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

COUNTABLE POLYMORPHIC MAY-MUST EFFECTS

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

MEMORIA PARA OPTAR AL TITULO DE INGENIERO CIVIL EN COMPUTACION

FABIÁN ANDRÉS MOSSO CHÁVEZ

PROFESOR GUÍA:
ÉRIC TANTER

PROFESOR CO-GUÍA:
MATÍAS TORO IPINZA

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
NANCY HIRSCHFELD KAHLER
MARCELO MENDOZA ROCHA

Este trabajo ha sido parcialmente financiado por el Proyecto FONDECYT Regular 1150017

SANTIAGO DE CHILE
2020

Resumen

Muchos programas realizan operaciones tales como mutar el heap, imprimir en pantalla, leer archivos, comunicarse con otros programas o generar números aleatorios. Estas interacciones son conocidas como efectos colaterales. Los efectos son útiles, pero pueden ser problemáticos: un programa podría sobre-escribir un archivo importante, o un programa podría contener *logs* de depuración en la etapa de producción, exponiendo información interna. Por lo tanto, existen técnicas como los sistemas de tipos y efectos, que nos permiten razonar estáticamente sobre programas con efectos.

En esta tesis, proponemos un cálculo formal con un sistema de efectos que permite a los programadores razonar sobre efectos que podrían y que deben suceder cuando un programa se ejecuta. El sistema de efectos diferencia entre efectos que podrían suceder y efectos que deben suceder. Adicionalmente el sistema de efectos nos permite restringir el número de veces que un efecto puede o debe realizarse durante la ejecución de un programa. El lenguaje presentado en esta tesis es llamado λ^{p}_{mme} , donde *mme* representa a los efectos opcionales y a los efectos obligatorios (*may-must effects*), y p representa efectos polimórficos, efectos genéricos que son representado por variables de efecto, y estas variables pueden ser reemplazadas por efectos concretos.

El trabajo de esta tesis consiste en dos partes. En la primera parte, definimos λ^{p}_{mme} , un cálculo formal con un sistema de tipo y efectos que combina tanto los efectos que podrían realizarse como los efectos que deben realizarse, y polimorfismo de efectos. La novedad de este trabajo consiste en la combinación de estos conceptos. En la segunda parte, desarrollamos el formalismo de λ^{p}_{mme} junto con la definición y prueba de la propiedad de coherencia (*soundness*) de su sistema de tipo y efectos. La propiedad de coherencia declara que la semántica estática de λ^{p}_{mme} es una aproximación acertada de la semántica dinámica, es decir, una función bien tipada respeta la especificación de efectos (sus privilegios y obligaciones).

El trabajo realizado se válida mediante la definición y demostración de la propiedad de coherencia. Esta propiedad indica que el lenguaje propuesto tiene sentido, que el lenguaje indica los efectos que podrían y que deben suceder cuando un programa bien tipado se ejecuta.

El sistema de tipos y efectos de λ^{p}_{mme} , le permite al programador diseñar y cumplir especificaciones mas fuertes de las que son posibles expresar en un sistema de efectos simple. A pesar que λ^{p}_{mme} es mas expresivo que un sistema de efectos simple, aun no hay una implementación del lenguaje. λ^{p}_{mme} tiene limitantes, tal como la falta de recursión, o tipos dependientes. Por lo tanto, λ^{p}_{mme} aun no seria útil para un programador.

Abstract

Many programs perform effectful operations such as mutating the heap, printing to the screen, reading files, communicating with other programs or generating random numbers. These interactions are known as side effects. Effects are useful but they can be problematic: a program may overwrite an important file, or a program may contain debug logs in the production stage, exposing internal information, however, there are techniques that allow us to statically reason about effectful programs, such as type and effect systems.

In this thesis we propose a core language and its effect system, which allows programmers to reason about effects that may and *must* happen when a program executes. The effect system differentiates effects that may happen from effects that must happen. Additionally the effect system allows us to restrict the number of times an effect may or must be performed during the execution of a program. The core language presented in this thesis is called λ^{p}_{mme}, where *mme* stands for may-must effects, and p stands for effect polymorphism.

The work of this thesis consists in two parts. The first part defines λ^{p}_{mme}, a core language with a type and effect system that combines both effect permissions and obligations, and effect polymorphism. The novelty of this work is the combination of these concepts. The second part develops the formalism of λ^{p}_{mme} along with the definition and proof of its type-and-effect soundness property. The soundness property states that the static semantics of λ^{p}_{mme} are a sound approximation of its dynamic semantics, i.e. a well-typed function respects its effect specification (privileges and obligations).

λ^{p}_{mme} is validated by defining and proving the soundness property. This property shows that the proposed language makes sense, that the language indicates correctly the effects that could and should happen when a well-typed program executes.

The type-and-effect system of λ^{p}_{mme} allows the programmer to design and meet stronger specifications than can be expressed in a simple type-and-effect system. Although, the type-and-effect system of λ^{p}_{mme} is more expressive than a simple type and effect system, but λ^{p}_{mme} does not have an implementation yet. λ^{p}_{mme} has limitations like lack of recursion, or dependent types. λ^{p}_{mme} would not be useful for a programmer.

Contents

List of Figures

Chapter 1

Introduction

Many programs perform effectful operations such as mutating the heap, printing to the screen, reading files, communicating with other programs or generating random numbers. These interactions are known as side effects. Effects are useful, but they can be problematic: a program may overwrite an important file, or a program may contain debug logs in the production stage, exposing internal information. On the other hand, programs that do not have side effects, named *pure* programs, benefit from satisfying the referential transparency property: the result of a program only depends on its inputs. Although effects are useful, they may be problematic. There are techniques, however, that allow us to statically reason about effectful programs such as type and effect systems.

A type is a classification of the kind of value that a program computes. Type systems are tools for statically reasoning about the behavior of programs; they allow us to reason about the type of a program and they ensure that a well-typed program does not get stuck. Effect systems are an extension of type systems allowing static reasoning about effects, thus providing stronger static guarantees than type systems. For example, an effect system could ensure that a function is pure, allowing us to execute that function in parallel without data races. *Classic* effect systems [17, 21, 18] only allow to specify the effects that *may* happen during the execution of a program. For instance, in Java; any function that *may* throw an exception has to be annotated. Thus an effect system gives us the guarantee that all exceptions that may happen in the program are captured. A limitation of classic effect systems is that they can only give guarantees about the absence of effects, not their presence.

A running example. Consider a procedure saveName that takes as argument a path and a name (a name is represented by a pair of strings, which represent the first and last name), and writes the name into the file. How do we know that saveName behaves properly? There are many verification techniques that could be used to ensure this requirement, like model checking [9] or certified programming [8]. Although type systems are less expressive, than these techniques, they are simpler to use for a programmer: a programmer only has to insert type annotations and the type system does the rest. In this example, one possible signature for saveName is:

$$\mathsf{saveName} : \mathsf{String} \times \mathsf{Name} \to \mathsf{Unit}$$

meaning that it receives as arguments a string of type String, a name of type Name and returns a unit of type Unit (returning unit is equivalent to returning void is C like languages). By just looking at the type of saveName, we can only reason about the input and output of the function; but not about the effectful operations of the function when it is applied.

An effect system allows us to reason about the effectful operations that happen when saveName is applied. For instance, a type signature extended with effect annotations for saveName may be:

$$\mathsf{saveName} : \mathsf{String} \times \mathsf{Name} \xrightarrow{\{\, open, write, close \,\}} \mathsf{Unit}$$

here the new annotation over the arrow represents the effect set that may happen when saveName is applied. *open*, *write* and *close* represent effects for the actions of opening, writing and closing a file respectively, i.e. when saveName is applied it is allowed to open, write or close a file. From the signature, we do not know if the function actually closes the file, or the amount of times that saveName opens, writes, or if it closes the file. We are interested in an effect system that is capable of ensuring statically that:

1. saveName opens the file exactly once.
2. saveName writes exactly two times, one for the first name and another for the last name.
3. saveName closes the file exactly once.

This small specification is not the strongest one that we could possible imagine. For instance, a stronger specification may require that saveName check the existence of the file before opening it, or to check that the name that is written in the file comes from the argument. These expressive specifications are out of scope of this thesis.

The hypothesis of this thesis is that it is possible to create a type-and-effect system capable of ensuring before executing a program, that the same program satisfies requirements such as those presented with saveName, i.e. that a program may or should perform an action and the number of times that action is performed. Our goal is only design and validate that a type-and-effect system with obligation is possible using as base a type-and-effect system that already allows to reason about the may effects.

In this thesis we propose a core language and its effect system that allows programmers to reason about effects that may and *must* happen when a program executes. The effect system differentiates effects that may happen from effects that must happen. Additionally the effect system allows us to restrict the number of times an effect may or must be performed during the execution of a program.

