds on type variables. The type  $\text{ListStr}\langle X\rangle$  is therefore equivalent to  $\text{ListStr}\langle X : \text{String}..T\rangle$ , where the notation  $X : A..B$  is used to denote that  $X$  is a type variable that ranges between  $A$  and  $B$ ). Note that for  $\text{ListStr}$  to be well-formed, the declassification type variable  $X$  must at least be a supertype of the safety type `String`.

Going back to declassification-polymorphic lists, if we want to allow the definition of methods like `contains`, we can further constrain the type variable  $X$  to be a subtype of

StringEq:

$$\text{ListEqStr}\langle X : \text{String}.. \text{StringEq} \rangle \triangleq$$

$$[ \dots, \text{tail} : \text{Unit}_L \rightarrow \text{ListEqStr}\langle X \rangle_L ]$$

The type `ListEqStr` denotes a recursive polymorphic declassification policy that allows a public observer to traverse the list and compare its elements for equality with a given public element. With this policy we can implement a generic `contains` function with publicly observable result:

```

BoolL contains <X : String..StringEq> (ListEqStr <X>L l, StringL s){
  if(l.isEmpty()) return false
  if(l.head().eq(s)) return true
  return contains(l.tail(),s)
}

```

The key here is that `l.head().eq(s)` is guaranteed to be publicly observable, because the actual declassification policy with which  $X$  will be instantiated necessarily includes (at least) the `eq` method. Thus, upper bounds on declassification variables are useful for supporting polymorphic *clients*.

As mentioned above, the lower bound of a type variable used for declassification must at least be the safety type for well-formedness. More interestingly, the lower bound plays a critical (dual) role for *implementors* of declassification-polymorphic functions. Consider a method with signature

$$\langle X : \text{String}.. \top \rangle \text{String} \triangleleft \text{StringLen} \rightarrow \text{String} \triangleleft X$$

Can this method return non-public values? For instance, can it be the identity function? No, because returning a string of type `String`  $\triangleleft$  `StringLen` would be unsound. Indeed, a client could instantiate  $X$  with `String`, yielding

$$\text{String} \triangleleft \text{StringLen} \rightarrow \text{String} \triangleleft \text{String}$$

Therefore, to be sound for all possible instantiations of  $X$ , the implementor of the method has no choice but to return a public string.

To recover flexibility and allow a polymorphic implementation to return non-public values, we can constrain the lower bound of  $X$ . For instance

$$\langle X : \text{StringLen}.. \top \rangle \text{String} \triangleleft \text{StringLen} \rightarrow \text{String} \triangleleft X$$

admits the identity function as an implementation, in addition to other implementations that produce public results. Returned values cannot be *more private* than specified by the lower bound of  $X$ ; their type must be a subtype of the lower bound.

Having illustrated the interest of upper and lower bounds of declassification type variables in isolation, we now present an example that combines both. Consider two lists of strings,

each with one of the following declassification policies:

$$\begin{aligned}
\text{ListStrLen} \langle X : \text{String}.. \text{StringLen} \rangle &\triangleq \\
&[ \text{isEmpty} : \text{Unit}_L \rightarrow \text{Bool}_L, \\
&\text{head} : \text{Unit}_L \rightarrow \text{String} \triangleleft X, \\
&\text{tail} : \text{Unit}_L \rightarrow \text{ListStrLen}_L ] \\
\text{ListStrFstLen} &\triangleq [ \text{isEmpty} : \text{Unit}_L \rightarrow \text{Bool}_L, \\
&\text{head} : \text{Unit}_L \rightarrow \text{String} \triangleleft \text{StrFstLen}, \\
&\text{tail} : \text{Unit}_L \rightarrow \text{ListStrFstLen}_L ]
\end{aligned}$$

`ListStrLen` is declassification polymorphic, ensuring that at least the length of its elements is declassified ( $X$  has upper bound `StringLen`). The second policy, `ListStrFstLen`, is monomorphic: it declassifies both the first character and the length of its elements. If we want a function able to concatenate these two string lists, its most general polymorphic signature ought to be:

$$\begin{aligned}
&\langle X : \text{StrFstLen}.. \text{StringLen} \rangle \\
&\text{ListStrLen} \langle X \rangle_L \times \text{ListStrFstLen}_L \rightarrow \text{ListStrLen} \langle X \rangle_L
\end{aligned}$$

The upper bound `StringLen` is required to have a valid instantiation of `ListStrLen`  $\langle X \rangle$ ; the lower bound `StrFstLen` is required to be able to add elements of the second list to the returned list.

### 4.2.3 Polymorphic relaxed noninterference, informally

Introducing polymorphism in declassification types yields an extended notion of type-based relaxed noninterference called *polymorphic relaxed noninterference* (PRNI). PRNI exactly characterizes that a program with polymorphic types must be secure for any instantiation of its type variables. To account for type variables, the judgment  $\text{PRNI}(\Delta, \Gamma, e, S)$  is parametrized by  $\Delta$ , a set of bounded type variables (*i.e.*  $\Delta ::= \cdot \mid \Delta, X : A..B$ ). As in  $\text{TRNI}(\Gamma, e, S)$ , the closing typing environment  $\Gamma$  specifies the secrecy of the inputs that  $e$  can use, and  $S$  specifies the observation level for the output.  $\Delta$  gives meaning to the type variables that can occur in both  $S$  and  $\Gamma$ :  $e$  is secure for *any* instantiation of type variables that respects the bounds.

For instance, given  $\Delta \triangleq X : \text{StrFstLen}.. \text{StringLen}$  and  $\Gamma \triangleq x : \text{String} \triangleleft X$ ,  $\text{PRNI}(\Delta, \Gamma, x.\text{length}(), \text{Int}_L)$  holds because for any type  $T$  such that  $\text{StrFstLen} \triangleleft T \triangleleft \text{StringLen}$ , and the knowledge that two input strings are related at  $\text{String} \triangleleft T$ , and hence at  $\text{String} \triangleleft \text{StringLen}$  (*i.e.* both strings have the same length), the public observer does not learn anything new by executing  $x.\text{length}()$ . However,  $\text{PRNI}(\Delta, \Gamma, x.\text{first}(), \text{String}_L)$  does not hold. Note that if we substitute  $X$  by `StringLen`, given two strings with the same length “abc” and “123”, the public observer is able to distinguish them by executing “abc”.first() and “123”.first() and observing the results “a” and “1” as public values.

Also,  $\text{PRNI}(\Delta, \Gamma, x, \text{String} \triangleleft \text{StringFst})$  does not hold. Again, we can substitute  $X$  by `StringLen`, and take input strings “abc” and “123”, which can be discriminated by the public observer at type  $\text{String} \triangleleft \text{StringFst}$ . However,  $\text{PRNI}(\Delta, \Gamma, x, \text{String} \triangleleft \text{StringLen})$  does hold: any

two equivalent values at  $\text{String} \triangleleft T$  where  $\text{StrFstLen} <: T <: \text{StringLen}$  have at least the same length.

The rest of this chapter dives into the formalization of polymorphic relaxed noninterference in a pure object-oriented setting (Sections 4.3 and 4.4), before discussing the necessary extensions to accommodate primitive types (Section 4.5).

## 4.3 Formal semantics

We model polymorphic type-based declassification in  $\text{Ob}_{\text{SEC}}^{\diamond}$ , an extension of the language  $\text{Ob}_{\text{SEC}}$  (Chapter 2) with polymorphic declassification.  $\text{Ob}_{\text{SEC}}$  is based on the object calculi of Abadi and Cardelli [1], and our treatment of type variables and bounded polymorphism is inspired by Featherweight Java [30] and DOT [48].

### 4.3.1 Syntax

Figure 4.1 presents the syntax of  $\text{Ob}_{\text{SEC}}^{\diamond}$ . We highlight the extension for polymorphic declassification, compared to the syntax of  $\text{Ob}_{\text{SEC}}$ .

The language has three kind of expressions: objects, method invocations and variables. Objects  $[z : S \Rightarrow \overline{m(x)} e]$  are collections of method definitions. Recall that the self variable  $z$  binds the current object.

A security type  $S$  is a faceted type  $T \triangleleft U$ , where  $T$  is called the *safety type* of  $S$ , and  $U$  is called the *declassification type* of  $S$ . Types  $T$  include object types  $O$  and self type variables  $\alpha$ . Declassification types  $U$  additionally feature type variables  $X$ , to express polymorphic declassification. We use metavariables  $A$  and  $B$  for declassification type bounds.

An object type  $\text{Obj}(\alpha). [\overline{m : M}]$  is a collection of method signatures with unique names (we sometimes use  $R$  to refer to just a collection of methods). The self type variable  $\alpha$  binds to the defined object type (*i.e.* object types are recursive types). A method signature  $\langle X : A..B \rangle S_1 \rightarrow S_2$  introduces the type variable  $X$  with lower bound  $A$  and upper bound  $B$ . To simplify the presentation of the calculus, we model single-argument methods with a single type variable.<sup>1</sup>

### 4.3.2 Subtyping

Figure 4.2 presents the  $\text{Ob}_{\text{SEC}}^{\diamond}$  subtyping judgment  $\Delta; \Phi \vdash U_1 <: U_2$ . The type variable environment  $\Delta$  is a set of type variables with their bounds, *i.e.*  $\Delta ::= \bullet \mid \Delta, X : A..B$ . The subtyping environment  $\Phi$  is a set of subtyping assumptions between self type variables, *i.e.*  $\Phi ::= \bullet \mid \Phi, \alpha <: \beta$

The rules for the monomorphic part of the language are similar to  $\text{Ob}_{\text{SEC}}$ . Rule (SObj) justifies subtyping between two object types; it holds if the methods of the left object type  $O_1$  are subtypes of the corresponding methods on  $O_2$ . Both width and depth subtyping are

<sup>1</sup>The implementation supports both multiple arguments and multiple type variables.

$e$	::=	$v \mid e.m \langle U \rangle (e) \mid x$	(terms)
$v$	::=	$o$	(values)
$o$	::=	$\left[ z : S \Rightarrow \overline{m(x)e} \right]$	(objects)
$S$	::=	$T \triangleleft U$	(security types)
$T$	::=	$O \mid \alpha$	(types)
$U, A, B$	::=	$T \mid X$	(declassification types)
$O$	::=	$\mathbf{Obj}(\alpha).R$	(object types)
$R$	::=	$\overline{m : M}$	(record types)
$M$	::=	$\langle X : A..B \rangle S \rightarrow S$	(method signatures)
$\Gamma$	::=	$\bullet \mid \Gamma, x : S$	(type environments)
$\Phi$	::=	$\bullet \mid \Phi, \alpha <: \beta$	(subtyping environments)
$\Delta$	::=	$\bullet \mid \Delta, X : A..B$	(type variable environments)
$\alpha, \beta$			(self type variables)

Figure 4.1:  $\mathbf{Ob}_{\text{SEC}}^{\langle \rangle}$ : Syntax

supported. Note that to verify subtyping of method collections, *i.e.*  $\Delta; \Phi, \alpha <: \beta \vdash R_1 <: R_2$ , we put in subtyping relation in  $\Phi$  the self variables  $\alpha$  and  $\beta$ . Rule (SVar) accounts for subtyping between self type variables and it holds if such subtyping relation exists in the subtyping environment.

The rules (STrans) and (SSubEq) justify subtyping by transitivity and type equivalence respectively. We consider type equivalence up to renaming and folding/unfolding of self type variables (Chapter 2).

The novel part of  $\mathbf{Ob}_{\text{SEC}}^{\langle \rangle}$  are type variables, handled by rules (SGVar1) and (SGVar2). We follow the approach of Rompf and Amin [48]. Rule (SGVar1) justifies subtyping between a type variable  $X$  and a type  $B$ , if  $B$  is the upper bound of the type variable in  $\Delta$ . Rule (SGVar2) is dual to (SGVar1), justifying that  $A <: X$  if  $A$  is the lower bound of  $X$ .

The judgment  $\Delta; \Phi \vdash R_1 <: R_2$  accounts for subtyping between collections of methods, and is used in rule (SObj). The judgment  $\Delta; \Phi \vdash M_1 <: M_2$  denotes subtyping between method signatures. For this judgment to hold, the type variable bounds  $A'..B'$  of the supertype (on the right) must be included within the bounds  $A..B$  of the subtype (on the left); this ensures that any instantiation on the right is valid on the left. Then, in a type variable environment extended with  $X : A'..B'$ , standard function subtyping must hold (contravariant on the argument type, covariant on the return type).

Finally, rule (SST) accounts for subtyping between security types, which requires facets to be pointwise subtypes.

$$\boxed{\Delta; \Phi \vdash U_1 <: U_2}$$

$$\begin{array}{c}
\text{(SObj)} \frac{O_1 \triangleq \mathbf{Obj}(\alpha).R_1 \quad O_2 \triangleq \mathbf{Obj}(\beta).R_2 \quad \Delta; \Phi, \alpha <: \beta \vdash R_1 <: R_2}{\Delta; \Phi \vdash O_1 <: O_2} \\
\text{(SVar)} \frac{\alpha <: \beta \in \Phi}{\Delta; \Phi \vdash \alpha <: \beta} \quad \text{(SSubEq)} \frac{O_1 \equiv O_2}{\Delta_X; \Phi \vdash O_1 <: O_2} \\
\text{(SGVar1)} \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash X <: B} \quad \text{(SGVar2)} \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash A <: X} \\
\text{(STrans)} \frac{\Delta; \Phi \vdash U_1 <: U_2 \quad \Delta; \Phi \vdash U_2 <: U_3}{\Delta; \Phi \vdash U_1 <: U_3}
\end{array}$$

$$\boxed{\Delta; \Phi \vdash R_1 <: R_2}$$

$$\text{(SR)} \frac{\overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies \Delta; \Phi \vdash M <: M'}{\Delta; \Phi \vdash [\overline{m} : M] <: [\overline{m'} : M']}$$

$$\boxed{\Delta; \Phi \vdash M_1 <: M_2}$$

$$\text{(SM)} \frac{\Delta; \Phi \vdash B' <: B \quad \Delta; \Phi \vdash A <: A' \quad \Delta, X : A'..B'; \Phi \vdash S'_1 <: S_1 \quad \Delta, X : A'..B'; \Phi \vdash S'_2 <: S'_2}{\Delta; \Phi \vdash \langle X : A..B \rangle S_1 \rightarrow S_2 <: \langle X : A'..B' \rangle S'_1 \rightarrow S'_2}$$

$$\boxed{\Delta; \Phi \vdash S_1 <: S_2}$$

$$\text{(SST)} \frac{\Delta; \Phi \vdash T_1 <: T_2 \quad \Delta; \Phi \vdash U_1 <: U_2}{\Delta; \Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 4.2:  $\mathbf{Ob}_{\text{SEC}}^\diamond$ : Subtyping rules

### 4.3.3 Type system

The typing rules of  $\mathbf{Ob}_{\text{SEC}}^\diamond$  appeal to some auxiliary definitions, given in Figure 4.3. Function  $\mathbf{ub}(\Delta, U)$  returns the upper bound of a type  $U$  in the type variable environment  $\Delta$ . Since  $\mathbf{Ob}_{\text{SEC}}^\diamond$  has a top type ( $\mathbf{Obj}(\alpha).[]$ ) this recursive definition of  $\mathbf{ub}$  is well-founded; as in Featherweight Java [30], we assume that  $\Delta$  does not contain cycles. The auxiliary judgment  $\Delta \vdash m \in U$  holds if method  $m$  belongs to type  $U$ . For a type variable, this means that the method is in the upper bound  $\mathbf{ub}(\Delta, X)$ . Function  $\mathbf{msig}(\Delta, U, m)$  returns the polymorphic method signature of method  $m$  in type  $U$ . The rule for type variables looks up the signature in the upper bound. The rule for object types is standard; note that it returns closed type signatures with respect to the self type variable. Finally, the judgment  $\Delta \vdash U \in A..B$  holds if the type  $U$  is a super type of  $A$  and a subtype of  $B$  in the type variable environment  $\Delta$ .

Figure 4.4 presents the typing judgment  $\Delta; \Gamma \vdash e : S$  for  $\mathbf{Ob}_{\text{SEC}}^\diamond$ , which denotes that “ex-

$$\boxed{\text{ub}(\Delta, U) = T}$$

$$\frac{T \neq X}{\text{ub}(\Delta, T) = T} \quad \frac{X : A..B \in \Delta}{\text{ub}(\Delta, X) = \text{ub}(\Delta, B)}$$

$$\boxed{\Delta \vdash m \in U}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). \overline{[m : M]}}{\Delta \vdash m_i \in O} \quad \frac{\Delta \vdash m \in \text{ub}(\Delta, X)}{\Delta \vdash m \in X}$$

$$\boxed{\text{msig}(\Delta, U, m) = M}$$

$$\overline{\text{msig}(\Delta, X, m) = \text{msig}(\_, \text{ub}(\Delta, X), m)}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). \overline{[m : M]}}{\text{msig}(\_, O, m_i) = M [O/\alpha]}$$

$$\boxed{\Delta \vdash U \in A..B}$$

$$\frac{\Delta; \bullet \vdash A <: U \quad \Delta; \bullet \vdash U <: B}{\Delta \vdash U \in A..B}$$

Figure 4.3:  $\mathbf{Obj}_{\text{SEC}}^\diamond$ : Some auxiliary definitions

pression  $e$  has type  $S$  under type variable environment  $\Delta$  and type environment  $\Gamma$ . Note that in our presentation, we omit well-formedness rules for types and environments. They are included in Appendix C.1.4.

The first three typing rules are standard: rule (TVar) types a variable according to the environment, rule (TSub) is the subsumption rule and rule (TObj) types an object. The method definitions of the object must be well-typed with respect to the method signatures taken from the safety type  $T$  of the security type  $S$  ascribed to the self variable  $z$ . For this, the method body  $e_i$  must be well-typed in an extended type variable environment with the type variable  $\Delta, X : A_i..B_i$ , and an extended type environment with the self variable and the method argument.

Rules (TmD) and (TmH) cover method invocation, and account for declassification. The actual argument type  $U'$  must satisfy the variable bounds  $\Delta \vdash U' \in A..B$ . On the one hand, rule (TmD) applies when the method  $m$  is in  $U$  with signature  $\langle X : A..B \rangle S_1 \rightarrow S_2$ ; this corresponds to a use of the object at its declassification interface. Then, the method invocation has type  $S_2$  substituting  $U'$  for  $X$ . On the other hand, rule (TmH) applies when  $m$  is not in  $U$ , but it is in  $T$ ; this corresponds to a use beyond declassification and should raise the security to high. This is why the result type is  $T_2 [X/U'] \triangleleft \top$ . This is all similar to the non-polymorphic rules (Figure 2.4), save for the type bounds check, and the type-level substitution.

$\Delta; \Gamma \vdash e : S$

$$\begin{array}{c}
\text{(TVar)} \frac{x \in \text{dom}(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \text{(TSub)} \frac{\Delta; \Gamma \vdash e : S' \quad \Delta; \bullet \vdash S' <: S}{\Delta; \Gamma \vdash e : S} \\
\\
\text{(TObj)} \frac{S \triangleq T \triangleleft U \quad \text{msig}(\_, T, m_i) = \langle X : A_i..B_i \rangle S'_i \rightarrow S''_i \quad \Delta, X : A_i..B_i; \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}{\Delta; \Gamma \vdash [z : S \Rightarrow \overline{m(x)} e] : S} \\
\\
\text{(TmD)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \in U \quad \text{msig}(\Delta; U, m) = \langle X : A..B \rangle S_1 \rightarrow S_2 \quad \Delta \vdash U' \in A..B \quad \Delta; \Gamma \vdash e_2 : S_1 [U'/X]}{\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : S_2 [U'/X]} \\
\\
\text{(TmH)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \notin U \quad \text{msig}(\Delta; T, m) = \langle X : A..B \rangle S_1 \rightarrow T_2 \triangleleft U_2 \quad \Delta \vdash U' \in A..B \quad \Delta; \Gamma \vdash e_2 : S_1 [U'/X]}{\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : T_2 [U'/X] \triangleleft \top}
\end{array}$$

Figure 4.4:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Static semantics

### 4.3.4 Dynamic semantics

The small-step dynamic semantics of  $\text{Ob}_{\text{SEC}}^{\diamond}$  are standard, given in Figure 4.5. They rely on evaluation contexts and use the auxiliary function  $\text{methimpl}(o.m)$  to lookup a method implementation. Note that types in general, and type variables in particular, do not play any role at runtime.

### 4.3.5 Safety

We first define what it means for a closed expression  $e$  to be *safe*: an expression is safe if it evaluates to a value, or diverges without getting stuck.

**Definition 15** (Safety).  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$

Well-typed  $\text{Ob}_{\text{SEC}}^{\diamond}$  closed terms are safe.

**Theorem 14** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

But of course, type safety is far from sufficient; we want to make sure that well-typed  $\text{Ob}_{\text{SEC}}^{\diamond}$  terms are *secure*. To this end, the next section formalizes the precise notion of security we consider in  $\text{Ob}_{\text{SEC}}^{\diamond}$ , and proves that it is implied by typing.



methimpl( $o, m$ ) =  $x.e$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m(x)e}]}{\text{methimpl}(o, m_i) = x.e_i}$$

$$E ::= [] \mid E.m(e) \mid v.m(E) \quad (\text{evaluation contexts})$$

$$(\text{EMInvO}) \frac{o \triangleq [z : \_ \Rightarrow \_] \quad \text{methimpl}(o, m) = x.e}{E[o.m \langle \_ \rangle (v)] \mapsto E[e[o/z] [v/x] ]}$$

Figure 4.5:  $\text{Ob}_{\text{SEC}}^\diamond$ : Dynamic semantics

## 4.4 Polymorphic relaxed noninterference, formally

We now formally define the security property of *polymorphic type-based relaxed noninterference* (PRNI), and prove that the  $\text{Ob}_{\text{SEC}}^\diamond$  type system soundly enforces PRNI.

### 4.4.1 Logical relation

We define how values, terms and environments are related through a step-indexed logical relation [4] (Figure 4.6). Step-indexing is needed to ensure that the logical relation is well-founded in the presence of recursive object types.

The main novelty of this logical relation with respect that of  $\text{Ob}_{\text{SEC}}$  is that it needs to give an interpretation to polymorphic security types of the form  $T \triangleleft X$ . We do this by quantifying over all possible actual types  $U$  for  $X$  and interpreting  $T \triangleleft U$ . The interpretation of a security type is expressed as sets of *atoms* of the form  $(k, e_1, e_2)$ , where  $k$  is a *step index* meaning that  $e_1$  and  $e_2$  are related for  $k$  steps.

The definition also appeals to a *simple* typing judgment  $\Gamma \vdash_1 e : T$ , which disregards the declassification types, and is therefore standard (Appendix C.1.9). We have that  $\Delta; \Gamma \vdash e : T \triangleleft U \Rightarrow \Gamma \vdash_1 e : T$ . The use of this simple type system in the logical relation clearly separates the definitions of security from its static enforcement by the type system of §4.3.3.

The logical relation uses several auxiliary definitions.  $\text{Atom}_n[T]$  requires  $e_1$  and  $e_2$  to be *simply well-typed* expressions of type  $T$  and the index  $k$  to be strictly less than  $n$ .  $\text{Atom}_n^{\text{val}}[T]$  restricts  $\text{Atom}_n[T]$  to values.  $\text{Atom}[T]$  are atoms of simply well-typed expressions of type  $T$  (*i.e.* for *any* step-index  $k$ ).

The definition of  $\mathcal{NV}[T \triangleleft O]$  relates two objects  $o_1, o_2$  for  $k$  steps if for any method  $m \in O$  and with signature  $\langle X : A..B \rangle S' \rightarrow S''$  and  $j < k$ , given related arguments for  $j$  steps at  $S'$ , invocations of  $m$  produce related results for  $j$  steps at  $S''$ . More specifically, given *any* actual type  $T'$  that satisfies the bounds of the type parameter  $X$  (*i.e.*,  $T' \in A..B$ ) and given related arguments in  $\mathcal{NV}[S' [T'/X]]$  we must obtain related computations in  $\mathcal{NV}[S'' [T'/X]]$ .

The relational interpretation of expressions  $\mathcal{NC}[T \triangleleft U]$  relates atoms of the form  $(k, e_1, e_2)$  that satisfy that for all  $j < k$ , if both expressions  $e_1$  and  $e_2$  reduce to values  $v_1$  and  $v_2$  in at

$$\begin{aligned}
\text{Atom}_n [T] &= \{(k, e_1, e_2) \mid k < n \wedge \vdash_1 e_1 : T \wedge \vdash_1 e_2 : T\} \\
\text{Atom}_n^{\text{val}} [T] &= \{(k, v_1, v_2) \in \text{Atom}_n [T]\} \\
\text{Atom} [T] &= \{(k, e_1, e_2) \in \bigcup_{n \geq 0} \text{Atom}_n [T]\} \\
\mathcal{NV}[T \triangleleft O] &= \{(k, v_1, v_2) \in \text{Atom} [T] \mid \\
&\quad (\forall m \in O. \text{msig}(O, m) = \langle X : A..B \rangle S' \rightarrow S'' \\
&\quad \quad \forall j < k, T', v'_1, v'_2. \vdash T' \wedge T' \in A..B \wedge \\
&\quad \quad (j, v_1, v_2) \in \mathcal{NV}[T \triangleleft O] \wedge (j, v'_1, v'_2) \in \mathcal{NV}[S' [T'/X]] \implies \\
&\quad \quad (j, v_1.m(\_) (v'_1), v_2.m(\_) (v'_2)) \in \mathcal{NC}[S'' [T'/X]])\} \\
\mathcal{NC}[T \triangleleft U] &= \{(k, e_1, e_2) \in \text{Atom} [T] \mid (\forall j < k. (e_1 \mapsto^{\leq j} v_1 \wedge e_2 \mapsto^{\leq j} v_2) \\
&\quad \implies (k - j, v_1, v_2) \in \mathcal{NV}[T \triangleleft U])\} \\
\mathcal{NG}[\cdot] &= \{(k, \emptyset, \emptyset)\} \\
\mathcal{NG}[\Gamma, x : S] &= \{(k, \gamma_1 [x \mapsto v_1], \gamma_2 [x \mapsto v_2]) \mid (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\Gamma] \wedge (k, v_1, v_2) \in \mathcal{NV}[S]\} \\
\mathcal{ND}[\cdot] &= \{\emptyset\} \\
\mathcal{ND}[\Delta, X : A..B] &= \{\sigma [X \mapsto T] \mid \sigma \in \mathcal{ND}[\Delta] \wedge \vdash T \wedge \Delta \vdash T \in A..B\}
\end{aligned}$$

Figure 4.6:  $\text{Ob}_{\text{SEC}}^{\langle \rangle}$  Step-indexed logical relation for type-based equivalence

most  $k$  steps then  $v_1$  and  $v_2$  must be equivalent for the remaining  $k - j$  steps. This definition is termination-insensitive: if one expression does not terminate in less than  $k$  steps, then the two expressions are trivially equivalent.

Type environments have standard interpretations.  $\mathcal{NG}[\Gamma]$  relates value substitutions  $\gamma$ , *i.e.* mappings from variables to closed values, as triples of the form  $(k, \gamma_1, \gamma_2)$ , where  $\gamma_1$  and  $\gamma_2$  are related if they have the same variables as  $\Gamma$ , and for any variable  $x$ , the associated values are related for  $k$  steps at type  $\Gamma(x)$ . Finally, a type substitution  $\sigma$ , *i.e.* a mapping from type variables to closed types, *satisfies* a type variable environment  $\Delta$ , noted  $\mathcal{ND}[\Delta]$ , if it has the same type variables that  $\Delta$  and the mapped type  $T$  is within the type variable bounds.

#### 4.4.2 Defining polymorphic relaxed noninterference

Having defined the logical relation, we can now formally define PRNI. As standard, noninterference properties allow modular reasoning about open terms with respect to (term-level) variables. For PRNI, we extend this modular reasoning principle to open terms with respect to *type* variables. Then, a simply well-typed expression  $e$  under  $\Delta$  and a well-formed  $\Gamma$  satisfies PRNI at well-formed type  $S$ , written  $\text{PRNI}(\Delta, \Gamma, e, S)$ , if for any type substitution  $\sigma$  that satisfies  $\Delta$  and two value substitutions  $\gamma_1$  and  $\gamma_2$  in the relational interpretation of  $\sigma(\Gamma)$ , applying the two value substitutions to the expression  $e$  produces equivalent expressions at type  $\sigma(S)$ . As usual, the definition quantifies universally on the step index  $k$ . We need only consider a single type substitution  $\sigma$ ; indeed, type variables happen only in declassification types, which express the observation power of the public observer. Therefore, for each security type of the form  $T \triangleleft X$  we only need to consider *one* actual type  $U$  within the bounds of  $X$  to pick the observation power of the public observer. The substitution  $\sigma$  captures all these choices.

**Definition 16** (Polymorphic relaxed noninterference).

$$\begin{aligned} \text{PRNI}(\Delta, \Gamma, e, S) &\iff \\ S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \quad \Delta \vdash \Gamma \quad \Delta \vdash S \quad & \\ \forall k \geq 0. \forall \sigma, \gamma_1, \gamma_2. \sigma \in \mathcal{ND}[\Delta]. (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)] & \\ \implies (k, \sigma(\gamma_1(e)), \sigma(\gamma_2(e))) \in \mathcal{NC}[\sigma(S)] & \end{aligned}$$

This definition captures the intuitive security notion that an expression is secure if it produces indistinguishable outputs up to the declassification power of the public observer (specified by  $S$ ), when linked with indistinguishable inputs up to their declassification (specified by  $\Gamma$ ).

### 4.4.3 Security type soundness

To establish that all well-typed  $\text{Ob}_{\text{SEC}}^{\diamond}$  terms satisfy PRNI, we first introduce a notion of logically-related open terms, and prove that if an expression is related to itself, then it satisfies PRNI. We then prove the fundamental property of the logical relation, which states that well-typed terms are logically-related to themselves.

Two open expressions  $e_1$  and  $e_2$  are logically related at type  $S$  in environments  $\Delta$  and  $\Gamma$  if, given a type substitution  $\sigma$  satisfying  $\Delta$  and value substitutions  $\gamma_1$  and  $\gamma_2$  in the relational interpretation of  $\sigma(\Gamma)$ , closing these expressions with the given substitutions produces related expressions related at type  $\sigma(S)$ .

**Definition 17** (Logical relatedness of open terms).

$$\begin{aligned} \Delta; \Gamma \vdash e_1 \approx e_2 : S &\iff \\ \Delta; \Gamma \vdash e_i : S \quad \Delta \vdash \Gamma \quad \Delta \vdash S \quad & \\ \forall k \geq 0. \forall \sigma, \gamma_1, \gamma_2. \sigma \in \mathcal{ND}[\Delta]. & \\ (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)] & \\ \implies (k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC}[\sigma(S)] & \end{aligned}$$

Trivially, if an expression is logically related to itself, then it satisfies PRNI.

**Lemma 15** (Self logical relation implies PRNI).

$$\Delta; \Gamma \vdash e \approx e : S \implies \text{PRNI}(\Delta, \Gamma, e, S)$$

We then turn to proving that all well-typed terms are logically-related to themselves, *i.e.* the fundamental property of the logical relation.

**Theorem 16** (Fundamental property).

$$\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \vdash e \approx e : S$$

*Proof.* The proof is by induction on the typing derivation of  $e$ . We use the common approach of defining compatibility lemmas for each typing rule [4]. Each case follows from the corresponding compatibility lemma. □

Security type soundness follows directly from Theorem 16 and Lemma 15.

**Theorem 17** (Security type soundness).

$$\Delta; \Gamma \vdash e : S \implies \text{PRNI}(\Delta, \Gamma, e, S)$$

Having proven that well-typed  $\text{Ob}_{\text{SEC}}^\diamond$  programs are secure, we are almost ready to revisit the examples of Section 4.2 to illustrate PRNI. We must first address the case of primitive types, discussed next.

## 4.5 Ad-hoc polymorphism for primitive types

Both  $\text{Ob}_{\text{SEC}}$  (Chapter 2) and  $\text{Ob}_{\text{SEC}}^\diamond$  (so far) ignore the case of primitive types, such as integers and strings. However, in an object-oriented language, primitive types are both necessary and challenging from a security viewpoint. In particular, integrating type-based declassification with faceted types requires appealing to a form of *ad hoc* polymorphism.

### 4.5.1 The need and challenge of primitive types

In a pure object-oriented calculus (as in a pure functional calculus) without primitive types, the only real observation that can be made on programs is termination. A termination-insensitive notion of noninterference is therefore useless in a pure setting: one needs some primitive types with a purely syntactic notion of equality. Indeed, all the examples we presented in earlier sections assume a syntactic notion of observation for strings and integers.

Introducing primitive types calls for some form of label polymorphism. Indeed, we do not want to fix the security level of primitive operations, as this would be either impractical for the public observer (if all security labels were high) or for the secret observer (if all security labels were low). This is why practical security-typed languages like FlowCaml [45] and Jif [36] use label-polymorphic primitive operators, specifying that the return label is the *least upper bound* of the argument labels. For instance, a binary integer operator would have type  $\forall \ell_1, \ell_2. \text{Int}_{\ell_1} \times \text{Int}_{\ell_2} \rightarrow \text{Int}_{\ell_1 \sqcup \ell_2}$ . In a monomorphic security language, the same principle is hardcoded in the typing rules for primitive operators [67].

Unfortunately this approach does not work with labels-as-types for objects, even in a label-monomorphic setting. Indeed, because labels are types, returning the join of the argument security labels means computing the *subtyping join* (denoted  $\sqcup_{<}$ ; hereafter) of the declassification types. This is both impractical, incorrect, and potentially unsound from a security viewpoint:

- *Impractical.* Consider a function of type  $\text{Bool} \triangleleft X_1 \times \text{Int} \triangleleft X_2 \rightarrow \text{Bool} \triangleleft (X_1 \sqcup_{<} X_2)$ . Given two public arguments (*i.e.*  $X_1 = \text{Bool}$ ,  $X_2 = \text{Int}$ ), then assuming  $\text{Bool} \sqcup_{<} \text{Int} = \top$ , the result is necessarily private. While sound, this is far too conservative; it is impractical for primitive operations to always return secret values even when given public inputs.
- *Incorrect.* Consider an integer comparison of type  $\text{Int} \triangleleft X_1 \times \text{Int} \triangleleft X_2 \rightarrow \text{Bool} \triangleleft (X_1 \sqcup_{<} X_2)$ . If we instantiate this signature with  $\text{Int}$  and  $\text{Int}$  we obtain an *ill-formed* return type,  $\text{Bool} \triangleleft \text{Int}$ .

- *Insecure.* Consider a unary integer operator  $\text{Int} \triangleleft X \rightarrow \text{Int} \triangleleft X$ ; this signature is not sound security-wise for all unary integer operators. Take an operator that trims the most-significant bit of its argument. If one declassifies only the parity of the argument, two equivalent inputs will not always yield two equivalent outputs (as the parity of the trimmed values might differ).

## 4.5.2 Sound signatures for primitive types

The observations above reveal one of the flip sides of expressive declassification policies: because declassification is captured semantically, declassification polymorphism is a very strong notion compared to standard label polymorphism. In the general case, without appealing to intricate semantic conditions, there are therefore only two simple syntactic principles to define sound signatures for primitive operators:<sup>2</sup>

- (P1) if every argument is public (*e.g.*  $\text{String}_L$ ) then the return type can be public.  
(P2) if any argument is not public (*e.g.*  $\text{String} \triangleleft \text{StringFst}$ ) then the return type must be secret (*e.g.*  $\text{String}_H$ ).

As is typical, we provide an object-oriented interface for primitive types (*e.g.*  $a + b$  is  $a.+(b)$  as in Scala for instance). Therefore the principles above must be extended to account for the status of the *receiver* object: if the primitive method invoked on the primitive value is part of its declassification type, then it is considered a public “argument”; otherwise, it is private.

Note that, without any form of polymorphism, *i.e.* picking a single syntactic principle above, primitive types would be impractical. Duplicating all definitions to offer both options is also not a viable approach.

## 4.5.3 Polymorphic primitive signatures

To support the two syntactic principles exposed above, we propose to use *ad-hoc* polymorphism (akin to overloading) for primitive types  $P$  in  $\text{Ob}_{\text{SEC}}^\diamond$ . We introduce *polymorphic primitive signatures*, written  $P \triangleleft * \rightarrow P \triangleleft *$ . A primitive security type  $P \triangleleft *$  is resolved polymorphically *at use site*, following principles (P1) and (P2) above. Object-oriented interfaces for primitive types are exclusively composed of polymorphic primitive signatures. For instance, in  $\text{Ob}_{\text{SEC}}^\diamond$  strings are primitives, declared by the following  $\text{String}$  type:

$$\text{String} \triangleq [ \text{concat} : \text{String} \triangleleft * \rightarrow \text{String} \triangleleft *, \\ \text{first} : \text{Unit} \triangleleft * \rightarrow \text{String} \triangleleft *, \\ \text{length} : \text{Unit} \triangleleft * \rightarrow \text{Int} \triangleleft *, \\ \text{eq} : \text{String} \triangleleft * \rightarrow \text{Bool} \triangleleft *, \\ \dots ]$$

To illustrate, assume  $a : \text{String}_L$ ,  $b : \text{String}_L$  and  $c : \text{String}_H$ . Then  $a.\text{eq}(b)$  has type  $\text{Bool}_L$ , while  $a.\text{eq}(c)$  has type  $\text{Bool}_H$ .

<sup>2</sup>The syntactic principles (P1) and (P2) are formally justified by the proof of Lemma 18, discussed in Section 4.5.5.

$e$	::= $\dots \mid e.m(e)$	(terms)
$v$	::= $\dots \mid \mathbf{b}$	(values)
$T$	::= $\dots \mid P$	(types)
$M$	::= $\dots \mid I$	(method signatures)
$S$	::= $\dots \mid P \triangleleft *$	(security types)
$I$	::= $P \triangleleft * \rightarrow P \triangleleft *$	(primitive signatures)
$P$	::= (eg. Int, String)	(primitive types)
$\Phi$	::= $\dots \mid \Phi, P \triangleleft: \beta$	(subtyping environments)

Figure 4.7:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Extended syntax for primitive types

Primitive types can also be subject to declassification policies. For instance, consider:

$$\text{StringEqPoly} \triangleq [\text{eq} : \text{String} \triangleleft * \rightarrow \text{Bool} \triangleleft *]$$

and  $d : \text{String} \triangleleft \text{StringEqPoly}$ . Then  $d.\text{eq}(b)$  has type  $\text{Bool}_L$ , while  $d.\text{concat}(a) : \text{String}_H$ .

Furthermore, one can use any type signature in a declassification policy for a primitive type, as long as it is sound. For instance,  $\text{StringEqL} \triangleq [\text{eq} : \text{String}_L \rightarrow \text{Bool}_L]$  respects principle 1). Conversely,  $\text{StringEqBad} \triangleq [\text{eq} : \text{String}_H \rightarrow \text{Bool}_L]$  cannot be used as it would violate the soundness principles above (in  $\text{Ob}_{\text{SEC}}^{\diamond}$ ,  $\text{String} \triangleleft \text{StringEqBad}$  is ill-formed).

#### 4.5.4 Formal semantics

We now formalize the treatment of primitive types in  $\text{Ob}_{\text{SEC}}^{\diamond}$ . Figure 4.7 presents the extended syntax to support primitive values  $\mathbf{b}$  and primitive types  $P$ . Expression  $e.m(e)$  is for method invocation on primitives; as explained previously, primitive types expose an object-oriented interface, so  $a + b$  is  $a.+(b)$ . We extend the category  $S$  with security types of the form  $P \triangleleft *$  and introduce a new category  $I$ , for primitive signatures  $P \triangleleft * \rightarrow P \triangleleft *$ . The security type  $P \triangleleft *$  can be used for standard signatures  $\langle X : A..B \rangle_{S_1} \rightarrow S_2$  as well.

The changes to subtyping are in Figure 4.8. Now, subtyping assumptions can be also made between a primitive type and a self type variable, *i.e.*  $\Phi ::= \bullet \mid \Phi, \alpha \triangleleft: \beta \mid \Phi, P \triangleleft: \beta$ , and function `meths` returns the methods of a primitive type. Rule (SPObj) justifies subtyping between a primitive type and an object type, and it is very similar to rule (SObj) of Figure 4.2 for object types. Rule (SPVar) accounts for subtyping between primitive types and type variables and it holds if such subtyping relation exists in the subtyping environment  $\Phi$ . Note that there is no rule for subtyping between an object type and a primitive type, because this would not be sound. Rule (SImpl) accounts for subtyping between the same primitive signature. There is no rule to justify subtyping between a primitive signature and a standard signature.

As we discussed at the end of Section 4.5.2, we need an extra condition for the well-formedness of security types to ensure sound declassification. Given the type  $T \triangleleft U$ , if  $T$  has a method  $m : I$ , the method signature of  $m$  in  $U$  must be either the same primitive

$$\boxed{\Delta; \Phi \vdash U_1 <: U_2}$$

$$\dots \quad (\text{SPObj}) \frac{\text{meths}(P) = R_1 \quad O \triangleq \mathbf{Obj}(\beta).R_2 \quad \Delta; \Phi, P <: \beta \vdash R_1 <: R_2}{\Delta; \Phi \vdash P <: O}$$

$$(\text{SPVar}) \frac{P <: \beta \in \Phi}{\Delta; \Phi \vdash P <: \beta}$$

$$\boxed{\Delta; \Phi \vdash M_1 <: M_2}$$

$$\dots \quad (\text{SImpl}) \frac{}{\Delta; \Phi \vdash I <: I}$$

Figure 4.8:  $\mathbf{Ob}_{\text{SEC}}^\diamond$ : Subtyping rules for primitive types

signature  $I$ , or a normal signature that is *sound*. We use the predicate `soundsig` to express that signature  $\langle X : A..B \rangle S_1 \rightarrow S_2$  is sound, which must satisfy that either the argument type is public, or the return type is private:

$$\text{soundsig}(\langle \_ \rangle P \triangleleft U_1 \rightarrow T_2 \triangleleft U_2) \iff U_1 = P \vee U_2 = \top$$

Figure 4.9 presents the extension to the typing rules of  $\mathbf{Ob}_{\text{SEC}}^\diamond$ . Rule (TPrim) justifies typing for primitive values, using a function  $\Theta$  that specifies each primitive type. The new typing rules (TPmD) and (TPmH) realize ad hoc polymorphism for primitive types. Rule (TPmD) is key: it applies when  $m$  is in the declassification type  $U$ , and uses the function `retlab` to calculate the declassification type of the return type, based on the type of the argument: if the argument is public, so is the returned value. Rule (TPmH) applies when  $m$  is not in the declassification type  $U$ , and similarly to (TmH), ensures that the returned value is private.

Figure 4.10 shows the extension to the dynamic semantics to support primitive values. Rule (EMInvP) executes a method invocation on a primitive value using the function  $\theta$ , which abstracts over the internal implementation of primitive values.

To prove type safety for  $\mathbf{Ob}_{\text{SEC}}^\diamond$  with primitives, we only need to assume that  $\Theta$  and  $\theta$ —which are parameters of the language—agree on the specification and implementation of all primitive types and their operations.

### 4.5.5 Logical relation for primitive types

Figure 4.11 presents the extension to the logical relation of Figure 4.6 to account for primitive types. First,  $\mathcal{NV}[[P \triangleleft P]]$  relates *syntactically equal* primitive values of type  $P$ . Second, the definition of  $\mathcal{NV}[[T \triangleleft O]]$  now accounts for primitive values that are observed with a declassification type  $O$ .  $\mathcal{NV}[[T \triangleleft O]]$  still relates values  $v_1$  and  $v_2$  if, for all methods of  $O$ , given related arguments, the invocations of  $m$  on  $v_1$  and  $v_2$  produce related computations. However, the definition now discriminates between each type of signatures. For a method  $m$  with primitive signature  $P_1 \triangleleft * \rightarrow P_2 \triangleleft *$ , we require one of the following conditions to hold.

$\Delta; \Gamma \vdash e : S$

$$\begin{array}{c}
\cdots \quad (\text{TPrim}) \frac{P = \Theta(\mathbf{b})}{\Delta; \Gamma \vdash \mathbf{b} : P \triangleleft P} \\
\\
\begin{array}{c}
\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \in U \\
\text{msig}(\Delta, U, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft * \\
\Delta; \Gamma \vdash e_2 : P_1 \triangleleft U_1 \\
\text{retlab}(P_1 \triangleleft U_1, P_2) = P'_2
\end{array} \\
(\text{TPmD}) \frac{}{\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft P'_2} \\
\\
\begin{array}{c}
\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \notin U \\
\text{msig}(\Delta, T, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft * \\
\Delta; \Gamma \vdash e_2 : P_1 \triangleleft U_1
\end{array} \\
(\text{TPmH}) \frac{}{\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft \top}
\end{array}$$

$\text{retlab}(P_1 \triangleleft U_1, P_1) = U$

$$\text{retlab}(P_1 \triangleleft U_1, P_2) = \begin{cases} P_2 & P_1 = U_1 \\ \top & \text{otherwise} \end{cases}$$

Figure 4.9:  $\text{Ob}_{\text{SEC}}^\diamond$ : Extended static semantics for primitive types

$$(\text{EMInvP}) \frac{}{E[\mathbf{b}_1.m(\mathbf{b}_2)] \mapsto E[\theta(m, \mathbf{b}_1, \mathbf{b}_2)]}$$

Figure 4.10:  $\text{Ob}_{\text{SEC}}^\diamond$ : Dynamic semantics of primitive values

If we get related arguments  $v'_1$  and  $v'_2$  at  $P_1 \triangleleft P_1$  (*i.e.* public values), method invocations  $v_1.m(v'_1)$  and  $v_2.m(v'_2)$  need to be related at the declassification type  $P_2 \triangleleft P_2$ . Otherwise, if the arguments  $v'_1$  and  $v'_2$  are related at a non-public type ( $P_1 \triangleleft U$ ,  $P_1 \neq U$ ), then  $v_1.m(v'_1)$  and  $v_2.m(v'_2)$  need to be related at the top type  $P \triangleleft \top$ . These conditions are expressed in the definition by requiring related method invocations in  $\mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$ .

Extending the fundamental property (Lemma 16) for primitive types requires the following lemma, which states that syntactically-equal primitive values of type  $P$  are in the object-oriented interpretation of any type  $P \triangleleft O$ —essentially, equal values cannot be discriminated.

**Lemma 18** (Equal values are logically related).

$\forall k \geq 0, \mathbf{b}, P, O.$

$$\vdash_1 \mathbf{b} : O \wedge P \triangleleft O \implies (k, \mathbf{b}, \mathbf{b}) \in \mathcal{NV}[[P \triangleleft O]]$$

*Proof.* Because  $P \triangleleft O$  is a well-formed security type,  $O$  consists of primitive signatures and standard, sound signatures. For the case of primitive signatures, at some point we have  $P_1 \triangleleft * \rightarrow P_2 \triangleleft *$  and equivalent values at  $(j, v_1, v_2) \in P_1 \triangleleft U_1$  and we have to prove that  $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$ . We do case analysis on  $U$ . If  $U_1 =$



$$\begin{aligned}
\mathcal{NV}\llbracket P \triangleleft P \rrbracket &= \{(k, \mathbf{b}, \mathbf{b}) \in \text{Atom}[P]\} \\
\mathcal{NV}\llbracket T \triangleleft O \rrbracket &= \{(k, v_1, v_2) \in \text{Atom}[T] \mid \dots \\
&\quad \forall m \in O. \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft * \\
&\quad \forall j < k, v'_1, v'_2. U_1 >: P_1 \\
&\quad ((j, v'_1, v'_2) \in \mathcal{NV}\llbracket P_1 \triangleleft U_1 \rrbracket \implies \\
&\quad (j, v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}\llbracket P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2) \rrbracket)\}
\end{aligned}$$

Figure 4.11:  $\text{Ob}_{\text{SEC}}^\diamond$ : Step-indexed logical relation with new definitions for primitive types

$P_1$ , we know that  $v_1 = v_2$  and hence if  $\delta(m, b, v_1)$  and  $\delta(m, b, v_2)$  are defined, their results are syntactically equal, so  $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{NC}\llbracket P_2 \triangleleft P_2 \rrbracket$ . If  $U \neq P_1$ , then the proof obligation is  $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{NC}\llbracket P_2 \triangleleft \top \rrbracket$ ; this is trivial because any two values are related at  $\top$ . For the case of standard signatures, at some point we have  $P_1 \triangleleft U_1 \rightarrow P_2 \triangleleft U_2$  and we have to prove that  $(j, \delta(m, b, v_1), \delta(m, b, v_2)) \in \mathcal{NC}\llbracket P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2) \rrbracket$ . Since the signature is sound, we know that either  $U_1 = P_1$  or  $U_2 = \top$ ; then the result follows similarly to the primitive signature case.  $\square$

Note that the two syntactic principles for sound signatures of primitive types introduced in Section 4.5.2 are justified by the need to establish Lemma 18. Principle (P1) is necessary because we cannot assume anything about two invocations of an *arbitrary* partial function  $\delta$ , except that given *syntactically* equal arguments, if it produces results, then those results are syntactically equal. Principle (P2) is justified because any two invocations of  $\delta$  are observationally equivalent at  $\top$  (like any computation in general), so the actual relation between the arguments does not matter. For any primitive operator signature that does not abide by either (P1) or (P2), it is possible to devise a  $\delta$  that violates Lemma 18

Consequently,  $\text{Ob}_{\text{SEC}}^\diamond$  with primitive types is a sound security-typed language, *i.e.* all well-typed programs satisfy PRNI (Theorem 17).

## 4.5.6 Illustration

In Section 4.2 we gave informal examples of secure and insecure programs with respect to PRNI. Now, armed with Theorem 17, and the definitions for primitive types, we can formally check if a given program is secure by typechecking it. The prototype implementation of  $\text{Ob}_{\text{SEC}}^\diamond$  features a number of examples and allows one to try out the language and typechecker. In this section, we unfold the reasoning underlying the proof of Theorem 17 for a specific example, in order to illustrate the technical details of PRNI and the relational interpretation of object types, including primitive signatures.

To lighten notation, we omit unused type parameters in method declarations and type instantiations in method invocations. Note that we introduce the `Unit` type with its unique unit primitive value.

We illustrate polymorphic declassification by considering type and variable environments  $\Delta \triangleq X : \text{String}.. \text{StringLen}$  and  $\Gamma \triangleq x : \text{String} \triangleleft X$ . We discuss two possible formal definitions for `StringLen`, either using standard method signatures, or using primitive signatures:

1.  $\mathbf{Obj}(\alpha). [\text{length} : \mathbf{Unit}_L \rightarrow \mathbf{Int}_L]$
2.  $\mathbf{Obj}(\alpha). [\text{length} : \mathbf{Unit} \triangleleft * \rightarrow \mathbf{Int} \triangleleft *]$

With definition 1) above, the program  $x.\text{length}(\text{unit})$  has type  $\mathbf{Int}_L$ ; *i.e.*  $\Delta; \Gamma \vdash x.\text{length}(\text{unit}) : \mathbf{Int}_L$ . Then, by Theorem 17, we know that  $\text{PRNI}(\Delta, \Gamma, x.\text{length}(\text{unit}), \mathbf{Int}_L)$  holds; the program is secure for any public observer.

Let us unfold  $\text{PRNI}(\Delta, \Gamma, x.\text{length}(\text{unit}), \mathbf{Int}_L)$  to verify why it holds. For any type substitution  $X \mapsto T \in \mathcal{ND}[\Delta]$  and equivalent value substitutions  $(k, x \mapsto v_1, x \mapsto v_2) \in \mathcal{NG}[\bullet, x : \mathbf{String} \triangleleft T]$ , we have that  $(k, v_1.\text{length}(\text{unit}), v_2.\text{length}(\text{unit})) \in \mathcal{NC}[\mathbf{Int} \triangleleft \mathbf{Int}]$ .

To verify this:

1. By  $(k, x \mapsto v_1, x \mapsto v_2) \in \mathcal{NG}[\bullet, x : \mathbf{String} \triangleleft T]$  we know that  $(k, v_1, v_2) \in \mathcal{NV}[\mathbf{String} \triangleleft T]$ . Because  $T <: \mathbf{StringLen}$ , we have  $\mathcal{NV}[\mathbf{String} \triangleleft T] \subseteq \mathcal{NV}[\mathbf{String} \triangleleft \mathbf{StringLen}]$  by a subtyping lemma, and hence  $(k, v_1, v_2) \in \mathcal{NV}[\mathbf{String} \triangleleft \mathbf{StringLen}]$ .
2. Then, instantiate the definition of  $\mathcal{NV}[\mathbf{String} \triangleleft \mathbf{StringLen}]$  with  $\text{length}, k, T, \text{unit}, \text{unit}$ . Note that:
  - $\text{length} \in \mathbf{StringLen}$
  - $\text{msig}(\bullet, \mathbf{StringLen}, \text{length}) = \mathbf{Unit}_L \rightarrow \mathbf{Int}_L$
  - $T \in \mathbf{String}..\mathbf{StringLen}$ , which follows from  $X \mapsto T \in \mathcal{ND}[\Delta]$
  - and  $(k, \text{unit}, \text{unit}) \in \mathcal{NV}[\mathbf{Unit} \triangleleft \mathbf{Unit}]$  (by definition of  $\mathcal{NV}[P \triangleleft P]$ ),

Then  $(k, v_1.\text{length}(\text{unit}), v_2.\text{length}(\text{unit})) \in \mathcal{NC}[\mathbf{Int} \triangleleft \mathbf{Int}]$ .

With definition 2) above of  $\mathbf{StringLen}$ , we apply the same steps until the instantiation of  $\mathcal{NV}[\mathbf{String} \triangleleft \mathbf{StringLen}]$ . At this point, since  $\text{length}$  has a primitive signature, we have to consider the extended case for primitive type signatures from Figure 4.11. Instantiate it with  $\text{length}, k, \text{unit}, \text{unit}$ , and observe that  $\text{msig}(\bullet, \mathbf{StringLen}, \text{length}) = \mathbf{Unit} \triangleleft * \rightarrow \mathbf{Int} \triangleleft *$ . Then, given that  $(k, \text{unit}, \text{unit}) \in \mathcal{NV}[\mathbf{Unit} \triangleleft \mathbf{Unit}]$ , we have that  $(k, v_1.\text{length}(\text{unit}), v_2.\text{length}(\text{unit})) \in \mathcal{NC}[\mathbf{Int} \triangleleft \text{retlab}(\mathbf{Unit} \triangleleft \mathbf{Unit}, \mathbf{Int})] = \mathcal{NC}[\mathbf{Int} \triangleleft \mathbf{Int}]$ .

## 4.6 Declassification polymorphism and existentials

So far in this chapter we have developed declassification polymorphism for the object-oriented labels-as-types approach, including the case of primitive types. In this section, we briefly comment on declassification polymorphism for the existential labels-as-types approach (Chapter 3). First, we note that the existential labels-as-types approach enjoys a limited form of declassification polymorphism. Second, we explain that for this form of declassification polymorphism, primitive types do not need any special treatment as was required for the object-oriented labels-as-types approach.

**Declassification polymorphism for the clients of a package** The existential labels-as-types approach uses existential types to express declassification policies. Existential types are closely related to universal types  $\forall X.T$ , the foundations of type polymorphism for functional languages. In particular, the client of a package of type  $\exists X.T$  must be polymorphic with

respect to the type  $X$ , modulo the type interface  $T$ . Then, we can use this type polymorphism of the client code with respect to the type variable  $X$  and the operations in  $T$  to express declassification polymorphism.

To illustrate the above point, we implement a polymorphic `contains` function, similarly to what we did in § 4.2.2. For that goal, we use a standard builtin polymorphic `List[T]` type constructor for functional languages and the standard operations for lists [41]:

- `nil[T]`: list constructor for the empty list.
- `cons[T]`: list constructor for a non-empty list. It takes an element of type  $T$  (the *head* element) and a list of type `List[T]` (the *tail* list).
- `isNil`: returns true if the list is empty, false in other case.
- `head`: returns the element at the head of the list. It fails for the empty list.
- `tail`: returns the tail list. It fails for the empty list.

We also reuse the type `AccountStore` of § 3.2 and we add a new operation: a function `eq` that returns a boolean value indicating if two secret passwords are equal. We omit faceted types here, since the point can be explained without them.

$$\text{AccountStore} \triangleq \exists X. [ \text{userPass} : \text{String} \rightarrow X \\ \text{eq} : X \rightarrow X \rightarrow \text{Bool} \\ \dots ]$$

Then, with a package `p` of type `AccountStore`, we can implement a polymorphic declassification function `contains`, which given a list and an element of type  $X$  returns whether or not the element is in the list:

```
open(X, store) = p in
...
Bool contains(List[X] l, X search){
  if(isNil(l)) false
  else
    if(store.eq(search, head(l))) true
    else contains(tail(l), search)
}
```

The function `contains` is polymorphic with respect to  $X$  up to the operations provided by `AccountStore`. In particular, it makes uses of the operation `eq` to verify that the element to search is in the list: `store.eq(search, head(l))`.

There are a few things worth noting in the definition of this function `contains`<sup>3</sup> with respect to the function `contains` of § 4.2.2, implemented for the object-oriented labels-as-types approach:

- In this function `contains` the variable  $X$  is already universally quantified because the `open` expression introduces it. Therefore, the client code is polymorphic with respect

---

<sup>3</sup>The implementation of `contains` is recursive and our  $\lambda_{\text{SEC}}^{\exists}$  model does not include recursive features such as recursive functions (`fix`) or recursive types. We could illustrate the concept of declassification polymorphism without recursion, however, it is more relevant to use the function `contains`, similarly as we did in § 4.2.2

to  $X$ . That is, the function `contains` does not introduce the variable  $X$ , as happens in the object-oriented labels-as-types approach.

- Here, to verify that two elements with the declassification type  $X$  are equal, we use the operation `eq` provided by the type `AccountStore`, while in the `contains` function of § 4.2.2 we use the `eq` method of `String` exposed via subtyping with the bounds  $X : \text{String}.. \text{StringEq}$ . Therefore, the operations of the existential type can be seen as “upper bound for the type  $X$ ” with respect to declassification polymorphism.

**Declassification polymorphism for primitives types** Finally, we explain why primitive types do not require special considerations for this form of declassification polymorphism. The main question of declassification polymorphism for primitive types is: what is the resulting declassification type of an operation *computing* a result on (possibly multiple) primitive values? As explained in § 4.5, in Jif, the resulting label of a primitive operator is the upper bound of the labels of its arguments. For the object-oriented label-as-types approach there are two fundamental principles (§ 4.5.2). For the existential labels-as-types approach, there is no dependency between the declassification type of the arguments and the declassification type of the result.

With existential types, declassification-polymorphic operations, *i.e.* operations that take abstract types and return abstract types, can be defined and implemented in two places:

1. In the package that provides the existential type. Here, the package implementation can treat values of the abstract types as values of the corresponding implementation types and thereby to compute over these values.
2. In the client code, polymorphic operations are implemented over operations of the existential type (as is the case of the function `contains` above) or they do not compute with values of the abstract type, *e.g.* the identity function.

Hence, the existential type interface explicitly defines the type of a polymorphic operation that computes over primitive values and thereby the resulting declassification type does not depend on the declassification types of the arguments (as it happens in Jif and  $\text{Ob}_{\text{SEC}}^{\diamond}$ ).

To illustrate the above observations, we use the following type `Declassifier` that provides a polymorphic operation `sum` over two secrets numbers, obtained with `n1` and `n2` at the types `Int@X` and `Int@Y`, respectively:

$$\text{Declassifier} \triangleq \exists X.Y.[ \text{n1} : \rightarrow \text{Int}@X \quad \text{n2} : \rightarrow \text{Int}@Y \quad \text{sum} : \text{Int}@X \rightarrow \text{Int}@Y \rightarrow \text{Int}@X \dots ]$$

Note that we *decide* to provide the result type of the operation `sum` as `Int@X`. However, we can also provide it as type `Int@Y`. That is, the type signature of the declassification-polymorphic operation is up to the person that defines the existential type interface. Based on the operation `sum`, we can implement other declassification-polymorphic operations in the client code. If we do not use the operation `sum`, then the declassification-polymorphic operation cannot compute over its arguments, for instance `id : Int@X → Int@X = λx : Int@X. x`.

In summary, the initial decision of making a computable operation on primitive types

declassification-polymorphic is at the existential type; hence, declassification polymorphism for primitive types do not need a different treatment with respect to user-defined types.

## 4.7 Related work

As noted by Sabelfeld and Sands [52], many declassification approaches of the *what* dimension can be expressed using partial equivalence relations to model the public observer knowledge. Here, we use the logical interpretation of security types (Figure 4.6) to specify the partial equivalence relation that a public observer can use to distinguish values and computations.

**Label polymorphism.** Support for label polymorphism in security-typed programming languages can be classified in two categories: static and dynamic label polymorphism. Static label polymorphism can either be provided via explicit syntactic constructions to introduce generic labels [36], or implicitly with constraint-based label inference [36, 45, 56]. The dynamic form of label polymorphism relies on labels as first-class entities that can be passed around like standard values [36, 54, 55].

The Jif language [36] supports all three forms of label polymorphism. It provides a direct syntax to introduce labels at the method and class levels, which can be constrained. Also, Jif features label inference: local variables are inferred to have a fresh generic label that is resolved using constraints from the context. Inferred fields and method arguments have default labels. In addition, Jif supports first-class labels. Our work focuses on the foundations of explicit declassification polymorphism, and currently does not address label inference and first-class labels. Because labels are types, label inference would boil down to fairly standard type inference; first-class labels however would require a notion of first-class types, which should be considered with care.

Sun et al. [56] design a constraint-based label inference mechanism for an object-oriented language with classes and inheritance. Classes and methods are label polymorphic. The programmer can rely on the inference mechanism to achieve label polymorphism or to specify generic labels at the class level; method-level explicit polymorphism is not considered. Stefan et al. [54, 55] provide label-polymorphism via first-class labels much like Jif.

**Declassification and Polymorphism.** When present, the declassification mechanisms of the label-polymorphic proposals discussed above [36, 45, 56, 55] are completely orthogonal to label polymorphism. The polymorphic labels-as-types approach developed in chapter allows us to reason about declassification and label polymorphism with the single and unified concept of standard types.

Our approach is closely related to that of Hicks et al. [29], which propose *trusted declassification* in an object-oriented language based on the DLM [37], where each label is composed of principals. Declassification is globally defined, associating principals to the *trusted methods* that can be used to declassify an expression to another principal. Because classes are polymorphic with respect to principals, this induces a form of implicit label polymorphism. More precisely, a class definition is checked at instantiation-time with the actual principals provided for the instantiation. This use-site polymorphism for principals is similar to our treatment of polymorphic primitive signatures (Section 4.5).

Tse and Zdancewic [60] propose certificate-based declassification and conditioned noninterference. They extend System  $F_{<}$  with monadic labels similarly to DCC [2], using DLM [37]. Declassification is modeled as a *read privilege* that a principal is allowed to give to another principal in a certain context. Their work merges standard types with labels, principals and privileges in the same syntactic category of types. Since System  $F_{<}$  supports type polymorphism, the language supports label polymorphism. However, it is not clear how to use label polymorphism to express polymorphic declassification in that setting.

Finally, the syntactic principles we introduce for primitive signatures are related to the work of Li and Zdancewic [31] on labels-as-functions. For local policies, the typing rules for integer primitive operators follow the same principles, but are more expressive. In particular, they provide typing rules for binary integer operators where one operand has an arbitrary declassification policy and the other operand is public; the resulting label is a functional composition of the operand label with a function wrapping the operator. As explained before, this semantic implication cannot be expressed with the labels-as-types approach, unless one is willing to consider much more advanced typing disciplines.

## 4.8 Conclusion

We extend relaxed noninterference for object-oriented labels-as-types approach to selective and expressive declassification in order to account for polymorphism. The proposed declassification polymorphism is novel and useful to precisely control declassification of polymorphic structures and to define procedures that are polymorphic over the declassification policies of their arguments. Bounded polymorphism further controls the guarantees and expectations of clients and providers with respect to declassification. Bringing type-based declassification to real-world programming also requires addressing the issue of primitive types, which were ignored in prior work. For this we introduce a novel form of ad-hoc polymorphism. We formalize the approach, prove its soundness, and provide a prototype implementation. This chapter provides a necessary and solid basis to integrate object-oriented type-based declassification in existing object-oriented languages.

# Chapter 5

## Conclusion and perspectives

In this thesis we presented a novel approach for declassification in information-flow type systems, which exploits familiar notions of type abstraction to model expressive declassification policies. We developed this labels-as-types approach in two settings: object-oriented languages where type abstraction is driven by subtyping ( $\text{Ob}_{\text{SEC}}$ ); and functional languages where type abstraction is driven by existential types ( $\lambda_{\text{SEC}}^{\exists}$ ). Furthermore, we developed declassification polymorphism for the object-oriented labels-as-types approach ( $\text{Ob}_{\text{SEC}}^{\diamond}$ ), and highlighted how the functional labels-as-types approach enjoys a form of declassification polymorphism. Therefore, this thesis provides the necessary foundations to integrate the labels-as-types approach in existing language infrastructures.

The labels-as-types approach features faceted types, which are necessary to express the views of a value for a privileged observer and a public observer. As we have extensively discussed, without faceted types, we can still exploit type abstraction to capture declassification policies, however, the approach only accounts for the view of the public observer and supporting different views of a value is a standard feature of information-flow type systems.

This thesis has opened new future research directions that we highlight below.

**Implementing type-based declassification in languages.** A first research direction is to implement the labels-as-types approach in a full-fledged programming language. For that, we must model faceted types, to account for security levels, and to implement the additional typing rules for elimination of secrets : the rule (TmH) of the  $\text{Ob}_{\text{SEC}}$  type system or the rules that use the auxiliary function  $[S_1]_{S_2}$  in the  $\lambda_{\text{SEC}}^{\exists}$  type system.

A particularly appealing candidate language is Scala: its type system is expressive enough to encode faceted types and the special semantics of the elimination rules for faceted types should be achievable via a compiler plugin. Scala is also a good target because it features all the type abstraction mechanisms exploited by the different realization of labels-as-types. Then, another interesting venue for future work is to study the combination of the functional approach with the object-oriented approach, thereby bridging the gap towards a practical implementation in a language like Scala.

Another research direction with practical implications is to explore how to *infer* the minimal knowledge that has to be exposed to a public observer in order to guarantee a relaxed noninterference at a given type. Inferring the minimal input declassifications of a secure program can for instance be useful to assess the impact of some refactoring or change has on security [33]

**Type-based declassification, authorization and integrity.** In this thesis, we have proposed an expressive mechanism for declassification policies related to *what* is declassified about a secret. However, a practical information flow control language needs to support other forms of declassification, as well as authorization and integrity policies. Therefore, an interesting research direction is to integrate our type-based approach for declassification with other approaches that handle authorization and integrity policies, such as the Decentralized Label Model (DLM) of Jif.

As introduced in Chapter 1, labels in Jif are formed of two components, one for the confidentiality policy and one for the integrity policy. The confidentiality policy defines the principals that see a value. Declassification in the context of Jif means allowing *external* principal to see that value and can be done arbitrarily (except that the integrity policy is used to verify that the declassification is not affected by an attacker). One idea to integrate our approach with the DLM is to extend the DLM labels to triplets, where the new component is a type (facet) expressing the declassification policy. Then, the declassification policy represents the view that an external principal has of a value. It should then be possible to enforce that any use of the `declassify` operator respects the declassification policy defined by the third component, thereby disabling arbitrary declassification.

**Expressiveness of declassification policies.** A last open research direction of this thesis is to explore how adding more expressiveness to types leads to more expressiveness in the policies that we can express. This thesis partially explored this direction. For instance, choosing objects, as opposed to records in `ObSEC`, allowed us to explore recursive declassification policies from the start, given that the essence of data abstraction in OOP are recursive types [18]. Also, using existential types as opposed to object types, allowed us to express extrinsic declassification policies. Beyond the typing disciplines covered in this thesis, it is relevant to study the impact of more advanced typing disciplines on the expressiveness of type-based declassification, especially refinement types [66, 62] and dependent types. Refinement types, as found in *e.g.* LiquidHaskell [62], enrich standard types with predicates over a decidable logic. Additionally, refinement types usually support a form of dependent types, allowing refinements to refer to variables in scope as well as function arguments. Combining such expressive types with our approach allows interesting declassification policies to be defined, such as restricting successive arguments of a progressive declassification. Dependent types enable computations at the level of types, therefore opening many possibility to express highly-specific declassification policies as types.



# Bibliography

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [2] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In POPL 1999 [44], pages 147–160.
- [3] A. Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.
- [4] A. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In P. Sestoft, editor, *Proceedings of the 15th European Symposium on Programming Languages and Systems (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83, Vienna, Austria, Mar. 2006. Springer-Verlag.
- [5] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.
- [6] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (S&P 2007)*, pages 207–221, Berleley/Oakland, California, USA, May 2007. IEEE Computer Society Press.
- [7] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In S. Chong and D. A. Naumann, editors, *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security (PLAS 2009)*, pages 113–124, Dublin, Ireland, June 2009. ACM Press.
- [8] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 165–178, Philadelphia, USA, Jan. 2012. ACM Press.
- [9] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Sept. 2005.
- [10] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the 29th IEEE Symposium on Security*

*and Privacy (S&P 2008)*, Oakland, California, USA, May 2008. IEEE Computer Society Press.

- [11] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In CSF 2008 [23], pages 83–97.
- [12] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In Piessens and Viganò [43], pages 46–67.
- [13] I. Bolosteanu and D. Garg. Asymmetric secure multi-execution with declassification. In Piessens and Viganò [43], pages 24–45.
- [14] W. J. Bowman and A. Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 101–113, Vancouver, Canada, Aug. 2015. ACM Press.
- [15] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 431–444, Madrid, Spain, Jan. 2010. ACM Press.
- [16] S. Chong and A. C. Myers. Security policies for downgrading. In V. Atluri, B. Pfitzmann, and P. D. McDaniel, editors, *Proceedings of the 11th ACM Conference on Computer and Communications (CCS 2004)*, pages 198–209, Washington, DC, USA, Oct. 2004. ACM Press.
- [17] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassifications. In CSF 2008 [23].
- [18] W. R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.
- [19] K. Crary. Modules, abstraction, and parametric polymorphism. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 100–113, Paris, France, Jan. 2017. ACM Press.
- [20] R. Cruz and É. Tanter. Existential types for relaxed noninterference. In *Proceedings of the 17th Asian Symposium on Programming Languages and Systems (APLAS 2019)*, Bali, Indonesia, Dec. 2019. To appear.
- [21] R. Cruz and É. Tanter. Polymorphic relaxed noninterference. In *Proceedings of the IEEE Secure Development Conference (SecDev 2019)*, McLean, VA, USA, Sept. 2019. IEEE Computer Society Press. URL <http://arxiv.org/abs/1906.04830>.
- [22] R. Cruz, T. Rezk, B. Serpette, and É. Tanter. Type abstraction for relaxed noninterference. In P. Müller, editor, *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:27, Barcelona, Spain, June 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

- [23] CSF 2008. *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, Pittsburgh, Pennsylvania, June 2008. IEEE Computer Society Press.
- [24] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [25] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)*, pages 109–124, Berkeley/Oakland, California, USA, May 2010. IEEE Computer Society Press.
- [26] C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 351–360, Chicago, Illinois, USA, Oct. 2011. ACM Press.
- [27] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 186–197, Venice, Italy, Jan. 2005. ACM Press.
- [28] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL 98)*, pages 365–377, San Diego, CA, USA, Jan. 1998. ACM Press.
- [29] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings of the workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 65–74, 2006.
- [30] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3): 396–450, 2001.
- [31] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 158–170, Long Beach, CA, USA, Jan. 2005. ACM Press.
- [32] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In C. Fournet, M. W. Hicks, and L. Viganò, editors, *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW 2005)*, pages 226–240, Aix-en-Provence, France, June 2005. IEEE Computer Society Press.
- [33] M. I. Meneses Cortés. Desclasificación basada en tipos en Dart: Implementación y elaboración de herramientas de inferencia. Technical report, Universidad de Chile - Facultad de Ciencias Físicas y Matemáticas, 2018. URL <http://repositorio.uchile.cl/handle/2250/168368>.
- [34] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. ISSN 0164-0925. doi: 10.1145/44501.45065. URL <http://doi.acm.org/10.1145/44501.45065>.

- [35] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL 1999* [44], pages 228–241.
- [36] A. C. Myers. Jif homepage. <http://www.cs.cornell.edu/jif/>, accessed March 2019.
- [37] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, Oct. 2000.
- [38] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW 2004)*, pages 172–186, Pacific Grove, CA, USA, June 2004. IEEE Computer Society Press.
- [39] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P 2011)*, pages 165–179, Berkeley, California, USA, May 2011. IEEE Computer Society Press.
- [40] M. Ngo, D. A. Naumann, and T. Rezk. Typed-based relaxed noninterference for free. *CoRR*, abs/1905.00922, 2019. URL <https://arxiv.org/abs/1906.04830>.
- [41] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.
- [42] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2005.
- [43] F. Piessens and L. Viganò, editors. *Proceedings of the 5th International Conference on Principles of Security and Trust (POST 2016)*, Lecture Notes in Computer Science, Eindhoven, The Netherlands, Apr. 2016. Springer-Verlag.
- [44] *POPL 1999*. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, San Antonio, TX, USA, Jan. 1999. ACM Press.
- [45] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
- [46] W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 33–48, New Orleans, LA, USA, June 2013. IEEE Computer Society Press.
- [47] J. C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. Elsevier, 1983.
- [48] T. Rompf and N. Amin. Type soundness for dependent object types (DOT). In E. Visser and Y. Smaragdakis, editors, *Proceedings of the 31st ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, (OOPSLA 2016), part of SPLASH 2016*, pages 624–641, Amsterdam, The Netherlands, Oct. 2016. ACM Press.

- [49] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF 2010)*, pages 186–199, Edinburgh, United Kingdom, July 2010. IEEE Computer Society Press.
- [50] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
- [51] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Next-NSF-JSPS International Symposium (ISSS 2003) Revised Papers*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191, Tokyo, Japan, Nov. 2004. Springer-Verlag.
- [52] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
- [53] N. Shikuma and A. Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In M. Okada and I. Satoh, editors, *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science (ASIAN 2006): Secure Software and Related Issues*, volume 4435 of *Lecture Notes in Computer Science*, pages 301–315, Tokyo, Japan, Dec. 2008. Springer-Verlag.
- [54] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, pages 95–106. ACM Press, 2011.
- [55] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming*, 27, 2017.
- [56] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Proceedings of the 11th Static Analysis Symposium (SAS 2004)*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99, Verona, Italy, Aug. 2004. Springer-Verlag.
- [57] N. Swamy and M. Hicks. Verified enforcement of stateful information release policies. In Ú. Erlingsson and M. Pistoia, editors, *Proceedings of the 3rd Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 21–32, Tucson, AZ, USA, June 2008. ACM Press.
- [58] M. Toro, R. Garcia, and É. Tanter. Type-driven gradual security with references. *ACM Transactions on Programming Languages and Systems*, 40(4):16:1–16:55, Nov. 2018.
- [59] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2004)*, pages 115–125, Snowbird, Utah, USA, Sept. 2004. ACM Press.
- [60] S. Tse and S. Zdancewic. A design for a security-typed language with certificate-based declassification. In S. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP 2005)*, volume 2986 of *Lecture Notes in Computer Science*, pages 279–294, Berlin, Heidelberg, 2005. Springer-Verlag.

- [61] M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF 2014)*, pages 293–307, Vienna, Austria, July 2014. IEEE Computer Society Press.
- [62] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282, Gothenburg, Sweden, Sept. 2014. ACM Press.
- [63] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.
- [64] P. Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, London, United Kingdom, 1989. ACM.
- [65] G. Washburn and S. Weirich. Generalizing parametricity using information-flow. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71, Chicago, IL, USA, June 2005. IEEE Computer Society Press.
- [66] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 249–257. ACM Press, 1998.
- [67] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, Aug. 2002.
- [68] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence, PLID 2004*, Aug. 2004.
- [69] S. Zdancewic and A. C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW 2001)*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.

# Appendices

# Appendix A

## Type-based relaxed noninterference

In this appendix we include the full definitions and proofs for Chapter 2.