The core language presented in this thesis is called λ^{p}_{mme} ; where *mme* stands for may-must effect and p stands for polymorphic effects [1]. A may-must effect has the following structure:

$$\varepsilon(n_1, n_2)$$

[1]The name may-must is a reference to may-must analysis, a technique from the area of program analysis [20].

where ε is the name of the effect, n_2 is the quantity of privileges, and n_1 is the quantity of obligations. The effect system of λ_{mme} allows us to specify saveName as:

$$\mathsf{saveName} : \{\, \mathit{open}(1,1), \mathit{write}(2,2), \mathit{close}(1,1) \,\}\, \mathsf{String} \times \mathsf{Name} \to \mathsf{Unit}\, \{\ \}$$

The set on the left, named input effects, indicates the privileges and obligations that the function requires in order to be applied, and the set on the right, named output effects, indicates the privileges and obligations that the function produces when it is applied. This is a different interpretation of effect that we presented previously; previously the effect annotations represented the effects that may happen during execution. In the signature above, saveName requires the privilege and the obligation of opening and closing a file exactly once; and writing exactly two times. The empty output effect set means that the function does not produce any remaining privileges or obligations. The type system ensures from the type that the specification defined previously is satisfied. Thus with λ^{p}_{mme} it is possible to write more expressive specifications than a classic effect system, this means that functions can have effect as parameters.

Opening a file, performing an action (like: writing, reading, updating, and deleting) with a name and closing the file is a common pattern. We want to write a generic function performName that receives as argument an effectful procedure, opens a file, performs the action received as argument and closes the file. In order to be able to write performName, λ^{p}_{mme} supports effect polymorphism [17]. Thus, the type of performName is:

$$\mathsf{performName} :< \alpha > \{\, \mathit{open}(1,1), \alpha, \mathit{close}(1,1) \,\}\, (\{\, \alpha \,\}\, \mathsf{Name} \to \mathsf{Unit}\, \{\ \}) \times \mathsf{String} \times \mathsf{Name} \to \mathsf{Unit}\, \{\ \}$$

where α represents an effect variable that has to be instantiated in order to apply the function. For example, we can define saveName using performName by replacing α for $\mathit{write}(2,2)$ and applying a procedure that allows to write.

To summarize, the contributions of this thesis are:

- We define λ^{p}_{mme}, a core language with a type and effect system that combines both effect permissions and obligations, and effect polymorphism. The novelty of this work is the combination of these concepts. The type and effect system of λ^{p}_{mme} gives us stronger guarantees than a standard type and effect system.
- We develop the formalism of λ^{p}_{mme} along with the definition and proof of its type-and-effect soundness property. The soundness property states that the static semantics of λ^{p}_{mme} are a sound approximation of its dynamic semantics, i.e. a well-typed function respects its effect specification (privileges and obligations).

The document is organized as follows: Chapter 2 introduces the background, where we specify the syntax and typing rules of a simple functional language (SFL) and we extend that language with effects. Chapter 3 presents λ_{mme}, a language with only may must effects: the static semantics (syntax, typing rules), dynamic semantics, and its properties. Chapter 4 presents λ^{p}_{mme}, an extension of λ_{mme} with effect polymorphism. Chapter 5 discusses other effect systems and compares them with λ_{mme}. Chapter 6 presents the conclusions and future work. Appendix A contains the complete proof of type-and-effect soundness for λ^{p}_{mme}.

Chapter 2

Background

This chapter provides the minimal background necessary for understanding the remaining chapters. First, we introduce a simple functional language (SFL); we provide its static semantics, i.e. syntax and typing rules, also we provide its dynamics semantics and we define its soundness property. Second, we illustrate why using a simple type system is not enough for the specification of the saveName example presented in the previous chapter. Third, we present SFLE, an extension of SFL with effects. We start by showing the supported effects of SFLE along its statics and dynamic semantics, and finally we discuss soundness for effects systems. Finally, we show the level of specification that SFLE accomplishes for saveName.

2.1 Type Systems

Type systems are described by a set of rules that classify expressions of a program by the kind of value they produce. Type systems verify that all expressions of a program have a proper type. For example, the expression $(t + 1)$ is only valid if t has a numerical type. The objective of a type system is to ensure statically that *some errors* do not occur when a well-typed program executes. For example, a simple type system prevents the user from applying a boolean value as if it were a function. A more expressive type system could reject a program that apply the function first to an empty list [1]. Furthermore a more expressive type system (for instance one with full dependent types) can express correctness properties in types [8].

In the next sections we provide the static and dynamic semantics of SFL and its soundness property. Also, we show that SFL cannot satisfy a rich enough specification of saveName.

[1] first is a function that given a list returns the first element. It typically throws a runtime error if given an empty list.

$$
\begin{array}{rcll}
 & T \in \textsc{Type}, \quad t \in \textsc{Term}, \quad x \in \mathrm{Var}, \quad n \in \mathsf{Nat}, \quad \mathsf{op} \in \mathrm{OP}, & & \\
 & \{\, \mathsf{Bool}, \mathsf{Nat}, \mathsf{Unit}, \ldots \,\} \in B, \quad \{\, \mathsf{true}, \mathsf{false}, n, \mathsf{unit}, \ldots \,\} \in b & & \\
T & ::= & B \mid T \times T \mid T \to T & \text{(types)} \\
v & ::= & b \mid \{\, v, v \,\} \mid (\lambda x : T \,.\, t) \mid \mathsf{op} & \text{(values)} \\
t & ::= & v \mid x \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid \{\, t, t \,\} \mid t.1 \mid t.2 & \text{(terms)} \\
E & ::= & \Box \mid \Box\ t \mid v\ \Box \mid \mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t & \\
 & & \mid \{\, \Box, t \,\} \mid \{\, v, \Box \,\} \mid \Box.1 \mid \Box.2 & \text{(contexts)} \\
\Gamma & ::= & \cdot \mid \Gamma, (x : T) & \text{(type context)}
\end{array}
$$

Figure 2.1: Syntax of SFL

2.1.1 Syntax and Static Semantics

Let us consider a simple functional language (SFL). The syntax of SFL is defined in Figure 2.1. The language is just the simply typed lambda calculus (STLC) extended with base types, base values, operations, pairs, and if. We use T as a metavariable that represents types, t represents terms, x represents variables, n represents natural numbers, and op represents primitive operations. The set of operations OP, base types B, and base values b are open sets, sets that does not contain its boundary elements.

As shown in Figure 2.1, a type T may be a base type B, a pair $T \times T$, or a function $T \to T$. All functions and primitive operators are modeled as unary. For example, the type $\mathsf{Bool} \to (\mathsf{Nat} \to \mathsf{Nat})$ is the type of a function that expects a boolean, and returns a function from number to number. Base types B include booleans Bool, natural numbers Nat, and Unit. A value v may be a base value b, a pair $\{\, v, v \,\}$, a lambda abstraction $(\lambda x : T \,.\, t)$, or a primitive operation op. Base values b include booleans true and false, natural numbers n, and unit unit. A term t may be a value v, a variable x, a pair $\{\, t, t \,\}$, a projection of the first element $t.1$, a projection of the second element $t.2$, a function application $t\ t$, or a conditional $\mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$. A context E may be a hole $\Box$, an application with a hole in function position $\Box\ t$, or an application with a hole in argument position $v\ \Box$, or if expression with a hole in the conditional position $\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t$, or pair with a hole at the first expression $\{\, \Box, t \,\}$, or a pair with a hole at the second position $\{\, v, \Box \,\}$, or projection of the first element with a hole $\Box.1$, or projection of the second element with a hole $\Box.2$.

Figure 2.2 defines the typing rules for SFL, these rules are the standard for the simple typed lambda calculus [23]. The typing judgment has the form $\Gamma \vdash t : T$, which represents that term t has type T, under type context Γ, where Γ is a finite mapping from variables to types. Where the symbol $\vdash$ reads as 'we can derive'. For example, $A \vdash B$ means 'from A we can derive B'. The typing rules are defined inductively by inference rules [2].