### A.1 Full Syntax and semantics

#### A.1.1 Syntax

$e$	$::= v \mid e.m(e) \mid x$	(terms)	
$v$	$::= [z : S \Rightarrow \overline{m(x)}e]$	(values)	$x, y, z$ (variables)
$T, U$	$::= O \mid \alpha$	(types)	$\alpha, \beta$ (type variables)
$O$	$::= \mathbf{Obj}(\alpha). [\overline{m : S \rightarrow S}]$	(object type)	$m$ (method labels)
$S$	$::= T \triangleleft U$	(security type)	

Figure A.1:  $\text{Ob}_{\text{SEC}}$ : Syntax

#### Syntax of environments and notations

$\Gamma$	$::= \cdot \mid \Gamma, x : S$	(type environment)
$\Phi$	$::= \cdot \mid \Phi, \alpha <: \beta$	(subtyping environment)
$\Delta_{\text{ok}}$	$::= \cdot \mid \Delta_{\text{ok}}, \alpha$	(type variable environment)
$\Sigma$	$::= \cdot \mid \Sigma, \alpha \triangleq O$	(type definition environment)

- $\Gamma$  is a finite map from variables to closed and well-formed security types.  $\Sigma$  is a finite map from type variables to object types.  $\Phi$  is a set of subtyping relations between type variables.  $\Delta_{\text{ok}}$  is a set of type variables
- $\text{dom}(Env)$  (where  $Env$  could be  $\Gamma$ ,  $\Sigma$  or  $\Phi$ ) is the set of variables for which the finite map  $Env$  is defined. In the case of  $\text{dom}(\Phi)$ , it is the set of the type variables in the left part of the subtyping relation.
- The notations  $\Gamma[x \mapsto S]$ ,  $\Sigma[\alpha \mapsto O]$ ,  $\Phi[\alpha <: \beta]$  extend the environments  $\Gamma$ ,  $\Sigma$ ,  $\Phi$  with a new binding or relation respectively. If  $x \in \text{dom}(\Gamma)$ ,  $\alpha \in \text{dom}(\Sigma)$  or either  $\alpha$  or  $\beta \in \text{dom}(\Phi) \cup \text{cod}(\Phi)$  the extension operation is not defined for the respective



environment. We also use  $\Gamma, x : S$  or  $\Sigma, \alpha \triangleq O$  or  $\Phi, \alpha <: \beta$  with the meaning of environment extensions when for the context is evident we want to extend the respective environment.

- The notation  $\Delta_{\text{ok}}, \alpha$  extends the set  $\Delta_{\text{ok}}$  with a new type variable. If  $\alpha \in \Delta_{\text{ok}}$  the operation is not defined.

We use the following functions to access to the elements of the environments.

- $\Gamma(x)$  returns the security type associated to  $x$  in  $\Gamma$ . If  $x \notin \text{dom}(\Gamma)$ , then  $\Gamma(x)$  is undefined.
- $\Sigma(\alpha)$  returns the type associated to  $\alpha$  in  $\Sigma$ . If  $\alpha \notin \text{dom}(\Sigma)$ , then  $\Sigma(\alpha)$  is undefined
- $\alpha <: \beta \in \Phi$  is true if  $\Phi(\alpha) = \beta$ , false otherwise.  $\Phi(\alpha)$  returns the type variable in the right part of the subtyping relation with  $\alpha$  in  $\Phi$ . If  $\alpha \notin \text{dom}(\Phi)$ , then  $\Phi(\alpha)$  is undefined

We use the function  $\Sigma[T]$  to close free variables of type  $T$  with the bindings in  $\Sigma$ .

### A.1.2 Well-formedness of environments and types

For the main results of the Section 2.5 to hold we need to ensure we work with well-formed security types.

**Well formed types.** We use the predicate  $\text{valid}(S)$  to denote that a security type  $S$  is closed and that the object types that  $S$  contains have unique method members. We do not discuss the definition of  $\text{valid}(S)$  because they are standard [1](Figure A.2). Then we focus on the presentation of the new rules of well-formed security types.

To check for *well-formed security types*, *i.e.* that the safety type is a subtype of the declassification type we define the judgment  $\Sigma \vdash_s S$  (Figure A.4). The (WFS-ST) rule is the most important. For this rule to hold, the subtyping relation between both facets must hold and also both facets must be well-formed regarding the security types that they contain. The presence of type variables in the facets of a security type and the corresponding subtyping constraint introduces subtle cases to manage before to use the subtyping judgment. The following object type illustrates that:

$$O \triangleq \mathbf{Obj}(\alpha). [\mathbf{m} : S \rightarrow \alpha \triangleleft \mathbf{Obj}(\beta). [\mathbf{m} : S \rightarrow \alpha \triangleleft \beta]]$$

For  $\vdash_s O$  to hold,  $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta). [\mathbf{m} : S \rightarrow \alpha \triangleleft \beta]$  must hold. It implies to check  $\vdash \alpha <: \mathbf{Obj}(\beta). [\mathbf{m} : S \rightarrow \alpha \triangleleft \beta]$ . Note that, we cannot justify that subtyping judgment, because we do not have a subtyping premise involving the type variable  $\alpha$ . To address this, we need to remember (in  $\Sigma$ ) the surrounding recursive object type  $O$  that binds  $\alpha$ , and to transform the check  $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta). [\mathbf{m} : S \rightarrow \alpha \triangleleft \beta]$  to  $\vdash O <: \mathbf{Obj}(\beta). [\mathbf{m} : S \rightarrow \alpha \triangleleft \beta]$  by closing  $\alpha$  with the mappings in  $\Phi$  (*i.e.*  $O$ ). The function  $\Sigma[T]$  closes the free variables of the type  $T$  with the binding in  $\Sigma$ .

Finally the rule (WF) holds if the a type is closed and a well-formed security type, denoted  $\vdash S$ .

$$\boxed{\Delta_{\text{ok}} \vdash T}$$

$$\begin{array}{c} T \equiv \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \\ (i \neq j \implies m_i \neq m_j) \\ \text{(WF-V)} \frac{\alpha \in \Delta_{\text{ok}}}{\Delta_{\text{ok}} \vdash \alpha} \quad \text{(WF-O)} \frac{\Delta_{\text{ok}}, \alpha \vdash S_{1i} \quad \Delta_{\text{ok}}, \alpha \vdash S_{2i}}{\Delta_{\text{ok}} \vdash T} \end{array}$$

Figure A.2: Standard well-formed rules for object types and type variables

$$\boxed{\Delta_{\text{ok}} \vdash S}$$

$$\begin{array}{c} \text{(WF-ST)} \frac{\Delta_{\text{ok}} \vdash T \quad \Delta_{\text{ok}} \vdash U}{\Delta_{\text{ok}} \vdash T \triangleleft U} \quad \frac{\cdot \vdash S}{\text{valid}(S)} \end{array}$$

Figure A.3: Lift of rules from Figure A.2 for security types

$$\boxed{\Sigma \vdash_s T}$$

$$\text{(WFS-V)} \frac{\frac{T \equiv \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \quad \Sigma, \alpha : T \vdash_s S_{1i} \quad \Sigma, \alpha : T \vdash_s S_{2i}}{\Sigma \vdash_s T}}{\Sigma \vdash_s \alpha}$$

$$\boxed{\Sigma \vdash_s S}$$

$$\boxed{\vdash S}$$

$$\begin{array}{c} \text{(WFS-ST)} \frac{\Sigma \vdash_s T \quad \Sigma \vdash_s U \quad \cdot \vdash \Sigma[T] <: \Sigma[U]}{\Sigma \vdash_s T \triangleleft U} \quad \text{(WF)} \frac{\text{valid}(S) \quad \cdot \vdash_s S}{\vdash S} \end{array}$$

Figure A.4: Well-formedness rules for security types

**Well-formed environment.** The judgment  $\vdash \Gamma$  states that a type environment is well formed if all types in the environment are well-formed.  $\vdash \Phi$  captures the well-formed conditions for the subtyping environments. This condition is checked in (SVar) and (SObj) rules.

$\vdash \Gamma$

$$\text{(EEnvOk)} \frac{}{\vdash \cdot} \quad \text{(EnvOk)} \frac{\vdash \Gamma \quad \vdash S \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x : S}$$

$\vdash \Phi$

$$\text{(EnvSubOk)} \frac{\vdash \Phi \quad \alpha_i \notin \text{dom}(\Phi) \cup \text{cod}(\Phi)}{\vdash \Phi, \alpha_1 <: \alpha_2}$$

Figure A.5: Well-formed environment

Once formalized the well-formed conditions for environments and types, we implicitly assume them in most definitions.

### A.1.3 Subtyping

The gray parts in the subtyping rules were not included in the main document. They prevent to justify inconsistent subtyping judgments by controlling the uses of type variables. For example:

$$T_1 \triangleq \mathbf{Obj}(\alpha). [n : S \rightarrow \mathbf{Obj}(\beta). [m_1 : \beta_L \rightarrow S' \quad m_2 : S_1 \rightarrow S_2]_L]$$

$$T_2 \triangleq \mathbf{Obj}(\beta). [n : S \rightarrow \mathbf{Obj}(\alpha). [m_1 : \alpha_L \rightarrow S']_L]$$

For  $\vdash T_1 <: T_2$  to hold, after using the rule (SObj) twice, the contravariance of  $m_1$  parameters  $\cdot, \alpha <: \beta, \beta <: \alpha \vdash \alpha <: \beta$  must hold. We can justify this by applying the rule (SVar) because we have the assumption  $\alpha <: \beta$  in the subtyping environment. So, we justify  $\vdash T_1 <: T_2$  and it is not the case that  $T_1$  is subtype of  $T_2$ . The problem is the occurrence of the variables  $\alpha$  and  $\beta$  in both types, that creates subtyping assumptions in both directions and it allows to justify subtyping between type variables that represent unrelated types (by subtyping). The well-formed condition of the subtyping environment  $\Phi$  prevents this kind of cases, because we cannot extend the environment with a subtyping premise, where one of the involved variables is already in the environment.

$$\boxed{\Phi \vdash T <: T}$$

$$\begin{array}{c} O_1 \triangleq \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \quad O_2 \triangleq \mathbf{Obj}(\beta). \overline{[m' : S'_1 \rightarrow S'_2]} \quad \overline{m'} \subseteq \overline{m} \\ m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j}) \\ \vdash \Phi \quad \text{dom}(\Phi) \cup \text{cod}(\Phi) \vdash O_i \\ \text{(SObj)} \frac{}{\Phi \vdash O_1 <: O_2} \end{array}$$

$$\begin{array}{c} \vdash \Phi \\ \text{(SVar)} \frac{\alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \quad \text{(SSubEq)} \frac{T_1 \equiv T_2}{\Phi \vdash T_1 <: T_2} \quad \text{(STrans)} \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3} \end{array}$$

$$\boxed{\Phi \vdash S <: S}$$

$$\text{(TSubST)} \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

### A.1.4 Type equivalence

Two types are equivalent if the equivalence can be derived for the congruence induced by rules (Alpha-Eq) and (Fold-Unfold). For example:  $\mathbf{Obj}(\alpha). [m : \alpha \rightarrow \alpha] \equiv \mathbf{Obj}(\beta). [m : \beta \rightarrow \beta]$   
 $\mathbf{Obj}(\alpha). [m : \top \rightarrow \alpha] \equiv \mathbf{Obj}(\alpha). [m : \top \rightarrow \mathbf{Obj}(\beta). [m : \top \rightarrow \beta]]$

$$\boxed{T \equiv T}$$

$$\begin{array}{c} \text{(Sym)} \frac{}{T \equiv T} \quad \text{(Refl)} \frac{T_1 \equiv T_2}{T_2 \equiv T_1} \quad \text{(Trans)} \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3} \\ \text{(O-Congr)} \frac{S_{1i} \equiv S'_{1i} \quad S_{2i} \equiv S'_{2i}}{\mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \equiv \mathbf{Obj}(\alpha). \overline{[m : S'_1 \rightarrow S'_2]}} \\ \text{(Alpha-Eq)} \frac{O \triangleq \mathbf{Obj}(\alpha). \overline{[m : S_1 \rightarrow S_2]} \quad \beta \text{ fresh}}{O \equiv O[\beta/\alpha]} \quad \text{(Fold-Unfold)} \frac{}{O \equiv O[O/\alpha]} \end{array}$$

$$\boxed{S \equiv S}$$

$$\frac{T_1 \equiv T_2 \quad U_1 \equiv U_2}{T_1 \triangleleft U_1 \equiv T_2 \triangleleft U_2}$$

### A.1.5 Simple type system

We define a typing judgment  $\Gamma \vdash_{\text{sf}} e : S$  that replaces the rule (TmD) and (TmH) by one rule (T1mI) that does not care about the declassification type. Furthermore we replace the subtyping judgment  $\Phi \vdash S_1 <: S_2$  by the simple subtyping judgment  $\Phi \vdash_{\text{sf}} S_1 <: S_2$  that only takes care of subtyping between the private facets of the security types <sup>1</sup>

<sup>1</sup>The definition of  $\Phi \vdash_{\text{sf}} S_1 <: S_2$  does not have anything special and we left out

$$\boxed{\Gamma \vdash_{\text{sf}} e : S}$$

$$\begin{array}{c}
\text{(T1Var)} \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash_{\text{sf}} x : \Gamma(x)} \quad \text{(T1Sub)} \frac{\Gamma \vdash_{\text{sf}} e : S' \quad \vdash_{\text{sf}} S' <: S \quad \vdash S}{\Gamma \vdash_{\text{sf}} e : S} \\
\text{(T1Obj)} \frac{\vdash S \quad S \triangleq T \triangleleft U \quad \text{msig}(T, m_i) = S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash_{\text{sf}} e_i : S''_i}{\Gamma \vdash_{\text{sf}} [z : S \Rightarrow \overline{m}(x) e] : S} \\
\text{(T1mI)} \frac{\Gamma \vdash_{\text{sf}} e_1 : T \triangleleft U \quad \text{msig}(T, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash_{\text{sf}} e_2 : S_1}{\Gamma \vdash_{\text{sf}} e_1.m(e_2) : S_2}
\end{array}$$

Now we can define a simple typing judgment.

$$\boxed{\Gamma \vdash_1 e : T}$$

$$\frac{\Gamma \vdash_{\text{sf}} e : T \triangleleft U}{\Gamma \vdash_1 e : T}$$

## A.2 Type safety: proofs

**Definition 18.**  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$

Notation:

- We denote an expression is irreducible with the predicate  $\text{irred}(e)$  which is defined as  $\nexists e'. e \mapsto^1 e'$ .
- We should read  $v \in \mathcal{V}_k[S]$  as "the value interpretation of type  $S$  for  $k$  steps". It means if we use  $v$  in a context with no more that  $k$  step of evaluation we will see that  $v$  has type  $S$ . However if you run it with more than  $k$  steps you might notice it does not belong to type  $S$ .
- We define  $\mathcal{C}_k[S]$  as the computational interpretation of type  $S$  for  $k$  steps

$$\begin{aligned}
\mathcal{V}_k[S] = & \{v = [z : S_1 \Rightarrow \_ ] \mid S \triangleq T \triangleleft U \quad \vdash S_1 <: S \wedge \\
& (\forall j < k. v \in \mathcal{V}_j[S_1] \wedge \\
& (\forall m \in T, v'. \text{msig}(T, m) = S' \rightarrow S'' \quad \text{methimpl}(v, m) = x.e \\
& v' \in \mathcal{V}_j[S'] \implies e[v/z][v'/x] \in \mathcal{C}_j[S'']))\}
\end{aligned}$$

$$\mathcal{C}_k[S] = \{e \mid \forall j < k. \forall e'. (e \mapsto^j e' \wedge \text{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[S]\}$$

Figure A.6:  $\text{Ob}_{\text{SEC}}$ : Unary logical relation for safety

A value substitution is a finite map from variables to values and the operation  $\gamma(e)$  substitutes all variables in  $e$  for its corresponding value in  $\gamma$ . We use the following algorithmic definition in proofs:

$$\gamma ::= \emptyset \mid \gamma[x \mapsto v] \quad \emptyset(e) = e \quad \gamma[x \mapsto v](e) = \gamma(e)[v/x]$$

**Definition 19** (Safe substitution). *We say  $\gamma$  is a safe substitution regarding  $\Gamma$  for  $k$  steps, written  $\gamma \in \mathcal{G}_k[\Gamma]$  if  $\text{dom}(\gamma) = \text{dom}(\Gamma)$  and  $\forall x \in \text{dom}(\Gamma). \gamma(x) \in \mathcal{V}_k[\Gamma(x)]$*

## A.2.1 Auxiliary Lemmas

**Lemma 19.**

$v \in \mathcal{V}_k[S]$  and  $j \leq k$  then  $v \in \mathcal{V}_j[S]$

*Proof.* Let us denote  $v \triangleq [z : S_1 \Rightarrow \_]$  and  $S \triangleq T \triangleleft U$   
To show  $e \in \mathcal{V}_j[S]$  we have two proofs obligations:

- $\vdash S_1 <: S$ . It follows directly from  $v \in \mathcal{V}_k[S]$
- Considering arbitrary  $i, m, v_1$  such that
  - $i < j$
  - $m \in T$  and  $\text{msig}(T, m) = S_1 \rightarrow S_2$  and  $\text{methimpl}(v, m) = x.e$
  - $v_1 \in \mathcal{V}_i[S_1]$

**Show:**

1.  $v \in \mathcal{V}_i[S_1]$ . It follows directly by instantiating the definition of  $v \in \mathcal{V}_k[S]$  with  $i$ .
2.  $e[v/z][v_1/x] \in \mathcal{C}_i[S_2]$ . We now instantiate the assumption  $v \in \mathcal{V}_k[S]$  with  $i, m, v_1$ .

Note we can do it because:

- $i < j \leq k$
- $m \in T$ .
- $v_1 \in \mathcal{V}_i[S_1]$ . (we considered it)

Hence:  $e[v/z][v_1/x] \in \mathcal{C}_i[S_2]$

□

**Lemma 20.**

*If  $\gamma \in \mathcal{G}_k[\Gamma]$  then  $\gamma \in \mathcal{G}_j[\Gamma]$  where  $j \leq k$*

*Proof.* Trivial by Lemma 19

□

**Lemma 21.**

*If  $v \in \mathcal{V}_k[S]$  and  $\vdash S <: S_1$  then  $v \in \mathcal{V}_k[S_1]$*

*Proof.* Let us denote  $S \triangleq T \triangleleft U$ ,  $S_1 \triangleq T_1 \triangleleft U_1$  and  $v = [z : S_2 \Rightarrow \_]$

The proof is by (strong) induction on  $k$ .

**Case ( $k = 0$ ).** *We only need to check  $\vdash S_2 <: S_1$ . It follows directly from  $v \in \mathcal{V}_k[S]$  and  $\vdash S <: S_1$*

**Case** ( $k > 0$ ). Expanding  $\mathcal{V}_k[[S_1]]$  we have three proof obligations:

- $\vdash S_2 <: S_1$ . It follows directly from  $v \in \mathcal{V}_k[[S]]$  and  $\vdash S <: S_1$
- Suppose arbitrary  $j < k$ , show:  $v \in \mathcal{V}_j[[S_2]]$ . It follows directly by instantiating  $v \in \mathcal{V}_k[[S]]$  with  $j$ .
- Suppose arbitrary  $j, m, v'_1$  such that:
  - $j < k$
  - $m \in T_1$  and  $\text{msig}(T_1, m) = S'_1 \rightarrow S''_2$  and  $\text{methimpl}(v, m) = x.e$
  - $v'_1 \in \mathcal{V}_j[[S'_1]]$

**Show:**  $e[v/z][v'_1/x] \in \mathcal{C}_j[[S''_2]]$

Expanding the above definition we have a new proof obligation over the following premises:

Consider arbitrary  $j_2, e'$  such that:

- $j_2 < j$
- $e[v/z][v'_1/x] \mapsto^{j_2} e'$
- $\text{irred}(e')$

**Show:**  $e' \in \mathcal{V}_{j-j_2}[[S''_2]]$

By the definition of  $v \in \mathcal{V}_k[[S]]$  we know:

1.  $\forall j < k. v \in \mathcal{V}_j[[S_2]] \wedge (\forall m \in T \text{ msig}(T, m) = S' \rightarrow S'' \text{ methimpl}(v, m) = x.e \quad \forall v' \in \mathcal{V}_j[[S']] \implies e[v/z][v'/x] \in \mathcal{C}_j[[S'']])$

We instantiate the definition of  $v \in \mathcal{V}_k[[S]]$  with  $j, m, v'_1$ . Note we can do it because

- $j < k$  (we considered it)
- $m \in T$ . We assume  $m \in T_1$ , and we know  $\vdash T <: T_1$ , which follows from  $\vdash S <: S_1$
- $v'_1 \in \mathcal{V}_j[[S']]$ . We consider  $v'_1 \in \mathcal{V}_j[[S'_1]]$  and we know  $\vdash S'_1 <: S'$  (derived from  $\vdash T <: T_1$ ), then we apply the induction hypothesis with  $j < k$ ,

Hence:  $e[v/z][v'_1/x] \in \mathcal{C}_j[[S'']]$ . We now instantiate this definition with  $j_2$  and  $e'$  and we obtain:

$e' \in \mathcal{V}_{j-j_2}[[S'']]$

Note that  $\vdash S'' <: S''_2$  then we can apply the hypothesis induction with  $j - j_2 < k$  (since  $0 \leq j_2 < j < k$ ), and we finally obtain:

$e' \in \mathcal{V}_{j-j_2}[[S''_2]]$

□

## Lemma 22.

If  $e \in \mathcal{C}_k[[S]]$  and  $\vdash S <: S_1$  then  $e \in \mathcal{C}_k[[S_1]]$

*Proof.* We expand the definition of  $e \in \mathcal{C}_k[[S]]$  and  $e \in \mathcal{C}_k[[S_1]]$  and we apply then the Lemma 21

□

**Lemma 23.**

If  $\vdash \Gamma$  and  $\Gamma \vdash e : S$  then  $\vdash S$

*Proof.* By induction on typing derivations of  $e$ .

**Case (TVar).**

$$(TVar) \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

**Show:**  $\vdash \Gamma(x)$ . This follows directly from  $\vdash \Gamma$ .

**Case (TSub).**

$$(TSub) \frac{\Gamma \vdash e : S' \quad \vdash S' <: S \quad \vdash S}{\Gamma \vdash e : S}$$

**Show:**  $\vdash S$ .

By the assumption of the rule we obtain  $\vdash S$ .

**Case (TObj).** The goal is directly checked in the rule.

**Case (TmD).**  $e = e_1.m(e_2)$  **Show:**  $\vdash S_2$

We apply IH on  $e_1$  and obtain  $\vdash S$  where  $S \triangleq T \triangleleft U$ . From this fact, we know  $\vdash S_2$  which is the required goal here.

**Case (TmH).**  $e = e_1.m(e_2)$ , **Show:**  $\vdash T_2 \triangleleft \top$

We apply IH on  $e_1$  and obtain  $\vdash S$  where  $S \triangleq T \triangleleft U$ . From this fact, we know that  $\vdash T_2 \triangleleft U_2$ . Then  $\vdash T_2 \triangleleft \top$ .

□

## A.2.2 Proof: Safety

**Lemma 24.**

$$\Gamma \vdash e : S \implies \Gamma \models e : S$$

*Proof.* By induction on the typing derivations of  $e$ .

**Case (TSub).**

$$(TSub) \frac{\Gamma \vdash e : S' \quad \vdash S' <: S \quad \vdash S}{\Gamma \vdash e : S}$$



**Show:**  $\Gamma \models e : S$ . Expanding this definition we have a new proof obligation under the following assumptions:

Consider arbitrary  $k, \gamma$  such that  $k \geq 0$  and  $\gamma \in \mathcal{G}_k[\Gamma]$

**Show:**  $\gamma(e) \in \mathcal{C}_k[S]$ .

From the IH on  $\Gamma \vdash e : S'$  we know:

$\forall k \geq 0, \forall \gamma. \gamma \in \mathcal{G}_k[\Gamma] \implies \gamma(e) \in \mathcal{C}_k[S']$ . Instantiating this with  $k$  and  $\gamma$  we get:

$\gamma(e) \in \mathcal{C}_k[S']$ . Then applying Lemma 22 with  $\vdash S' <: S$  we get:

$\gamma(e) \in \mathcal{C}_k[S]$

**Case (TObj).**

$$(TObj) \frac{\vdash S \quad S \triangleq T \triangleleft U \quad \text{msig}(T, m_i) = S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}{\Gamma \vdash [z : S \Rightarrow \overline{m(x)e}] : S}$$

**Show:**  $\Gamma \models [z : S \Rightarrow \overline{m(x)e}] : S$

Expanding the above definition we have a new proof obligation:

Consider arbitrary  $k, \gamma$  such that:

- $k \geq 0, \quad \gamma \in \mathcal{G}_k[\Gamma]$

We need **to show:**  $\gamma([z : S \Rightarrow \overline{m(x)e}]) \in \mathcal{C}_k[S]$ . Expanding this definition we have a new proof obligation:

Consider arbitrary  $j, e'$  such that:

- $j < k \quad \gamma([z : S \Rightarrow \overline{m(x)e}]) \mapsto^j e' \wedge \text{irred}(e')$

**Show:**  $e' \in \mathcal{V}_{k-j}[S]$ .

Note that:  $\gamma([z : S \Rightarrow \overline{m(x)e}]) \equiv [z : S \Rightarrow \overline{m(x)\gamma(e)}]$  and it is already irreducible and a value, so  $j = 0$ .

Then we need **to show:**  $v = [z : S \Rightarrow \overline{m(x)\gamma(e)}] \in \mathcal{V}_k[S]$ . To prove this we use strong induction on  $k$ . The case  $k = 0$  trivially holds, then we focus in an arbitrary  $k > 0$ :

Expanding this definition we have the following proof obligations:

- $\vdash S <: S$ . It holds by (SSubEq) rule.
- Suppose  $j_1 < k$ , show  $v \in \mathcal{V}_{j_1}[S]$ . This follows directly by induction hypothesis on  $j_1$  ( $j_1 < k$ )

- Consider arbitrary  $j_1, m, v'$  such that:
    - $j_1 < k$
    - $m \in T$  and  $\text{methimpl}(v, m) = x.\gamma(e)$
    - $v' \in \mathcal{V}_{j_1}[[S']]$
- Show:**  $\gamma(e) [v/z] [v'/x] \in \mathcal{C}_{j_1}[[S'']]$

By IH on  $\Gamma, z : S, x : S' \vdash e : S''$  we get  $\Gamma, z : S, x : S' \models e$ . We instantiate this definition with  $j_1, \gamma [z \mapsto v] [x \mapsto v']$ . Note we can do it because:

- $j_1 \geq 0$
- $\gamma [z \mapsto v] [x \mapsto v'] \in \mathcal{G}_{j_1}[[\Gamma, z : S, x : S']]$ .
  - First we establish  $\gamma \in \mathcal{G}_{j_1}[[\Gamma]]$  by applying Lemma 20 to  $\gamma \in \mathcal{G}_k[[\Gamma]]$  with  $j_1 < k$
  - Second we considered  $v' \in \mathcal{V}_{j_1}[[S']]$
  - Third we already show  $v \in \mathcal{V}_{j_1}[[S]]$

Hence:  $\gamma [z \mapsto v] [x \mapsto v'] (e) \in \mathcal{C}_{j_1}[[S'']] \equiv$

$\gamma(e) [v/z] [v'/x] \in \mathcal{C}_{j_1}[[S'']]$

**Case (TVar).**

$$(TVar) \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

**Show:**  $\Gamma \models x : S$

Expanding this definition we have a new proof obligation:

Consider arbitrary  $k, \gamma$  such that:

- $k \geq 0$  and  $\gamma \in \mathcal{G}_k[[\Gamma]]$

**Show:**  $\gamma(x) \in \mathcal{G}_k[[\Gamma]]$ . Expanding this definition we have a new proof obligation:

Consider arbitrary  $j, e'$  such that:

- $j < k$
- $\gamma(x) \mapsto^j e'$
- $\text{irred}(e')$

**Show:**  $e' \in \mathcal{V}_{k-j}[[S]]$ . Let us denote  $v = \gamma(x)$ . Note that  $j = 0$ , so we need to **show:**  $v \in \mathcal{V}_k[[S]]$  which follows from  $\gamma \in \mathcal{G}_k[[\Gamma]]$ .

**Case (TmD).**

$$(TmD) \frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \in U \quad \text{msig}(U, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2}$$

**Show:**  $\Gamma \models e_1.m(e_2) : S_2$ . Expanding this definition we have a new proof obligation:

Consider arbitrary  $k, \gamma$  such that:

- $k \geq 0, \gamma \in \mathcal{G}_k[\Gamma]$

We need **to show:**  $\gamma(e_1.m(e_2)) \in \mathcal{C}_k[S_2] \equiv \gamma(e_1).m(\gamma(e_2)) \in \mathcal{C}_k[S_2]$ . Expanding this definition we have a new proof obligation:

Consider arbitrary  $j, e'$  such that:

- $j < k$
- $\gamma(e_1).m(\gamma(e_2)) \mapsto^j e' \wedge \text{irred}(e')$

We need **to show:**  $e' \in \mathcal{V}_{k-j}[S_2]$

Inspecting the dynamic semantics we know that there exists  $j_1, e'_1$  such that:

- $\gamma(e_1) \mapsto^{j_1} e'_1$
- $\text{irred}(e'_1)$
- $j_1 \leq j$

By IH on  $\Gamma \vdash e_1 : S$  where  $S \triangleq T \triangleleft U$  we obtain  $\Gamma \models e_1 : S$ . We instantiate this definition with  $k, \gamma$ . We can do it because:

- We already considered  $k \geq 0$  and  $\gamma \in \mathcal{G}_k[\Gamma]$

Hence:  $\gamma(e_1) \in \mathcal{C}_k[S]$ . Then we instantiate this definition with  $j_1, e'_1$ . We can do it because:

- $j_1 \leq j < k$
- $\gamma(e_1) \mapsto^{j_1} e'_1$  and  $\text{irred}(e'_1)$ .

Hence:  $e'_1 \in \mathcal{V}_{k-j_1}[S]$ .

Hence:  $e'_1 \triangleq v_1 \triangleq [z : S_a \Rightarrow \_ ] \vdash S_a <: S$

Note that:

$$\begin{aligned} & \gamma(e_1).m(\gamma(e_2)) \\ & \mapsto^{j_1} [z : S_a \Rightarrow \_ ].m(\gamma(e_2)) \\ & \mapsto^{j-j_1} e' \end{aligned}$$

Inspecting the dynamic semantics again, we know that there exists  $j_2$  such that:

- $\gamma(e_2) \mapsto^{j_2} e'_2$
- $\text{irred}(e'_2)$
- $j_2 \leq j - j_1$

Now we use the IH on  $\Gamma \vdash e_2 : S_1$  to get  $\Gamma \models e_2 : S_1$ . We instantiate this definition with

$k - j_1, \gamma$ . Note we can do it because:

- $k - j_1 \geq 0$  (because  $j_1 < k$ )
- $\gamma \in \mathcal{G}_{k-j_1}[\Gamma]$  which follows from Lemma 20 applied to  $\gamma \in \mathcal{G}_k[\Gamma]$  with  $k - j_1 < k$

Hence:  $\gamma(e_2) \in \mathcal{C}_{k-j_1}[\mathbb{S}_1]$ . Then we instantiate this definition with  $j_2, e'_2$ . We can do it because:

- $j_2 < k - j_1$  which follows from  $j_2 \leq j - j_1$  and  $j < k$ .
- We already concluded  $\gamma(e_2) \mapsto^{j_2} e'_2$  and  $\text{irred}(e'_2)$ .

Hence:  $e'_2 \in \mathcal{V}_{k-j_1-j_2}[\mathbb{S}_1]$ .

Hence:  $e'_2 \equiv v_2 \equiv [y : S_b \Rightarrow \_ ] \quad \vdash S_b <: S_1$

Note that:

$$\begin{aligned}
& \gamma(e_1.m(e_2)) \equiv \gamma(e_1).m(\gamma(e_2)) \\
& \mapsto^{j_1} e'_1.m(\gamma(e_2)) \\
& \equiv v_1.m(\gamma(e_2)) \mapsto^{j_2} v_1.m(e'_2) \\
& \equiv [z : S_a \Rightarrow \_ ].m([y : S_b \Rightarrow \_ ]) \\
& \mapsto^1 e_m [v_1/z] [v_2/x] \\
& \mapsto^{j_3} e' \\
& \text{where } j = j_1 + j_2 + 1 + j_3 \\
& \text{methimpl}(v_1, m) = x.e_m
\end{aligned}$$

Then we instantiate the definition of  $v_1 \in \mathcal{V}_{k-j_1}[\mathbb{S}]$  with  $k - j_1 - j_2 - 1, m, v_2$ . Note we can do it because:

- $k - j_1 - j_2 - 1 < k - j_1$  which is direct.
- $m \in T$  which follows from  $m \in U$  and  $\vdash T <: U$  ( $\vdash_s T \triangleleft U$ )
- $v_2 \in \mathcal{V}_{k-j_1-j_2-1}[\mathbb{S}_1]$  which follows from Lemma 19 applied to  $v_2 \in \mathcal{V}_{k-j_1-j_2}[\mathbb{S}_1]$  and  $k - j_1 - j_2 - 1 < k - j_1 - j_2$

Hence:  $e_m [v_1/z] [v_2/x] \in \mathcal{C}_{k-j_1-j_2-1}[\mathbb{S}_2]$ . Then we instantiate this with  $j_3, e'$ . Note we can do it because:

- $j_3 < k - j_1 - j_2 - 1$  which follows from  $j_3 = j - j_1 - j_2 - 1$  and  $j < k$
- $e_m [v_1/z] [v_2/x] \mapsto^{j_3} e'$
- $\text{irred}(e')$

Hence:  $e' \in \mathcal{V}_{k-j_1-j_2-1-j_3}[\mathbb{S}_2] \equiv e' \in \mathcal{V}_{k-j}[\mathbb{S}_2]$

Case (TmH).

$$\text{(TmH)} \frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \notin U \quad \text{msig}(T, m) = S_1 \rightarrow T_2 \triangleleft U_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \mathbb{T}}$$

The proof structure is exactly the same that case [TmD]

□

**Lemma 5** (Syntactic typing implies semantic typing).  $\vdash e : S \implies \models e : S$

*Proof.* Directly by applying Lemma 24 with an empty type environment.

□

**Lemma 4** (Semantic type safety).  $\models e : S \implies \text{safe}(e)$ .

*Proof.* To show  $\text{safe}(e)$  we need to consider arbitrary  $e'$  such that  $e \mapsto^* e'$  and then we need **to show:**  $e' = v$  or  $\exists e'' . e' \mapsto e''$

Let us consider an arbitrary  $j_1$  to count the step that takes  $e \mapsto^* e'$ . Let us denote  $l = j_1 + 1$ . By expanding the definition of  $\models e : S$  we have  $\forall k \geq 0 . e \in \mathcal{C}_k \llbracket S \rrbracket$ . We instantiate this with  $k = l$  to obtain  $e \in \mathcal{C}_l \llbracket S \rrbracket$ . By unfolding this we have:

$\forall j < l . \forall e_1 . (e \mapsto^j e_1 \wedge \text{irred}(e_1)) \implies e_1 \in \mathcal{V}_{k-j} \llbracket S \rrbracket$ . We instantiate this definition with  $j_1$  and  $e'$  and we obtain:  $(e \mapsto^{j_1} e' \wedge \text{irred}(e')) \implies e' \in \mathcal{V}_{k-j_1} \llbracket S \rrbracket$ .

We have two cases here:  $\neg \text{irred}(e')$  and  $\text{irred}(e')$ . The  $\neg \text{irred}(e')$  case is trivial, because it must exist  $\exists e'' . e' \mapsto e''$ . For the  $\text{irred}(e')$ , we have that  $e' \in \mathcal{V}_{k-j} \llbracket S \rrbracket$ , so  $e'$  is a value. □

**Theorem 6** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

*Proof.* The proof is straightforward by composing Lemma 5 with Lemma 4.

□

## A.3 Type-based relaxed noninterference: proofs

### A.3.1 Logical relation

The logical relation is defined bellow:

$$\begin{aligned} v_1 \approx_k v_2 : \mathcal{V} \llbracket S \rrbracket &\iff S \triangleq T \triangleleft U \quad v_i \triangleq [z : \_ \Rightarrow \_] \\ &\vdash_1 v_i : T \wedge (\forall m \in U . \text{msig}(U, m) = S' \rightarrow S'' \quad \text{methimpl}(v_i, m) = x.e_i \\ &\forall j < k, v'_1, v'_2 . v_1 \approx_j v_2 : \mathcal{V} \llbracket S \rrbracket \wedge \\ &\quad (v'_1 \approx_j v'_2 : \mathcal{V} \llbracket S' \rrbracket \implies e_1 [v_1/z] [v'_1/x] \approx_j e_2 [v_2/z] [v'_2/x] : \mathcal{C} \llbracket S'' \rrbracket)) \end{aligned}$$

$$\begin{aligned} e_1 \approx_k e_2 : \mathcal{C} \llbracket S \rrbracket &\iff S \triangleq T \triangleleft U \\ &\vdash_1 e_i : T \wedge (\forall j < k . (e_1 \mapsto^{\leq j} v_1 \wedge e_2 \mapsto^{\leq j} v_2) \implies v_1 \approx_{k-j} v_2 : \mathcal{V} \llbracket S \rrbracket) \end{aligned}$$

**Definition 20** (Satisfactory substitution). A substitution  $\gamma$  satisfies type environment  $\Gamma$ , noted  $\gamma \models \Gamma$ , iff  $\text{dom}(\gamma) = \text{dom}(\Gamma) \wedge \forall x \in \text{dom}(\Gamma) . \vdash_1 \gamma(x) : T$  where  $\Gamma(x) \triangleq T \triangleleft U$

**Definition 21** (Related substitutions). *Two substitutions  $\gamma_1$  and  $\gamma_2$  are equivalent for  $k$  steps with respect to a type environment  $\Gamma$ , noted  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$ , if  $\gamma_i \models \Gamma$  and*

$$\forall x \in \text{dom}(\Gamma). \gamma_1(x) \approx_k \gamma_2(x) : \mathcal{V}[\Gamma(x)]$$

**Definition 22** (Type-based relaxed noninterference).

$$\begin{aligned} \text{TRNI}(\Gamma, e, S) \iff S \triangleq T \triangleleft U \quad \wedge \quad \Gamma \vdash_1 e : T \\ \forall k \geq 0. \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma] \implies \gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[S] \end{aligned}$$

**Definition 23** (Equivalence of open terms).

$$\begin{aligned} \Gamma \vdash e_1 \approx e_2 : S \iff S \triangleq T \triangleleft U \\ \Gamma \vdash_1 e_i : T \quad \wedge \\ \forall k \geq 0. \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma] \implies \gamma_1(e_1) \approx_k \gamma_2(e_2) : \mathcal{C}[S] \end{aligned}$$

### A.3.2 Auxiliary lemmas: Simple type system

**Lemma 25.** *If  $\vdash \Gamma$  and  $\Gamma \vdash e : T \triangleleft U$  then  $\Gamma \vdash_1 e : T$*

*Proof.* By induction on typing derivations of  $e$ .

**Case (TVar).**

$$(TVar) \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

Let us denote  $\Gamma(x) \triangleq S \triangleq T \triangleleft U$

**Show:**  $\Gamma \vdash_1 x : T$ , which is in fact **to show:**  $\Gamma \vdash_{\text{sf}} x : T \triangleleft U'$  for some  $U'$

This follows from  $\Gamma(x) \triangleq T \triangleleft U$  by using the rule (T1Var)

**Case (TObj).**

$$(TObj) \frac{S \triangleq T \triangleleft U \quad \text{msig}(T, m_i) = S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}{\Gamma \vdash [z : S \Rightarrow \overline{m(x)} e] : S}$$

**Show:**  $\Gamma \vdash_1 e : T$ , which is in fact to show:  $\Gamma \vdash_{\text{sf}} [z : S \Rightarrow \overline{m(x)} e] : T \triangleleft U_1$  for some  $U_1$

We apply the rule (T1Obj) over the following facts:

- $\forall m_i \in T \quad \Gamma, z : S, x_i : S'_i \vdash_{\text{sf}} e : S''_i$ , where  $S''_i \triangleq T''_i \triangleleft U''_i$ . It follows from to apply (T1Sub) on:

- $\Gamma, z : S, x_i : S'_i \vdash_{\text{sf}} e : T''_i \triangleleft U''_{i1}$  from some  $U''_{i1}$ . This follows from IH.
- $\vdash_{\text{sf}} T''_i \triangleleft U''_{i1} < : T''_i \triangleleft U''_i$

: Hence:  $\Gamma \vdash_{\text{sf}} [z : S \Rightarrow \overline{m(x)}e] : S$ , and then  $U_1 = U$ .

**Case (TSub).**

$$(TSub) \frac{\Gamma \vdash e : S' \quad \cdot \vdash S' < : S \quad \vdash S}{\Gamma \vdash e : S}$$

Let us denote  $S \triangleq T \triangleleft U$  and  $S' \triangleq T' \triangleleft U'$

**Show:**  $\Gamma \vdash_1 e : T$ , which is in fact **to show:**  $\Gamma \vdash_{\text{sf}} e : T \triangleleft U_1$  for some  $U_1$

We conclude that, applying the (T1Sub) rule the following facts:

- $\Gamma \vdash_{\text{sf}} e : T' \triangleleft U'_1$  which follows by IH on  $\Gamma \vdash e : T' \triangleleft U'$ .
- We can build  $S_1 \triangleq T \triangleleft T$ . Note that  $\cdot \vdash_{\text{sf}} S' < : S$  holds, because  $\cdot \vdash T' < : T$  (concluded from  $\cdot \vdash S' < : S$ ) and the  $\cdot \vdash_{\text{sf}} S' < : S$  does not care about the declassification type.

**Case (TmD).**

$$(TmD) \frac{\Gamma \vdash e_1 : T_1 \triangleleft U_1 \quad m \in U_1 \quad \text{msig}(U_1, m) = S' \rightarrow S'' \quad \Gamma \vdash e_2 : S'}{\Gamma \vdash e_1.m(e_2) : S''}$$

Denoting  $S' \equiv T' \triangleleft U'$  and  $S'' \equiv T'' \triangleleft S''$

**Show:**  $\Gamma \vdash e_1.m(e_2) : S'' \implies \Gamma \vdash_1 e_1.m(e_2) : T''$

by IH we have:

- $\Gamma \vdash e_1 : T_1 \triangleleft U_1 \implies \Gamma \vdash_1 e_1 : T_1$ , which is:  $\Gamma \vdash_{\text{sf}} e_1 : T_1 \triangleleft T_{11}$  for some  $T_{11}$
- $\Gamma \vdash e_2 : T' \triangleleft U' \implies \Gamma \vdash_1 e_2 : T'$ , which is:  $\Gamma \vdash_{\text{sf}} e_2 : T' \triangleleft U'_{11}$  for some  $U'_{11}$

We can apply rule (T1mI) with the following facts:

- $m \in T_1$ . This follows from  $T_1 < : U_1$  from  $\cdot \vdash_s T_1 \triangleleft U_1$  obtained with Lemma 23. Then we denote  $\text{msig}(T_1, m) = S'_1 \rightarrow S''_2$  where  $\cdot \vdash S'_1 \rightarrow S''_2 < : S' \rightarrow S''$  (because  $\cdot \vdash T_1 < : U_1$ ).
- $\Gamma \vdash_{\text{sf}} e_2 : S'_1$ . We derived it with the rule (T1Sub) applied to:
  - $\Gamma \vdash_{\text{sf}} e_2 : T' \triangleleft U'_{11}$  for some  $T_{11}$  (by IH)
  - $\vdash_{\text{sf}} T' \triangleleft U'_{11} < : S'_1$ 
    - \*  $\vdash_{\text{sf}} S' < : S'_1$  (because  $\vdash S'_1 \rightarrow S''_2 < : S' \rightarrow S''$ )
    - \*  $\vdash_{\text{sf}} T' \triangleleft U'_{11} < : T' \triangleleft U'$ . Note  $\cdot \vdash T' < : T'$  holds by reflexivity of subtyping.
 The we apply transitivity of subtyping.

Hence:  $\Gamma \vdash_{\text{sf}} e_1.m(e_2) : S''_2$ . Let us denote  $S''_2 \triangleq T''_2 \triangleleft U''_2$ .

Then we can apply the rule (T1Sub) over:

- $\Gamma \vdash_{\text{sf}} e_1.m(e_2) : S_2''$
- $\vdash_{\text{sf}} S_2'' <: S''$  because  $\vdash S_2'' <: S''$  (it follows from  $\vdash S_1' \rightarrow S_2'' <: S' \rightarrow S''$ ).

Hence:  $\Gamma \vdash_{\text{sf}} e_1.m(e_2) : S''$  which is equivalent to:  
 $\Gamma \vdash_{\text{sf}} e_1.m(e_2) : T'' \triangleleft U''$

Hence:  $\Gamma \vdash_1 e_1.m(e_2) : T''$

**Case (TmH).** Same structure that the case (TmD). It is simpler because we do not need to use (T1Sub) after apply (T1mI).

□

### Lemma 26.

$$\vdash \Gamma \wedge \Gamma \vdash_1 e : T \implies \exists U. \Gamma \vdash e : T \triangleleft U$$

*Proof.* By induction of the typing derivation of  $\Gamma \vdash_1 e : T$ . In all the case we choice  $U$  equals to the safety type  $T$ . □

### A.3.3 Auxiliary lemmas

We define some auxiliary lemmas that we use in the proof of RNI.

**Lemma 27** (Downward closed/Monotonicity).

*If  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  and  $j \leq k$  then  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$*

*Proof.* The proof is by induction hypothesis in the structure of close type  $S$ . We have only one case.

**Case** ( $S \triangleq O_1 \triangleleft O_2$ ). We have  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  and we need **to show**:  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$ . We proof this by strong induction on  $j$ . The case  $j = 0$  is trivial, because we only need to prove that both objects are (simple) well-typed (which it is also a requirement of the case  $k > 0$ ). Then we focus on an arbitrary  $k > 0$

Let us denote  $v_i \triangleq [z : \_ \Rightarrow \_]$ . By unfolding the definition of  $\mathcal{V}[[S]]$  we have two proof obligations:

- $\vdash_1 v_i : O_1$ . Directly from definition of  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$
- Considering arbitrary  $m, i, v_1', v_2'$  such that:
  - $m \in O_2$     $\text{msig}(U, m) = S' \rightarrow S''$     $\text{methimpl}(v_1, m) = x.e_1$     $\text{methimpl}(v_2, m) = x.e_2$
  - $i < j$ .    $v_1' \approx_i v_2' : \mathcal{V}[[S']]$

**Show:**

1.  $v_1 \approx_i v_2 : \mathcal{V}[[S]]$ . This follows by induction hypothesis on  $i$  ( $i < j$ ).



2.  $e_1 [v_1/z] [v'_1/x] \approx_i e_2 [v_2/z] [v'_2/x] : \mathcal{C}[[S'']]$ .

Since  $i < j \leq k$ , we can instantiate the definition of  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  with  $m, i, v'_1, v'_2$  and we obtain:

$e_1 [v_1/z] [v'_1/x] \approx_i e_2 [v_2/z] [v'_2/x] : \mathcal{C}[[S'']]$

□

**Lemma 28** (Monotonicity of related substitutions). *If  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[[\Gamma]]$  and  $j \leq k$  then  $\gamma_1 \approx_j \gamma_2 : \mathcal{G}[[\Gamma]]$*

*Proof.* Directly from the fact that a type substitution is a partial map (*i.e.* it does not repeat variables) and Lemma 27 □

**Lemma 29** (Subtyping relation).

*If  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  and  $\vdash S <: S'$  then  $v_1 \approx_k v_2 : \mathcal{V}[[S']]$*

*Proof.* By strong induction on  $k$ . The case  $k = 0$  is trivial so we focus in an arbitrary  $k$ .

Let us denote  $S \triangleq T \triangleleft U$  and  $S' \triangleq T' \triangleleft U'$

By unfolding the definition of  $v_1 \approx_k v_2 : \mathcal{V}[[S']]$  we have the following proof obligations:

- $\vdash_1 v_i : T'$ . From  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  we know  $\vdash_1 v_i : T$ . From the hypothesis  $\vdash S <: S'$  we know  $\vdash_{\text{sf}} T <: T'$ . Then we can use the rule (T1Sub) to conclude  $\vdash_1 v_i : T'$
- Assuming arbitrary  $m, j, v'_1, v'_2$  such that
  - $m \in U'$   $\text{msig}(U', m) = S'_1 \rightarrow S'_2$  where  $S'_1 \triangleq T'_1 \triangleleft U'_1$  and  $S'_2 \triangleq T'_2 \triangleleft U'_2$
  - $\text{methimpl}(v_1, m) = x.e_1$  and  $\text{methimpl}(v_2, m) = x.e_2$
  - $j < k$ .  $v'_1 \approx_j v'_2 : \mathcal{V}[[S'_1]]$

**Show:**

1.  $v_1 \approx_j v_2 : \mathcal{V}[[S']]$ . By instantiating the definition of  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  with  $j < k$  (the other parameters are not important in this case) we obtain  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$ . Then by applying induction hypothesis on  $j < k$  and  $\vdash S <: S'$ , we obtain  $v_1 \approx_j v_2 : \mathcal{V}[[S']]$
2.  $e_1 [v_1/z] [v'_1/x] \approx_j e_2 [v_2/z] [v'_2/x] : \mathcal{C}[[S'_2]]$

By expanding the above definition we have a new proof obligation under the following assumptions:

Consider arbitrary  $j', v''_1, v''_2$  such that

- $j' < j$
- $e_1 [v_1/z_1] [v'_1/x] \mapsto^{j_1} v''_1$
- $e_2 [v_2/z_2] [v'_2/x] \mapsto^{j_2} v''_2$
- $j_i < j'$

**Show:**  $v''_1 \approx_{j-j'} v''_2 : \mathcal{V}[[S'_2]]$

We instantiate now the second conjunction of the  $v_1 \approx_k v_2 : \mathcal{V}[[S]]$  with  $j, m, v'_1$  and  $v'_2$ . Note we can do it because:

- $m \in T$ . This follows from  $m \in T'$  and  $\vdash T <: T'$ . Then let us denote  $\text{msig}(U, m) = S_1 \rightarrow S_2$
- $j < k$
- $v'_1 \approx_j v'_2 : \mathcal{V}[[S_1]]$ . This follows from the induction hypothesis applied to  $v'_1 \approx_j v'_2 : \mathcal{V}[[S'_1]]$  and  $\vdash S'_1 <: S_1$  (which follows from  $\vdash S <: S'$ )

Hence we obtain  $e_1 [v_1/z] [v'_1/x] \approx_j e_2 [v_2/z] [v'_2/x] : \mathcal{C}[[S_2]]$

Now we instantiate the above definition with  $j'$ . Note we can do it because  $j' < j$ . Then we obtain:

$v''_1 \approx_{j-j'} v''_2 : \mathcal{V}[[S_2]]$ . Note  $j - j' < k$  and  $\vdash S_2 <: S'_2$ . Then we can apply the induction hypothesis to obtain:

$v''_1 \approx_{j-j'} v''_2 : \mathcal{V}[[S'_2]]$

□

**Lemma 30.** *If  $e_1 \approx_k e_2 : \mathcal{C}[[S]]$  and  $S <: S'$  then  $e_1 \approx_k e_2 : \mathcal{C}[[S']]$*

*Proof.* The proof is straightforward by unfolding the definitions  $e_1 \approx_k e_2 : \mathcal{C}[[S]]$  and  $e_1 \approx_k e_2 : \mathcal{C}[[S']]$  and then applying Lemma 29 □

**Lemma 31** (Substitution preserves typing).

*If  $\Gamma, x : T' \triangleleft U' \vdash_1 e : T$  and  $\vdash_1 v : T'$  then  $\Gamma \vdash_1 e [v/x] : T$*

*Proof.* By induction on the typing derivations of  $e$ . The (T1Var), (T1Sub), (T1mI) cases are standard [41]. Cases (T1Obj) is trivial too, it uses the standard lemmas: “permutation” and “weakening”. □

**Lemma 32** (Value substitution preserves simple typing).

*If  $\Gamma \vdash_1 e : T$  and  $\gamma \models \Gamma$  then  $\vdash_1 \gamma(e) : T$*

*Proof.* By induction on  $\Gamma$

**Case** ( $\Gamma = \cdot$ ). *We conclude  $\gamma = \emptyset$ . Then we need to **show**:  $\vdash_1 e : T$  which is exactly the hypothesis.*

**Case** ( $\Gamma = \Gamma_1, x : T_1 \triangleleft U_1$ ). *From  $\gamma \models \Gamma$  we know that there exists  $\gamma_1$  and  $x \mapsto v$  such that  $\gamma_1 \models \Gamma$  and  $\vdash_1 v : T_1$ . (This holds because  $\Gamma$  does not repeat variables).*

*Then we need to **show** :  $\vdash_1 \gamma_1 [x \mapsto v] (e) : T$*

*By applying Lemma 31 on  $\Gamma, x : T_1 \triangleleft U_1 \vdash_1 e : T$  and  $\vdash_1 v : T_1$  we obtain  $\Gamma_1 \vdash_1 e [v/x] : T$ . Then we apply the IH on  $\Gamma_1 \vdash_1 e [v/x] : T$  and  $\gamma_1 \models \Gamma_1$  and we obtain:*

$\vdash_1 \gamma_1(e[v/x]) : T$ , hence:  $\vdash_1 \gamma(e) : T$ . (the order of application of substitutions does not matter)

□

**Lemma 33** (Object inversion lemma).

If  $\vdash_1 [z : T \triangleleft U \Rightarrow \_ ] : T$  and  $\text{msig}(T, m) = S_1 \rightarrow T_2 \triangleleft U_2$  then  $\text{methimpl}(v_1, m) = x.e_{11}$  and  $\cdot, z : T \triangleleft U, x : S_1 \vdash_1 e_{11} : T_2$  for some  $x$  and  $e_{11}$ .

**Lemma 34** (Subject reduction).

If  $\vdash_1 e : T$  and  $e \mapsto^* e'$  then  $\vdash_1 e' : T$

*Proof.* Straightforward induction on the typing derivation of  $e$ . The case (T1Var) have no sense, because  $x$  is not well-typed under an empty type environment.

**Case (T1Sub).** By IH we obtain  $\vdash_1 e : T'$ . Then we apply (T1Sub) to obtain  $\vdash_1 e' : T$

**Case (T1Obj).** The expression  $e$  is a value, then  $e = e'$ , so the result is direct.

**Case (T1mI).**

$$(T1mI) \frac{\Gamma \vdash_{\text{sf}} e_1 : T \triangleleft U \quad \text{msig}(T, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash_{\text{sf}} e_2 : S_1}{\Gamma \vdash_{\text{sf}} e_1.m(e_2) : S_2}$$

$e = e_1.m(e_2)$

Let us denote:  $S_1 \triangleq T_1 \triangleleft T_1$     $S_2 \triangleq T_2 \triangleleft U_2$

**Show:**  $\vdash_1 e' : T_2$

We have three sub-cases:

- $e_1 \mapsto^* e'_1$ . Apply IH with  $e_1$ ,  $T$  and  $e'_1$  to obtain  $\vdash_1 e'_1 : T$ . Then apply rule T1mI to obtain that  $\vdash_1 e'_1.m(e_2) : T_2$
- $e_1 = v_1, e_2 \mapsto^* e'_2$ . Apply IH with  $e_2$ ,  $T_1$  and  $e'_2$  to obtain  $\vdash_1 e'_2 : T_1$ . Then apply rule T1mI to obtain that  $\vdash_1 e_1.m(e'_2) : T_2$
- $e_1 = v_1, e_2 = v_2$ . Let us denote  $v_1 \triangleq [z : T \triangleleft U \Rightarrow \_ ]$ . Apply Lemma 33 with  $\vdash_1 [z : T \triangleleft U \Rightarrow \_ ] : T$  and  $\text{msig}(T, m) = S_1 \rightarrow T_2 \triangleleft U_2$  to obtain that  $\text{methimpl}(v_1, m) = x.e_{11}$  and  $\cdot, z : T \triangleleft U, x : S_1 \vdash_1 e_{11} : T_2$  for some  $x$  and  $e_{11}$ .

**Show:**  $\vdash_1 e_{11}[z/v_1][x/v_2] : T_2$

Let us denote  $\gamma \triangleq \emptyset [z \mapsto v_1][x \mapsto v_2]$  and  $\Gamma \triangleq \cdot, z : T \triangleleft U, x : S_1$ . Note that  $\gamma \models \Gamma$

Our goal can be rewritten **to show:**  $\vdash_1 \gamma(e_{11}) : T_2$

Then we apply Lemma 32 on  $\Gamma \vdash_1 e_{11} : T_2$  and  $\gamma \models \Gamma$  to obtain  $\vdash_1 \gamma(e_{11}) : T_2$ .

□

### A.3.4 Proof of the Fundamental Property

**Lemma 8** (Fundamental property).  $\Gamma \vdash e : S \implies \Gamma \vdash e \approx e : S$

*Proof.* By induction on the typing derivation of  $e$ , where we assume  $\vdash \Gamma$ .

**Case** (TVar).

$$(TVar) \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

Denoting  $S \triangleq T \triangleleft U$

**Show:**  $\Gamma \vdash x \approx x : T \triangleleft U$

Expanding this definition we have two proof obligations:

- $\Gamma \vdash_1 x : T$ . We have  $\Gamma \vdash x : T \triangleleft U$  and  $\vdash \Gamma$  from assumption. Then we apply Lemma 25 to obtain  $\Gamma \vdash_1 x : T$ .
- $\forall k \geq 0. \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma] \implies \gamma_1(x) \approx_k \gamma_2(x) : \mathcal{C}[S]$   
Considering arbitrary  $k, \gamma_1, \gamma_2$  such that  $k \geq 0$  and  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$ , we are required **to show:**  $\gamma_1(x) \approx_k \gamma_2(x) : \mathcal{C}[S]$   
This follows directly from the definition of  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$  given that  $x \in \text{dom}(\Gamma)$ .

**Case** (TSub).

$$(TSub) \frac{\Gamma \vdash e : S' \quad \vdash S' <: S \quad \vdash S}{\Gamma \vdash e : S}$$

**Show:**  $\Gamma \models e \approx e : S$ . By unfolding this definition we have the new proof obligations:  
Let us denote  $S \triangleq T \triangleleft U$

- $\Gamma \vdash_1 e : T$ . It follows from Lemma 25
- Consider arbitrary  $k, \gamma_1, \gamma_2$  such that  $k \geq 0$  and  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$

**Show:**  $\gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[S]$ .

By IH on  $\Gamma \vdash e : S'$  we obtain  $\Gamma \models e \approx e : S'$ . Instantiating this definition with  $k, \gamma_1, \gamma_2$  we obtain:

$\gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[S']$ . From hypothesis we know  $\vdash S' <: S$  then we can apply Lemma 30 to obtain:

$\gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[S]$

Case (TObj).

$$(TObj) \frac{\vdash S \quad S \triangleq T \triangleleft U \quad \text{msig}(T, m_i) = S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}{\Gamma \vdash [z : S \Rightarrow \overline{m(x)e}] : S}$$

**Show:**  $\Gamma \vdash [z : S \Rightarrow \overline{m(x)e}] \approx [z : S \Rightarrow \overline{m(x)e}] : S$

Expanding the above definition we have two proof obligations:

- $\Gamma \vdash_1 [z : S \Rightarrow \overline{m(x)e}] : T$ . It follows from Lemma 25
- Consider arbitrary  $k, \gamma_1, \gamma_2$  such that  $k \geq 0$  and  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$

**Show:**  $\gamma_1([z : S \Rightarrow \overline{m(x)e}]) \approx_k \gamma_2([z : S \Rightarrow \overline{m(x)e}]) : \mathcal{C}[S] \equiv$   
 $[z : S \Rightarrow \overline{m(x)\gamma_1(e)}] \approx_k [z : S \Rightarrow \overline{m(x)\gamma_2(e)}] : \mathcal{C}[S]$

Let us denote  $e_1 \triangleq [z : S \Rightarrow \overline{m(x)\gamma_1(e)}]$  and  $e_2 \triangleq [z : S \Rightarrow \overline{m(x)\gamma_2(e)}]$

Then expanding the definition of  $\mathcal{C}[S]$  we have new proof obligations:

- $\vdash_1 e_1 : T$ . By applying Lemma 25 on  $\Gamma \vdash [z : S \Rightarrow \overline{m(x)e}] : S$  we obtain  $\Gamma \vdash_1 [z : S \Rightarrow \overline{m(x)e}] : T$ . We also have  $\gamma_1 \models \Gamma$  from  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$ . Then we can apply Lemma 32 and obtain:  $\vdash_1 \gamma_1([z : S \Rightarrow \overline{m(x)e}]) : T$ .
- $\vdash_1 e_2 : T$ . The same reasoning that above.
- Consider arbitrary  $j, v_1$  and  $v_2$  such that:
  - \*  $j < k$
  - \*  $e_1 \mapsto^{j_1} v_1$
  - \*  $e_2 \mapsto^{j_2} v_2$
  - \*  $j_i \leq j$

**Show:**  $v_1 \approx_{k-j} v_2 : \mathcal{V}[S]$

Since  $e_1 \triangleq [z : S \Rightarrow \overline{m(x)\gamma_1(e)}]$  and  $e_2 \triangleq [z : S \Rightarrow \overline{m(x)\gamma_2(e)}]$  are already values we have  $j_i = 0$ , so  $v_1 = e_1$  and  $v_2 = e_2$ .

Then we need **to show**  $v_1 \approx_k v_2 : \mathcal{V}[S]$ .

We prove this goal using strong induction on  $k$ . The case  $k = 0$  is trivial, since we only need to show both  $v_1$  and  $v_2$  are (simple) well-typed and this is also part of the general case. We focus then in an arbitrary  $k > 0$ .

Expanding the definition of  $v_1 \approx_k v_2 : \mathcal{V}[S]$  we have the following proof obligations:

- $\vdash_1 v_i : T$ . This is directly, because  $e_i = v_i$  and we already showed  $\vdash_1 e_i : T$
- Consider arbitrary  $m_l, j', v'_1, v'_2$  such that:
  - $m_l \in U$  and  $\text{msig}(U, m_l) = S' \rightarrow S''$  where  $S' \triangleq T' \triangleleft U'$
  - $\text{methimpl}(v_1, m_l) = x.\gamma_1(e_l)$  and  $\text{methimpl}(v_2, m_l) = x.\gamma_2(e_l)$
  - $j' < k$ .  $v'_1 \approx_{j'} v'_2 : \mathcal{V}[S']$

**Show:**

1.  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$ . This follows by induction hypothesis on  $j'$  ( $j' < k$ )
2.  $\gamma_1(e_l) [v_1/z] [v'_1/x] \approx_{j'} \gamma_2(e_l) [v_2/z] [v'_2/x] : \mathcal{C}[[S'']]$ .

*Proof of point 2.*

Let us denote  $\text{msig}(T, m_l) = S'_l \rightarrow S''_l$ ,  $S'_l \triangleq T'_l \triangleleft U'_l$  and  $S''_l \triangleq T''_l \triangleleft U''_l$   
 By IH on  $\Gamma, z : S, x : S'_l \vdash e_l : S''_l$  we obtain  $\Gamma, z : S, x : S'_l \vdash e_l \approx e_l : S''_l$ .

From this definition we obtain:

$$- \forall k_1 \geq 0 \quad \forall \gamma'_1, \gamma'_2. \gamma'_1 \approx_{k_1} \gamma'_2 : \mathcal{G}[[\Gamma, z : S, x : S'_l]] \implies \gamma'_1(e_l) \approx_{k_1} \gamma'_2(e_l) : \mathcal{C}[[S''_l]]$$

We instantiate the above fact with  $j'$  as  $k_1$ ,  $\gamma_1 [z \mapsto v_1] [x \mapsto v'_1]$  as  $\gamma'_1$  and  $\gamma_2 [z \mapsto v_2] [x \mapsto v'_2]$  as  $\gamma'_2$ . Note we can do it because:

- $j' \geq 0$  (indexes are natural numbers)
- $\gamma_1 [z \mapsto v_1] [x \mapsto v'_1] \approx_{j'} \gamma_2 [z \mapsto v_2] [x \mapsto v'_2] : \mathcal{G}[[\Gamma, z : S, x : S'_l]]$ . We can ensure this because:
  - \*  $\gamma_i [z \mapsto v_i] [x \mapsto v'_i] \models \Gamma, z : S, x : S'_l$ . It follows from:
    - $z \notin \text{dom}(\Gamma)$  and  $x \notin \text{dom}(\Gamma)$  (we considered environments that do not repeated variables)
    - $\vdash_1 v_i : T$ . It follows from  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$
    - $\vdash_1 v'_i : T'_l$ . From  $v'_1 \approx_{j'} v'_2 : \mathcal{V}[[S']]$ , we can conclude  $\vdash_1 v'_i : T'$ . From  $\vdash S$  (well-formedness of  $S$ ) we know  $\vdash T' <: T'_l$ . Then we apply (T1Sub) to conclude our goal.
  - \*  $\gamma_1 \approx_{j'} \gamma_2 : \mathcal{G}[[\Gamma]]$ . This follows applying Lemma 28 on  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[[\Gamma]]$  and  $j' < k$ .
  - \*  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$  (we already prove it in the point 1. )
  - \*  $v'_1 \approx_{j'} v'_2 : \mathcal{V}[[S'_l]]$ . We considered  $v'_1 \approx_{j'} v'_2 : \mathcal{V}[[S']]$ . From  $\vdash S$  we know  $\vdash S' <: S'_l$ . Then we can apply Lemma 29 to obtain the goal.

Hence  $\gamma_1 [z \mapsto v_1] [x \mapsto v'_1] (e_l) \approx_{j'} \gamma_2 [z \mapsto v_2] [x \mapsto v'_2] (e_l) : \mathcal{C}[[S''_l]]$ , which is equivalent to:  $\gamma_1(e_l) [v_1/z] [v'_1/x] \approx_{j'} \gamma_2(e_l) [v_2/z] [v'_2/x] : \mathcal{C}[[S''_l]]$ .

From well-formed constraint  $\vdash S$ , we know  $\vdash S''_l <: S''$ . Then we apply Lemma 30 to obtain:

$$\gamma_1(e_l) [v_1/z] [v'_1/x] \approx_{j'} \gamma_2(e_l) [v_2/z] [v'_2/x] : \mathcal{C}[[S'']]$$

**Case (TmD).**

$$\frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \in U \quad \text{msig}(U, m) = S_1 \rightarrow S_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2}$$

Denoting  $S_2 \triangleq T_2 \triangleleft U_2$

**Show:**  $\Gamma \vdash e_1.m(e_2) \approx e_1.m(e_2) : S_2$

Expanding this definition have we two proof obligations:

- $\Gamma \vdash_1 e_1.m(e_2) : T_2$ . It follows from Lemma 25

- Consider arbitrary  $k, \gamma_1, \gamma_2$  such that  $k \geq 0$ ,  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$

**Show:**  $\gamma_1(e_1.m(e_2)) \approx_k \gamma_2(e_1.m(e_2)) : \mathcal{C}[\mathbb{S}_2]$ .

Let us denote  $e'_1 \triangleq \gamma_1(e_1.m(e_2))$  and  $e'_2 \triangleq \gamma_2(e_1.m(e_2))$

Expanding the definition of  $e'_1 \approx_k e'_2 : \mathcal{C}[\mathbb{S}_2]$  we have the following new proof obligations:

- $\vdash_1 e'_1 : T_2$ . From the definition of  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\Gamma]$  we get  $\gamma_1 \models \Gamma$ . We also know  $\Gamma \vdash_1 e_1.m(e_2) : T_2$ , then we apply Lemma 32 to get  $\vdash_1 e'_1 : T_2$
- Consider arbitrary  $j, v_1, v_2$  such that:
  - \*  $j < k$
  - \*  $\gamma_1(e_1).m(\gamma_1(e_2)) \mapsto^{l_1} v_1$
  - \*  $\gamma_2(e_1).m(\gamma_2(e_2)) \mapsto^{l_2} v_2$
  - \*  $l_i \leq j$

**Show:**  $v_1 \approx_{k-j} v_2 : \mathcal{V}[\mathbb{S}_2]$

Because of the above assumptions and by inspecting the dynamic semantic we know that there exists  $j_{11}, j_{21}, j_{12}, j_{22}, j_{13}$  and  $j_{23}$  such that:

$$\begin{array}{ll}
 \gamma_1(e_1.m(e_2)) \equiv \gamma_1(e_1).m(\gamma_1(e_2)) & \gamma_2(e_1.m(e_2)) \equiv \gamma_2(e_1).m(\gamma_2(e_2)) \\
 \mapsto^{j_{11}} v_{11}.m(\gamma_1(e_2)) \mapsto^{j_{12}} v_{11}.m(v_{12}) & \mapsto^{j_{21}} v_{21}.m(\gamma_2(e_2)) \mapsto^{j_{22}} v_{21}.m(v_{22}) \\
 \mapsto^1 e_{11} [v_{11}/z] [v_{12}/x] & \mapsto^1 e_{21} [v_{21}/z] [v_{22}/x] \\
 \mapsto^{j_{13}} v_1 & \mapsto^{j_{23}} v_2 \\
 \text{where } l_1 = j_{11} + j_{12} + 1 + j_{13} & \text{where } l_2 = j_{21} + j_{22} + 1 + j_{23} \\
 v_{11} = [z : \_ \Rightarrow \_] & v_{21} = [z : \_ \Rightarrow \_] \\
 \text{methimpl}(v_{11}, m) = x.e_{11} & \text{methimpl}(v_{21}, m) = x.e_{21}
 \end{array}$$

Let us denote  $S' \triangleq T' \triangleleft U'$ .

By IH on  $\Gamma \vdash e_1 : S$  and  $\Gamma \vdash e_2 : S_1$  we obtain:

- $\gamma_1(e_1) \approx_k \gamma_2(e_1) : \mathcal{C}[S]$ .
- $\gamma_1(e_2) \approx_k \gamma_2(e_2) : \mathcal{C}[S_1]$ .

Instantiating the definition of  $\mathcal{C}[S]$  with  $j_1, v_{11}, v_{21}$  and  $\mathcal{C}[S']$  with  $j_2, v_{12}, v_{22}$  we obtain:

- $v_{11} \approx_{k-j_1} v_{21} : \mathcal{V}[S]$  where  $j_1 = \max(j_{11}, j_{21})$
- $v_{12} \approx_{k-j_2} v_{22} : \mathcal{V}[S_1]$  where  $j_2 = \max(j_{12}, j_{22})$

We denote  $j' = \max(j_1, j_2) + 1$ . (The “smart” choice of  $j'$  must satisfies  $j' > j_1$ ,  $j' > j_2$  and  $\max(j_{13}, j_{23}) < j - j'$ )

Now we instantiate the final conjunct of  $v_{11} \approx_{k-j_1} v_{21} : \mathcal{V}[S]$  with  $k - j'$ ,  $m, v_{12}, v_{22}$ . Note we can do it because:

- $m \in U$ . It follows from the assumption in the typing rule.
- $k - j' < k - j_1$  ( $j' > j_1$ )
- $v_{12} \approx_{k-j'} v_{22} : \mathcal{V}[S_1]$ . This follows from to apply Lemma 27 to  $v_{12} \approx_{k-j_2} v_{22} : \mathcal{V}[S_1]$  with  $k - j' < k - j_2$  ( $j' > j_2$ )

Hence:  $e_{11} [v_{11}/z] [v_{12}/x] \approx_{k-j'} e_{21} [v_{21}/z] [v_{22}/x] : \mathcal{C}[\mathbb{S}_2]$ .

We denote  $j_3 = \max(j_{13}, j_{23})$  and then we instantiate the above definition (of  $\mathcal{C}[\mathbb{S}_2]$ ) with  $j_3, v_1, v_2$ . Note with can do it because:

$$- j_3 < k - j' \quad (j_3 \leq j - j' < k - j')$$

Hence:  $v_1 \approx_{k-j'-j_3} v_2 : \mathcal{V}[[S_2]]$ . We know that  $j \geq j' + j_3$ , then  $k - j \leq k - j' - j_3$  and we can apply Lemma 27 to obtain:

$$v_1 \approx_{k-j} v_2 : \mathcal{V}[[S_2]]$$

**Case (TmH).**

$$\frac{\Gamma \vdash e_1 : T \triangleleft U \quad m \notin U \quad \text{msig}(T, m) = S_1 \rightarrow T_2 \triangleleft U_2 \quad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \top}$$

Let us denote  $S_2 \triangleq T_2 \triangleleft \top$

**Show:**  $\Gamma \vdash e_1.m(e_2) \approx e_1.m(e_2) : T_2 \triangleleft \top$

Expanding the above definition we have new proof obligations:

- $\Gamma \vdash_1 e_1.m(e_2) : T_2$ . It follows from Lemma 25
- Consider arbitrary  $k, \gamma_1, \gamma_2$  such that  $k \leq 0$  and  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[[\Gamma]]$

**Show:**  $\gamma_1(e_1.m(e_2)) \approx_k \gamma_2(e_1.m(e_2)) : \mathcal{C}[[S_2]]$

Let us denote  $e'_1 = \gamma_1(e_1.m(e_2))$  and  $e'_2 \triangleq \gamma_2(e_1.m(e_2))$  Expanding the above definition we have new proof obligations:

- $\vdash_1 e'_1 : T_2$ . From the definition of  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[[\Gamma]]$  we get  $\gamma_1 \models \Gamma$ . We also know  $\Gamma \vdash_1 e_1.m(e_2) : T_2$ , then we apply Lemma 32 to get  $\vdash_1 e'_1 : T_2$
- Consider arbitrary  $j, v_1, v_2$  such that:
  - \*  $j < k$
  - \*  $\gamma_1(e_1.m(e_2)) \mapsto^{j_1} v_1$ .
  - \*  $\gamma_2(e_1.m(e_2)) \mapsto^{j_2} v_2$ .
  - \*  $j_i \leq j$

**Show:**  $v_1 \approx_j v_2 : \mathcal{V}[[S]]$ . This follows directly from:

- \*  $\vdash_1 v_i : T_2$ . This follows from the subject reduction (Lemma 34)
- \* There is no  $m$  in  $\top$  which vacuously satisfies the final conjunct.

□

**Lemma 7 (Self-equivalence).**  $\Gamma \vdash e \approx e : S \implies \text{TRNI}(\Gamma, e, S)$

*Proof.* **Show:**  $\text{TRNI}(\Gamma, e, S)$

Let us denote  $S \triangleq T \triangleleft U$

Unfolding the above definition we have the following proof obligations:

- $\Gamma \vdash_1 e : T$ . It follows directly from  $\Gamma \vdash e \approx e : S$
- Considering arbitrary  $k, \gamma_1, \gamma_2$  such that:  $k \geq 0$  and  $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[[\Gamma]]$   
**Show:**  $\gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[[S]]$ . We obtain this by instantiating the last conjunction of  $\Gamma \vdash e \approx e : S$  with  $k, \gamma_1$  and  $\gamma_2$ .



□

# Appendix B

## Existential relaxed noninterference

We include now the complete definitions and proofs for Chapter 3.