Rule (Tx) assigns a variable the type that is associated to it in the context Γ. Rule (Tb) assigns to b a type defined by the function μ, where the function is a finite mapping defined in Figure 2.3 that associates base values with base Types. Rule (Tλ) types a function with type $T_1 \to T_2$ if the body of the function has type T_2 under the context Γ extended with the

[2] Inference rules state that given the premises (statements above the line) are true, then the conclusion (below the line) is true.

$$\boxed{\Gamma \vdash t : T}$$

$$(\text{Tx})\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad (\text{Tb})\frac{\mu(b) = B}{\Gamma \vdash b : B} \qquad (\text{T}\lambda)\frac{\Gamma, (x : T_1) \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1 \,.\, t) : T_1 \to T_2}$$

$$(\text{Top})\frac{\zeta(\mathsf{op}) = T}{\Gamma \vdash \mathsf{op} : T} \qquad (\text{Tpair})\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{\, t_1, t_2 \,\} : T_1 \times T_2} \qquad (\text{Tproj1})\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.1 : T_1}$$

$$(\text{Tproj2})\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.2 : T_2} \qquad (\text{Tapp})\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$(\text{Tif})\frac{\Gamma \vdash t_1 : \mathsf{Bool} \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

Figure 2.2: Type rules for SFL

$$\begin{aligned} \mu : b &\to B \\ \mu(n) &= \mathsf{Nat} \\ \mu(\mathsf{true}) &= \mathsf{Bool} \\ \mu(\mathsf{false}) &= \mathsf{Bool} \\ \mu(\mathsf{unit}) &= \mathsf{Unit} \end{aligned}$$

Figure 2.3: Base values mapped to a base type

variable x mapped to type T_1. Rule (Top) assigns to op a type given by the function ζ, where ζ is a mapping from primitive operations to types. Rule (Tif) enforces that the conditional t_1 has type Bool, and both branches have the same type T.

For instance consider, the following type derivation:

$$\text{Tapp}\frac{\text{T}\lambda\ \dfrac{\text{Tvar}\ \dfrac{}{x : \mathsf{Nat} \vdash x : \mathsf{Nat}}}{\cdot \vdash (\lambda x : \mathsf{Nat} \,.\, x) : \mathsf{Nat} \to \mathsf{Nat}} \qquad \text{Tb}\ \dfrac{\mu(3) = \mathsf{Nat}}{\cdot \vdash 3 : \mathsf{Nat}}}{\cdot \vdash (\lambda x : \mathsf{Nat} \,.\, x)\ 3 : \mathsf{Nat}}$$

This typing derivation shows an application of the function $(\lambda x : \mathsf{Nat} \,.\, x)$ to 3, in an empty context. Thus this expression is well-typed [3].

2.1.2 Dynamic Semantics

The dynamics semantics of SFL are defined by Figure 2.4 and 2.5. Where Figure 2.4, called contextual reduction, describes the evaluation of any program from SFL and the reduction is defined from the notion of computation $\to$. The evaluation judgment has the form $t \mapsto t'$, which represents that term t evaluates to t' in one step. Like type rules, the dynamic

[3] A term is well-typed if the type of the expression can be derived by the typing rules.

$$\text{(Context)}\frac{t \to t'}{E[t] \mapsto E[t']}$$

Figure 2.4: Contextual reduction

$$\boxed{t \to t'}$$

$$\text{(E-Proj1)}\frac{}{\{\, v_1, v_2 \,\}.1 \to v_1} \qquad \text{(E-Proj2)}\frac{}{\{\, v_1, v_2 \,\}.2 \to v_2} \qquad \text{(E-Op)}\frac{\mathsf{op}\ v_1 \to_\zeta v_2}{\mathsf{op}\ v_1 \to v_2}$$

$$\text{(E-App)}\frac{}{(\lambda x : T_1\ .\ t)\ v \to t[x \to v]} \qquad \text{(E-IfT)}\frac{}{\mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f \to t_t}$$

$$\text{(E-IfF)}\frac{}{\mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f \to t_f}$$

Figure 2.5: Notions of reduction

semantics are defined inductively by inference rules. Rule (Context) defines that if a term t evaluates to t' by the notion of reduction $\to$, then the same term inserted in any context can also perform a step to t' inserted in the same context.

Figure 2.5 defines the notions of reduction $\to$, that describes *interesting* computations. Rule (E-Proj1) reduces pair $\{\, v_1, v_2 \,\}$ by extracting the first element v_1. Similarly, rule (E-Proj2) reduces pair $\{\, v_1, v_2 \,\}$ by extracting the second element v_2. Rule (E-Op) reduces the expression $\mathsf{op}\ v_1$ to v_2 by applying the relation $\to_\zeta$, where the relation relates an operation op, with an input v_1, and its correspond output v_2. Rule (E-App) reduces the function application $(\lambda x : T.t)\ v$ by applying the substitution $t[x \to v]$ [4]. Rule (E-IfT) reduces the expression $\mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f$ by selecting the then branch t_t. Similarly to the previous rule, rule (E-IfF) reduces the expression $\mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f$ by selecting the else branch t_f.

2.1.3 Soundness

A sound type system is one that guarantee that a well-typed program fulfill certain property. Where in simple type systems, like the one that we presented with SFL, we are interested that well type programs do not get stuck, the property is called type safety. Formally it is defined as follows:

Theorem 2.1 (Type Soundness)**.**

$$\text{If}\ \ \cdot \vdash t : T\ \ \text{and}\ \ t \mapsto^* v\ \ \text{then}\ \mathsf{value}(v)\ \text{or}\ t \mapsto t'$$

Where $\mapsto^*$ is the transitive closure of the small-step evaluation $\mapsto$, i.e. $t \mapsto^* v$ means that it takes t zero or more steps to reach v. If a term t is well typed, then either it is going to

[4] The substitution means that we replace any free occurrence of x on t by v

$$
\begin{aligned}
\zeta : \text{OP} &\rightarrow \textsc{Type} \\
\zeta(\mathsf{getFile}) &= \mathsf{String} \rightarrow \mathsf{File} \\
\zeta(\mathsf{open}) &= \mathsf{Unit} \rightarrow \mathsf{Unit} \\
\zeta(\mathsf{writeName}) &= \mathsf{Name} \rightarrow \mathsf{Unit} \\
\zeta(\mathsf{close}) &= \mathsf{Unit} \rightarrow \mathsf{Unit}
\end{aligned}
$$

Figure 2.6: Operations mapped to a type

$$
\begin{aligned}
\rightarrow_\zeta : \text{OP} \times v &\rightarrow v \\
(\mathsf{getFile}\ s) &\rightarrow_\zeta f \\
(\mathsf{open}\ \mathsf{unit}) &\rightarrow_\zeta \mathsf{unit} \\
(\mathsf{writeName}\ n) &\rightarrow_\zeta \mathsf{unit} \\
(\mathsf{close}\ \mathsf{unit}) &\rightarrow_\zeta \mathsf{unit}
\end{aligned}
$$

Figure 2.7: Operations evaluated

reduce to a value v, or it can perform another step [5].

2.1.4 Using a Simple Type System for saveName

For illustration purposes, we extend SFL with base types File, String and Name, and we define operations as primitives from the I/O domain:

$$\{\, \mathsf{getFile}, \mathsf{open}, \mathsf{writeName}, \mathsf{close} \,\} \in \text{OP}$$

Figure 2.6 shows the formal definition of ζ for the operations that we defined previously. getFile receives as argument a String and returns a File, open receives a Unit and return Unit, writeName receives as arguments a Name and returns a Unit, and close receives a Unit and returns a Unit. Operations open, writeName, and close are elements of a file i.e. the operations open opens, writeName writes, and close closes the file that they belongs to. For example, let consider the next program with a syntax *à la* Scala:

```
1   val f : File = getFile("text.txt")
2   f.open()
3   f.close()
```

where at the first line, the program creates a file, the second one opens it and, the last one closes it.

Figure 2.7 shows the definition of $\rightarrow_\zeta$ for the operations that we defined previously. getFile and a string s evaluate to a file f, open and unit evaluate to unit, writeName and a name n evaluates to unit, and close and unit evaluate to unit.

The type system of SFL gives saveName the following type:

$$\mathsf{saveName} : \mathsf{String} \times \mathsf{Name} \rightarrow \mathsf{Unit}$$

[5] Note that any program that diverges, also it is a program that is sound, and that SFL cannot diverge. The simple lambda calculus cannot diverge [23], and SFL is just simple lambda calculus with operations.

$T \in \textsc{Type}, \quad t \in \textsc{Term}, \quad x \in \text{Var}, \quad n \in \mathsf{Nat}, \quad \varepsilon \in \text{Effects}, \quad \mathsf{op} \in \text{OP}$

T	::=	$B \mid T \times T \mid T_1 \xrightarrow{\Phi} T2$	(types)
B	::=	$\mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{Unit}$	(base types)
v	::=	$b \mid \{\, v, v \,\} \mid (\lambda x : T \,.\, t) \mid \mathsf{op}$	(values)
b	::=	$n \mid \mathsf{unit} \mid \mathsf{true} \mid \mathsf{false}$	(booleans)
t	::=	$v \mid x \mid \{\, t, t \,\} \mid t.1 \mid t.2 \mid t\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t$	(terms)
E	::=	$\square \mid \square\ t \mid v\ \square \mid \mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t$	
		$\mid \{\, \square, t \,\} \mid \{\, v, \square \,\} \mid \square.1 \mid \square.2$	(contexts)
Γ	::=	$\cdot \mid \Gamma, (x : T)$	(type context)
Φ	::=	$\cdot \mid \Phi, \varepsilon$	(effect context)

Figure 2.8: Syntax of SFLE

a function that receives a String, and a Name as arguments, and returns a Unit. As we discussed previously, simple type systems can only give guarantees about the input and the output of a function, and that a well typed function does not get stuck when applied. Here, we are interested in the behavior of saveName, not only in its end result. A simple type system cannot track effects. In this case the specification (saveName opens the file exactly once, writes two times, and closes the file exactly once) that we are interested in is about the side effects of saveName. Thus a type and effect system seems to be appropriate to use.

2.2 Type and Effect Systems

As discussed in Chapter 1, type and effects systems allow us to reason statically about side effects. We extend SFL with effects (SFLE) following an approach similar to the system presented by Lucassen and Gifford [17].

2.2.1 Syntax and Static Semantics

The syntax of SFLE is defined in Figure 2.8. SFLE extends the syntax of SFL by adding the metavariable ε that represents effects. Primitives operations can have effects as well. The effect context Φ is defined as a set of effects.

The function type $T_1 \xrightarrow{\Phi} T2$ is annotated with the set of latent effects Φ. The effect set represents the effects that the function *may* perform when it is applied, hence the name latent effects.

Figure 2.9 presents the typing rules of SFLE. The type judgment is defined as $\Gamma; \Phi \vdash t : T$ where the term t has type T, under type context Γ, and effect context Φ, which represents the effects that it may perform.

Rules for variables and base types are presented in Figure 2.2. Rule (Tλ) is similar to the rule from SFL, but now the body is typed with the effect set Φ_l that represents the latent

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

$$(\text{Tx})\frac{x : T \in \Gamma}{\Gamma; \Phi \vdash x : T} \qquad (\text{Tb})\frac{\mu(b) = B}{\Gamma; \Phi \vdash b : B} \qquad (\text{T}\lambda)\frac{\Gamma, (x : T_1); \Phi_l \vdash t : T_2}{\Gamma; \Phi \vdash (\lambda x : T_1 \,.\, t) : T_1 \xrightarrow{\Phi_l} T2}$$

$$(\text{Top})\frac{\zeta(\mathsf{op}) = T}{\Gamma; \Phi \vdash \mathsf{op} : T} \qquad (\text{Tpair})\frac{\Gamma; \Phi_1 \vdash t_1 : T_1 \qquad \Gamma; \Phi_2 \vdash t_2 : T_2}{\Gamma; \Phi_1 \cup \Phi_2 \vdash \{\, t_1, t_2 \,\} : T_1 \times T_2}$$

$$(\text{Tproj1})\frac{\Gamma; \Phi \vdash t : T_1 \times T_2}{\Gamma; \Phi \vdash t.1 : T_1} \qquad (\text{Tproj2})\frac{\Gamma; \Phi \vdash t : T_1 \times T_2}{\Gamma; \Phi \vdash t.2 : T_2}$$

$$(\text{Tapp})\frac{\Gamma; \Phi_1 \vdash t_1 : T_1 \xrightarrow{\Phi_l} T_2 \qquad \Gamma; \Phi_2 \vdash t_2 : T_1}{\Gamma; \Phi_1 \cup \Phi_2 \cup \Phi_l \vdash t_1\ t_2 : T_2}$$

$$(\text{Tif})\frac{\begin{array}{c}\Gamma; \Phi_1 \vdash t_1 : \mathsf{Bool} \\ \Gamma; \Phi_2 \vdash t_2 : T \qquad \Gamma; \Phi_3 \vdash t_3 : T\end{array}}{\Gamma; \Phi_1 \cup \Phi_2 \cup \Phi_3 \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

Figure 2.9: Type-and-effect system for SFLE

$$\begin{array}{rcl} \zeta : \text{OP} \to \textsc{Type} & & \\ \zeta(\mathsf{open}) & = & \mathsf{String} \xrightarrow{\{\, open \,\}} \mathsf{File} \\ \zeta(\mathsf{write}) & = & \mathsf{File} \times \mathsf{String} \xrightarrow{\{\, write \,\}} \mathsf{Unit} \\ \zeta(\mathsf{close}) & = & \mathsf{File} \xrightarrow{\{\, close \,\}} \mathsf{Unit} \end{array}$$

Figure 2.10: Operations mapped to a type

effects of the body of the function. Rule (Top) is as before. Rule (Tpair) assigns type $T_1 \times T_2$ to a pair, and the effect set of the pair are the union of the effect sets of its elements. Rule (Tproj1) assigns type T_1 to the first projection, and the effect set of the first projection is the effect set of the pair. Rule (Tproj2) assigns type T_2 to the second projection, and the effect set of the second projection, as the first projection, os the effect set of the pair. Rule (Tapp) assigns type T_2 if the term t_1 has type $T_1 \xrightarrow{\Phi_l} T_2$, the type t_2 has type T_1, and the effect context of an application must subsumes the effects sets of t_1, t_2, and the latent effects of t_1. Rule (Tif) assigns type T to a conditional, and the effect set of the conditional is the union of the effect sets of its sub-terms t_1, t_2, and t_3.

For instance, consider SFLE extended with types String, Name, File, and the operations from the I/O domain we presented previously (open, write, and close). We define the effects of that domain as:

$$\varepsilon \in \{\, open, write, close \,\}$$

Figure 2.10 defines ζ, where the operations are mapped to a type with effect annotations. ζ assigns to open the type of a function that receives as argument a String, may open a file, and returns File; write receives as arguments a File and a String, may write to a file, and returns Unit; and close receive as argument a File, may close the file, and return Unit.

$$\text{(Context)}\frac{\langle t, \Phi\rangle \to \langle t', \Phi'\rangle}{\langle E[t], \Phi\rangle \mapsto \langle E[t'], \Phi'\rangle}$$

Figure 2.11: Contextual reduction

The following type judgments show how the effect set in the context is used in order to type a function application:

$$\text{Tapp}\frac{\text{Top}\ \dfrac{\zeta(\mathsf{open}) = \mathsf{String} \xrightarrow{\{\, open \,\}} \mathsf{File}}{\cdot;\cdot \vdash \mathsf{open} : \mathsf{String} \xrightarrow{\{\, open \,\}} \mathsf{File}} \qquad \dfrac{}{\cdot;\cdot \vdash \text{`./main.txt'} : \mathsf{String}}}{\cdot;\{\, open \,\} \vdash \mathsf{open}(\text{`./main.txt'}) : \mathsf{File}}$$

The function open requires the effect set $\{\, open \,\}$ in order to be well typed. Trying to do the same application with an empty effect context does not type check.

2.2.2 Dynamic semantics

Figure 2.11 and 2.12 define the dynamic semantics for SFLE, where their semantics are similar to the dynamic semantics of SFL, that were presented at Figure 2.4 and Figure 2.5, with the difference that SFLE keeps track of the effects produced during execution. The evaluation judgment has the form $\langle t, \Phi\rangle \mapsto \langle t', \Phi'\rangle$, where the effects sets Φ and Φ' represent the effects that have been performed until that point of execution.

They are only two rules that are different from SFL (the other rules do not alter the effect set): rule (Context) defines that if, a pair composed by term t and effect set Φ reduces to the pair composed by term t' and effect set Φ' by the notion of reduction $\to$, then the pair composed by the same term inserted in any context and the effect set Φ, can also do a step to the pair composed by the term t' inserted in the same context and the effect set Φ'; and rule (E-Op) reduces the pair composed by expression op v_1 and effect set Φ by applying the relation op $v_1 \to_\zeta v_2$ and modifying the effect set Φ by adding the latent effects of op, Φ_l.