### B.1 Full syntax and semantics

#### B.1.1 Syntax

$e$	$::=$	$\lambda x : S.e \mid e e \mid x \mid \mathbf{b} \mid e \oplus e \mid \langle \rangle \mid \langle e, e \rangle$	(terms)
		$\mid \mathbf{fst} e \mid \mathbf{snd} e \mid \mathbf{inl} e \mid \mathbf{inr} e \mid \mathbf{case} t \text{ of } \mathbf{inl} x.e \mid \mathbf{inr} x.e$	
		$\mid \mathbf{pack}(T, e) \text{ as } T \mid \mathbf{open}(X, x) = e \text{ in } e$	
$v$	$::=$	$\lambda x : S.e \mid \mathbf{b} \mid \langle \rangle \mid \langle v, v \rangle \mid \mathbf{inl} v \mid \mathbf{inr} v \mid \mathbf{pack}(T, v) \text{ as } T$	(values)
$T, U$	$::=$	$S \rightarrow S \mid P \mid \mathbf{1} \mid S + S \mid S \times S \mid \exists X.T \mid X \mid \top$	(types)
$S$	$::=$	$T@U$	(security types)
$P$	$::=$	( <i>e.g.</i> $\mathbf{Int}, \mathbf{String}$ )	(primitive types)
$\Gamma$	$::=$	$\bullet \mid \Gamma, x : S$	(type environments)
$\Delta$	$::=$	$\bullet \mid \Delta, X : T$	(type variable environments)

$x, y, z$	(variables)		
$X$	(type variables)	$T_{\mathbf{L}} \triangleq T@T$	$T_{\mathbf{H}} \triangleq T@\top$

Figure B.1:  $\lambda_{\text{SEC}}^{\exists}$ : Syntax (Appendix)

## B.1.2 Subtyping

$S <: S$

$$\frac{}{T@U <: T@U} \quad \frac{}{T@U <: T@T} \quad \frac{}{T@T <: T@X}$$

Figure B.2:  $\lambda_{\text{SEC}}^{\exists}$ : Subtyping of security types

### B.1.3 Auxiliary definitions

$$\text{sftype}(\exists X.T) = \begin{cases} T' & \text{sftypes}(X, T) \setminus X = \{T'\} \setminus X \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\text{sftypes}(X, 1) = \{\}$$

$$\text{sftypes}(X, X) = \{\}$$

$$\text{sftypes}(X, Y) = \{\}$$

$$\text{sftypes}(X, P) = \{\}$$

$$\text{sftypes}(X, S_1 \rightarrow S_2) = \text{sftypes}(X, S_1) \cup \text{sftypes}(X, S_2)$$

$$\text{sftypes}(X, S_1 \times S_2) = \text{sftypes}(X, S_1) \cup \text{sftypes}(X, S_2)$$

$$\text{sftypes}(X, S_1 \times S_2) = \text{sftypes}(X, S_1) \cup \text{sftypes}(X, S_2)$$

$$\text{sftypes}(X, S_1 \times S_2) = \text{sftypes}(X, S_1) \cup \text{sftypes}(X, S_2)$$

$$\text{sftypes}(X, \exists Y.T) = \text{sftypes}(X, T)$$

$$\text{sftypes}(X, \exists X.T) = \{\}$$

$$\text{sftypes}(X, T@X) = \text{sftypes}(X, T) \cup \{T\}$$

$$\text{sftypes}(X, T@Y) = \text{sftypes}(X, T)$$

$$\text{sftypes}(X, T@\top) = \text{sftypes}(X, T)$$

$$\text{sftypes}(X, T@T) = \text{sftypes}(X, T)$$

$$\boxed{T \sqsubseteq T}$$

$$\frac{}{T \sqsubseteq T} \quad \frac{}{T \sqsubseteq X}$$

### B.1.4 Well-formedness of types: closed types

The Figure B.3 shows the judgments  $\Delta \vdash U$  and  $\Delta \vdash S$  to justify that the type  $U$  and  $S$  are *closed* under the type variable environment  $\Delta$  respectively.

$\Delta \vdash U$

$$\begin{array}{c}
 \frac{X : \_ \in \Delta}{\Delta \vdash X} \quad \frac{}{\Delta \vdash P} \quad \frac{}{\Delta \vdash 1} \quad \frac{}{\Delta \vdash \top} \quad \frac{\Delta \vdash S_1 \quad \Delta \vdash S_2}{\Delta \vdash S_1 \rightarrow S_2} \\
 \\
 \frac{\Delta \vdash S_1 \quad \Delta \vdash S_2}{\Delta \vdash S_1 \times S_2} \quad \frac{\Delta \vdash S_1 \quad \Delta \vdash S_2}{\Delta \vdash S_1 + S_2} \quad \frac{\Delta, X : \_ \vdash T}{\Delta \vdash \exists X.T}
 \end{array}$$

$\Delta \vdash S$

$$\frac{\Delta \vdash T \quad \Delta \vdash U}{\Delta \vdash T@U}$$

Figure B.3:  $\lambda_{\text{SEC}}^{\exists}$ : Closed types

### B.1.5 Facet-wise well-formedness of security types

The Figure B.4 shows three judgments. The judgment  $\Delta \vdash_{\text{fw}} U$  (resp.  $\Delta \vdash_{\text{fw}} S$ ) holds if the type  $U$  (resp.  $S$ ) is facet-wise well-formed.

The judgment  $\Delta \models S$  holds if the security type  $S$  is facet-wise well-formed ( $\Delta \vdash_{\text{fw}} S$ ) and closed with respect to type variables ( $\Delta \vdash S$ ).

When we later refer that a security type  $S$  is (just) well-formed under a type environment  $\Delta$  we mean that  $\Delta \models S$  holds.

$$\boxed{\Delta \vdash_{\text{fw}} U}$$

$$\frac{}{\Delta \vdash_{\text{fw}} X} \quad \frac{}{\Delta \vdash_{\text{fw}} P} \quad \frac{}{\Delta \vdash_{\text{fw}} 1} \quad \frac{}{\Delta \vdash_{\text{fw}} \top} \quad \frac{\Delta \vdash_{\text{fw}} S_1 \quad \Delta \vdash S_2}{\Delta \vdash_{\text{fw}} S_1 \rightarrow S_2}$$

$$\frac{\Delta \vdash_{\text{fw}} S_1 \quad \Delta \vdash S_2}{\Delta \vdash_{\text{fw}} S_1 \times S_2} \quad \frac{\Delta \vdash_{\text{fw}} S_1 \quad \Delta \vdash_{\text{fw}} S_2}{\Delta \vdash_{\text{fw}} S_1 + S_2} \quad \frac{\text{sftype}(\exists X.T) = T' \quad \Delta \vdash_{\text{fw}} T}{\Delta \vdash_{\text{fw}} \exists X.T}$$

$$\boxed{\Delta \vdash_{\text{fw}} S}$$

$$\frac{\Delta \vdash_{\text{fw}} T}{\Delta \vdash_{\text{fw}} T@T} \quad \frac{\Delta \vdash_{\text{fw}} T}{\Delta \vdash_{\text{fw}} T@\top} \quad \frac{\Delta \vdash_{\text{fw}} T \quad \Delta(X) = T}{\Delta \vdash_{\text{fw}} T@X}$$

$$\boxed{\Delta \models S}$$

$$\frac{\Delta \vdash S \quad \Delta \vdash_{\text{fw}} S}{\Delta \models S}$$

$$\boxed{\Delta \models T}$$

$$\frac{\Delta \vdash T \quad \Delta \vdash_{\text{fw}} T}{\Delta \models T}$$

Figure B.4:  $\lambda_{\text{SEC}}^{\exists}$ : Facet-wise well-formedness of security types

### B.1.6 Well-formedness of environments

The Figure B.5 shows the judgments  $\Delta \models \Gamma$  and  $\models \Delta$ .  $\Delta \models \Gamma$  holds if the type environment  $\Gamma$  is well-formed under the type variable environment  $\Delta$ , which mean two things: 1)  $\Gamma$  does not have repeated variables and 2) each mapped type  $S$  is well-formed under  $\Delta$ . The judgment  $\models \Delta$  holds if a type environment  $\Delta$  does not have repeated type variables names and if all the mapped types  $T$  are also well-formed.

Having formalized well-formedness of environments and types, we assume them in all definitions.

$$\boxed{\Delta \models \Gamma}$$

$$\frac{}{\Delta \models \bullet} \quad \frac{\Delta \models \Gamma \quad \Delta \models S \quad x \notin \text{dom}(\Gamma)}{\Delta \models \Gamma, x : S}$$

$$\boxed{\models \Delta}$$

$$\frac{}{\models \bullet} \quad \frac{\models \Delta \quad X \notin \text{dom}(\Delta) \quad \Delta \models T}{\models \Delta, X : T}$$

Figure B.5:  $\lambda_{\text{SEC}}^{\exists}$ : Well-formedness of environments

## B.1.7 Type system

The Figure B.6 shows the typing rules for  $\lambda_{\text{SEC}}^{\exists}$ .

$$\boxed{\Delta; \Gamma \vdash e : S}$$

$$\begin{array}{c}
\text{(TVar)} \frac{x \in \text{dom}(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \text{(TS)} \frac{\Delta; \Gamma \vdash e : S' \quad S' <: S}{\Delta; \Gamma \vdash e : S} \quad \text{(TP)} \frac{P = \Theta(b)}{\Delta; \Gamma \vdash b : P_{\text{L}}} \\
\text{(TFun)} \frac{\Delta; \Gamma, x : S \vdash e : S'}{\Delta; \Gamma \vdash \lambda x : S. e : (S \rightarrow S')_{\text{L}}} \quad \text{(TPair)} \frac{\Delta; \Gamma \vdash e_1 : S_1 \quad \Delta; \Gamma \vdash e_2 : S_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : (S_1 \times S_2)_{\text{L}}} \\
\text{(TInl)} \frac{\Delta; \Gamma \vdash e : S_1}{\Delta; \Gamma \vdash \text{inl } e : (S_1 + S_2)_{\text{L}}} \quad \text{(TInr)} \frac{\Delta; \Gamma \vdash e : S_2}{\Delta; \Gamma \vdash \text{inr } e : (S_1 + S_2)_{\text{L}}} \\
\text{(TPack)} \frac{\Delta; \Gamma \vdash e : (T [T'/X])_{\text{L}} \quad T' \sqsubseteq \text{sftype}(\exists X.T)}{\Delta; \Gamma \vdash \text{pack}(T', e) \text{ as } \exists X.T : (\exists X.T)_{\text{L}}} \\
\text{(TApp)} \frac{\Delta; \Gamma \vdash e_1 : S \quad S = (S_1 \rightarrow S_2)@U \quad \Delta; \Gamma \vdash e_2 : S_1}{\Delta; \Gamma \vdash e_1 e_2 : [S_2]_S} \\
\text{(TOp)} \frac{\Theta(\oplus) : P \times P' \rightarrow P'' \quad \Delta; \Gamma \vdash e_1 : P@U \quad \Delta; \Gamma \vdash e_2 : P'@U'}{\Delta; \Gamma \vdash e_1 \oplus e_2 : [[P''@P'']]_{P@U} P'@U'} \\
\text{(TFst)} \frac{\Delta; \Gamma \vdash e : S \quad S = (S_1 \times S_2)@U}{\Delta; \Gamma \vdash \text{fst } e : [S_1]_S} \quad \text{(TSnd)} \frac{\Delta; \Gamma \vdash e : S \quad S = (S_1 \times S_2)@U}{\Delta; \Gamma \vdash \text{snd } e : [S_2]_S} \\
\text{(TCase)} \frac{\Delta; \Gamma \vdash e : S \quad S = (S_1 + S_2)@U \quad \Delta; \Gamma, x_1 : S_1 \vdash e_1 : S' \quad \Delta; \Gamma, x_2 : S_2 \vdash e_2 : S'}{\Delta; \Gamma \vdash \text{case } e \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 : [S']_S} \\
\text{(TOpen)} \frac{\Delta; \Gamma \vdash e : S \quad S = (\exists X.T)@U \quad T' \triangleq \text{sftype}(\exists X.T) \quad \Delta, X : T'; \Gamma, x : T_{\text{L}} \vdash e' : S' \quad \Delta \models S'}{\Delta; \Gamma \vdash \text{open}(X, x) = e \text{ in } e' : [S']_S}
\end{array}$$

Figure B.6:  $\lambda_{\text{SEC}}^{\exists}$ : Typing rules



## B.1.8 Dynamic semantics

The Figure B.7 shows the dynamic semantics of  $\lambda_{\text{SEC}}^{\exists}$ . To execute primitive operators, we use a *function*  $\theta$  to abstract over an internal runtime for primitive values.

$E ::= [] \mid \text{fst } E \mid \text{snd } E \mid \text{case } E \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mid E \text{ e} \mid v \text{ E} \quad (\text{evaluation contexts})$   
 $\mid E \oplus e \mid v \oplus E \mid \text{open}(X, x) = E \text{ in } e'$

$\text{fst } \langle v_1, v_2 \rangle \mapsto v_1$	$\frac{e \mapsto e'}{E[e] \mapsto E[e']}$
$\text{snd } \langle v_1, v_2 \rangle \mapsto v_2$	
$\text{case inl } v \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mapsto e_1[v/x_1]$	
$\text{case inr } v \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \mapsto e_2[v/x_2]$	
$(\lambda x : S. e) v \mapsto e[v/x]$	
$\mathbf{b}_1 \oplus \mathbf{b}_2 \mapsto \theta(\oplus, \mathbf{b}_1, \mathbf{b}_2)$	
$\text{open}(X, x) = (\text{pack}(T', v) \text{ as } \exists X.T) \text{ in } e' \mapsto e'[v/x][T'/X]$	

Figure B.7:  $\lambda_{\text{SEC}}^{\exists}$ : Dynamic semantics

## B.1.9 Simple type system

The Figure B.8 defines a simple type system that does not consider the declassification type. The judgment  $\Delta; \Gamma \vdash_{\text{sf}} e : S$  holds if the expression  $e$  is simply well-typed for type  $S$  under the type variable environment  $\Delta$  and the type environment  $\Gamma$ . We make the following changes to not consider the declassification types:

- We use a simply subtyping judgment  $S' <_{\text{sf}} S$  that boils down to type equality modulo the safety type (recursively).
- We remove the protection to the return type of the elimination rules.

The judgment  $\Delta; \Gamma \vdash_1 e : T$  is the front-end of the simple type system and it uses the judgment  $\Delta; \Gamma \vdash_{\text{sf}} e : S$ .

$$\boxed{\Delta; \Gamma \vdash_{\text{sf}} e : S}$$

$$\frac{x \in \text{dom}(\Gamma)}{\Delta; \Gamma \vdash_{\text{sf}} x : \Gamma(x)} \quad \frac{\Delta; \Gamma \vdash_{\text{sf}} e : S' \quad S' <_{\text{sf}} S}{\Delta; \Gamma \vdash_{\text{sf}} e : S} \quad \frac{P = \Theta(\mathbf{b})}{\Delta; \Gamma \vdash_{\text{sf}} \mathbf{b} : P@U}$$

$$\frac{\Delta; \Gamma, x : S \vdash_{\text{sf}} e : S'}{\Delta; \Gamma \vdash_{\text{sf}} \lambda x : S. e : (S \rightarrow S')_{\text{L}}} \quad \frac{\Delta; \Gamma \vdash_{\text{sf}} e_1 : S_1 \quad \Delta; \Gamma \vdash_{\text{sf}} e_2 : S_2}{\Delta; \Gamma \vdash_{\text{sf}} \langle e_1, e_2 \rangle : (S_1 \times S_2)_{\text{L}}}$$

$$\frac{\Delta; \Delta; \Gamma \vdash_{\text{sf}} e : S_1}{\Delta; \Gamma \vdash_{\text{sf}} \text{inl } e : (S_1 + S_2)_{\text{L}}} \quad \frac{\Delta; \Gamma \vdash_{\text{sf}} e : S_2}{\Delta; \Gamma \vdash_{\text{sf}} \text{inr } e : (S_1 + S_2)_{\text{L}}}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e : T [T'/X]@U}{\Delta; \Gamma \vdash_{\text{sf}} \text{pack}(T', e) \text{ as } \exists X.T : (\exists X.T)_{\text{L}}}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e_1 : (S_1 \rightarrow S_2)@U \quad \Delta; \Gamma \vdash_{\text{sf}} e_2 : S_1}{\Delta; \Gamma \vdash_{\text{sf}} e_1 e_2 : S_2}$$

$$\frac{\Theta(\oplus) : P \times P' \rightarrow P'' \quad \Delta; \Gamma \vdash_{\text{sf}} e_1 : P@U \quad \Delta; \Gamma \vdash_{\text{sf}} e_2 : P'@U'}{\Delta; \Delta; \Gamma \vdash_{\text{sf}} e_1 \oplus e_2 : P''@U''}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e : (S_1 \times S_2)@U}{\Delta; \Gamma \vdash_{\text{sf}} \text{fst } e : S_1} \quad \frac{\Delta; \Gamma \vdash_{\text{sf}} e : (S_1 \times S_2)@U}{\Delta; \Gamma \vdash_{\text{sf}} \text{snd } e : S_2}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e : S \quad S = (S_1 + S_2)@U \quad \Delta; \Gamma, x_1 : S_1 \vdash_{\text{sf}} e_1 : S' \quad \Delta; \Gamma, x_2 : S_2 \vdash_{\text{sf}} e_2 : S'}{\Delta; \Gamma \vdash_{\text{sf}} \text{case } e \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 : S'}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e : S \quad S = (\exists X.T)@U \quad \Delta, X : \_ ; \Gamma, x : T@T \vdash_{\text{sf}} e' : S' \quad \Delta \vdash S'}{\Delta; \Gamma \vdash_{\text{sf}} \text{open}(X, x) = e \text{ in } e' : S'}$$

$$\boxed{\Delta; \Gamma \vdash_1 e : T}$$

$$\frac{\Delta; \Gamma \vdash_{\text{sf}} e : T@U}{\Delta; \Gamma \vdash_1 e : T}$$

Figure B.8:  $\lambda_{\text{SEC}}^{\exists}$  Simple typing, defined in terms of single-facet typing

## B.2 Type safety

The type safety result for the  $\lambda_{\text{SEC}}^{\exists}$  language is standard. Note that faceted types does not change the safety meaning of a program. Here, we just highlight the standard path to prove type safety using logical predicates [4].

**Definition 12** (Safety).  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$

**Unary model.** We use logical predicates  $\mathcal{V}[[T]]_s$  and  $\mathcal{C}[[T]]_s$  for safety in the spirit of the unary model of Ahmed [4]. The logical predicates  $\mathcal{V}[[T]]_s$  and  $\mathcal{C}[[T]]_s$  define the safety meaning of the safety type  $T$  as values and expressions, therefore the declassification type does not play any role in the definitions. With that in mind, our logical predicates are straightforward adaptations of the unary model of Ahmed [4].

**Definition 24** (Semantic typing).  $\models e : T@U \iff e \in \mathcal{C}[[T]]_s$ .

**Lemma 35** (Semantic type safety).  $\models e : S \implies \text{safe}(e)$

*Proof.* The proof follows directly from definitions 12 and 24 . □

**Lemma 36** (Type Safety).  $\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \models e : S$

*Proof.* The proof is by induction on the typing derivation of  $e$ . □

**Theorem 14** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

*Proof.* This follows directly from Lemmas 35 and 36 □

## B.3 Existential relaxed noninterference: proofs

### B.3.1 Logical relation

The Figure B.9 shows the logical relation for existential relaxed noninterference.

$$\begin{array}{lcl}
\text{Atom}[T_1, T_2] & = & \{(e_1, e_2) \mid \bullet; \bullet \vdash_1 e_1 : T_1 \wedge \bullet; \bullet \vdash_1 e_2 : T_2\} \\
\text{Atom}_\rho[T] & = & \text{Atom}[\rho_1(T), \rho_2(T)] \\
\text{Rel}[T_1, T_2] & = & \{R \subseteq \text{Atom}[T_1, T_2]\} \\
\mathcal{V}[\mathbf{1}]\rho & = & \{(\langle \rangle, \langle \rangle) \in \text{Atom}_\rho[\mathbf{1}]\} \\
\mathcal{V}[P]\rho & = & \{(\mathbf{b}, \mathbf{b}) \in \text{Atom}_\rho[P]\} \\
\mathcal{V}[S_1 \rightarrow S_2]\rho & = & \{(v_1, v_2) \in \text{Atom}_\rho[S_1 \rightarrow S_2] \mid \\
& & \forall v'_1, v'_2. (v'_1, v'_2) \in \mathcal{V}[S_1]\rho \implies (v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2]\rho\} \\
\mathcal{V}[S_1 \times S_2]\rho & = & \{(\langle v_1, v'_1 \rangle, \langle v_2, v'_2 \rangle) \in \text{Atom}_\rho[S_1 \times S_2] \mid (v_1, v_2) \in \mathcal{V}[S_1]\rho \wedge (v'_1, v'_2) \in \mathcal{V}[S_2]\rho\} \\
\mathcal{V}[S_1 + S_2]\rho & = & \{(\text{inl } v_1, \text{inl } v_2) \in \text{Atom}_\rho[S_1 + S_2] \mid (v_1, v_2) \in \mathcal{V}[S_1]\rho\} \\
& & \cup \{(\text{inr } v_1, \text{inr } v_2) \in \text{Atom}_\rho[S_1 + S_2] \mid (v_1, v_2) \in \mathcal{V}[S_2]\rho\} \\
\mathcal{V}[\exists X.T]\rho & = & \{(\text{pack}(T_1, v_1) \text{ as } \exists X.T, \text{pack}(T_2, v_2) \text{ as } \exists X.T) \in \text{Atom}_\rho[\exists X.T] \mid \\
& & T_1 \sqsubseteq \text{sftype}(\exists X.T) \wedge T_2 \sqsubseteq \text{sftype}(\exists X.T) \wedge \\
& & \exists R \in \text{Rel}[T_1, T_2]. (v_1, v_2) \in \mathcal{V}[T]\rho[X \mapsto (T_1, T_2, R)]\} \\
\mathcal{V}[X]\rho & = & \rho_R(X) \\
\mathcal{V}[T@X]\rho & = & \rho_R(X) \cup \mathcal{V}[T]\rho \\
\mathcal{V}[T@T]\rho & = & \text{Atom}[\rho_1(T), \rho_2(T)] \\
\mathcal{V}[T@T]\rho & = & \mathcal{V}[T]\rho \\
\mathcal{C}[T@U]\rho & = & \{(e_1, e_2) \in \text{Atom}_\rho[T] \mid e_1 \mapsto^* v_1 \wedge e_2 \mapsto^* v_2 \wedge (v_1, v_2) \in \mathcal{V}[T@U]\rho\} \\
\mathcal{G}[\cdot]\rho & = & \{(\emptyset, \emptyset)\} \\
\mathcal{G}[\Gamma; x : S]\rho & = & \{(\gamma_1[x \mapsto v_1], \gamma_2[x \mapsto v_2]) \mid (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \wedge (v_1, v_2) \in \mathcal{V}[S]\rho\} \\
\mathcal{D}[\cdot] & = & \{\emptyset\} \\
\mathcal{D}[\Delta; X : T] & = & \{\rho[X \mapsto (T_1, T_2, R)] \mid \rho \in \mathcal{D}[\Delta] \wedge T_1 \sqsubseteq T \wedge T_2 \sqsubseteq T \wedge R \in \text{Rel}[T_1, T_2]\} \\
\text{ERNI}(\Delta, \Gamma, e, S) & \iff & S \triangleq T@U \quad \Delta; \Gamma \vdash_1 e : T \wedge \Delta \models \Gamma \wedge \Delta \models S \wedge \\
& & \forall \rho, \gamma_1, \gamma_2. \rho \in \mathcal{D}[\Delta]. (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \\
& & \implies (\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e))) \in \mathcal{C}[S]\rho \\
\Delta; \Gamma \vdash e_1 \approx e_2 : S & \iff & \Delta; \Gamma \vdash e_1 : S \wedge \Delta \models \Gamma \wedge \Delta \models S \wedge \\
& & \forall \rho, \gamma_1, \gamma_2. \rho \in \mathcal{D}[\Delta]. (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho \\
& & \implies (\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \mathcal{C}[S]\rho
\end{array}$$

Figure B.9:  $\lambda_{\text{SEC}}^{\exists}$  Logical relation

### B.3.2 Auxiliary lemmas: Simple typing

**Lemma 37** (Well-typed programs are simple well-typed).

If  $\Delta; \Gamma \vdash e : T@U$  then  $\Delta; \Gamma \vdash_1 e : T$

**Lemma 38** (Simple typing is preserved by value substitutions).

If  $\Delta; \Gamma \vdash_1 e : T$  and  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$  then  $\bullet; \bullet \vdash_1 \gamma_1(e) : T$  and  $\bullet; \bullet \vdash_1 \gamma_2(e) : T$

*Proof.* By induction on  $\Gamma$

□

**Lemma 39** (Simple typing is preserved by type substitutions).

If  $\Delta; \Gamma \vdash_1 e : T$  and  $\rho \in \mathcal{D}[\Delta]$

Then  $\Delta; \Gamma \vdash_1 \rho_1(e) : T$  and  $\Delta; \Gamma \vdash_1 \rho_2(e) : T$

### B.3.3 Strong normalization

**Theorem 40** (Strong Normalization).

$\bullet; \bullet \vdash_1 e : T \implies e \Downarrow$

*Proof.* The  $\lambda_{\text{SEC}}^{\exists}$  language has no non-terminating expressions. The proof is completely standard, by defining a logical predicate [41].  $\square$

### B.3.4 Auxiliary lemmas: logical relation

### B.3.5 Auxiliary definitions

**Definition 25** (Simple satisfactory value substitution). *A value substitution  $\gamma$  satisfies a type environment  $\Gamma$  under a type variable environment  $\Delta$ , noted  $\Delta \models \gamma : \Gamma$  iff  $\text{dom}(\gamma) = \text{dom}(\Gamma)$  and  $\forall x : T@U \in \Gamma, \Delta; \bullet \vdash_1 \gamma(x) : T$*

#### Substitutions preserve simple typing

**Lemma 41** (Substitutions preserve simple typing).

*Let  $\Delta, \Gamma \vdash e_1 : T_1@U_1$  and  $\Delta, \Gamma \vdash e_2 : T_1@U_1$*

*Let  $\rho \in \mathcal{D}[\Delta]$  and  $\Delta \models \gamma_1 : \Gamma$  and  $\Delta \models \gamma_2 : \Gamma$*

*Then  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e_2))) \in \text{Atom}[\rho_1(T_1), \rho_2(T_1)]$*

#### PER simple typing

**Lemma 42** (PER simple typing).

*If  $(v_1, v_2) \in \mathcal{V}[T@U]\rho$  then  $(v_1, v_2) \in \text{Atom}_\rho[T]$*

*Proof.* By case analysis on the relational interpretation of security types  $\mathcal{V}[S]\rho$  and types  $\mathcal{V}[T]\rho$ . All the relational interpretation are defined on atoms.  $\square$

#### PER Subtyping

**Lemma 43** (PER Subtyping).

*Let  $\Delta \models S, \Delta \models S', \rho \in \mathcal{D}[\Delta]$  and  $\Delta \vdash S' <: S$*

*(1) If  $(v_1, v_2) \in \mathcal{V}[S']\rho$  then  $(v_1, v_2) \in \mathcal{V}[S]\rho$*

*(2) If  $(e_1, e_2) \in \mathcal{C}[S']\rho$  then  $(e_1, e_2) \in \mathcal{C}[S]\rho$*

*Proof.* We proof both statements (1) and (2) simultaneously.

The proof is by induction on the structure of the faceted type  $S$ .

Statement (1):

**Case** ( $S = T@T$ ). *By inversion of the subtyping relation  $\Delta \vdash S' <: T@T$ , we know that  $S' = T@T$*

*Hence, it is direct that  $(v_1, v_2) \in \mathcal{V}[T@T]\rho$  implies  $(v_1, v_2) \in \mathcal{V}[T@T]\rho$ .*

**Case** ( $S = T@T$ ). *Then our goal is:*

$$(v_1, v_2) \in \mathcal{V}[T@T]\rho = \text{Atom}[\rho_1(T), \rho_2(T)].$$

*By inversion of the subtyping relation  $\Delta \vdash S' <: T@T$ , we know that  $S' = T@U$ .*

Apply Lemma 42 with  $(v_1, v_2) \in \mathcal{V}[[T@U]]\rho$ .

Hence,  $(v_1, v_2) \in \text{Atom}[T]$ .

**Case**  $(S = T@X)$ . Then our goal is:

$$(v_1, v_2) \in \mathcal{V}[[T@X]]\rho = \rho_{\mathbf{R}}(X) \cup \mathcal{V}[[T]]\rho.$$

By inversion of the subtyping relation  $\Delta \vdash S' <: T@X$  we know that  $S$  is either  $T@X$  or  $T@T$ .

If  $S = T@X$ , then the proof is direct.

If  $S = T@T$ , then  $(v_1, v_2) \in \mathcal{V}[[T@T]]\rho = \mathcal{V}[[T]]\rho \subseteq \rho_{\mathbf{R}}(X) \cup \mathcal{V}[[T]]\rho$ .

□

## Related values are related terms

**Lemma 44** (Related values are related terms). If  $(v_1, v_2) \in \mathcal{V}[[S]]\rho$  then  $(v_1, v_2) \in \mathcal{C}[[S]]\rho$

*Proof.* Direct from the definition of  $(v_1, v_2) \in \mathcal{C}[[S]]\rho$

□

## Atom subject reduction

**Lemma 45** (Atom reduction).

Let  $e_1 \mapsto^* e'_1$  and  $e_2 \mapsto^* e'_2$

Let  $(e_1, e_2) \in \text{Atom}_\rho[T] \Rightarrow (e'_1, e'_2) \in \text{Atom}_\rho[T]$

Then  $(e'_1, e'_2) \in \text{Atom}_\rho[T] \Rightarrow (e_1, e_2) \in \text{Atom}_\rho[T]$

*Proof.* The proof is straightforward. Each subgoal of each direction follows by induction on the typing derivation of  $e_i$  ( $i \in \{1, 2\}$ ). □

## Anti reduction/Backward preservation

**Lemma 46** (Anti reduction/Backward preservation).

Let  $e_1 \mapsto^* e'_1$  and  $e_2 \mapsto^* e'_2$

Let  $(e'_1, e'_2) \in \mathcal{C}[[S]]\rho$

Then  $(e_1, e_2) \in \mathcal{C}[[S]]\rho$

*Proof.* Denote  $S = T@U$ .

Proof obligations:

1.  $(e_1, e_2) \in \text{Atom}[T]$ . Apply Lemma 45 (Atom reduction) with  $(e'_1, e'_2) \in \text{Atom}[T]$  which follows from  $(e'_1, e'_2) \in \mathcal{C}[[S]]\rho$



2. **Show:** that there exists  $v_1, v_2$  such as:

- $e_1 \mapsto^* v_1$ . It follows from  $e_1 \mapsto^* e'_1$  and  $e'_1 \mapsto^* v_1$
- $e_2 \mapsto^* v_2$ . It follows from  $e_2 \mapsto^* e'_2$  and  $e'_2 \mapsto^* v_2$
- $(v_1, v_2) \in \mathcal{V}[[S]]\rho$ . It follows from  $(e'_1, e'_2) \in \mathcal{C}[[S]]\rho$  and  $e'_1 \mapsto^* v_1$  and  $e'_2 \mapsto^* v_2$

□

## Atoms are related a top

**Lemma 47** (Atoms are related a top).  $(e_1, e_2) \in \text{Atom}_\rho[T] \Rightarrow (e_1, e_2) \in \mathcal{C}[[T@T]]\rho$

*Proof.* Proof obligations:

There exists  $v_1$  and  $v_2$  such as:

- $e_1 \mapsto^* v_1$ . Apply Theorem 40 (Strong Normalization).
- $e_2 \mapsto^* v_2$ . Apply Theorem 40 (Strong Normalization).
- $(v_1, v_2) \in \mathcal{V}[[T@T]]\rho$ . By the definition of  $\mathcal{V}[[T@T]]\rho$ , this is equivalent to show  $(v_1, v_2) \in \text{Atom}_\rho[T]$ . Apply Lemma 45 (Atom reduction).

□

## Monadic bind

**Lemma 48** (Monadic bind).

If  $(e_1, e_2) \in \mathcal{C}[[S]]\rho$

and  $\forall v_1, v_2. (v_1, v_2) \in \mathcal{V}[[S]]\rho \implies (E[v_1], E[v_2]) \in \mathcal{C}[[S']]\rho$

then  $(E[e_1], E[e_2]) \in \mathcal{C}[[S']]\rho$

*Proof.* Unfold  $(e_1, e_2) \in \mathcal{C}[[S]]\rho$  to know that there exists  $v'_1$  and  $v'_2$  such as

- $e_1 \mapsto^* v'_1$
- $e_2 \mapsto^* v'_2$
- $(v'_1, v'_2) \in \mathcal{V}[[S]]\rho$

Inspect the dynamic semantics to know:

$$\begin{aligned} E[e_1] &\mapsto^* E[v'_1] \\ E[e_2] &\mapsto^* E[v'_2] \end{aligned}$$

Instantiate the second premise with  $v'_1, v'_2$ . Note that  $(v'_1, v'_2) \in \mathcal{V}[[S]]\rho$ .

Hence,  $(E[v'_1], E[v'_2]) \in \mathcal{C}[[S']]\rho$

Instantiate Lemma 46 (Anti reduction/Backward preservation). Note that:

- $E[e_1] \mapsto^* E[v'_1]$
- $E[e_2] \mapsto^* E[v'_2]$
- $(E[v'_1], E[v'_2]) \in \mathcal{C}[[S']]\rho$

Hence,  $(E[e_1], E[e_2]) \in \mathcal{C}[[S']]\rho$

□

## PER Type Substitution: Compositionality

**Lemma 49** (PER Type Substitution: Compositionality).

Let  $\rho \in \mathcal{D}[\Delta]$

Let  $\Delta \models T'$  and  $\Delta, X : \_ \vdash T$

Let  $R = \mathcal{V}[T'@T']\rho$

(1)  $\mathcal{V}[T]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[T [T'/X]]\rho$

(2)  $\mathcal{V}[S]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[S [T'/X]]\rho$

(3)  $\mathcal{C}[S]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{C}[S [T'/X]]\rho$

*Proof.* We prove the three claims above simultaneously, by induction on the structure of  $T$  and  $S$ ,

1.  $\mathcal{V}[T]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[T [T'/X]]\rho$

**Case** ( $T = S_1 \rightarrow S_2$ ). We first show the  $\Rightarrow$  direction:

$$(v_1, v_2) \in \mathcal{V}[S_1 \rightarrow S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] \implies (v_1, v_2) \in \mathcal{V}[(S_1 \rightarrow S_2) [T'/X]]\rho.$$

Assuming  $(v'_1, v'_2) \in \mathcal{V}[S_1 [T'/X]]\rho$

**Show:**

$$(v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2 [T'/X]]\rho$$

Instantiate  $(v_1, v_2) \in \mathcal{V}[S_1 \rightarrow S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$  with  $v'_1, v'_2$ . Note that:

- $(v'_1, v'_2) \in \mathcal{V}[S_1]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$  which follows from the second claim:  
 $\mathcal{V}[S_1]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[S_1 [T'/X]]\rho$

Hence,  $(v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$

Apply the claim (3) above with  $S_2$

Hence,  $(v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2 [T'/X]]\rho$

Now we show the  $\Leftarrow$  direction:

$$(v_1, v_2) \in \mathcal{V}[(S_1 \rightarrow S_2) [T'/X]]\rho \implies (v_1, v_2) \in \mathcal{V}[S_1 \rightarrow S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)].$$

Assuming  $(v'_1, v'_2) \in \mathcal{V}[S_1]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$

**Show:**

$$(v_1 v'_1, v_2 v'_2) \in \mathcal{V}[S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$$

Instantiate  $(v_1, v_2) \in \mathcal{V}[(S_1 \rightarrow S_2) [T'/X]]\rho$  with  $v'_1, v'_2$ . Note that:

- $(v'_1, v'_2) \in \mathcal{V}[S_1 [T'/X]]\rho$  which follows from IH on the second claim:  $\mathcal{V}[S_1]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[S_1 [T'/X]]\rho$  applied with the assumption above.

Hence,  $(v_1 v'_1, v_2 v'_2) \in \mathcal{V}[S_2 [T'/X]]\rho$

Apply the IH of claim (3) with  $S_2$

Hence,  $(v_1 v'_1, v_2 v'_2) \in \mathcal{C}[S_2]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$

**Case** ( $T = P$ ). The proof is direct.

**Case** ( $T = S_1 + S_2$ ). *The proof is direct. It simply applies the IH of the second claim with  $S_1$  and  $S_2$ .*

**Case** ( $T = S_1 \times S_2$ ). *The proof is direct. It applies the IH of the second claim with  $S_1$  (resp.  $S_2$ ) for the case of left-injected (resp. left-injected) values.*

**Case** ( $T = \exists Y.T''$ ). *We first show the  $\Rightarrow$  direction:*

$$(v_1, v_2) \in \mathcal{V}[\exists Y.T'']\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] \implies (v_1, v_2) \in \mathcal{V}[(\exists Y.T'') [T'/X]]\rho.$$

Denote  $v_1 = \text{pack}(T'_1, v'_1)$  as  $\exists Y.T''$  and  $v_2 = \text{pack}(T'_2, v'_2)$  as  $\exists Y.T''$

**Show:**

$$\exists R' \in \text{Rel}[T'_1, T'_2] \implies (v'_1, v'_2) \in \mathcal{V}[T'' [T'/X]]\rho[Y \mapsto (T'_1, T'_2, R')].$$

From  $(v_1, v_2) \in \mathcal{V}[\exists Y.T'']\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$  we know that:

$$(A) \exists R'' \in \text{Rel}[T'_1, T'_2] \text{ such as } (v'_1, v'_2) \in \mathcal{V}[T'']\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] [Y \mapsto (T'_1, T'_2, R'')]$$

Take  $R = R''$ . Note that:

- $R'' \in \text{Rel}[T'_1, T'_2]$ .

**Show:**

$$(v'_1, v'_2) \in \mathcal{V}[T'' [T'/X]]\rho[Y \mapsto (T'_1, T'_2, R'')].$$

Instantiate (A) with  $R''$ , hence:

$$(v'_1, v'_2) \in \mathcal{V}[T'']\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] [Y \mapsto (T'_1, T'_2, R'')]$$

$$(v'_1, v'_2) \in \mathcal{V}[T'']\rho[Y \mapsto (T'_1, T'_2, R'')] [X \mapsto (\rho_1(T'), \rho_2(T'), R)]$$

Apply IH with the first claim and  $T''$ ,  $\rho = \rho[Y \mapsto (T'_1, T'_2, R'')]$ , hence

$$(v'_1, v'_2) \in \mathcal{V}[T'' [T'/X]]\rho[Y \mapsto (T'_1, T'_2, R'')].$$

The proof of the  $\Leftarrow$  direction is quite similar to the  $\Rightarrow$  direction.