2.2.3 Soundness

Effect systems conservatively approximate the effect set that a program may perform during execution. Thus a sound effect system is one that does a sound approximation of the effects, i.e. a sound effect system guarantees that when a well type program executes, the predicted effect set contains at least all the effects that program performs [6]. This is called the effect soundness property and can be defined as follows:
Theorem 2.2 (Effect Soundness)**.**

$$\text{If}\ \ \cdot;\Phi_s \vdash t : T\ \ \text{and}\ \ \langle t, \cdot\rangle \mapsto^* \langle v, \Phi_{\mathrm{d}}\rangle\ \text{then}\ \Phi_{\mathrm{d}} \subseteq \Phi_s$$

[6] Usually this property is enforced by dynamic checks [18], but SFLE does not have dynamic checks.

$$\boxed{\langle t, \Phi\rangle \to \langle t', \Phi'\rangle}$$

$$\text{(E-Proj1)}\ \frac{}{\langle \{\, v_1, v_2 \,\}.1, \Phi\rangle \to \langle v_1, \Phi\rangle} \qquad \text{(E-Proj2)}\ \frac{}{\langle \{\, v_1, v_2 \,\}.2, \Phi\rangle \to \langle v_2, \Phi\rangle}$$

$$\text{(E-Op)}\ \frac{\zeta(\mathsf{op}) = T_1 \xrightarrow{\Phi_l} T_2 \quad \mathsf{op}\ v_1 \to_\zeta v_2}{\langle \mathsf{op}\ v_1, \Phi\rangle \to \langle v_2, \Phi \cup \Phi_l\rangle} \qquad \text{(E-App)}\ \frac{}{\langle (\lambda x : T_1\ .\ t)\ v, \Phi\rangle \to \langle t[x \to v], \Phi\rangle}$$

$$\text{(E-IfT)}\ \frac{}{\langle \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_t, \Phi\rangle} \qquad \text{(E-IfF)}\ \frac{}{\langle \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_f, \Phi\rangle}$$

Figure 2.12: Notions of reduction of SFLE

The effect soundness property of SFLE states that if a term t is well typed to an effect set Φ_s, and the term with an empty effect reduces to a pair of value v and effect set Φ_d, [7] then effect set Φ_s has more elements than effect set Φ_d.

2.2.4 Using a Type and Effect System for saveName

The type-and-effect system of SFLE assigns to saveName the following type:

$$\mathsf{saveName} : \mathsf{String} \times \mathsf{Name} \xrightarrow{\{\, open, write, close \,\}} \mathsf{Unit}$$

i.e. a function that receives a path to a file (as a string) and a name as arguments, and may perform the *open*, *write*, and *close* effects, returning unit. A possible implementation that type checks with a syntax *à la* Scala is as follows:

```
1 def saveName {open, write, close} (p: String, n: Name) : Unit ={
2     val f = getFile(p)
3     f.open()
4     f.write(n.firstName)
5 }
```

The effect system accepts this program although this program violates the specification: the function only writes the first name, and never closes the file. There is no way to enforce that an action must be performed a given number of times by using a simple type and effect system.

[7] Even though SFLE cannot diverge, because SFLE is SFL with effect annotation and SFL cannot diverge, the soundness definition is more general, any program that diverges fulfill the definition.

2.3 Summary

In this chapter, we provided the static and dynamics semantics, as well as the soundness property of SFL, a simple functional language. Then we showed that a type system is not enough for the specification defined in Chapter 1. We defined a simple functional language extended with effects SFLE, provided its static and dynamics semantics, and soundness property; we showed that with SFLE we can obtain stronger specifications than SFL, but strong not enough for the level of specification that we want. In the following chapter we present λ_{mme} whose effect system allows us to statically guarantee that a program meets the specification.

Chapter 3

Lambda May-Must

We present λ_{mme} whose type-and-effect system supports may-must effects. May-must effects are different from the effects presented in SFLE: they represent both the privileges and obligations that a program requires and produces. First, we present the static semantics in Section 3.1. Second, we provide the dynamic semantics in Section 3.2. Finally in Section 3.3, we present the soundness property of the language.

3.1 Static Semantics

In this section we present the static semantics of λ_{mme}. We present its syntax, effect sets, typing rules, well formedness, subtyping, and the minimum effect set ($\rightsquigarrow$) function, which calculates the minimum effect set to type check an expression.

3.1.1 Syntax

The syntax for λ_{mme} is presented in the Figure 3.1. The set Eff corresponds to the set of all side effects. A side effect $\varepsilon(n, n)$ is composed of a tag ε which represents the name of an effect (e.g. *write* represents the side effect of writing a file), and the number of privileges and obligations of the given effect. For instance, the effect $write(1, 3)$ represents the obligation to write at least once, and the privilege to write up to 3 times. These numbers belongs to the set Nat^{∞} which includes all natural numbers plus the infinite ∞ (and the 0).

A type T may be a base type B (examples of base types could be: Unit or Bool), a pair $T \times T$, or a function $T^{\Phi} \to T^{\Phi}$. A function type ${T_1}^{\Phi_1} \to {T_2}^{\Phi_2}$ denotes a function that receives as argument a term of type T_1, requires the effect set Φ_1 in order to be applied, returns a term of type T_2, and produces as output the effect set Φ_2. A raw value u may be a base value b, a pair $\{ u, u \}$, a lambda abstraction $(\lambda x : T \,.\, t)$, or an operation op. A base value b may be a boolean true or false, a natural number n, or unit. A value v is a raw value u ascribed to a type T. Without losing generality, all values are ascribed to simplify

$$T \in \textsc{Type}, \quad n \in \textsc{Nat}^{\infty}, \quad \varepsilon \in \textsc{Effects},$$
$$\varepsilon(n,n) \in \textsc{Eff}, \quad \mathrm{op} \in \mathrm{OP}, \quad b \in \{\, n, \mathsf{true}, \mathsf{false}, \mathsf{unit}... \,\}, \quad t \in \textsc{Term},$$
$$x \in \textsc{Var}, \quad B \in \textsc{BaseType}$$

T	$::=$	$B \mid T \times T \mid T^{\Phi} \to T^{\Phi}$	(types)
v	$::=$	$u :: T$	(values)
u	$::=$	$b \mid \{\, u, u \,\} \mid (\lambda x : T \,.\, t) \mid \mathrm{op}$	(raw values)
t	$::=$	$v \mid x \mid \{\, t, t \,\} \mid t.1 \mid t.2 \mid t\ v \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t :: T$	(terms)
E	$::=$	$\Box \mid \Box\ v \mid \{\, \Box, t \,\} \mid \{\, v, \Box \,\} \mid \Box.1 \mid \Box.2 \mid \mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t$	
	$\mid$	$\Box :: T$	(evaluation context)
Φ	$::=$	$\cdot \mid \Phi, \varepsilon(n,n)$	(effect set)
Γ	$::=$	$\cdot \mid \Gamma, (x : T)$	(type context)

Figure 3.1: Syntax

$$\Phi_1 \sqsubseteq \Phi_2 \stackrel{\text{def}}{=} (\Phi_1 \leq_p \Phi_2) \text{ and } (\Phi_2 \leq_o \Phi_1)$$

$$(\text{EmptyP})\frac{}{\{\,\} \leq_p \Phi} \qquad (\text{NumberP})\frac{\Phi_1 \leq_p \Phi_2 \qquad n_2 \leq n_2'}{\Phi_1, \varepsilon(n_1,n_2) \leq_p \Phi_2, \varepsilon(n_1',n_2')}$$

$$(\text{EmptyO})\frac{}{\{\,\} \leq_o \Phi} \qquad (\text{NumberP})\frac{\Phi_1 \leq_o \Phi_2 \qquad n_1' \leq n_1}{\Phi_1, \varepsilon(n_1,n_2) \leq_o \Phi_2, \varepsilon(n_1',n_2')}$$

Figure 3.2: Effect set containment and privilege and obligation ordering

the proofs of soundness. A term t may be a value v, a term variable x, an application $t\ t$, an if expression, a pair $\{\, t, t \,\}$, or a type ascription $t :: T$. An evaluation context E may be a hole $\Box$, an application with a hole in function position $\Box\ t$, a hole in argument position $v\ \Box$, an if expression with a hole in the conditional position $\mathsf{if}\ \Box\ \mathsf{then}\ t\ \mathsf{else}\ t$, a pair with a hole at the first expression $\{\, \Box, t \,\}$, a pair with a hole at the second position $\{\, v, \Box \,\}$, a projection of the first element with a hole $\Box.1$, a projection of the second element with a hole $\Box.2$, or an ascription expression with a hole $\Box$. Γ is a finite map that associates a term with a type. Φ is a set of effects $\varepsilon(n,n)$.

3.1.2 Effect Set Operations

In this section we define operators over effect sets and their corresponding privileges and obligations. We define effect set containment in Figure 3.2. We say that the effect set Φ_1 is contained in the effect set Φ_2, notation $\Phi_1 \sqsubseteq \Phi_2$, if the effect set Φ_2 has more privileges and less obligations than the effect set Φ_1. Figure 3.2 presents the definition of $\leq_p$ (privilege ordering) and $\leq_o$ (obligation ordering). Rule (EmptyP) specifies that an empty set has less privileges than any effect set. Rule (NumberP) defines that $\Phi_1, \varepsilon(n_1,n_2)$ has less privileges than $\Phi_2, \varepsilon(n_1,n_2')$ if $n_2 \leq n_2'$, and inductively Φ_1 has less privileges than Φ_2. The definition of obligation ordering is similar to the definition of $\leq_p$. Let us present some examples for $\sqsubseteq$:

$$\{\, a(2,5), b(1,2) \,\} \sqsubseteq \{\, a(1,10), b(0,5) \,\}$$

The effect set on the left has less privileges, and more obligations than the effect set on the right, therefore the effect set of the left is contained in the effect set of the right. Consider now the following effect sets:

$$\{\, a(2,5) \,\} \not\sqsubseteq \{\, a(3,10) \,\}$$

The effect set on the left is not smaller than the effect set on the right, because the effect set on the right have more obligations than the effect set on the left. Now, consider the following effect sets:

$$\{\, a(0,1) \,\} \not\sqsubseteq \{\, b(0,1) \,\}$$

The effect set on the left contains a privilege from the effect a that the effect on the right does not contain, thus the effect set on the left is not contained by the effect set on the right. Finally, consider the following effect set:

$$\{\,\} \sqsubseteq \{\, a(0,1) \,\}$$

The empty effect set is contained by any effect set that does not contain any obligation.

Figure 3.3 defines the algebra between effect sets and effects. The intersection of effect sets $\Cap$ is defined as the set of minimum privileges and maximum obligations for any given pair of effects with the same tag. For example, consider effect set $\{\, a(3,5) \,\}$ and $\{\, a(4,7) \,\}$, the intersection is computed as follows:

$$\{\, a(3,5) \,\} \Cap \{\, a(4,7) \,\} = \{\, a(4,5) \,\}$$

Dually, the union of effect sets $\Cup$ is defined as the set of maximum privileges and the minimum obligations for any given pair of effects with the same tag. Thus, the following effect set is the result of the union:

$$\{\, a(3,5) \,\} \Cup \{\, a(4,7) \,\} = \{\, a(3,7) \,\}$$

Subtraction between effect sets is defined in two cases. The first case is when both effects sets have an effect tag in common. In that case, it subtracts the privileges and obligations from the minuend effect set [1]. The second case is for all effect tags not present in the subtracted effect set: the result is the same effects present in the minuend effect set. The subtraction for privileges and obligations is defined by cases. The subtraction between ∞ and ∞ is zero, the subtraction between ∞ and any number is ∞, and the subtraction between two numbers is the standard subtraction between natural numbers. The addition of effect sets, like the subtraction, is defined by two cases. If both effect sets contain the same effect tag, then the result is the standard addition of natural numbers (including infinite) between the components of the effect, otherwise if only one effect set contains an effect tag, then the result of the addition is the effect.

3.1.3 Well Formedness

In this section, we define well formedness for types and effect sets. The judgment for well formedness for types $\vdash T$ defines that the type T is well formed. Similarly the judgment for effect $\vdash \Phi$ defines that the effect set Φ is well formed.

[1] We use the same symbol ($\dot{-}$) for subtracting effect sets and effects.

$$\Phi_1 \Cap \Phi_2 = \{\varepsilon(\max(n_1, n_4), \min(n_2, n_3)) \mid \varepsilon(n_1, n_2) \in \Phi_1 \wedge \varepsilon(n_3, n_4) \in \Phi_2\}$$

$$\begin{aligned}\Phi_1 \Cup \Phi_2 = \; & \{\varepsilon(\min(n_1, n_4), \max(n_2, n_3)) \mid \varepsilon(n_1, n_2) \in \Phi_1 \wedge \varepsilon(n_3, n_4) \in \Phi_2\} \\ & \cup \\ & \{\, \varepsilon(n_1, n_2) \mid \varepsilon(n_1, n_2) \in \Phi_1 \vee \varepsilon(n_1, n_2) \in \Phi_2 \,\}\end{aligned}$$

$$\begin{aligned}\Phi_1 \dot{-} \Phi_2 = \; & \{\, \varepsilon(\mathrm{i}_1 \dot{-} \mathrm{i}_3, \mathrm{i}_2 \dot{-} \mathrm{i}_4) \mid \varepsilon(\mathrm{i}_1, \mathrm{i}_2) \in \Phi_1 \wedge \varepsilon(\mathrm{i}_3, \mathrm{i}_4) \in \Phi_2 \,\} \\ & \cup \\ & \{\, \varepsilon(\mathrm{i}_1, \mathrm{i}_2) \mid \varepsilon(\mathrm{i}_1, \mathrm{i}_2) \in \Phi_1 \wedge \varepsilon \notin \Phi_2 \,\}\end{aligned}$$

$$\begin{aligned}\infty \dot{-} \infty &= 0 && \\ \infty \dot{-} n &= \infty && \text{If } n \neq \infty \\ n_1 \dot{-} n_2 &= n_1 - n_2 && \text{If } n_1 \geq n_2 \wedge n_1 \neq \infty \\ n_1 \dot{-} n_2 &= 0 && \text{If } n_1 \leq n_2 \\ & \text{undefined otherwise} &&\end{aligned}$$

$$\begin{aligned}\Phi_1 + \Phi_2 = \; & \{\varepsilon(n_1 + n_3, n_2 + n_4) \mid \varepsilon(n_1, n_2) \in \Phi_1 \wedge \varepsilon(n_3, n_4) \in \Phi_2\} \\ & \cup \\ & \{\, \varepsilon(n_1, n_2) \mid \varepsilon(n_1, n_2) \in \Phi_1 \vee \varepsilon(n_1, n_2) \in \Phi_2 \,\}\end{aligned}$$

Figure 3.3: Algebra of may-must effects

$$\boxed{\vdash T}$$

$$\frac{}{\vdash B} \qquad \frac{\vdash T_1 \qquad \vdash T_2}{\vdash T_1 \times T_2} \qquad \frac{\vdash T_1 \qquad \vdash T_2 \qquad \vdash \Phi_c \qquad \vdash \Phi_p}{\vdash {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}}$$

Figure 3.4: Well-formedness of types

Well formedness of types is presented in Figure 3.4. For a type be well formed, it either has to be a base type B, a pair type $T_1 \times T_2$ where both elements of the pair are also well formed, or a function type ${T_1}^{\Phi_c} \to {T_2}^{\Phi_p}$ where the types T_1 and T_2, and the effect sets Φ_c and Φ_p are well formed. Well formedness of effect sets is presented in Figure 3.5. An effect set Φ is well formed if it is empty, or for each effect the number of privileges and obligations belong to $\textsc{Nat}^\infty$, and there are more privileges than obligations.

3.1.4 Subtyping

Figure 3.6 presents the rules of subtyping for λ_{mme}. Rules (Sb) defines that a base type B is a subtype of itself. Rule (Spair) defines that pair $T_1 \times T_2$ is subtype of $T_3 \times T_4$ if type T_1 is subtype of type T_3 and type T_2 is subtype of type T_4. Rule (Sλ) defines that a function type ${T_1}^{\Phi_{c1}} \to {T_2}^{\Phi_{p1}}$ is subtype of ${T_3}^{\Phi_{c2}} \to {T_4}^{\Phi_{p2}}$ if type T_3 is subtype of type T_1, effect set Φ_{c1} is

$$\boxed{\vdash \Phi}$$

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Phi \quad n_1 \in \text{NAT}^\infty \quad n_2 \in \text{NAT}^\infty \quad n_1 \leq n_2}{\vdash \Phi, \varepsilon(n_1, n_2)}$$

Figure 3.5: Well-formedness of effect sets

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

$$\text{(Sb)}\frac{}{B <: B} \qquad \text{(Spair)}\frac{T_1 <: T_3 \quad T_2 <: T_4}{T_1 \times T_2 <: T_3 \times T_4}$$

$$\text{(S}\lambda\text{)}\frac{T_3 <: T_1 \quad \Phi_{c1} \sqsubseteq \Phi_{c2} \quad T_2 <: T_4 \quad \Phi_{p2} \sqsubseteq \Phi_{p1}}{{T_1}^{\Phi_{c1}} \to {T_2}^{\Phi_{p1}} <: {T_3}^{\Phi_{c2}} \to {T_4}^{\Phi_{p2}}}$$

Figure 3.6: Subtyping

contained in the effect set Φ_{c2}, type T_2 is subtype of type T_4, and effect set Φ_{p2} is contained in the effect set Φ_{p1}.

3.1.5 Typing Rules

λ_{mme} has two typing judgments, one for values and another for terms. Both typing rules, raw values and terms, are mutually defined. The typing rules for raw values are presented in the Figure 3.7. Raw values do not consume any side effect or produce any side effect. A typing judgment for $\Gamma \vdash_u u : T$ denotes that raw value u has type T under type context Γ.