**Case** ( $T = Y$ ). *Do case analysis on  $Y = X$  and  $Y \not\approx X$ . The proof for both cases easily follows.*

$$2. \mathcal{V}[S]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[S [T'/X]]\rho:$$

**Case** ( $S = T \circledast T$ ). *This is direct. Use IH of the first claim.*

**Case** ( $S = T \circledast \top$ ). *This is direct. Both interpretations are  $\text{Atom}[T]$*

**Case** ( $S = T \circledast Y$ ). • *Suppose  $X = Y$ , then :*

*By well-formedness of  $T \circledast X$  we know that  $T = I$  or  $T = Y$ . The step below are for the case  $T = I$ . The case  $T = Y$  easily follows for IH using the first claim:*

*On one hand:*

$$\mathcal{V}[(T \circledast X) [T'/X]]\rho = \mathcal{V}[(T' \circledast X) [T'/X]]\rho = \mathcal{V}[T' \circledast T']\rho = \mathcal{V}[T']\rho$$

*And on the other hand:*

$$\mathcal{V}[T \circledast X]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[T' \circledast X]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)]$$

$$\mathcal{V}[T' \circledast X]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \rho_R(X) \cup \mathcal{V}[T']\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = R \cup \mathcal{V}[T']\rho$$

$$= \mathcal{V}[T' \circledast T']\rho \cup \mathcal{V}[T']\rho$$

$$= \mathcal{V}[T']\rho \cup \mathcal{V}[T']\rho = \mathcal{V}[T']\rho$$

Hence,  $\mathcal{V}[(T \triangleleft X) [T'/X]]\rho = \mathcal{V}[(T \circledast X) [T'/X]]\rho$

• *Suppose  $X \neq Y$  then:*

$$\mathcal{V}[(T \circledast Y) [T'/X]]\rho = \mathcal{V}[T [T'/Y] \circledast Y]\rho = \rho_R(Y) \cup \mathcal{V}[T [T'/X]]\rho$$

$$\begin{aligned} \mathcal{V}[[T@Y]]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] &= \rho_{\mathbf{R}}(Y) \cup \mathcal{V}[[T [T'/X]]]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] \\ &= \rho_{\mathbf{R}}(Y) \cup \mathcal{V}[[T [T'/X]]]\rho \end{aligned}$$

$$\text{Hence } \mathcal{V}[[T@Y]]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{V}[[T@Y] [T'/X]]\rho.$$

3.  $\mathcal{C}[[S]]\rho[X \mapsto (\rho_1(T'), \rho_2(T'), R)] = \mathcal{C}[[S [T'/X]]]\rho$  The proof is direct. Unfold the definition of related computations and then apply the IH with the claim (2).

□

### B.3.6 Proof of the Fundamental Property

#### Compatibility Var

**Lemma 50** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Var).

$$\Delta; \Gamma \vdash x \approx x : \Gamma(x)$$

*Proof.* First, let us denote  $S \triangleq \Gamma(x)$ .

Proof obligations:

1.  $\Delta; \Gamma \vdash x : S$  which is direct.
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as:  $\rho \in \mathcal{D}[\Delta], \gamma_1, \gamma_2, (k, \gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show**

$$\begin{aligned} (\rho(\gamma_1(x)), \rho(\gamma_2(x))) &\in \mathcal{C}[S]\rho \\ &\equiv (\gamma_1(x), \gamma_2(x)) \in \mathcal{C}[S]\rho \end{aligned}$$

From  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$  we know that exists  $v_1, v_2$  such as:

- $\gamma_1(x) = v_1$
- $\gamma_2(x) = v_2$
- $(v_1, v_2) \in \mathcal{V}[S]\rho$

Apply Lemma 44 (Related values are related terms) with  $(k, \gamma_1(x), \gamma_2(x)) \in \mathcal{V}[S]\rho$  to obtain  $(\gamma_1(x), \gamma_2(x)) \in \mathcal{C}[S]\rho$

□

#### Compatibility Prim

**Lemma 51** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Prim).

Let  $P = \Theta(b)$ .

Then  $\Delta; \Gamma \vdash b \approx b : P@P$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash b : P@P$ . Apply rule (TPrim)
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as:
  - $\rho \in \mathcal{D}[\Delta], (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$\begin{aligned} (\rho(\gamma_1(b)), \rho(\gamma_2(b))) &\in \mathcal{C}[P@P]\rho \\ &\equiv (b, b) \in \mathcal{C}[P@P]\rho \end{aligned}$$

Apply Lemma 44 (Related values are related terms) with  $(b, b) \in \mathcal{V}[P@P]\rho$  to obtain:  $(b, b) \in \mathcal{C}[P@P]\rho$

□

### Compatibility Unit

**Lemma 52** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Unit).

$$\Delta; \Gamma \vdash \langle \rangle \approx \langle \rangle : \text{unit@unit}$$

*Proof.* The proof is a straightforward replication of steps for the Lemma 51. □

### Compatibility Subsumption

**Lemma 53** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Subsumption).

Let  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$ . Let  $\Delta \vdash S' <: S$ .

Then  $\Delta; \Gamma \vdash e_1 \approx e_2 : S$ .

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1 : S$  and  $\Delta; \Gamma \vdash e_2 : S$ . Apply rule (TSub) with  $\Delta; \Gamma \vdash e_i : S'$  (obtained from  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$ ) and  $\Delta \vdash S' <: S$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as :
  - $\rho \in \mathcal{ND}[\Delta]$ ,  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$(\rho(\gamma_1(e_1)), \rho(\gamma_2(e_2))) \in \mathcal{C}[S]\rho.$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$  with  $\rho, \gamma_1, \gamma_2$  to obtain:

$$(\rho(\gamma_1(e_1)), \rho(\gamma_2(e_2))) \in \mathcal{C}[S']\rho.$$

Apply Lemma 43 (PER Subtyping) with  $\Delta \vdash S' <: S$ .

Hence,  $(\rho(\gamma_1(e_1)), \rho(\gamma_2(e_2))) \in \mathcal{C}[S]\rho$ .

□

### Compatibility Function

**Lemma 54** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Function).

If  $\Delta; \Gamma, x : S \vdash e_1 \approx e_2 : S'$

Then  $\Delta; \Gamma \vdash \lambda x : S. e_1 \approx \lambda x : S. e_2 : (S \rightarrow S')_L$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \lambda x : S. e_1 : [S_2]_U$  and  $\Delta; \Gamma \vdash \lambda x : S. e_2 : (S \rightarrow S')'_L$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{C}[\Gamma]\rho$

**Show:**

$$(\rho_1(\gamma_1(\lambda x : S. e_1)), \rho_2(\gamma_2(\lambda x : S. e_2))) \in \mathcal{C}[(S \rightarrow S')_{\perp}]\rho \equiv$$

$$(\lambda x : S. \rho_1(\gamma_1(e_1)), \lambda x : S. \rho_2(\gamma_2(e_2))) \in \mathcal{C}[(S \rightarrow S')_{\perp}]\rho$$

Use Lemma 44 (Related values are related terms) to rewrite the goal to:

**Show:**

$$(\lambda x : S. \rho_1(\gamma_1(e_1)), \lambda x : S. \rho_2(\gamma_2(e_2))) \in \mathcal{V}[(S \rightarrow S')_{\perp}]\rho$$

$$(\lambda x : S. \rho_1(\gamma_1(e_1)), \lambda x : S. \rho_2(\gamma_2(e_2))) \in \mathcal{V}[S \rightarrow S']\rho$$

Assuming  $(v'_1, v'_2) \in \mathcal{V}[S]\rho$

**Show:**

$$((\lambda x : S. \rho_1(\gamma_1(e_1))) v'_1, (\lambda x : S. \rho_2(\gamma_2(e_2))) v'_2) \in \mathcal{C}[S']\rho$$

$$(\rho_1(\gamma_1(e_1)) [x/v'_1], \rho_2(\gamma_2(e_2)) [x/v'_2]) \in \mathcal{C}[S']\rho$$

Instantiate  $\Delta; \Gamma, x : S \vdash e_1 \approx e_2 : S_2$  with  $\rho, \gamma_1 [x \mapsto v'_1], \gamma_2 [x \mapsto v'_2]$ . Note that:

- $\rho \in \mathcal{D}[\Delta]$  which follows from above.
- $(\gamma_1 [x \mapsto v'_1], \gamma_2 [x \mapsto v'_2]) \in \mathcal{G}[\Gamma, x : S]\rho$  which follows from:
  - $(\gamma_1, \gamma_2) \in \mathcal{C}[\Gamma]\rho$ .
  - $(v'_1, v'_2) \in \mathcal{V}[S]\rho$ .

Hence,  $(\rho_1(\gamma_1 [x \mapsto v'_1](e_1)), \rho_2(\gamma_2 [x \mapsto v'_2](e_2))) \in \mathcal{C}[S']\rho$

Both  $v'_1$  and  $v'_2$  are closed values, therefore the above statement is equivalent to:

$$(\rho_1(\gamma_1(e_1)) [x/v'_1], \rho_2(\gamma_2(e_2)) [x/v'_2]) \in \mathcal{C}[S']\rho$$

□

## Compatibility Application

**Lemma 55** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Application).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : S, S \triangleq (S_1 \rightarrow S_2)@U$

Let  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_1$

Then  $\Delta; \Gamma \vdash e_1 e_2 \approx e'_1 e'_2 : [S_2]_U$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1 e_2 : [S_2]_U$  and  $\Delta; \Gamma \vdash e'_1 e'_2 : [S_2]_U$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$(\rho_1(\gamma_1(e_1 e_2)), \rho_2(\gamma_2(e'_1 e'_2))) \in \mathcal{C}[[S_2]_U]\rho \equiv$$

$$(\rho_1(\gamma_1(e_1)) \rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_1)) \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_2]_U]\rho$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e'_1 : (S_1 \rightarrow S_2)@U$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[(S_1 \rightarrow S_2)@U]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[(S_1 \rightarrow S_2)@U]\rho$  to rewrite the goal to:

Assuming  $(v_1, v'_1) \in \mathcal{V}[(S_1 \rightarrow S_2)@U]\rho$

**Show:**

$$(v_1 \rho_1(\gamma_1(e_2)), v'_1 \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_2]_U]\rho$$

Now, instantiate  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_1$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_1]]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_1]]\rho$  to rewrite the goal to:

Assuming  $(v_2, v'_2) \in \mathcal{V}[[S_1]]\rho$

**Show:**

$$(v_1 v_2, v'_1 v'_2) \in \mathcal{C}[[S_2]_U]\rho$$

Do case analysis on  $U$ .

**Case**  $(U = S_1 \rightarrow S_2)$ . Then  $\mathcal{V}[(S_1 \rightarrow S_2)@S_1 \rightarrow S_2]\rho = \mathcal{V}[[S_1 \rightarrow S_2]]\rho$  Instantiate  $(v_1, v'_1) \in \mathcal{V}[[S_1 \rightarrow S_2]]\rho$

*The rest is straightforward.*

**Case**  $(U = \top \vee U = X)$ . Denote  $S_2 \triangleq T_2@U_2$

**Show:**

$$(v_1 v_2, v'_1 v'_2) \in \mathcal{C}[[T_2@T]]\rho$$

From  $(v_1, v'_1) \in \mathcal{V}[(S_1 \rightarrow S_2)@U]\rho$ , we know  $(v_1, v'_1) \in \text{Atom}[S_1 \rightarrow S_2] = \text{Atom}[S_1 \rightarrow T_2@U_2]$

Denote  $S_1 \triangleq T_1@U_1$ . We also know that  $(v_2, v'_2) \in \text{Atom}[T_1]$ .

Use the simple type system of Figure B.8 to derive  $(v_1 v_2, v'_1 v'_2) \in \text{Atom}[T_2]$ .

Apply Lemma 47 with  $(v_1 v_2, v'_1 v'_2) \in \text{Atom}[T_2]$

Hence,  $(v_1 v_2, v'_1 v'_2) \in \mathcal{C}[[T_2@T]]\rho$

□

## Compatibility BinOp

**Lemma 56** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-BinOp).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : P@U$  and  $\Delta; \Gamma \vdash e_2 \approx e'_2 : P'@U'$

Let  $\Theta_{op}(\oplus) : P \times P' \rightarrow P''$

Then  $\Delta; \Gamma \vdash e_1 \oplus e_2 \approx e'_1 \oplus e'_2 : [[P''_L]_{P@U}]_{P'@U'}$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1 \oplus e_2 : [[P''_L]_{P@U}]_{P'@U'}$  and  $\Delta; \Gamma \vdash e'_1 \oplus e'_2 : [[P''_L]_{P@U}]_{P'@U'}$



2. Assuming arbitrary  $\rho, \gamma_1$  and  $\gamma_2$ , such as  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$\begin{aligned} & (\rho_1(\gamma_1(e_1 \oplus e_2)), \rho_2(\gamma_2(e'_1 \oplus e'_2))) \in \mathcal{C}[[[P_L'']_{P@U}]_{P'@U'}]\rho \\ & (\rho_1(\gamma_1(e_1)) \oplus \rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_1)) \oplus \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[[P_L'']_{P@U}]_{P'@U'}]\rho \end{aligned}$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e'_1 : P@U$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[P@U]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[P@U]\rho$  to rewrite the goal to:

Assuming  $(v_1, v'_1) \in \mathcal{V}[P@U]\rho$

**Show:**

$$(v_1 \oplus \rho_1(\gamma_1(e_2)), v'_1 \oplus \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[[P_L'']_{P@U}]_{P'@U'}]\rho$$

Now, instantiate  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_1$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[P'@U']\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[P'@U']\rho$  to rewrite the goal to:

Assuming  $(v_2, v'_2) \in \mathcal{V}[P'@U']\rho$

**Show:**

$$(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \mathcal{C}[[[P_L'']_{P@U}]_{P'@U'}]\rho$$

Do case analysis on  $U$  and  $U'$ .

**Case**  $(U = P \wedge U' = P')$ . **Show:**

$$(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \mathcal{C}[P_L'']\rho$$

From  $(v_1, v'_1) \in \mathcal{V}[P@P]\rho$  and  $(v_2, v'_2) \in \mathcal{V}[P'@P']\rho$  we know that  $v_1 = v'_1$  and  $v_2$  and  $v'_2$ .

Inspect the dynamic semantic to know that:  $v_1 \oplus v_2 \mapsto v_3$  and  $v'_1 \oplus v'_2 \mapsto v'_3$  and that  $v_3 = v'_3$ .

Hence  $(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \mathcal{C}[P_L'']\rho$

**Case**  $(U \neq P \vee U' \neq P')$ .

**Show:**

$$(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \mathcal{C}[P''@T]\rho$$

Use the simple type system of Figure B.8 to derive  $(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \text{Atom}[P'']$ .

Apply Lemma 47 with  $(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \text{Atom}[P'']$

Hence,  $(v_1 \oplus v_2, v'_1 \oplus v'_2) \in \mathcal{C}[P''@T]\rho$

□

## Compatibility Inl

**Lemma 57** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Inl).

If  $\Delta; \Gamma \vdash e \approx e' : S_1$

Then  $\Delta; \Gamma \vdash \text{inl } e \approx \text{inl } e' : (S_1 + S_2)_L$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \text{inl } e : (S_1 + S_2)_L$  and  $\Delta; \Gamma \vdash \text{inl } e' : (S_1 + S_2)_L$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{C}[\Gamma]\rho$

**Show:**

$$\begin{aligned} (\rho_1(\gamma_1(\text{inl } e)), \rho_2(\gamma_2(\text{inl } e'))) &\in \mathcal{C}[(S_1 + S_2)_L]\rho \equiv \\ (\text{inl } \rho_1(\gamma_1(e)), \text{inl } \rho_2(\gamma_2(e'_1))) &\in \mathcal{C}[(S_1 + S_2)_L]\rho \end{aligned}$$

Instantiate  $\Delta; \Gamma \vdash e \approx e' : S_1$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[S_1]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[S_1]\rho$  to rewrite the goal to:

Assuming  $(v, v') \in \mathcal{V}[S_1]\rho$

**Show:**

$$(\text{inl } v, \text{inl } v') \in \mathcal{C}[(S_1 + S_2)_L]\rho$$

From  $(v, v') \in \mathcal{V}[S_1]\rho$  and the definition of  $\mathcal{V}[S_1 + S_2]\rho$ , derive that:

$$(\text{inl } v, \text{inl } v') \in \mathcal{V}[S_1 + S_2]\rho,$$

Hence,  $(\text{inl } v, \text{inl } v') \in \mathcal{V}[(S_1 + S_2)_L]\rho$ . Apply Lemma 44 (Related values are related terms)

Hence,  $(\text{inl } v, \text{inl } v') \in \mathcal{C}[(S_1 + S_2)_L]\rho$ .

□

## Compatibility Inr

**Lemma 58** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Inr).

*If*  $\Delta; \Gamma \vdash e \approx e' : S_2$   
*Then*  $\Delta; \Gamma \vdash \text{inr } e \approx \text{inr } e' : (S_1 + S_2)_L$

*Proof.* The proof is a mere replication of the steps for the Lemma 57 adapted for expressions  $\text{inr } e$  and  $\text{inr } e'$ . □

## Compatibility Case

**Lemma 59** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Case).

*Let*  $\Delta; \Gamma \vdash e \approx e' : S$ ,  $S \triangleq (S_1 + S_2)@U$   
*Let*  $\Delta; \Gamma, x_1 : [S_1]_S \vdash e_1 \approx e'_1 : S'$  and  $\Delta; \Gamma, x_2 : [S_2]_S \vdash e_2 \approx e'_2 : S'$   
*Then*  $\Delta; \Gamma \vdash \text{case } e \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 \approx \text{case } e' \text{ of inl } x_1.e'_1 \mid \text{inr } x_2.e'_2 : [S']_S$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \text{case } e \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2 : [S']_S$  and  $\Delta; \Gamma \vdash \text{case } e' \text{ of inl } x_1.e'_1 \mid \text{inr } x_2.e'_2 : [S']_S$

2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$\begin{aligned} (\rho_1(\gamma_1(\text{case } e \text{ of inl } x_1.e_1 \mid \text{inr } x_2.e_2)), \rho_2(\gamma_2(\text{case } e' \text{ of inl } x_1.e'_1 \mid \text{inr } x_2.e'_2))) \in \mathcal{C}[[S']_S]\rho \equiv \\ (\text{case } \rho_1(\gamma_1(e)) \text{ of inl } x_1.\rho_1(\gamma_1(e_1)) \mid \text{inr } x_2.\rho_1(\gamma_1(e_2))), \\ \text{case } \rho_2(\gamma_2(e')) \text{ of inl } x_1.\rho_2(\gamma_2(e'_1)) \mid \text{inr } x_2.\rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S']_S]\rho \end{aligned}$$

Instantiate  $\Delta; \Gamma \vdash e \approx e' : (S_1 + S_2)@U$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e')) \in \mathcal{C}[(S_1 + S_2)@U]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e')) \in \mathcal{C}[(S_1 + S_2)@U]\rho$  to rewrite the goal to:

Assuming  $(v, v') \in \mathcal{V}[(S_1 + S_2)@U]\rho$

**Show:**

$$\begin{aligned} (\text{case } v \text{ of inl } x_1.\rho_1(\gamma_1(e_1)) \mid \text{inr } x_2.\rho_1(\gamma_1(e_2))), \\ \text{case } v' \text{ of inl } x_1.\rho_2(\gamma_2(e'_1)) \mid \text{inr } x_2.\rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S']_S]\rho \end{aligned}$$

Do case analysis on  $U$ .

**Case**  $(U = S_1 + S_2)$ . Then  $\mathcal{V}[(S_1 + S_2)@U]\rho = \mathcal{V}[S_1 + S_2]\rho$  and  $[S']_S = S'$

Do case analysis on the definition of  $\mathcal{V}[S_1 + S_2]\rho$ :

- $v = \text{inl } v_l$  and  $v' = \text{inl } v'_l$  such as  $(v_l, v'_l) \in \mathcal{V}[S_1]\rho$

Inspect the dynamic semantics to rewrite the goal as:

$$\begin{aligned} (\rho_1(\gamma_1(e_1)) [v_l/x_1], \rho_2(\gamma_2(e'_1)) [v'_l/x_2]) \in \mathcal{C}[S']\rho \equiv \\ (\rho_1(\gamma_1(e_1) [v_l/x_1]), \rho_2(\gamma_2(e'_1) [v'_l/x_2])) \in \mathcal{C}[S']\rho \equiv \\ (\rho_1(\gamma_1 [x_1 \mapsto v_l](e_1)), \rho_2(\gamma_2 [x_2 \mapsto v'_l](e'_1))) \in \mathcal{C}[S']\rho \equiv \end{aligned}$$

Instantiate the second hypothesis with  $\rho, \gamma_1 [x_1 \mapsto v_l]$  and  $\gamma_2 [x_2 \mapsto v'_l]$ . Note that:

$$- (\gamma_1 [x_1 \mapsto v_l], \gamma_2 [x_2 \mapsto v'_l]) \in \mathcal{G}[\Gamma, x : S_1]\rho$$

Hence,  $(\rho_1(\gamma_1 [x_1 \mapsto v_l](e_1)), \rho_2(\gamma_2 [x_2 \mapsto v'_l](e'_1))) \in \mathcal{C}[S']\rho$

□

- $v = \text{inr } v_r$  and  $v' = \text{inl } v'_r$

Follow the same steps that for the case above.

**Case**  $(U = \top \vee U = \perp)$ . Denote  $S' \stackrel{\Delta}{=} T'@U'$ , then  $[T'@U']_S = T'@\top$  and our goal is to show:

$$\begin{aligned} (\text{case } v \text{ of inl } x_1.\rho_1(\gamma_1(e_1)) \mid \text{inr } x_2.\rho_1(\gamma_1(e_2))), \\ \text{case } v' \text{ of inl } x_1.\rho_2(\gamma_2(e'_1)) \mid \text{inr } x_2.\rho_2(\gamma_2(e'_2))) \in \mathcal{C}[T'@\top]\rho \end{aligned}$$

From  $(v, v') \in \mathcal{V}[(S_1 + S_2)@U]\rho$ , we know  $(v, v') \in \text{Atom}[S_1 + S_2]$ .

Do case analysis on  $v$  and  $v'$ . We have four cases:  $(v, v') = (\text{inl } v_l, \text{inl } v'_l)$ ,  $(v, v') = (\text{inr } v_l, \text{inr } v'_r)$ ,  $(v, v') = (\text{inl } v_l, \text{inr } v'_r)$ ,  $(v, v') = (\text{inr } v_l, \text{inl } v'_l)$ . The ones that combines left and right injections are the most interesting. We cover one of them, the rest are quite similar.

Case:  $(v, v') = (\text{inl } v_l, \text{inr } v'_r)$

Inspect the dynamic semantics to rewrite the goal to

**Show:**

$$(\rho_1(\gamma_1(e_1)) [v_l/x_1], \rho_2(\gamma_2(e'_2)) [v'_r/x_2]) \in \mathcal{C}[T'@\top]\rho$$

Apply the Lemma 41 (Substitutions preserve simple typing)

Hence,  $(\rho_1(\gamma_1(e_1)) [v_l/x_1], \rho_2(\gamma_2(e'_2)) [v_r/x_2]) \in \text{Atom}[T'@T]$

Apply Lemma 47 (Atoms are related a top).

Hence,  $(\rho_1(\gamma_1(e_1)) [v_l/x_1], \rho_2(\gamma_2(e'_2)) [v_r/x_2]) \in \mathcal{C}[[T'@T]]\rho$

□

## Compatibility Pair

**Lemma 60** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Pair).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : S_1$  and  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_2$

Then  $\Delta; \Gamma \vdash \langle e_1, e_2 \rangle \approx \langle e'_1, e'_2 \rangle : (S_1 \times S_2)_L$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : (S_1 + S_2)_L$  and  $\Delta; \Gamma \vdash \langle e'_1, e'_2 \rangle : (S_1 \times S_2)_L$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[[\Delta]]$  and  $(\gamma_1, \gamma_2) \in \mathcal{G}[[\Gamma]]\rho$

**Show:**

$$\begin{aligned} & (\rho_1(\gamma_1(\langle e_1, e_2 \rangle)), \rho_2(\gamma_2(\langle e'_1, e'_2 \rangle))) \in \mathcal{C}[(S_1 \times S_2)_L]\rho \equiv \\ & (\langle \rho_1(\gamma_1(e_1)), \rho_1(\gamma_1(e_2)) \rangle, \langle \rho_1(\gamma_1(e'_1)), \rho_1(\gamma_1(e'_2)) \rangle) \in \mathcal{C}[(S_1 \times S_2)_L]\rho \end{aligned}$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e'_1 : S_1$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[[S_1]]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[[S_1]]\rho$  to rewrite the goal to:

Assuming  $(v_1, v'_1) \in \mathcal{V}[[S_1]]\rho$

**Show:**

$$(\langle v_1, \rho_1(\gamma_1(e_2)) \rangle, \langle v'_1, \rho_1(\gamma_1(e'_2)) \rangle) \in \mathcal{C}[(S_1 \times S_2)_L]\rho$$

Instantiate  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_2$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_2]]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_2)), \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S_2]]\rho$  to rewrite the goal to:

Assuming  $(v_2, v'_2) \in \mathcal{V}[[S_2]]\rho$

**Show:**

$$(\langle v_1, v_2 \rangle, \langle v'_1, v'_2 \rangle) \in \mathcal{C}[(S_1 \times S_2)_L]\rho$$

From the  $\mathcal{V}[[S_1 \times S_2]]\rho$  and the assumptions above, we derive that:

$$(\langle v_1, v_2 \rangle, \langle v'_1, v'_2 \rangle) \in \mathcal{V}[[S_1 \times S_2]]\rho,$$

Hence,  $(\langle v_1, v_2 \rangle, \langle v'_1, v'_2 \rangle) \in \mathcal{V}[(S_1 \times S_2)_L]\rho$ . Apply Lemma 44 (Related values are related terms)

Hence,  $(\langle v_1, v_2 \rangle, \langle v'_1, v'_2 \rangle) \in \mathcal{V}[(S_1 \times S_2)_L]\rho$

□

## Compatibility Fst

**Lemma 61** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Fst).

If  $\Delta; \Gamma \vdash e \approx e' : S$ ,  $S \triangleq (S_1 \times S_2)@U$

Then  $\Delta; \Gamma \vdash \text{fst } e \approx \text{fst } e' : \lceil S_1 \rceil_S$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \text{fst } e : \lceil S_1 \rceil_U$  and  $\Delta; \Gamma \vdash \text{fst } e' : \lceil S_1 \rceil_U$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{C}[\Gamma]\rho$

**Show:**

$$\begin{aligned} (\rho_1(\gamma_1(\text{fst } e)), \rho_2(\gamma_2(\text{fst } e'))) &\in \mathcal{C}[\lceil S_1 \rceil_U]\rho \equiv \\ (\text{fst } \rho_1(\gamma_1(e)), \text{fst } \rho_2(\gamma_2(e'))) &\in \mathcal{C}[\lceil S_1 \rceil_U]\rho \end{aligned}$$

Instantiate  $\Delta; \Gamma \vdash e \approx e' : (S_1 \times S_2)@U$  with  $\rho, \gamma_1, \gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e'))) \in \mathcal{C}[(S_1 \times S_2)@U]\rho$

Apply Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e'))) \in \mathcal{C}[(S_1 \times S_2)@U]\rho$  to rewrite the goal to:

Assuming  $(v, v') \in \mathcal{V}[(S_1 \times S_2)@U]\rho$

**Show:**

$$(\text{fst } v, \text{fst } v') \in \mathcal{C}[\lceil S_1 \rceil_U]\rho$$

Do case analysis on  $U$ :

**Case** ( $U = S_1 \times S_2$ ). Then  $\mathcal{V}[(S_1 \times S_2)@U]\rho = \mathcal{V}[S_1 \times S_2]\rho$

and our goal is **to show**

$$(\text{fst } v, \text{fst } v') \in \mathcal{C}[S_1]\rho$$

From  $(v, v') \in \mathcal{V}[S_1 \times S_2]\rho$  that  $v = \langle v_1, v_2 \rangle$  and  $\langle v'_1, v'_2 \rangle$  and  $(v_1, v'_1) \in \mathcal{V}[S_1]\rho$

Inspect the dynamic semantics to get  $\text{fst } v = v_1$  and  $\text{fst } v' = v'_1$

Hence our goal reduces to:

$$(v_1, v'_1) \in \mathcal{C}[S_1]\rho$$

which follows from above.

Hence,  $(\text{fst } v, \text{fst } v') \in \mathcal{C}[S_1]\rho$

**Case** ( $U = \top \vee U = X$ ). Denote  $S_1 \triangleq T_1@U_1$ , our goal rewrites to:

$$(\text{fst } v, \text{fst } v') \in \mathcal{C}[T_1@T]\rho$$

From  $(v, v') \in \mathcal{V}[(S_1 \times S_2)@U]\rho$  we know that  $(v, v') \in \text{Atom}[S_1 \times S_2]$

Apply the simple type system rules (Figure B.8) to obtain:

$$(\text{fst } v, \text{fst } v') \in \text{Atom}[T_1]$$

Apply the Lemma 47 (Atoms are related a top).

Hence,  $(\text{fst } v, \text{fst } v') \in \mathcal{C}[T_1@T]\rho$

□

## Compatibility Snd

**Lemma 62** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Snd).

If  $\Delta; \Gamma \vdash e \approx e' : S$ ,  $S \triangleq (S_1 \times S_2)@U$   
Then  $\Delta; \Gamma \vdash \text{snd } e \approx \text{snd } e' : [S_2]_S$

*Proof.* The proof is a mere replication of the steps for the Lemma 61 adapted for expressions  $\text{snd } e$  and  $\text{snd } e'$ .  $\square$

## Compatibility Pack

**Lemma 63** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Pack).

Let  $\Delta; \Gamma \vdash e \approx e' : (T [T'/X])_{\text{L}}$

Let  $T' \sqsubseteq \text{sftype}(\exists X.T)$

Then  $\Delta; \Gamma \vdash \text{pack}(T', e) \text{ as } \exists X.T \approx \text{pack}(T', e') \text{ as } \exists X.T : (\exists X.T)_{\text{L}}$

*Proof.* Denote  $T'' = \exists X.T$

Proof obligations:

1.  $\Delta; \Gamma \vdash \text{pack}(T', e_1) \text{ as } \exists X.T : (\exists X.T)_{\text{L}}$  and  $\Delta; \Gamma \vdash \text{pack}(T', e_1) \text{ as } \exists X.T : (\exists X.T)_{\text{L}}$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as:  $\rho \in \mathcal{D}[\Delta], (\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

**Show:**

$$(\rho_1(\gamma_1(\text{pack}(T', e) \text{ as } T'')), \rho_2(\gamma_2(\text{pack}(T', e') \text{ as } T''))) \in \mathcal{C}[(\exists X.T)_{\text{L}}]\rho$$

Instantiate  $\Delta; \Gamma \vdash e \approx e' : (T [T'/X])_{\text{L}}$  with  $\rho, \gamma_1, \gamma_2$ . Note that:

- $\rho \in \mathcal{D}[\Delta]$
- $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$

Hence,  $(\rho_1(\gamma_1(e)), \rho_2(\gamma_2(e))) \in \mathcal{C}[(T [T'/X])_{\text{L}}]\rho$

Apply the Lemma 48 (Monadic bind) to rewrite our goal to :

Assuming  $(v, v') \in \mathcal{V}[T [T'/X]_{\text{L}}]\rho$

**Show:**

$$\begin{aligned} & (\rho_1(\gamma_1(\text{pack}(T', v) \text{ as } T'')), \rho_2(\gamma_2(\text{pack}(T', v') \text{ as } T''))) \in \mathcal{C}[(\exists X.T)_{\text{L}}]\rho \\ \equiv & (\text{pack}(\rho_1(T'), \rho_1(\gamma_1(v)) \text{ as } \rho_1(T'')), \text{pack}(\rho_2(T'), \rho_2(\gamma_2(v')) \text{ as } \rho_2(T''))) \in \\ & \mathcal{C}[(\exists X.T)_{\text{L}}]\rho \end{aligned}$$

Since both pack expressions are already values, our remaining goal is **to show:**

$$(\text{pack}(\rho_1(T'), \rho_1(\gamma_1(v)) \text{ as } \rho_1(T'')), \text{pack}(\rho_2(T'), \rho_2(\gamma_2(v')) \text{ as } \rho_2(T''))) \in \mathcal{V}[(\exists X.T)_{\text{L}}]\rho$$

New proof obligations:

- $(\text{pack}(\rho_1(T'), \rho_1(\gamma_1(v)) \text{ as } \rho_2(T'')), \text{pack}(\rho_2(T'), \rho_2(\gamma_2(v')) \text{ as } \rho_2(T''))) \in \text{Atom}_{\rho} [\exists X.T]$   
Apply Lemma 41 (Substitutions preserve simple typing).
- $\exists R \in \text{Rel} [\rho_1(T'), \rho_2(T')] \Rightarrow (v, v') \in \mathcal{V}[T]\rho [X \mapsto (\rho_1(T'), \rho_2(T'), R)]$

Pick  $R = \mathcal{V}[T]_{\text{L}}\rho$ . Note that:

- $\mathcal{V}[T]_{\text{L}}\rho \in \text{Rel} [\rho_1(T'), \rho_2(T')]$ . This is equivalent to show:  $\mathcal{V}[T]_{\text{L}}\rho \subseteq \text{Atom} [\rho_1(T'), \rho_2(T')]$   
which follows by applying the Lemma 42 (PER simple typing).

Apply the Lemma 49 (PER Type Substitution: Compositionality) with  $\rho, T'$ . Note that:

- $\rho \in \mathcal{D}[\Delta]$
- $(v, v') \in \mathcal{V}[T [T'/X]_{\perp}] \rho$

Hence:  $(v, v') \in \mathcal{V}[T] \rho [X \mapsto (\rho_1(T'), \rho_1(T'), R)]$

□

## Compatibility Open

**Lemma 64** ( $\lambda_{\text{SEC}}^{\exists}$  Compatibility-Open).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : S, S \triangleq (\exists X.T)@U$

Let  $T'' = \text{sftype}(\exists X.T)$

Let  $\Delta, X : T''; \Gamma, x : [T@T]_S \vdash e_2 \approx e'_2 : S'$

Let  $\Delta \vdash S'$

Then  $\Delta; \Gamma \vdash \text{open}(X, x) = e_1 \text{ in } e_2 \approx \text{open}(X, x) = e'_1 \text{ in } e'_2 : [S']_S$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash \text{open}(X, x) = e_1 \text{ in } e_2 : [S']_S$  and  $\Delta; \Gamma \vdash \text{open}(X, x) = e'_1 \text{ in } e'_2 : [S']_S$
2. Assuming arbitrary  $\rho, \gamma_1, \gamma_2$  such as  $\rho \in \mathcal{D}[\Delta]$  and  $(\gamma_1, \gamma_2) \in \mathcal{C}[\Gamma] \rho$

**Show:**

$$(\rho_1(\gamma_1(\text{open}(X, x) = e_1 \text{ in } e_2)), \rho_2(\gamma_2(\text{open}(X, x) = e'_1 \text{ in } e'_2))) \in \mathcal{C}[[S']_S] \rho \equiv \\ (\text{open}(X, x) = \rho_1(\gamma_1(e_1)) \text{ in } \rho_1(\gamma_1(e_2)), \text{open}(X, x) = \rho_2(\gamma_2(e'_1)) \text{ in } \rho_2(\gamma_2(e'_2))) \in \\ \mathcal{C}[[S']_S] \rho$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e'_1 : (\exists X.T)@U$  with  $\rho, \gamma_1$  and  $\gamma_2$ .

Hence,  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[(\exists X.T)@U] \rho$

Apply the Lemma 48 (Monadic bind) with  $(\rho_1(\gamma_1(e_1)), \rho_2(\gamma_2(e'_1))) \in \mathcal{C}[(\exists X.T)@U] \rho$  to rewrite the goal to:

Assuming  $(v_1, v'_1) \in \mathcal{V}[(\exists X.T)@U] \rho$ .

**Show:**

$$(\text{open}(X, x) = v_1 \text{ in } \rho_1(\gamma_1(e_2)), \text{open}(X, x) = v'_1 \text{ in } \rho_2(\gamma_2(e'_2))) \in \mathcal{C}[[S']_S] \rho$$

Denote  $v_1 = \text{pack}(T''_1, v_{11})$  as  $\exists X.T$  and  $v'_1 = \text{pack}(T''_2, v'_{11})$  as  $\exists X.T$  for some  $T''_1$  and  $T''_2$ . Note that  $(v_1, v'_1) \in \text{Atom}_{\rho}[\exists X.T]$

Inspect the dynamic semantic to reduce the goal **to show:**

$$(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T''_1/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T''_2/X]) \in \mathcal{C}[[S']_S] \rho$$

By well-formedness (Figure B.4) of  $S \triangleq (\exists X.T)@U$ , which follows from the first hypothesis, the type  $U$  must be of the form:  $\exists X.T, \top$  or some type variable  $Y$ . Then, do case analysis on  $U$ .

**Case** ( $U = \exists X.T$ ). Then  $\mathcal{V}[(\exists X.T)@U] \rho = \mathcal{V}[\exists X.T] \rho$

and our goal rewrite **to show:**

$$(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T''_1/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T''_2/X]) \in \mathcal{C}[[S']_S] \rho$$

From  $(v_1, v'_1) \in \mathcal{V}[\exists X.T] \rho$  we know that:

(a)  $T_1'' \sqsubseteq \text{sftype}(\exists X.T) \wedge T_2'' \sqsubseteq \text{sftype}(\exists X.T)$

(b)  $\exists R.R \in \text{Rel}[T_1'', T_2'']$ .

(c)  $(v_{11}, v'_{11}) \in \mathcal{V}[[T]]\rho[X \mapsto (T_1'', T_2'', R)]$ .

Instantiate  $\Delta, X : T''; \Gamma, x : [T@T]_S \vdash e_2 \approx e'_2 : S$  with  $\rho[X \mapsto (T_1'', T_2'', R)]$ ,  $\gamma_1[x \mapsto v_{11}]$  and  $\gamma_2[x \mapsto v'_{11}]$ . Note that:

- $\rho[X \mapsto (T_1'', T_2'', R)] \in \mathcal{D}[\Delta, X : T'']$ , which requires to show:
  - $\rho \in \mathcal{D}[\Delta]$ . It follows from the above hypothesis.
  - $T_1'' \sqsubseteq T'' \wedge T_2'' \sqsubseteq T''$ . From hypothesis, we have that  $T'' = \text{sftype}(\exists X.T)$ , and  $T_1'' \sqsubseteq \text{sftype}(\exists X.T)$  follows from above (a).
  - $R \in \text{Rel}[T_1'', T_2'']$ . It follows from above (b).
- $(\gamma_1[x \mapsto v_{11}], \gamma_2[x \mapsto v'_{11}]) \in \mathcal{G}[\Gamma, x : [T@T]_S]\rho[X \mapsto (T_1'', T_2'', R)] = \mathcal{G}[\Gamma, x : T_1'']\rho[X \mapsto (T_1'', T_2'', R)]$ . Note that:
  - $(\gamma_1, \gamma_2) \in \mathcal{G}[\Gamma]\rho$ , given by hypothesis
  - $(v_{11}, v'_{11}) \in \mathcal{V}[[T]]\rho[X \mapsto (T_1'', T_2'', R)]$  which follows from above

Hence,  $(\rho[X \mapsto (T_1'', T_2'', R)](\gamma_1[x \mapsto v_{11}](e_2)), (\rho[X \mapsto (T_1'', T_2'', R)](\gamma_2[x \mapsto v'_{11}](e'_2))) \in \mathcal{C}[[S']]\rho[X \mapsto (T_1'', T_2'', R)]$

From hypothesis we know that  $\Delta \vdash S'$ , then rewrite the above statement as:

$(\rho[X \mapsto (T_1'', T_2'', R)](\gamma_1[x \mapsto v_{11}](e_2)), \rho[X \mapsto (T_1'', T_2'', R)](\gamma_2[x \mapsto v'_{11}](e'_2))) \in \mathcal{C}[[S']]\rho$  which is equivalent to our pending goal.