Rule (Tb), types a constant b by using the function μ. Rule (Tλ) types $(\lambda x : T_1 \,.\, t)$ as ${T_1}^{\Phi_c} \to {T_2}^{\Phi_p}$ where Φ_c is computed by using the function minimum effect set ($\rightsquigarrow$) is defined in Section 3.1.6. The body is well typed using Φ_c. Because the body is a term, we use the type judgment for terms ($\vdash$). Rule (Top) types op by using the mapping function ζ. Rule (Tvalpair) defines that the the type of a pair of raw values $\{ u_1, u_2 \}$ has type $T_1 \times T_2$ if raw value u_1 has type T_1 and raw value u_2 has type T_2.

Type rules for terms are presented in the Figure 3.8. A typing judgment for terms $\Phi; \Gamma \vdash t : T; \Phi'$ denotes that term t has type T under type context Γ, consuming effect set Φ and producing effect set Φ'. Rule (Tx) assigns type T to variable x if x maps to type T in context Γ. Rule (Tv) assigns type T to value v if the value v has type T. Rule (Tpair), given the effect set Φ, assigns to pair $\{ t_1, t_2 \}$ the type $T_1 \times T_2$ and the effect set Φ_2 if sub-term t_1 given the effect set Φ has type T_1 and produces the effect set Φ_1, and t_2 given the same effect set,

$$\boxed{\Gamma \vdash_u u : T}$$

$$\text{(Tb)}\frac{\mu(b) = B}{\Gamma \vdash_u b : B} \qquad \text{(T}\lambda\text{)}\frac{\begin{array}{c}\Gamma, (x : T_1) \vdash t \rightsquigarrow \Phi_c \\ \Phi_c; \Gamma, (x : T_1) \ \vdash t : T_2; \Phi_p\end{array}}{\Gamma \vdash_u (\lambda x : T_1 \ . \ t) : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}} \qquad \text{(Top)}\frac{\zeta(op) = T}{\Gamma \vdash_u \text{op} : T}$$

$$\text{(Tvalpair)}\frac{\Gamma \vdash_u u_1 : T_1 \qquad \Gamma \vdash_u u_2 : T_2}{\Gamma \vdash_u \{\, u_1, u_2 \,\} : T_1 \times T_2}$$

Figure 3.7: Typing for base values

$$\boxed{\Phi; \Gamma \vdash t : T; \Phi'}$$

$$\text{(Tx)}\frac{x : T \in \Gamma}{\Phi; \Gamma \vdash x : T; \Phi} \qquad \text{(Tv)}\frac{\begin{array}{c}\Gamma \vdash_u u : T' \\ T' <: T\end{array}}{\Phi; \Gamma \vdash u :: T : T; \Phi}$$

$$\text{(Tpair)}\frac{\Phi; \Gamma \vdash t_1 : T_1; \Phi_1 \qquad \Phi_1; \Gamma \vdash t_2 : T_2; \Phi_2}{\Phi; \Gamma \vdash \{\, t_1, t_2 \,\} : T_1 \times T_2; \Phi_2} \qquad \text{(Tproj1)}\frac{\Phi; \Gamma \vdash t : T_1 \times T_2; \Phi_1}{\Phi; \Gamma \vdash t.1 : T_1; \Phi_1}$$

$$\text{(Tproj2)}\frac{\Phi; \Gamma \vdash t : T_1 \times T_2; \Phi_1}{\Phi; \Gamma \vdash t.2 : T_2; \Phi_1} \qquad \text{(Tapp)}\frac{\begin{array}{c}\Phi; \Gamma \vdash t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi_1 \\ \Phi_1; \Gamma \vdash t_2 : T_1; \Phi_2 \qquad \Phi_c \leq_p \Phi_2\end{array}}{\Phi; \Gamma \vdash t_1 \ t_2 : T_2; \Phi_2 \dot{-} \Phi_c + \Phi_p}$$

$$\text{(Tif)}\frac{\begin{array}{c}\Phi; \Gamma \vdash t_1 : \mathsf{Bool}; \Phi_1 \\ \Phi_1; \Gamma \vdash t_2 : T; \Phi_2 \\ \Phi_1; \Gamma \vdash t_3 : T; \Phi_3\end{array}}{\Phi; \Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T; \Phi_2 \Cap \Phi_3} \qquad \text{(Tascribe)}\frac{\begin{array}{cc}\vdash T & T' <: T \\ \Phi; \Gamma \vdash t : T'; \Phi_1\end{array}}{\Phi; \Gamma \vdash t :: T : T; \Phi_1}$$

Figure 3.8: Typing rules for terms

that is produced by typing t_1, Φ_1 has type T_2 and produces effect set Φ_2; we use the effect set Φ_1 produced by typing t_1 for typing t_2 because, t_1 could consume or produce effects. Rule (Tproj1), given the effect set Φ, assigns to the first projection $t.1$ type T_1 and effect set Φ_1 if t has type $T_1 \times T_2$ and produces the effect set Φ_1 under the effect set Φ. Rule (Tproj2), given the effect set Φ, assigns to the second projection $t.2$ type T_2 and the effect set Φ_1 if t, given the effect set Φ has type $T_1 \times T_2$ and produces effect set Φ_1. Rule (Tapp) types $t_1\ t_2$ as T_2 with the output effect set $\Phi_2 \dot{-} \Phi_c + \Phi_p$. This operation represents that the body, given effect set Φ_2, is going to consume Φ_c and is going to produce Φ_p, if t given the effect set Φ has type ${T_1}^{\Phi_c} \to {T_2}^{\Phi_p}$ and produces effect set Φ_1, and t_2 given the effect set Φ_1 has type T_1 and produces the effect set Φ_2. Furthermore, the amount of privileges needed Φ_c must be lower than the amount of privileges available at the moment of the application Φ_1.

Let us present an example of typing an application. We assume that we have the following

function:

$$f : \mathsf{Unit}^{\{\, a(2,2) \,\}} \to \mathsf{Unit}^{\{\, b(0,3) \,\}}$$

function f consumes 2 obligations and 2 privileges of the effect a, and produces 0 obligation and 3 privileges of the effect b. Thus the type derivation of applying the function f to unit, under an initial effect set $\{\, a(1,6) \,\}$ and an empty context is:

$$\frac{\begin{array}{c}\{\, a(1,6) \,\}; \cdot \vdash f : \mathsf{Unit}^{\{\, a(2,2) \,\}} \to \mathsf{Unit}^{\{\, b(0,3) \,\}}; \{\, a(1,6) \,\} \\ \{\, a(1,6) \,\}; \cdot \vdash \mathsf{unit} : \mathsf{Unit}; \{\, a(1,6) \,\} \\ \{\, a(2,2) \,\} \leq_p \{\, a(1,6) \,\}\end{array}}{\{\, a(1,6) \,\}; \cdot \vdash f\ \mathsf{unit} : \mathsf{Unit}; \Phi'}$$

where effect set Φ' is defined as:

$$\Phi' \stackrel{\text{def}}{=} \{\, a(1,6) \,\} \dot{-} \{\, a(2,2) \,\} + \{\, b(0,3) \,\} \stackrel{\text{def}}{=} \{\, a(0,4), b(0,3) \,\}$$

therefore:

$$\{\, a(1,6) \,\}; \cdot \vdash f\ \mathsf{unit} : \mathsf{Unit}; \{\, a(0,4), b(0,3) \,\}$$

Rule (Tif) assigns to $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ type T, and produces effect set $\Phi_2 \Cap \Phi_3$, if t_1 has type Bool under effect set Φ and produces effect set Φ_1, t_2 has type T under effect set Φ_1 and produces effect set Φ_2, and t_3 has type T under effect set Φ_1 and produces effect set Φ_3. The intuition behind this rule is that the type system of λ_{mme} is conservative, thus λ_{mme} produces the minimum amount of privileges and the maximum amount of obligations for each effect present in both branches. Let us present an example of typing an if expression. We assume that we have the following two expressions:

$$\{\, a(1,1) \,\}; \cdot \vdash t_1 : \mathsf{Unit}; \{\, b(2,4) \,\}$$

$$\{\, a(1,1) \,\}; \cdot \vdash t_2 : \mathsf{Unit}; \{\, b(0,3) \,\}$$

Term t_1 consumes 1 obligation and 1 privilege of the effect a, and produces 2 obligations and 4 privileges of the effect b. Similarly, term t_2 consumes 1 obligation and 1 privilege of the effect a, and produces 0 obligations and 3 privileges of the effect b. Therefore, the typing derivation of the following if expression, with an initial effect set $\{\, a(1,1) \,\}$ and empty context Γ is:

$$\frac{\begin{array}{c}\{\, a(1,1) \,\}; \cdot \vdash \mathsf{true} : \mathsf{Bool}; \{\, a(1,1) \,\} \\ \{\, a(1,1) \,\}; \cdot \vdash t_1 : \mathsf{Unit}; \{\, b(2,4) \,\} \\ \{\, a(1,1) \,\}; \cdot \vdash t_2 : \mathsf{Unit}; \{\, b(0,3) \,\}\end{array}}{\{\, a(1,1) \,\}; \cdot \vdash \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 : \mathsf{Unit}; \Phi'}$$

where effect set Φ' is defined as:

$$\Phi' \stackrel{\text{def}}{=} \{\, b(2,4) \,\} \Cap \{\, b(0,3) \,\} \stackrel{\text{def}}{=} \{\, b(2,3) \,\}$$

therefore:

$$\{\, a(1,1) \,\}; \cdot \vdash \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 : \mathsf{Unit}; \{\, b(2,3) \,\}$$