**Case** ( $U = \top$ ). Denote  $S' = T'@U'$ , then  $[S']_S = [T'@U']_{(\exists X.T)@T} = T'@T$  and our goal rewrite **to show**:

$$(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T_1''/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T_2''/X]) \in \mathcal{C}[[T'@T]]\rho$$

From  $(v_1, v'_1) \in \mathcal{V}[(\exists X.T)@U]\rho$  we know that  $(v_1, v'_1) \in \text{Atom}_\rho[(\exists X.T)]$ , hence  $\vdash_1 v_1 : \rho_1(\exists X.T)$  and  $\vdash_1 v'_1 : \rho_2(\exists X.T)$

Then, use inversion of the simply-typed judgment to have  $\vdash_1 v_{11} : \rho_1[X \mapsto T_1''](T)$  and  $\vdash_1 v'_{11} : \rho_1[X \mapsto T_2''](T)$

Apply the Lemma 41 (Substitutions preserve simple typing)

Hence,  $(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T_1''/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T_2''/X]) \in \text{Atom}_\rho[T']$

Apply Lemma 47 (Atoms are related a top).

Hence,  $(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T_1''/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T_2''/X]) \in \mathcal{C}[[T'@T]]\rho$

**Case** ( $U = Y$ ). Denote  $S' = T'@U'$ , then  $[S']_S = [T'@U']_{(\exists X.T)@Y} = T'@T$  and our goal rewrite **to show**:

$$(\rho_1(\gamma_1(e_2)) [v_{11}/x] [T_1''/X], \rho_2(\gamma_2(e'_2)) [v'_{11}/x] [T_2''/X]) \in \mathcal{C}[[T'@T]]\rho$$

Apply the same steps that for the case  $U = \top$

□

## Fundamental property

**Theorem 12** (Fundamental property).  $\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \vdash e \approx e : S$



*Proof.* The proof is by induction on the typing derivation of  $\Delta; \Gamma \vdash e : S$ .

Each case follows directly from the corresponding compatibility lemma (Lemmas 50 ... 64) □

**Lemma 11** (Self logical relation implies PRNI).

$\Delta; \Gamma \vdash e \approx e : S \implies \text{ERNI}(\Delta, \Gamma, e, S)$

*Proof.* The proof is direct given the similarity of both definitions. The only difference between both definitions is that  $\Delta, \Gamma \vdash e \approx e : S$  uses the security type system (Figure 3.3) and  $\text{ERNI}(\Delta, \Gamma, e, S)$  uses the simple type system (Figure B.8). We use Lemma 37 (Well-typed programs are simple well-typed) to show that  $\Delta; \Gamma \vdash e : T@U \implies \Delta; \Gamma \vdash_1 e : T$  □

# Appendix C

## Polymorphic relaxed noninterference

We include now the full definitions and proofs for Chapter 4.

### C.1 Full syntax and semantics

#### C.1.1 Syntax

$e$	::=	$v \mid e.m(e) \mid e.m \langle U \rangle (e) \mid x \mid \mathbf{b}$	(terms)
$o$	::=	$[z : S \Rightarrow \overline{m(x)} e]$	(objects)
$v$	::=	$o \mid \mathbf{b}$	(values)
$T$	::=	$O \mid \alpha \mid P$	(types)
$U, A, B$	::=	$T \mid X$	(policies)
$O$	::=	$\mathbf{Obj}(\alpha). [m : M]$	(object type)
$R$	::=	$[m : M]$	(record types)
$M$	::=	$\langle X : A..B \rangle S \rightarrow S \mid I$	(method signature)
$I$	::=	$P \triangleleft * \rightarrow P \triangleleft *$	(primitive signatures)
$S$	::=	$T \triangleleft U \mid P \triangleleft *$	(security types)
$P$	::=	(eg. Int)	(primitive type)

Figure C.1:  $\mathbf{Ob}_{\text{SEC}}^{\langle \rangle}$ : Full Syntax

#### Environments

$\Gamma$	::=	$\bullet \mid \Gamma, x : S$	(type environment)
$\Delta$	::=	$\bullet \mid \Delta, X : A..B$	(generic type variable environment)
$\Phi$	::=	$\bullet \mid \Phi, \alpha <: \beta \mid \Phi, P <: \beta$	(subtyping environment)
$\Delta_{\text{ok}}$	::=	$\bullet \mid \Delta_{\text{ok}}, \alpha \mid \Delta_{\text{ok}}, X : A..B$	(type variable environment)
$\Sigma$	::=	$\bullet \mid \Sigma, \alpha \triangleq O$	(type definition environment)

The type variable environment  $\Delta_{\text{ok}}$  keeps track of the type variable definitions, both self type variables and generic type variables. This environment is not directly used in the static

semantics presented in the body of the Chapter 4. Recall that in the typing judgment  $\Delta; \Gamma \vdash e : S$  of Figure 4.4, we extend the typing environment  $\Gamma$  with closed types regarding self type variables. For this reason we use  $\Delta$  instead of  $\Delta_{\text{ok}}$ . Note that  $\Delta$  just keeps track of generic type variable definition. Then, the environment  $\Delta_{\text{ok}}$  is just an auxiliary mechanism to verify that an object type is well-formed (Figure C.5)

### C.1.2 Type equivalence

The type equivalence judgment (Figure C.2) is essentially the same that the one in §A.1.4, with minor modifications to the rule (O-congr) to support type variables.

Two types are equivalent (Figure C.2) if the equivalence can be derived through the congruence induced by rules (Alpha-Eq) and (Fold-Unfold). For example:

$$\mathbf{Obj}(\alpha). [m : \langle X : \alpha.. \top \rangle \alpha \rightarrow \alpha] \equiv \mathbf{Obj}(\beta). [m : \langle X : \beta.. \top \rangle \beta \rightarrow \beta]$$

$$\mathbf{Obj}(\alpha). [m : \langle X : \alpha.. \top \rangle \top \rightarrow \alpha] \equiv \mathbf{Obj}(\alpha). [m : \langle X : \alpha.. \top \rangle \top \rightarrow \mathbf{Obj}(\beta). [m : \langle Y : \beta.. \top \rangle \top \rightarrow \beta]]$$

$$\boxed{U \equiv U}$$

$$\text{(Sym)} \frac{}{U \equiv U} \quad \text{(Refl)} \frac{U_1 \equiv U_2}{U_2 \equiv U_1} \quad \text{(Trans)} \frac{U_1 \equiv U_2 \quad U_2 \equiv U_3}{U_1 \equiv U_3}$$

$$\text{(O-Congr)} \frac{L_i \equiv L'_i \quad U_i \equiv U'_i \quad S_{1i} \equiv S'_{1i} \quad S_{2i} \equiv S'_{2i}}{\mathbf{Obj}(\alpha). [m : \langle X : A..B \rangle S_1 \rightarrow S_2] \equiv \mathbf{Obj}(\alpha). [m : \langle X : L'..U' \rangle S'_1 \rightarrow S'_2]}$$

$$\text{(Alpha-Eq)} \frac{O \triangleq \mathbf{Obj}(\alpha). [m : M] \quad \beta \text{ fresh}}{O \equiv O[\beta/\alpha]} \quad \text{(Fold-Unfold)} \frac{}{O \equiv O[O/\alpha]}$$

$$\boxed{S \equiv S}$$

$$\frac{T_1 \equiv T_2 \quad U_1 \equiv U_2}{T_1 \triangleleft U_1 \equiv T_2 \triangleleft U_2}$$

Figure C.2:  $\mathbf{Obj}_{\text{SEC}}^{\diamond}$ : Type equivalence

### C.1.3 Subtyping

$$\boxed{\Delta; \Phi \vdash U <: U}$$

$$\text{(SPObj)} \frac{\text{meths}(P) = R_1 \quad O \triangleq \mathbf{Obj}(\beta).R_2}{\Delta; \Phi, P <: \beta \vdash R_1 <: R_2} \quad \text{(SPVar)} \frac{P <: \beta \in \Phi}{\Delta; \Phi \vdash P <: \beta}$$

$$\begin{aligned} \text{(SObj)} & \frac{O_1 \triangleq \mathbf{Obj}(\alpha).R_1 \quad O_2 \triangleq \mathbf{Obj}(\beta).R_2}{\Delta; \Phi, \alpha <: \beta \vdash R_1 <: R_2} & \text{(SGVar1)} & \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash X <: B} & \text{(SGVar2)} & \frac{X : A..B \in \Delta}{\Delta; \Phi \vdash A <: X} \\ \text{(SVar)} & \frac{\alpha <: \beta \in \Phi}{\Delta; \Phi \vdash \alpha <: \beta} & \text{(SSubEq)} & \frac{O_1 \equiv O_2}{\Delta; \Phi \vdash O_1 <: O_2} \\ \text{(STrans)} & \frac{\Delta; \Phi \vdash U_1 <: U_2 \quad \Delta; \Phi \vdash U_2 <: U_3}{\Delta; \Phi \vdash U_1 <: U_3} \end{aligned}$$

$$\boxed{\Delta; \Phi \vdash R_1 <: R_2}$$

$$\text{(SR)} \frac{\overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies \Delta; \Phi \vdash M <: M'}{\Delta; \Phi \vdash [m : M] <: [m' : M']}$$

$$\boxed{\Delta; \Phi \vdash M <: M}$$

$$\text{(SM)} \frac{\Delta; \Phi \vdash B' <: B \quad \Delta; \Phi \vdash A <: A' \quad \Delta, X : A'..B'; \Phi \vdash S'_1 <: S_1 \quad \Delta, X : A'..B'; \Phi \vdash S_2 <: S'_2}{\Delta; \Phi \vdash \langle X : A..B \rangle S_1 \rightarrow S_2 <: \langle X : A'..B' \rangle S'_1 \rightarrow S'_2} \quad \text{(SImpl)} \frac{}{\Delta; \Phi \vdash I <: I}$$

$$\boxed{\Delta; \Phi \vdash S <: S}$$

$$\text{(TSubST)} \frac{\Delta; \Phi \vdash T_1 <: T_2 \quad \Delta; \Phi \vdash U_1 <: U_2}{\Delta; \Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure C.3:  $\mathbf{Ob}_{\text{SEC}}^{\diamond}$ : Subtyping rules

### C.1.4 Standard well formedness of types and environments

The Fig. C.4 and Fig. C.5 show the rules for well formedness of environments and types respectively.

Since  $\Delta$  match the syntax of  $\Delta_{\text{ok}}$ , we can use the judgments  $\Delta_{\text{ok}} \vdash U$  and  $\Delta_{\text{ok}} \vdash S$  with  $\Delta$  and they hold if  $U$  (an  $S$  respectively) are closed with respecto to self type variables  $\alpha$ .

$\Delta \vdash \Gamma$

$$\frac{}{\Delta \vdash \cdot} \quad \frac{\Delta \vdash \Gamma \quad \Delta \models S \quad x \notin \text{dom}(\Gamma)}{\Delta \vdash \Gamma, x : S}$$

$\vdash \Delta$

$$\frac{}{\vdash \cdot} \quad \frac{\vdash \Delta \quad X \notin \text{dom}(\Delta) \quad \Delta \vdash A \quad \Delta \vdash B}{\vdash \Delta, X : A..B}$$

$\vdash \Phi$

$$\frac{}{\vdash \cdot} \quad \frac{\vdash \Phi \quad \alpha_1 \notin \text{dom}(\Phi) \cup \text{cod}(\Phi)}{\vdash \Phi, \alpha_1 <: \alpha_2} \quad \frac{\vdash \Phi \quad \alpha_2 \notin \text{dom}(\Phi) \cup \text{cod}(\Phi) \quad \models P}{\vdash \Phi, P <: \alpha_2}$$

Figure C.4:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Environments well-formedness

$\Delta_{\text{ok}} \vdash U$

$$\frac{\models P}{\Delta_{\text{ok}} \vdash P} \quad \frac{X \in \Delta_{\text{ok}}}{\Delta_{\text{ok}} \vdash X} \quad \frac{\alpha \in \Delta_{\text{ok}}}{\Delta_{\text{ok}} \vdash \alpha} \quad \frac{O \equiv \mathbf{Obj}(\alpha). \left[ m : \langle X : A..B \rangle S_1 \rightarrow S_2 \right] \quad (i \neq j \implies m_i \neq m_j) \quad \Delta_{\text{ok}}, \alpha \vdash A \quad \Delta_{\text{ok}}, \alpha \vdash B}{\Delta_{\text{ok}}, \alpha, X : A..B \vdash S_{1i} \quad \Delta_{\text{ok}}, \alpha, X : A..B \vdash S_{2i}}}{\Delta_{\text{ok}} \vdash O}$$

$\Delta_{\text{ok}} \vdash S$

$$\frac{\Delta_{\text{ok}} \vdash T \quad \Delta_{\text{ok}} \vdash U}{\Delta_{\text{ok}} \vdash T \triangleleft U}$$

Figure C.5:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Well-formedness of types

### C.1.5 Well-formedness of security types facet-wise

$$\text{soundsig}(\langle \_ \rangle P_1 \triangleleft U_1 \rightarrow T_2 \triangleleft U_2) \iff P_1 = U_1 \vee U_2 = \top$$

Fig. C.6 shows extended subtyping judgments that are the same the ones of Figure C.3, except that the judgment  $\Delta; \Phi \vdash M \blacktriangleleft M$  adds the rule (IG) justifying that is OK to declassify a primitive signature with a concrete sound signature.

The judgment  $\Delta \models S$  holds if the type  $S$  is a closed type and a well-formed security type.

Having formalized well-formedness of environments and types, we assume them in most definitions.

$$\begin{array}{c}
 \boxed{\Delta; \Phi \vdash U \blacktriangleleft U} \\
 \dots \\
 \boxed{\Delta; \Phi \vdash R_1 \blacktriangleleft R_2} \\
 \dots \\
 \boxed{\Delta; \Phi \vdash M \blacktriangleleft M} \\
 \dots \quad \text{(IG)} \frac{\Delta; \Phi \vdash T_1 \blacktriangleleft P_1 \quad \Delta; \Phi \vdash P_2 \blacktriangleleft T_2 \quad \text{soundsig}(\langle \_ \rangle T_1 \triangleleft U_1 \rightarrow T_2 \triangleleft U_2)}{\Delta; \Phi \vdash P_1 \triangleleft * \rightarrow P_2 \triangleleft * \blacktriangleleft \langle \_ \rangle T_1 \triangleleft U_1 \rightarrow T_2 \triangleleft U_2} \\
 \boxed{\Delta; \Phi \vdash S \blacktriangleleft S} \\
 \dots
 \end{array}$$

Figure C.6:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Rules for valid relation between facets

$$\boxed{\Delta; \Sigma \vdash_{<} U}$$

$$\frac{O \equiv \mathbf{Obj}(\alpha). [m : M] \quad \Delta; \Sigma, \alpha : O \vdash_{<} M}{\Delta; \Sigma \vdash_{<} O} \quad \frac{}{\Delta; \Sigma \vdash_{<} \alpha} \quad \frac{}{\Delta; \Sigma \vdash_{<} X} \quad \frac{}{\Delta; \Sigma \vdash_{<} P}$$

$$\boxed{\Delta; \Sigma \vdash_{<} M}$$

$$\frac{\Delta, X : A..B; \Sigma \vdash_{<} S_1 \quad \Delta, X : A..B; \Sigma \vdash_{<} S_2}{\Delta; \Sigma \vdash_{<} \langle X : A..B \rangle S_1 \rightarrow S_2} \quad \frac{}{\Delta; \Sigma \vdash_{<} I}$$

$$\boxed{\Delta; \Sigma \vdash_{<} S}$$

$$\frac{\Delta; \Sigma \vdash_{<} T \quad \Delta; \Sigma \vdash_{<} U \quad \Delta; \cdot \vdash \Sigma [T] \blacktriangleleft \Sigma [U]}{\Delta; \Sigma \vdash_{<} T \triangleleft U}$$

$$\boxed{\Delta \models S}$$

$$\frac{\Delta \vdash S \quad \Delta; \cdot \vdash_{<} S}{\Delta \models S}$$

Figure C.7:  $\mathbf{Ob}_{\text{SEC}}^{\langle \rangle}$ : Well-formedness of security types facet-wise

## C.1.6 Auxiliary definitions

$$\boxed{\text{ub}(\Delta, U) = T}$$

$$\frac{T \neq X}{\text{ub}(\Delta, T) = T} \quad \frac{X : A..B \in \Delta \quad \text{ub}(\Delta, B) = T_2}{\text{ub}(\Delta, X) = T_2}$$

$$\boxed{\Delta \vdash m \in U}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). [\overline{m : M}]}{\Delta \vdash m_i \in O} \quad \frac{\text{meths}(P) = [\overline{m : I}]}{\Delta \vdash m_i \in P} \quad \frac{\Delta \vdash m \in \text{ub}(\Delta, X)}{\Delta \vdash m \in X}$$

$$\boxed{\text{msig}(\Delta, U, m) = M}$$

$$\frac{O \triangleq \mathbf{Obj}(\alpha). [\overline{m : M_i}]}{\text{msig}(\_, O, m_i) = M_i [O/\alpha]} \quad \frac{\text{meths}(P) = [\overline{m : I}]}{\text{msig}(\_, P, m_i) = I_i}$$

$$\frac{}{\text{msig}(\Delta, X, m) = \text{msig}(\_, \text{ub}(\Delta, X), m)}$$

$$\boxed{\text{methimpl}(o, m) = x.e}$$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m(x)e}]}{\text{methimpl}(o, m_i) = x.e_i}$$

$$\boxed{\Delta \vdash U \in A..B}$$

$$\frac{\Delta; \bullet \vdash A <: U \quad \Delta; \bullet \vdash U <: B}{\Delta \vdash U \in A..B}$$

$$\boxed{\text{retlab}(P \triangleleft U, P) = U}$$

$$\text{retlab}(P_1 \triangleleft U_1, P_2) = \begin{cases} P_2 & P_1 = U_1 \\ \top & \text{otherwise} \end{cases}$$

Figure C.8:  $\mathbf{Ob}_{\text{SEC}}^{\diamond}$ : Auxiliary definitions



## C.1.7 Typing

$\Delta; \Gamma \vdash e : S$

$$\begin{array}{c}
\text{(TVar)} \frac{x \in \text{dom}(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)} \quad \text{(TSub)} \frac{\Delta; \Gamma \vdash e : S' \quad \vdash S' <: S}{\Delta; \Gamma \vdash e : S} \\
\\
\text{(TPrim)} \frac{P = \Delta_b(\mathbf{b})}{\Gamma \vdash \mathbf{b} : P \triangleleft P} \quad \text{(TObj)} \frac{S \triangleq T \triangleleft U \quad \text{msig}(\_, T, m_i) = \langle X : A_i..B_i \rangle S'_i \rightarrow S''_i}{\Delta, X : A_i..B_i; \Gamma, z : S, x_i : S'_i \vdash e_i : S''_i}}{\Delta; \Gamma \vdash [z : S \Rightarrow \overline{m(x)}e] : S} \\
\\
\text{(TmD)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \in U \quad \text{msig}(\Delta, U, m) = \langle X : A..B \rangle S_1 \rightarrow S_2}{\Delta \vdash U' \in A..B \quad \Delta; \Gamma \vdash e_2 : S_1 [U'/X]}{\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : S_2 [U'/X]} \\
\\
\text{(TmH)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \notin U \quad \text{msig}(\_, T, m) \langle X : A..B \rangle S_1 \rightarrow T_2 \triangleleft U_2}{\Delta \vdash U' \in A..B \quad \Delta; \Gamma \vdash e_2 : S_1 [U'/X]}{\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : T_2 [U'/X] \triangleleft \top} \\
\\
\text{(TPmD)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \in U \quad \text{msig}(\Delta, U, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *}{\Delta; \Gamma \vdash e_2 : P_1 \triangleleft U_1 \quad \text{retlab}(P_1 \triangleleft U_1, P_2) = P'_2}}{\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft P'_2} \\
\\
\text{(TPmH)} \frac{\Delta; \Gamma \vdash e_1 : T \triangleleft U \quad \Delta \vdash m \notin U \quad \text{msig}(\Delta, T, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *}{\Delta; \Gamma \vdash e_2 : P_1 \triangleleft U_1}}{\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft \top}
\end{array}$$

Figure C.9:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Static semantics

### C.1.8 Dynamic semantics

$E ::= [] \mid E.m(e) \mid v.m(E)$  (evaluation contexts)

$$\text{(EMInvO)} \frac{o \triangleq [z : \_ \Rightarrow \_] \quad \text{methimpl}(o, m) = x.e}{E[o.m \langle \_ \rangle (v)] \mapsto E[e [o/z] [v/x] ]}$$

$$\text{(EMInvP)} \frac{}{E[\mathbf{b}_1.m(\mathbf{b}_2)] \mapsto E[\theta(m, \mathbf{b}_1, \mathbf{b}_2)]}$$

Figure C.10:  $\text{Ob}_{\text{SEC}}^{\diamond}$ : Full dynamic semantics

### C.1.9 Simple type System

Figure C.11 and Figure C.12 define a simple type system and simple subtyping judgments respectively that do not take into account the declassification type. The typing judgment  $\Gamma \vdash_{\text{sf}} e : S$  uses the subtyping judgment  $\Phi \vdash_{\text{sf}} S <: S$ . The definition of the judgment  $\Phi \vdash_{\text{sf}} T <: T$  is straightforward and then omit here.

$$\boxed{\Gamma \vdash_{\text{sf}} e : S}$$

$$\begin{array}{c}
 \text{(T1Var)} \frac{x \in \text{dom}(\Gamma)}{\Gamma \vdash_{\text{sf}} x : \Gamma(x)} \quad \text{(T1Sub)} \frac{\Gamma \vdash_{\text{sf}} e : S' \quad \vdash_{\text{sf}} S' <: S}{\Gamma \vdash_{\text{sf}} e : S} \quad \text{(T1Prim)} \frac{P = \Theta(b)}{\Gamma \vdash_{\text{sf}} b : P \triangleleft P} \\
 \\
 \text{(T1Obj)} \frac{\text{msig}(\_, T, m_i) = \langle \_ \rangle S'_i \rightarrow S''_i \quad \Gamma, z : S, x_i : S'_i \vdash_{\text{sf}} e_i : S''_i}{\Gamma \vdash_{\text{sf}} [z : S \Rightarrow \overline{m}(x)] e : S} \\
 \\
 \text{(T1mI)} \frac{\Gamma \vdash_{\text{sf}} e_1 : T \triangleleft U \quad \text{msig}(\_, T, m) = \langle \_ \rangle S_1 \rightarrow S_2 \quad \Gamma \vdash_{\text{sf}} e_2 : S_1}{\Gamma \vdash_{\text{sf}} e_1.m \langle \_ \rangle (e_2) : S_2} \\
 \\
 \text{(T1PmI)} \frac{\Gamma \vdash_{\text{sf}} e_1 : T \triangleleft U \quad \text{msig}(\_, T, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft * \quad \Gamma \vdash_{\text{sf}} e_2 : P_1 \triangleleft *}{\Gamma \vdash_{\text{sf}} e_1.m(e_2) : P_2 \triangleleft *}
 \end{array}$$

$$\boxed{\Gamma \vdash_1 e : T}$$

$$\frac{\Gamma \vdash_{\text{sf}} e : T \triangleleft U}{\Gamma \vdash_1 e : T}$$

Figure C.11:  $\text{Ob}_{\text{SEC}}^{\diamond}$  simple typing, defined in terms of single-facet typing

$$\boxed{\Phi \vdash_{\text{sf}} T <: T}$$

...

$$\boxed{\Phi \vdash_{\text{sf}} S <: S}$$

$$\frac{\Phi \vdash_{\text{sf}} T_1 <: U_1}{\Phi \vdash_{\text{sf}} T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure C.12:  $\text{Ob}_{\text{SEC}}^{\diamond}$  simple subtyping

## C.2 Type safety: proofs

## C.3 Unary model

We present the logical predicate for safety (Figure C.13). This logical predicate gives a safety meaning to a safety type, hence the declassification type does not play any role in the definitions.

$$\begin{aligned}
\mathcal{V}_k[[P]] &= \{ \mathbf{b} \mid \\
&\quad (\forall j < k. v \in \mathcal{V}_j[[P]] \wedge \\
&\quad (\forall m \in P, \mathbf{b}'. \text{msig}(\bullet, P, m) = P' \triangleleft^* \rightarrow P'' \triangleleft^* \\
&\quad \mathbf{b}' \in \mathcal{V}_j[[P']] \implies \theta(m, \mathbf{b}, \mathbf{b}') \in \mathcal{C}_j[[P''])) \} \\
\mathcal{V}_k[[O]] &= \{ v = [z : O_1 \triangleleft \_ \Rightarrow \_] \mid \vdash_1 O_1 \triangleleft \_ < : O \triangleleft \_ \wedge \\
&\quad (\forall j < k. v \in \mathcal{V}_j[[O_1]] \wedge \\
&\quad (\forall m \in O, v'. \text{msig}(\bullet; O, m) = \langle \_ \rangle T' \triangleleft \_ \rightarrow T'' \triangleleft \_ \quad \text{methimpl}(v, m) = x.e \\
&\quad v' \in \mathcal{V}_j[[T']] \implies e[v/z][v'/x] \in \mathcal{C}_j[[T''])) \} \\
\mathcal{C}_k[[T]] &= \{ e \mid \forall j < k. \forall e'. (e \mapsto^j e' \wedge \text{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[[T]] \}
\end{aligned}$$

Figure C.13: Unary logical relation for safety

## C.4 Type safety

**Definition 15** (Safety).  $\text{safe}(e) \iff \forall e'. e \mapsto^* e' \implies e' = v \text{ or } \exists e''. e' \mapsto e''$

**Definition 26** (Semantic typing).  $\models e : T \triangleleft U \iff \forall k \geq 0. e \in \mathcal{C}_k[[T]]$ .

**Lemma 65** (Semantic type safety).  $\models e : S \implies \text{safe}(e)$

*Proof.* The proof follows directly from definitions 15 and 26 . □

**Lemma 66** (Type Safety).  $\Gamma \vdash e : S \implies \Gamma \models e : S$

*Proof.* The proof is by induction on the typing derivation of  $e$ . The different case with respect to the proof of type safety of  $\text{Ob}_{\text{SEC}}$  (Chapter 2) is the case (TPrim). For that case we only need to assume that  $\Theta$  and  $\theta$ —which are parameters of the language—agree on the specification and implementation of all primitive types and their operations. □

**Theorem 14** (Syntactic type safety).  $\vdash e : S \implies \text{safe}(e)$

*Proof.* This follows directly from Lemmas 65 and 66 □

## C.5 Polymorphic relaxed noninterference: proofs

### C.5.1 Logical relation

Figure C.14 shows the full logical relation for PRNI

$$\begin{array}{lcl}
\sigma & ::= & \emptyset \mid \sigma[X \mapsto T] \\
\text{Atom}_n [T] & = & \{(k, e_1, e_2) \mid k < n \wedge \vdash_1 e_1 : T \wedge \vdash_1 e_2 : T\} \\
\text{Atom}_n^{\text{val}} [T] & = & \{(k, v_1, v_2) \in \text{Atom}_n [T]\} \\
\text{Atom} [T] & = & \{(k, e_1, e_2) \in \bigcup_{n \geq 0} \text{Atom}_n [T]\} \\
\mathcal{NV} [P \triangleleft P] & = & \{(k, \mathbf{b}, \mathbf{b}) \in \text{Atom} [P]\} \\
\mathcal{NV} [T \triangleleft O] & = & \{(k, v_1, v_2) \in \text{Atom} [T] \mid \\
& & (\forall m \in O. \text{msig}(O, m) = \langle X : A..B \rangle S' \rightarrow S'' \\
& & \quad \forall j < k, T', v'_1, v'_2. \vdash T' \wedge T' \in A..B \wedge \\
& & \quad (j, v_1, v_2) \in \mathcal{NV} [T \triangleleft O] \wedge (j, v'_1, v'_2) \in \mathcal{NV} [S' [T'/X]] \implies \\
& & \quad (j, v_1.m \langle \_ \rangle (v'_1), v_2.m \langle \_ \rangle (v'_2)) \in \mathcal{NC} [S'' [T'/X]]) \wedge \\
& & (\forall m \in O. \text{msig}(O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft * \\
& & \quad \forall j < k, v'_1, v'_2. U_1 > : P_1 \\
& & \quad ((j, v'_1, v'_2) \in \mathcal{NV} [P_1 \triangleleft U_1]) \implies (j, v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC} [P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)])\} \\
\mathcal{NC} [T \triangleleft U] & = & \{(k, e_1, e_2) \in \text{Atom} [T] \mid (\forall j < k. (e_1 \mapsto^{\leq j} v_1 \wedge e_2 \mapsto^{\leq j} v_2) \implies (k - j, v_1, v_2) \in \mathcal{NV} [T \triangleleft U])\} \\
\mathcal{NG} [\cdot] & = & \{(k, \emptyset, \emptyset)\} \\
\mathcal{NG} [\Gamma, x : S] & = & \{(k, \gamma_1 [x \mapsto v_1], \gamma_2 [x \mapsto v_2]) \mid (k, \gamma_1, \gamma_2) \in \mathcal{NG} [\Gamma] \wedge (k, v_1, v_2) \in \mathcal{NV} [S]\} \\
\mathcal{ND} [\cdot] & = & \{\emptyset\} \\
\mathcal{ND} [\Delta, X : A..B] & = & \{\sigma [X \mapsto T] \mid \sigma \in \mathcal{ND} [\Delta] \wedge \Delta \vdash T \in A..B\} \\
\text{PRNI}(\Delta, \Gamma, e, S) & \iff & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \wedge \Delta \vdash \Gamma \wedge \Delta \vdash S \\
& & \forall k \geq 0. \forall \sigma, \gamma_1, \gamma_2. \sigma \in \mathcal{ND} [\Delta]. (k, \gamma_1, \gamma_2) \in \mathcal{NG} [\sigma(\Gamma)] \\
& & \implies (k, \sigma(\gamma_1(e)), \sigma(\gamma_2(e))) \in \mathcal{NC} [\sigma(S)] \\
\Delta; \Gamma \vdash e_1 \approx e_2 : S & \iff & \Delta; \Gamma \vdash e_i : S \wedge \forall k \geq 0. \forall \sigma, \gamma_1, \gamma_2. \sigma \in \mathcal{ND} [\Delta]. \\
& & (k, \gamma_1, \gamma_2) \in \mathcal{NG} [\sigma(\Gamma)] \implies (k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC} [\sigma(S)]
\end{array}$$

Figure C.14:  $\text{Ob}_{\text{SEC}}^{\diamond}$ . Step-indexed logical relations for PRNI

$$\begin{array}{lcl}
\gamma & ::= & \emptyset \mid \gamma [x \mapsto v] \\
\emptyset \models \Gamma & & \\
\gamma [x \mapsto v] \models \Gamma, x : S & \iff & \gamma \models \Gamma \wedge \bullet; \bullet \vdash v : S
\end{array}$$

Figure C.15:  $\text{Ob}_{\text{SEC}}^{\diamond}$ . PRNI. Auxiliary definition (used in proofs)

## C.5.2 Auxiliary lemmas: simple typing

**Lemma 67** (Well-type programs are simple well-typed).

*If  $\Delta; \Gamma \vdash e : T \triangleleft U$  then  $\Gamma \vdash_1 e : T$*

**Lemma 68** (Security subtyping implies simple subtyping).

*If  $\Delta; \Phi \vdash T' <: T$  then  $\Phi \vdash_{\text{sf}} T' <: T$*

**Lemma 69** (Value substitution preserves simple typing).

*If  $\Gamma \vdash_1 e : T$  and  $\gamma \models \Gamma$  then  $\vdash_1 \gamma(e) : T$*

*Proof.* By induction on  $\Gamma$ . □

### C.5.3 Auxiliary lemmas: logical relation

#### Atom subtyping

**Lemma 70** (Atom subtyping).

If  $(k, v_1, v_2) \in \text{Atom}[T']$  and  $\bullet; \bullet \vdash T' <: T$   
then  $(k, v_1, v_2) \in \text{Atom}[T]$

*Proof.* Proof obligations:

$\vdash_1 e_i : T$  ( $i \in \{1, 2\}$ ). Apply rule (T1Sub). Note that  $\bullet; \bullet \vdash T' <: T \Rightarrow \bullet \vdash_{\text{sf}} T' <: T$   
(Lemma 68, Security subtyping implies simple subtyping)  $\square$

#### Atom reduction

**Lemma 71** (Atom reduction).

Let  $e_1 \mapsto^* e'_1$  and  $e_2 \mapsto^* e'_2$   
Let  $(k, e_1, e_2) \in \text{Atom}[T]$   
Then  $(k, e'_1, e'_2) \in \text{Atom}[T]$

*Proof.* The proof is straightforward. Each subgoal follows by induction on the typing derivation of  $e_i$  ( $i \in \{1, 2\}$ ).  $\square$

#### Type substitution preserves subtyping

**Lemma 72** (Type substitution preserves subtyping).

Let  $\sigma \in \mathcal{ND}[\Delta]$   
If  $\Delta; \bullet \vdash S <: S'$  then  $\bullet; \bullet \vdash \sigma(S) <: \sigma(S')$   
If  $\Delta; \bullet \vdash T <: T'$  then  $\bullet; \bullet \vdash \sigma(T) <: \sigma(T')$

#### Type substitution preserves interval subtyping

**Lemma 73** (Type substitution preserves interval subtyping).

Let  $\sigma \in \mathcal{ND}[\Delta]$  and  $\Delta \vdash U$ ,  $\Delta \vdash A$ ,  $\Delta \vdash B$   
If  $\Delta \vdash U \in A..B$   
then  $\sigma(U) \in \sigma(A).. \sigma(B)$

*Proof.* Apply Lemma 72 (Type substitution preserves subtyping) for each subgoal.  $\square$

#### Interval subtyping expansion

**Lemma 74** (Interval subtyping expansion).

Let  $\sigma \in \mathcal{ND}[\Delta]$  and  $\bullet \vdash U$ ,  $\Delta \vdash A$ ,  $\Delta \vdash B$   
If  $\bullet \vdash U \in \sigma(A).. \sigma(B)$   
then  $\Delta \vdash U \in A..B$



## Downward closed/Monotonicity

**Lemma 75** (Downward closed/Monotonicity).

Let  $\bullet \vdash S$

If  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$  and  $j \leq k$

then  $(j, v_1, v_2) \in \mathcal{NV}[[S]]$

*Proof.* The proof is by induction on  $S$ . All valid cases boil down to  $P \triangleleft P$  and  $T \triangleleft O$

**Case** ( $S = P \triangleleft P$ ). This is direct from the definition of  $\mathcal{NV}[[P \triangleleft P]]$

**Case** ( $S = T \triangleleft O$ ).

*Proof obligations:*

1.  $(j, v_1, v_2) \in \text{Atom}[T]$ . This follows directly from  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$
2. Assuming arbitrary  $m, j', T', v'_1, v'_2$  such as:
  - $m \in O \wedge \text{msig}(\bullet; O, m) = \langle X : A..B \rangle S_1 \rightarrow S_2$
  - $j' < j$
  - $\vdash T' \wedge T' \in A..B$
  - $(j', v_1, v_2) \in \mathcal{NV}[[S]]$
  - $(j', v'_1, v'_2) \in \mathcal{NV}[[S_1 [X/T']]]$

**Show:**

$$(j, v_1.m \langle T' \rangle (v'_1), v_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[[S'' [X/T']]]$$

Instantiate the first conjunct of  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$  with  $m, j', T', v'_1, v'_2$ . Note that:

- $m \in O \wedge \text{msig}(\bullet; O, m) = \langle X : A..B \rangle S_1 \rightarrow S_2$
- $j' < k$ . It follows from  $j' < j \leq k$
- $\vdash T' \wedge T' \in A..B$
- $(j', v_1, v_2) \in \mathcal{NV}[[S]]$
- $(j', v'_1, v'_2) \in \mathcal{NV}[[S_1 [X/T']]]$

Hence  $(j, v_1.m \langle T' \rangle (v'_1), v_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[[S'' [X/T']]]$

3. Assuming arbitrary  $m, j, v'_1, v'_2$  such as:

- $m \in O \quad \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$
- $(j, v'_1, v'_2) \in \mathcal{NV}[[P_1 \triangleleft U_1]]$

**Show:**

$$(j', v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$$

Instantiate the second conjunct of  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$  with  $m, j', v'_1, v'_2$ . Note that:

- $m \in O \quad \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$
- $j' < k$ . It follows from  $j' < j \leq k$
- $(j, v'_1, v'_2) \in \mathcal{NV}[[P_1 \triangleleft U_1]]$

Hence  $(j', v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$

□

## Syntactic equivalences implies semantic equivalence

**Lemma 18** (Equal values are logically related).

$\forall k \geq 0, \mathbf{b}, P, O.$

$$\vdash_1 \mathbf{b} : O \wedge P <: O \implies (k, \mathbf{b}, \mathbf{b}) \in \mathcal{NV}[[P \triangleleft O]]$$

*Proof.* Assuming arbitrary  $m, j, v'_1, v'_2$  such as:

- $m \in O \quad \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$
- $(j, v'_1, v'_2) \in \mathcal{NV}[[P_1 \triangleleft U_1]]$

**Show:**

$$(j, v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$$

**Case** ( $U_1 = P_1$ ). *The*  $v'_1 = v'_2 = \mathbf{b}'$  *and we have to show*

$$\begin{aligned} & (j, \mathbf{b}.m(\mathbf{b}'), \mathbf{b}.m(\mathbf{b}')) \in \mathcal{NC}[[P_2 \triangleleft P_2]] \\ \equiv & (j, \theta(m, \mathbf{b}, \mathbf{b}'), \theta(m, \mathbf{b}, \mathbf{b}')) \in \mathcal{NC}[[P_2 \triangleleft P_2]] \end{aligned}$$

*which follows from the assumption that  $\theta$  is partial function that respects that signatures of  $\text{meths}(P)$ .*

**Case** ( $U_1 \neq P_1$ ). *Then we have to show*

$$(j, \mathbf{b}.m(v'_1), \mathbf{b}.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \top]]$$

*which trivially follows by using Lemma 80 (Well-typed terms are related at top)*

□

## PER Subtyping

**Lemma 76** (PER Subtyping).

*Let*  $\bullet \vdash S$ ,  $\bullet \vdash S'$  *and*  $\bullet; \bullet \vdash S' <: S$

- (1) *If*  $(k, v_1, v_2) \in \mathcal{NV}[[S']]$  *then*  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$
- (2) *If*  $(k, e_1, e_2) \in \mathcal{NC}[[S']]$  *then*  $(k, e_1, e_2) \in \mathcal{NC}[[S]]$

*Proof.* We proof the statements (1) and (2) simultaneously.

Induction on  $k$  and then nested induction on  $S$ .

We focus on  $k > 0$  (the case  $k = 0$  is trivial).

Statement (1):

All valid cases for  $S$  boils down to  $P \triangleleft P$  and  $T \triangleleft O$ .

**Case** ( $S = P \triangleleft P$ ).

*Proof obligations:*

1.  $(k, v_1, v_2) \in \text{Atom}[P]$ . Apply Lemma 70 (*Atom subtyping*)
2.  $v_1 = v_2 = b$ . From the third hypothesis we know that  $S' = P \triangleleft P$  and for the second one we know that  $v_1 = v_2 = b$

**Case** ( $S = T \triangleleft O$ ).

Denote  $S' = T' \triangleleft O'$

*Proof obligations:*

1.  $(k, v_1, v_2) \in \text{Atom}[T]$ . Apply Lemma 70 (*Atom subtyping*)
2. Assuming arbitrary  $m, j < k, T', v'_1, v'_2$  such as:

- $m \in O \quad \text{msig}(\bullet, O, m) = \langle X : A..B \rangle S_1 \rightarrow S_2$
- $\vdash T' \wedge T' \in A..B$
- $(j, v_1, v_2) \in \mathcal{NV}[[S]]$
- $(j, v'_1, v'_2) \in \mathcal{NV}[[S_1[X/T']]]$

**Show:**

$$(j, o_1.m \langle T' \rangle (v'_1), o_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[[S_2[X/T']]]$$

Instantiate  $(k, v_1, v_2) \in \mathcal{NV}[[T' \triangleleft O']]$  with  $m, j, T', v'_1, v'_2$ . Note that:

- $m \in O' \quad \text{msig}(\bullet, O', m) = \langle X : L'..U' \rangle S'_1 \rightarrow S'_2$
- $j < k$
- $\vdash T' \wedge T' \in A'..B'$ .
- $(j, v'_1, v'_2) \in \mathcal{NV}[[S'_1[X/T']]]$ . Apply the IH with  $(j, v'_1, v'_2) \in \mathcal{NV}[[S_1[X/T']]]$  and  $\bullet; \bullet \vdash S_1[X/T'] <: S'_1[X/T']$

Hence,  $(j, o_1.m \langle T' \rangle (v'_1), o_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[[S'_2[X/T']]]$

Apply IH, statement (2) with  $\bullet; \bullet \vdash S'_2[X/T'] <: S_2[X/T']$  to obtain

$$(j, o_1.m \langle T' \rangle (v'_1), o_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[[S_2[X/T']]]$$

3. Assuming arbitrary  $m, j, v'_1, v'_2$  such as:

- $m \in O \quad \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$
- $(j, v'_1, v'_2) \in \mathcal{NV}[[P_1 \triangleleft U_1]]$

**Show:**

$$(j, v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$$

Do a case analysis on  $S'$ . Each case reduces to  $S' = T' \triangleleft O'$  or  $S' = P' \triangleleft P'$

**Case** ( $S' = T' \triangleleft O'$ ). Instantiate  $(k, v_1, v_2) \in \mathcal{NV}[[T' \triangleleft O']]$  with  $m, j, v'_1, v'_2$ . Note that:

- $m \in O \quad \text{msig}(\bullet, O, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$ . Recall, that there is no subtyping rules between primitive types.

- $(j, v'_1, v'_2) \in \mathcal{NV}[[P_1 \triangleleft U_1]]$

Hence  $(j, v_1.m(v'_1), v_2.m(v'_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft U_1, P_2)]]$

**Case**  $(S' = P' \triangleleft P')$ .

It means that  $v_1 = v_2 = b$ . Apply Lemma 18 (Equal values are logically related) with  $b, P'$  and  $O$ . Note that  $\vdash_1 b : P'$  and  $P' <: O$ .

Statement (2):

Denote  $S \triangleq T \triangleleft U$  and  $S' \triangleq T' \triangleleft U'$

Proof obligations:

1.  $(k, e_1, e_2) \in \text{Atom}[T]$ . Apply Lemma 70 (Atom subtyping) with  $(k, e_1, e_2) \in \text{Atom}[T']$  ( $(k, e_1, e_2) \in \mathcal{NC}[[S']]$ ) and  $\bullet; \bullet \vdash T' <: T$
2. Assuming arbitrary  $j < k, v_1, v_2$  such as  $j < k$ :
  - $e_1 \mapsto^j v_1$
  - $e_2 \mapsto^j v_2$

**Show:**

$$(j, v_1, v_2) \in \mathcal{NV}[[S]]$$

Instantiate  $(k, e_1, e_2) \in \mathcal{NC}[[S']]$  with  $j, v_1, v_2$  to obtain:  $(k, v_1, v_2) \in \mathcal{NV}[[S']]$ .

Apply the IH, statement (1) with  $(k, v_1, v_2) \in \mathcal{NV}[[S']]$  and  $\bullet; \bullet \vdash S' <: S$  to obtain:

$$(k, v_1, v_2) \in \mathcal{NV}[[S]]$$

□

## Anti reduction

**Lemma 77** (Anti reduction).

Let  $S \triangleq T \triangleleft U$

Let  $(j, e_1, e_2) \in \text{Atom}[T]$

Let  $j' \leq j$  and  $j \leq j' + k$

Let  $e_1 \mapsto^{\leq k} e'_1$  and  $e_2 \mapsto^{\leq k} e'_2$

Let  $(j', e'_1, e'_2) \in \mathcal{NC}[[S]]$

Then  $(j, e_1, e_2) \in \mathcal{NC}[[S]]$

*Proof.* Denote  $S \triangleq T \triangleleft U$ .

Proof obligations:

1.  $(j, e_1, e_2) \in \text{Atom}[T]$ . Apply Lemma 71 (Atom reduction) with  $(j', e'_1, e'_2) \in \text{Atom}[T]$  which follows from  $(j', e'_1, e'_2) \in \mathcal{NC}[[S]]$
2. Assuming  $j_1 < j, v_1, v_2$  such as:
  - $e_1 \mapsto^{\leq j_1} v_1$
  - $e_2 \mapsto^{\leq j_1} v_2$

**Show:**

$$(j - j_1, v_1, v_2) \in \mathcal{NV}[[S]]$$

We have that:

$$\begin{aligned} e_1 &\mapsto^{\leq k} e'_1 \mapsto^{j'_1} v_1 \\ e_2 &\mapsto^{\leq k} e'_2 \mapsto^{j'_1} v_2 \end{aligned}$$

where  $j'_1 < j'$ .

Instantiate  $(j', e'_1, e'_2) \in \mathcal{NC}[[S]]$  with  $j'_1$ . Note that  $j'_1 < j'$ .

Hence,  $(j' - j'_1, v_1, v_2) \in \mathcal{NV}[[S]]$ .

Apply Lemma 75 (Downward closed/Monotonicity) with  $j - j_1 \leq j - (k + j'_1) \leq j' - j'_1$  ( $j - k \leq j'$ ) to obtain:

$$(j - j_1, v_1, v_2) \in \mathcal{NV}[[S]]$$

□

## Monadic bind

**Lemma 78** (Monadic bind).

If  $(k, e_1, e_2) \in \mathcal{NC}[[S]]$

and  $\forall j \leq k. \forall v_1, v_2. (j, v_1, v_2) \in \mathcal{NV}[[S]] \implies (j, E[v_1], E[v_2]) \in \mathcal{NC}[[S']]$

then  $(k, E[e_1], E[e_2]) \in \mathcal{NC}[[S']]$

*Proof.* Let us assume that  $e_2 \mapsto^{\leq j'} v'_1$  and  $e_2 \mapsto^{\leq j'} v'_2$  where  $j' \leq k$  (in other case the lemma vacuously holds)

Instantiate  $(k, e_1, e_2) \in \mathcal{NC}[[S]]$  with  $j', v'_1, v'_2$ .

Hence,  $(k - j', v'_1, v'_2) \in \mathcal{NV}[[S']]$

By the dynamic semantics we have know that:

$$\begin{aligned} E[e_1] &\mapsto^{\leq j'} E[v'_1] \\ E[e_2] &\mapsto^{\leq j'} E[v'_2] \end{aligned}$$

Instantiate the second premise with  $k - j', v'_1, v'_2$ . Note that  $k - j' \leq k$  and  $(k - j', v'_1, v'_2) \in \mathcal{NV}[[S']]$ .

Hence,  $(k - j', E[v'_1], E[v'_2]) \in \mathcal{NC}[[S']]$

Instantiate Lemma 77 (Anti reduction). Note that:

- $k - j' \leq k$
- $k \leq k - j' + j'$
- $E[e_1] \mapsto^{\leq j'} E[v'_1]$
- $E[e_2] \mapsto^{\leq j'} E[v'_2]$
- $(k - j', E[v'_1], E[v'_2]) \in \mathcal{NC}[[S']]$

Hence,  $(k, E[e_1], E[e_2]) \in \mathcal{NC}[[S']]$  □

### Substitutions preserve simple typing

**Lemma 79** (Substitutions preserve simple typing).

Let  $\Delta, \Gamma \vdash e_1 : T_1 \triangleleft U_1$  and  $\Delta, \Gamma \vdash e_2 : T_1 \triangleleft U_1$

Let  $\sigma \in \mathcal{ND}[[\Delta]]$  and  $(k, \gamma_1, \gamma_2) \in \mathcal{NG}[[\sigma(\Gamma)]]$

Then  $(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \text{Atom}[\sigma(T_1)]$

*Proof.* The proof is straightforward. Then the goal is equivalent to show:  $(k, \gamma_1(e_1), \gamma_2(e_2)) \in \text{Atom}[\sigma(T_1)]$  (because type variable are not taken into account by the simple type system).

For each value substitution apply Lemma 69 (Value substitution preserves simple typing) with  $\Gamma \vdash_1: e_i$  and  $\gamma_i \models \Gamma$  to obtain  $\vdash_1 \gamma_i(e_i) : \sigma(T_1)$  □

### Well-typed terms are related at top

**Lemma 80** (Well-typed terms are related at top).

Let  $\sigma \in \mathcal{ND}[[\Delta]]$  and  $(k, \gamma_1, \gamma_2) \in \mathcal{NG}[[\sigma(\Gamma)]]$

Let  $\Delta; \Gamma \vdash e_1 : T \triangleleft \top$

Let  $\Delta; \Gamma \vdash e_2 : T \triangleleft \top$

Then  $(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC}[[\sigma(T) \triangleleft \top]]$

*Proof.* Proof obligations

1.  $(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \text{Atom}[\sigma(U)]$ . It follows from Lemma 79 (Substitutions preserve simple typing).
2. Assuming  $j < k$ ,  $v_1, v_2$  such as:
  - $\sigma(\gamma_1(e_1)) \mapsto^{\leq j} v_1$
  - $\sigma(\gamma_2(e_2)) \mapsto^{\leq j} v_2$

**Show**

$$(k - j, v_1, v_2) \in \mathcal{NV}[[\sigma(T) \triangleleft \top]]$$

Which is equivalent to show

$$(k - j, v_1, v_2) \in \text{Atom}[\sigma(T)]$$

Apply Lemma 71 (Atom reduction) with  $(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \text{Atom}[\sigma(T)]$ ,  $v_1$  and  $v_2$  to obtain

$$(k - j, v_1, v_2) \in \text{Atom}[\sigma(T)]$$

□

### Related values are related terms

**Lemma 81** (Related values are related terms). *If  $(k, v_1, v_2) \in \mathcal{NV}[[S]]$  then  $(k, v_1, v_2) \in \mathcal{NC}[[S]]$*

*Proof.* The proof trivially follows.

□

## C.5.4 Proof of the Fundamental Property

### Pre-compatibility: Method Invocation

**Lemma 82** (Pre-compatibility: Method Invocation).

Let  $\sigma \in \mathcal{ND}[\Delta]$ ,  $\Delta \vdash T_1 \triangleleft U_1$

Let  $k'' \leq k' \leq k$

Let  $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T_1 \triangleleft U_1)]$

Let  $\Delta \vdash m \in U_1$  and  $\text{msig}(\Delta; U_1, m) = \langle X : A..B \rangle S_2 \rightarrow S$

Let  $\Delta \vdash U' \wedge \Delta \vdash U' \in A..B$

Let  $(k, v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2 [U'/X])]$

then  $(k'', v_1.m \langle \sigma(U') \rangle (v_1''), v_2.m \langle \sigma(U') \rangle (v_2'')) \in \mathcal{NC}[\sigma(S [U'/X])]$

*Proof.* Instantiate  $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T_1 \triangleleft U_1)]$  with:  $m, k'', \sigma(U'), v_1'', v_2''$ . Note that:

- $m \in \sigma(T_1)$ . It follows from  $\Delta \vdash m \in U_1$ ,  $\sigma \in \mathcal{ND}[\Delta]$  and  $\Delta \vdash T_1 \triangleleft U_1$ . Then  $\text{msig}(\bullet, \sigma(T_1), m) = \langle X : \sigma(A).. \sigma(B) \rangle \sigma(S_2) \rightarrow \sigma(S)$ .
- $k'' < k$  which follows directly from hypothesis.
- $\vdash \sigma(U')$  which is direct from  $\Delta \vdash U'$  and  $\rho \in \mathcal{ND}[\Delta]$ .
- $\sigma(U') \in \sigma(A).. \sigma(B)$ . Apply Lemma 73 (Type substitution preserves interval subtyping) with  $\sigma \in \mathcal{ND}[\Delta]$  and  $\Delta \vdash U' \in A..B$
- $(k'', v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2) [\sigma(U')/X]]$ . It follows from
  - $(k, v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2) [\sigma(U')/X]]$ . Note that  $\sigma(S_2 [U'/X]) = \sigma(S_2) [\sigma(U')/X]$  and  $(k, v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2 [U'/X])]$  is given in hypothesis
  - Apply Lemma 75 (Downward closed/Monotonicity) with  $(k, v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2) [\sigma(U')/X]]$  and  $k'' \leq k$  we obtain  $(k'', v_1'', v_2'') \in \mathcal{NV}[\sigma(S_2) [\sigma(U')/X]]$ .

Hence,  $(k'', v_1.m \langle \sigma(U') \rangle (v_1''), v_2.m \langle \sigma(U') \rangle (v_2'')) \in \mathcal{NC}[\sigma(S) [\sigma(U')/X]]$

Note  $\sigma(S [U'/X]) = \sigma(S) [\sigma(U')/X]$ .

Hence,  $(k'', v_1.m \langle \sigma(U') \rangle (v_1''), v_2.m \langle \sigma(U') \rangle (v_2'')) \in \mathcal{NC}[\sigma(S [U'/X])]$  □

### Compatibility-Var

**Lemma 83** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility-Var).

$\Delta; \Gamma \vdash x \approx x : \Gamma(x)$

*Proof.* First, let us denote  $S \triangleq \Gamma(x)$ .

Proof obligations:

1.  $\Delta; \Gamma \vdash x : S$  which is direct.
2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[\Delta], \gamma_1, \gamma_2, (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\rho(\Gamma)]$

**Show**



$$\begin{aligned} (k, \sigma(\gamma_1(x)), \sigma(\gamma_2(x))) &\in \mathcal{NC}[\sigma(S)] \\ &\equiv (k, \gamma_1(x), \gamma_2(x)) \in \mathcal{NC}[\sigma(S)] \end{aligned}$$

From  $(k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$  we know that exists  $v_1, v_2$  such as:

- $\gamma_1(x) = v_1$
- $\gamma_2(x) = v_2$
- $(k, v_1, v_2) \in \mathcal{NV}[\sigma(S)]$

Apply Lemma 81 (Related values are related terms) with  $(k, \gamma_1(x), \gamma_2(x)) \in \mathcal{NV}[\sigma(S)]$  to obtain  $(k, \gamma_1(x), \gamma_2(x)) \in \mathcal{NC}[\sigma(S)]$

□

## Compatibility-Prim

**Lemma 84** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility-Prim).

Let  $P = \Delta_b(\mathbf{b})$ .

Then  $\Delta; \Gamma \vdash b \approx b : P \triangleleft P$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash b : P \triangleleft P$ . Apply rule (TPrim)
2. Assuming arbitrary  $k, \rho, \gamma_1, \gamma_2$  such as:
  - $k \geq 0, \sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$\begin{aligned} (k, \sigma(\gamma_1(b)), \sigma(\gamma_2(b))) &\in \mathcal{NC}[\sigma(P \triangleleft P)] \\ &\equiv (k, b, b) \in \mathcal{NC}[P \triangleleft P] \end{aligned}$$

Apply Lemma 81 (Related values are related terms) with  $(k, b, b) \in \mathcal{NV}[P \triangleleft P]$  to obtain:

$$(k, b, b) \in \mathcal{NC}[P \triangleleft P]$$

□

## Compatibility Subsumption

**Lemma 85** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility-Subsumption).

Let  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$ . Let  $\Delta; \bullet \vdash S' <: S$ .

Then  $\Delta; \Gamma \vdash e_1 \approx e_2 : S$ .

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1 : S$  and  $\Delta; \Gamma \vdash e_2 : S$ . Apply rule (TSub) with  $\Delta; \Gamma \vdash e_i : S'$  (obtained from  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$ ) and  $\Delta; \bullet \vdash S' <: S$

2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as :

- $k \geq 0, \rho \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC}[\rho(S)].$$

Instantiate  $\Delta; \Gamma \vdash e_1 \approx e_2 : S'$  with  $k, \sigma, \gamma_1, \gamma_2$  to obtain:

$$(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC}[\sigma(S')].$$

Apply Lemma 76 (PER Subtyping) with  $\bullet; \bullet \vdash \sigma(S') <: \sigma(S)$ . Note that:

- $\bullet; \bullet \vdash \sigma(S') <: \sigma(S)$  follows from Lemma 72 (Type substitution preserves subtyping) applied to  $\sigma \in \mathcal{ND}[\Delta]$  and  $\Delta; \bullet \vdash S' <: S$ .

Hence,  $(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e_2))) \in \mathcal{NC}[\rho(S)]$ .

□

## Compatibility Object

**Lemma 86** ( $\text{Obj}_{\text{SEC}}^{\diamond}$  Compatibility-Object).

Let be  $S \triangleq O \triangleleft U$

Let be  $O \triangleq \mathbf{Obj}(\alpha). \left[ \overline{m : \langle X : A..B \rangle S' \rightarrow S''} \right]$

Then:

$$\frac{\text{msig}(\Delta, O, m_i) = \langle X : U_{li}..U_{ui} \rangle S'_i \rightarrow S''_i \quad \Delta, X : U_{li}..U_{ui}; \Gamma, z : S, x_i : S'_i \vdash e_i \approx e'_i : S''_i}{\Delta; \Gamma \vdash \left[ z : S \Rightarrow \overline{m(x)e} \right] \approx \left[ z : S \Rightarrow \overline{m(x)e'} \right] : S}$$

*Proof.* Denote  $o = \left[ z : S \Rightarrow \overline{m(x)e} \right]$  and  $o' = \left[ z : S \Rightarrow \overline{m(x)e'} \right]$

Proof obligations:

1.  $\Delta; \Gamma \vdash o : S \wedge \Delta; \Gamma \vdash o' : S$ . Apply rule (TObj)
2. Consider arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$\begin{aligned} & (k, \sigma(\gamma_1(o)), \sigma(\gamma_2(o'))) : \mathcal{NC}[\sigma(S)] \\ \equiv & (k, \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_1(e))} \right], \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_2(e'))} \right]) : \mathcal{NC}[\sigma(S)] \end{aligned}$$

Apply Lemma 81 (Related values are related terms) to transform the goal to

$$(k, \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_1(e))} \right], \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_2(e'))} \right]) : \mathcal{NV}[\sigma(S)]$$

Let us denote  $o_1 = \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_1(e))} \right], o_2 = \left[ z : \sigma(S) \Rightarrow \overline{m(x)\sigma(\gamma_2(e'))} \right]$

By well-formedness of the type  $S$  (Figure C.7) we know that  $\rho(U)$  is necessarily an object type (*i.e.* it is not a primitive type)

Proof of  $(k, o_1, o_2) \in \mathcal{NV}[\sigma(O) \triangleleft \sigma(U)]$ .

Sub goals

- $(k, o_1, o_2) \in \text{Atom}[\sigma(O)]$ . Apply Lemma 79 (Substitutions preserve simple typing) with  $\Delta; \Gamma \vdash o : S, \Delta; \Gamma \vdash o' : S, \sigma \in \mathcal{ND}[\Delta]$  and  $(k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$
- Assuming arbitrary  $m, j, T', v'_1, v'_2$  such as:

- $m \in \rho(U) \quad \text{msig}(\bullet, \sigma(O), m) = \langle X : \sigma(A).. \sigma(B) \rangle \sigma(S') \rightarrow \sigma(S'')$
- $j < k$
- $\vdash T' \wedge T' \in \sigma(A).. \sigma(B)$
- $(j, o_1, o_2) \in \mathcal{NV}[\sigma(S)]$
- $(j, v'_1, v'_2) \in \mathcal{NV}[\sigma(S')]$

**Show:**

$$(j, o_1.m \langle T' \rangle (v'_1), o_2.m \langle T' \rangle (v'_2)) \in \mathcal{NC}[\sigma(S'')]$$

Denote  $\text{methimpl}(o_1, m) = x.\sigma(\gamma_1(e))$  and  $\text{methimpl}(o_2, m) = x.\sigma(\gamma_2(e'))$

Then, the above goal rewrites to

$$(j, \sigma(\gamma_1(e)) [T'/X] [o_1/z] [v'_1/x], \sigma(\gamma_2(e')) [T'/X] [o_2/z] [v'_2/x]) \in \mathcal{NC}[\sigma(S'')]$$

$$(j, \sigma [X \mapsto T'] (\gamma_1(e)) [o_1/z] [v'_1/x], \sigma [X \mapsto T'] (\gamma_2(e')) [o_2/z] [v'_2/x]) \in \mathcal{NC}[\sigma(S'')]$$

Instantiate the second conjunct of the IH  $\Delta, X : A..B; \Gamma, z : S, x : S' \vdash e \approx e' : S''$  with

$j, \sigma' = \sigma [X \mapsto T'], \gamma'_1 = \gamma_1 [z \mapsto o_1] [x \mapsto v'_1], \gamma'_2 = \gamma_2 [z \mapsto o_2] [x \mapsto v'_2]$ . Note that:

- $j \geq 0$
- $\sigma [X \mapsto T'] \in \mathcal{ND}[\Delta, X : A..B]$ . It follows from:
  - \*  $\sigma \in \mathcal{ND}[\Delta]$ . It follows from above.
  - \*  $\Delta \vdash T' \in A..B$ . Apply Lemma 74 (Interval subtyping expansion) with  $\sigma \in \mathcal{ND}[\Delta]$  and  $\vdash T'$  and  $T' \in \sigma(A).. \sigma(B)$
- $(j, \gamma'_1, \gamma'_2) \in \mathcal{NG}[\sigma(\Gamma), z : \sigma(S), x : \sigma(S')]$ . It follows from:
  - \*  $(j, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$ . It follows from above.
  - \*  $(j, o_1, o_2) \in \mathcal{NV}[\sigma(S)]$ . It follows from the above.
  - \*  $(j, v'_1, v'_2) \in \mathcal{NV}[\sigma(S')]$ . It follows from the above.

Hence  $(j, \sigma [X \mapsto T'] (\gamma_1 [z \mapsto o_1] [x \mapsto v'_2] (e)), \sigma [X \mapsto T'] (\gamma_2 [z \mapsto o_2] [x \mapsto v'_2] (e'))) \in \mathcal{NC}[\sigma(S'')]$

Since  $o_1, o_2, v'_1, v'_2$  are closed values with respect to type variables we can rewrite this as:

$$(j, \sigma [X \mapsto T'] (\gamma_1(e)) [o_1/z] [v'_1/x], \sigma [X \mapsto T'] (\gamma_2(e')) [o_2/z] [v'_2/x]) \in \mathcal{NC}[\sigma(S'')]$$

□

## Compatibility Method Invocation Declassification

**Lemma 87** ( $\text{Ob}_{\text{SEC}}^\diamond$  Compatibility-Method-Invocation-Declassification).

Let  $S_1 \triangleq T_1 \triangleleft U_1, S_2 \triangleq T_2 \triangleleft U_2$

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T_1 \triangleleft U_1$

Let  $\Delta \vdash m \in U_1, \text{msig}(\Delta; U_1, m) = \langle X : A..B \rangle S_2 \rightarrow S$

Let  $\Delta \vdash U' \wedge \Delta \vdash U' \in A..B$

Let  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_2 [U'/X]$

Then  $\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) \approx e'_1.m \langle U' \rangle (e'_2) : S [U'/X]$

*Proof.* Let us denote  $e = e_1.m \langle U' \rangle (e_2)$  and  $e' = e'_1.m \langle U' \rangle (e'_2)$

Proof obligations:

1.  $\Delta; \Gamma \vdash e : S [U'/X]$  and  $\Delta; \Gamma \vdash e' : S [U'/X]$  which follow directly from the premises and the rule (TmD).
2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$\begin{aligned} & (k, \sigma(\gamma_1(e_1.m \langle U' \rangle (e_2))), \sigma(\gamma_2(e'_1.m \langle U' \rangle (e'_2)))) \in \mathcal{NC}[\sigma(S [U'/X])] \\ \equiv & (k, \sigma(\gamma_1(e_1)).m \langle \sigma(U') \rangle (\sigma(\gamma_1(e_2))), \sigma(\gamma_2(e'_1)).m \langle \sigma(U') \rangle (\sigma(\gamma_2(e'_2)))) \in \\ & \mathcal{NC}[\sigma(S [U'/X])] \end{aligned}$$

(because  $X \notin \text{dom}(\sigma)$ )

Instantiate the hypothesis  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T_1 \triangleleft U_1$  with  $k, \sigma, \gamma_1, \gamma_2$ , hence:

$$(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e'_1))) \in \mathcal{NC}[\sigma(T_1 \triangleleft U_1)]$$

Let  $k' \leq k$  and let  $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T_1 \triangleleft U_1)]$ . By Lemma 78 (Monadic bind) we can rewrite the goal **to show**

$$(k', v_1.m \langle \sigma(U') \rangle (\sigma(\gamma_1(e_2))), v_2.m \langle \sigma(U') \rangle (\sigma(\gamma_2(e'_2)))) \in \mathcal{NC}[S [U'/X]]$$

Instantiate the hypothesis  $\Delta; \Gamma \vdash e_2 \approx e'_2 : S_2 [U'/X]$  with  $k, \rho, \gamma_1, \gamma_2$ , hence:

$$(k, \sigma(\gamma_1(e_2)), \sigma(\gamma_2(e'_2))) \in \mathcal{NC}[\sigma(S_2 [U'/X])]$$

Let  $k'' \leq k'$  and let  $(k'', v''_1, v''_2) \in \mathcal{NV}[\sigma(S_2 [U'/X])]$ . By Lemma 78 (Monadic bind) we can rewrite the goal **to show**

$$(k'', v_1.m \langle \sigma(U') \rangle (v''_1), v_2.m \langle \sigma(U') \rangle (v''_2)) \in \mathcal{NC}[\sigma(S [U'/X])]$$

Then, we apply the Lemma 82 (Pre-compatibility: Method Invocation) with :

- $\sigma \in \mathcal{ND}[\Delta]$
- $\Delta \vdash T_1 \triangleleft U_1$
- $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T_1 \triangleleft U_1)]$
- $\Delta \vdash m \in U_1$  and  $\text{msig}(\Delta; U_1, m) = \langle X : A..B \rangle S_2 \rightarrow S$
- $\Delta \vdash U' \wedge \Delta \vdash U' \in A..B$
- $(k'', v''_1, v''_2) \in \mathcal{NV}[\sigma(S_2 [U'/X])]$

to obtain:  $(k'', v_1.m \langle \sigma(U') \rangle (v''_1), v_2.m \langle \sigma(U') \rangle (v''_2)) \in \mathcal{NC}[\sigma(S [U'/X])]$

□

## Compatibility-Method-Invocation-High

**Lemma 88** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility-Method-Invocation-High).

Let  $S_1 \triangleq T_1 \triangleleft U_1, S_2 \triangleq T_2 \triangleleft U_2$

If  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T_1 \triangleleft U_1$

$\Delta \vdash m \notin U_1, \text{msig}(\Delta; T_1, m) = \langle X : A..B \rangle S_2 \rightarrow T \triangleleft U$

$\Delta \vdash U' \wedge \Delta \vdash U' \in A..B$

$\Delta; \Gamma \vdash e_2 \approx e'_2 : S_2 [U'/X]$

then  $\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) \approx e'_1.m \langle U' \rangle (e'_2) : T [U'/X] \triangleleft \top$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : T [U'/X] \triangleleft \top$  and  $\Delta; \Gamma \vdash e'_1.m \langle U' \rangle (e'_2) : T [U'/X] \triangleleft \top$ . Apply rule (TmH).
2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$(k, \sigma(\gamma_1(e_1.m \langle U' \rangle (e_2))), \sigma(\gamma_2(e'_1.m \langle U' \rangle (e'_2)))) \in \mathcal{NC}[\sigma(T [U'/X]) \triangleleft \top]$$

Apply Lemma 80 (Well-typed terms are related at top) with:

- $\Delta; \Gamma \vdash e_1.m \langle U' \rangle (e_2) : T [U'/X] \triangleleft \top$  and  $\Delta; \Gamma \vdash e'_1.m \langle U' \rangle (e'_2) : T [U'/X] \triangleleft \top$ . It follows from above.
- $\sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$ . It follows from above.

Hence,  $(k, \sigma(\gamma_1(e_1.m \langle U' \rangle (e_2))), \sigma(\gamma_2(e'_1.m \langle U' \rangle (e'_2)))) \in \mathcal{NC}[\sigma(T [U'/X]) \triangleleft \top]$

□

## Compatibility TPmD

**Lemma 89** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility TPmD).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T \triangleleft U$

Let  $\Delta \vdash m \in U, \text{msig}(\Delta, U, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$

Let  $\Delta; \Gamma \vdash e_2 \approx e'_2 : P_1 \triangleleft U_1$

Let  $\text{retlab}(P_1 \triangleleft U_1, P_2) = P'_2$

Then  $\Delta; \Gamma \vdash e_1.m(e_2) \approx e'_1.m(e'_2) : P_2 \triangleleft P'_2$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft P'_2$  and  $\Delta; \Gamma \vdash e'_1.m(e'_2) : P_2 \triangleleft P'_2$ . Apply rule (TPmD).
2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[\Delta], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[\sigma(\Gamma)]$

**Show:**

$$\begin{aligned} & (k, \sigma(\gamma_1(e_1.m(e_2))), \sigma(\gamma_2(e'_1.m(e'_2)))) \in \mathcal{NC}[\sigma(P_2 \triangleleft P'_2)] \\ & \equiv (k, \sigma(\gamma_1(e_1)).m(\sigma(\gamma_1(e_2))), \sigma(\gamma_2(e'_1)).m(\sigma(\gamma_2(e'_2)))) \in \mathcal{NC}[\sigma(P_2 \triangleleft P'_2)] \\ & \equiv (k, \sigma(\gamma_1(e_1)).m(\sigma(\gamma_1(e_2))), \sigma(\gamma_2(e'_1)).m(\sigma(\gamma_2(e'_2)))) \in \mathcal{NC}[\sigma(P_2 \triangleleft P'_2)] \end{aligned}$$

Instantiate the hypothesis  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T \triangleleft U$  with  $k, \sigma, \gamma_1, \gamma_2$ , hence:

$$(k, \sigma(\gamma_1(e_1)), \sigma(\gamma_2(e'_1))) \in \mathcal{NC}[\sigma(T \triangleleft U)]$$

Let  $k' \leq k$  and let  $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T \triangleleft U)]$ . By Lemma 78 (Monadic bind) we can rewrite the goal **to show**

$$(k', v_1.m(\sigma(\gamma_1(e_2))), v_2.m(\sigma(\gamma_2(e'_2)))) \in \mathcal{NC}[[P_2 \triangleleft P'_2]]$$

Instantiate the hypothesis  $\Delta; \Gamma \vdash e_2 \approx e'_2 : P_1 \triangleleft U_1$  with  $k, \rho, \gamma_1, \gamma_2$ , hence:

$$(k, \sigma(\gamma_1(e_2)), \sigma(\gamma_2(e'_2))) \in \mathcal{NC}[[\sigma(P_1 \triangleleft U_1)]]$$

Let  $k'' \leq k'$  and let  $(k'', v''_1, v''_2) \in \mathcal{NV}[\sigma(P_1 \triangleleft U_1)]$ . By Lemma 78 (Monadic bind) we can rewrite the goal **to show**

$$(k'', v_1.m(v''_1), v_2.m(v''_2)) \in \mathcal{NC}[[P_2 \triangleleft P'_2]]$$

Instantiate  $(k', v_1, v_2) \in \mathcal{NV}[\sigma(T \triangleleft U)]$  with  $m, k'', v''_1, v''_2$ . Note that:

- $m \in \sigma(U)$ . It follows from  $\Delta \vdash m \in U$ ,  $\sigma \in \mathcal{ND}[[\Delta]]$  and  $\Delta \vdash T \triangleleft U$ . Then  $\text{msig}(\bullet, \sigma(T_1), m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$ .
- $k'' < k$  which follows above assumptions.
- $(k'', v''_1, v''_2) \in \mathcal{NV}[[P_1 \triangleleft \sigma(U_1)]]$ . Apply Lemma 75 (Downward closed/Monotonicity) with  $(k, v''_1, v''_2) \in \mathcal{NV}[[P_1 \triangleleft \sigma(U_1)]]$  and  $k'' \leq k$

Hence,  $(k'', v_1.m(v''_1), v_2.m(v''_2)) \in \mathcal{NC}[[P_2 \triangleleft \text{retlab}(P_1 \triangleleft \rho(U_1), P_2)]]$

Then, note that  $\text{retlab}(P_1 \triangleleft \rho(U_1), P_2) = \text{retlab}(P_1 \triangleleft U_1, P_2) = P'_2$ .

Hence,  $(k'', v_1.m(v''_1), v_2.m(v''_2)) \in \mathcal{NC}[[P_2 \triangleleft P'_2]]$

□

## Compatibility TPmH

**Lemma 90** ( $\text{Ob}_{\text{SEC}}^{\diamond}$  Compatibility TPmH).

Let  $\Delta; \Gamma \vdash e_1 \approx e'_1 : T \triangleleft U$

Let  $\Delta \vdash m \in T$ ,  $\text{msig}(\Delta, T, m) = P_1 \triangleleft * \rightarrow P_2 \triangleleft *$

Let  $\Delta; \Gamma \vdash e_2 \approx e'_2 : P_1 \triangleleft U_1$

Then  $\Delta; \Gamma \vdash e_1.m(e_2) \approx e'_1.m(e'_2) : P_2 \triangleleft \top$

*Proof.* Proof obligations:

1.  $\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft \top$  and  $\Delta; \Gamma \vdash e'_1.m(e'_2) : P_2 \triangleleft \top$ . Apply rule (TPmH).
2. Assuming arbitrary  $k, \sigma, \gamma_1, \gamma_2$  such as:  $k \geq 0, \sigma \in \mathcal{ND}[[\Delta]], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[[\sigma(\Gamma)]]$

**Show:**

$$(k, \sigma(\gamma_1(e_1.m(e_2))), \sigma(\gamma_2(e'_1.m(e'_2)))) \in \mathcal{NC}[[\sigma(P_2) \triangleleft \top]]$$

Apply Lemma 80 (Well-typed terms are related at top) with:

- $\Delta; \Gamma \vdash e_1.m(e_2) : P_2 \triangleleft \top$  and  $\Delta; \Gamma \vdash e'_1.m(e'_2) : P_2 \triangleleft \top$ . It follows from above
- $\sigma \in \mathcal{ND}[[\Delta]], (k, \gamma_1, \gamma_2) \in \mathcal{NG}[[\sigma(\Gamma)]]$ . It follows from above.

Hence,  $(k, \sigma(\gamma_1(e_1.m(e_2))), \sigma(\gamma_2(e'_1.m(e'_2)))) \in \mathcal{NC}[[\sigma(P_2) \triangleleft \top]]$

□

## Fundamental property

**Theorem 16** (Fundamental property).

$$\Delta; \Gamma \vdash e : S \implies \Delta; \Gamma \vdash e \approx e : S$$

*Proof.* The proof is by induction on the typing derivation of  $\Delta; \Gamma \vdash e : S$ .

Each case follows directly from the corresponding compatibility lemma (Lemmas 83 ... 90)  $\square$

**Lemma 15** (Self logical relation implies PRNI).

$$\Delta; \Gamma \vdash e \approx e : S \implies \text{PRNI}(\Delta, \Gamma, e, S)$$

*Proof.* The proof is direct given the similarity of both definitions. The only difference between both definitions is that  $\Delta, \Gamma \vdash e \approx e : S$  uses the security type system (Figure 4.4) and  $\text{PRNI}(\Delta, \Gamma, e, S)$  uses the simple type system (Figure C.11). We use Lemma 67 (Well-type programs are simple well-typed) to show that  $\Delta; \Gamma \vdash e : T \triangleleft U \implies \Gamma \vdash_1 e : T$   $\square$