Rule (Tascribe) types $t :: T$ as T and produces effect set Φ_1, if type T is well formed, term t is typed T' and produces effect set Φ_1, and type T' is sub-type of type T.

$$\boxed{\Gamma \vdash t \rightsquigarrow \Phi}$$

$$\text{SMv}\frac{}{\Gamma \vdash v \rightsquigarrow \{\ \}} \qquad \text{SMx}\frac{}{\Gamma \vdash x \rightsquigarrow \{\ \}} \qquad \text{SMpair}\frac{\begin{array}{c}\Gamma \vdash t_1 \rightsquigarrow \Phi_{t_1} \\ \Gamma \vdash t_2 \rightsquigarrow \Phi_{t_2} \\ \Phi_{t_1}; \Gamma \vdash t_1 : T_1; \Phi'_{t_1}\end{array}}{\Gamma \vdash \{\, t_1, t_2 \,\} \rightsquigarrow \Phi_{t_1} + \Phi_{t_2} \dot{-} \Phi'_{t_1}}$$

$$\text{SMproj1}\frac{\Gamma \vdash t \rightsquigarrow \Phi}{\Gamma \vdash t.1 \rightsquigarrow \Phi} \qquad \text{SMproj2}\frac{\Gamma \vdash t \rightsquigarrow \Phi}{\Gamma \vdash t.2 \rightsquigarrow \Phi}$$

$$\text{SMapp}\frac{\begin{array}{c}\Gamma \vdash t_1 \rightsquigarrow \Phi_{t_1} \qquad \Gamma \vdash t_2 \rightsquigarrow \Phi_{t_2} \\ \Phi_{t_1}; \Gamma \vdash t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi'_{t_1} \\ \Phi_{t_2}; \Gamma \vdash t_2 : T_1; \Phi'_{t_2}\end{array}}{\Gamma \vdash (t_1\ t_2) \rightsquigarrow \Phi_{t_1} + \Phi_c \dot{-} \Phi'_{t_2} + \Phi_{t_2} \dot{-} \Phi'_{t_1}}$$

$$\text{SMif}\frac{\begin{array}{c}\Gamma \vdash t_1 \rightsquigarrow \Phi_{t_1} \qquad \Phi_{t_1}; \Gamma \vdash t_1 : \mathsf{Bool}; \Phi'_{t_1} \\ \Gamma \vdash t_2 \rightsquigarrow \Phi_{t_2} \\ \Gamma \vdash t_3 \rightsquigarrow \Phi_{t_3}\end{array}}{\Gamma \vdash (\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) \rightsquigarrow \Phi_{t_1} + \Phi_{t_2} \Cup \Phi_{t_3} \dot{-} \Phi'_{t_1}} \qquad \text{SMasc}\frac{\Gamma \vdash t \rightsquigarrow \Phi}{\Gamma \vdash t :: T \rightsquigarrow \Phi}$$

Figure 3.9: Minimum effect set

3.1.6 Minimum Effect Set

Figure 3.9 defines the minimum effect set ($\rightsquigarrow$) function to compute the minimum effect set required to type a term, i.e. the minimum privileges needed and the maximum obligations produced. The judgment $\Gamma \vdash t \rightsquigarrow \Phi$ defines that given a context Γ, the term t requires the effect set Φ.

Rule (SMv) and (SMx) defines the smallest set for values and variables respectively, the smallest set in both cases is empty, they do not need any effect for typing. Rule (SMpair) defines that the smallest set in order to type the pair $\{\, t_1, t_2 \,\}$ is $(\Phi_{t_1} + (\Phi_{t_2} \dot{-} \Phi'_{t_1}))$ where Φ_{t_1} is the smallest set needed to type t_1, Φ_{t_2} is the smallest set needed to type t_2, and Φ'_{t_1} is the effect set produced by typing t_1 with its smallest set. Rule (SMproj1) defines that the smallest set for typing the first projection of a pair $t.1$ is the smallest set required for the pair t. Similarly, the smallest set needed for type the second projection of a pair t_2 is the smallest set required for type a pair t. Rule (SMapp) defines that the minimum effect set to type application $t_1\ t_2$ is the effect set $\Phi_{t_1} + \Phi_c \dot{-} \Phi'_{t_2} + \Phi_{t_2} \dot{-} \Phi'_{t_1}$, where Φ_{t_1} is the smallest set that requires t_1, Φ_c is the smallest set that requires the body of t_1 (t_1 is a term that has the type of a function), Φ'_{t_1} and Φ'_{t_2} are the sets produced by typing t_1 and t_2 respectively with their corresponding smallest sets. The intuition behind the minimal effect set is that if neither of the terms produces any privilege or obligation, then the minimal effect set is just the addition between the effect sets required by the subterms Φ_{t_1}, Φ_{t_2}, and the latent effect

set Φ_c.

$$\Phi'_{t_1} = \{\ \} \qquad\qquad \Phi'_{t_2} = \{\ \}$$

$$\Gamma \vdash (t_1\ t_2) \rightsquigarrow \Phi_{t_1} + \Phi_c + \Phi_{t_2}$$

If the term in function position produces infinite privileges in Φ'_{t_1}, then the only requirement is the effect set that requires term t_1 (as we defined previously, for any number n then $n \dot{-} \infty = 0$), thus if Φ'_{t_1} contains infinite privileges, the right subtraction is zero:

$$\Phi_{t_1} + \Phi_c \dot{-} \Phi'_{t_2} + \Phi_{t_2} \dot{-} \Phi'_{t_1}$$

$$\Phi_{t_1}$$

If the term t_2 produces infinite privileges, then the requirement of the application is only the addition of Φ_{t_1} with Φ_{t_2} minus the effect set produced by t_1, $\Phi_{t'_1}$.

$$\Phi_{t_1} + \Phi_{t_2} \dot{-} \Phi'_{t_1}$$

For example, consider the following expressions:

$$\cdot \vdash t_1 \rightsquigarrow \{\ a(1,3)\ \} \qquad\qquad \cdot \vdash t_2 \rightsquigarrow \{\ a(2,5)\ \}$$

$$\{\ a(1,3)\ \}; \cdot \vdash t_1 : \mathsf{Unit}^{\{\ a(2,2)\ \}} \to \mathsf{Unit}^{\{\ \cdot\ \}}; \{\ a(0,1)\ \}$$

$$\{\ a(2,5)\ \}; \cdot \vdash t_2 : \mathsf{Unit}; \{\ a(3,3)\ \}$$

Then, the minimal effect set required for type the application $t_1\ t_2$ is:

$$\cdot \vdash (t_1\ t_2) \rightsquigarrow \{\ a(1,3)\ \} + \{\ a(2,2)\ \} \dot{-} \{\ a(3,3)\ \} + \{\ a(2,5)\ \} \dot{-} \{\ a(0,1)\ \}$$

$$\cdot \vdash (t_1\ t_2) \rightsquigarrow \{\ a(1,3)\ \} + \{\ a(2,5)\ \} \dot{-} \{\ a(0,1)\ \}$$

$$\cdot \vdash (t_1\ t_2) \rightsquigarrow \{\ a(1,3)\ \} + \{\ a(2,4)\ \}$$

$$\cdot \vdash (t_1\ t_2) \rightsquigarrow \{\ a(3,7)\ \}$$

In order to type t_2, we required at least 5 privileges, and t_1 consumes 2 privileges, thus the application $t_1\ t_2$ requires at least 7 privileges:

$$\{\ a(3,7)\ \}; \cdot \vdash t_1 : \mathsf{Unit}^{\{\ a(2,2)\ \}} \to \mathsf{Unit}^{\{\ \cdot\ \}}; \{\ a(2,5)\ \}$$

$$\{\ a(2,5)\ \}; \cdot \vdash t_2 : \mathsf{Unit}; \{\ a(3,3)\ \}$$

Rule (SMif) defines the smallest set to typecheck $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ is $\Phi_{t_1} + ((\Phi_{t_2} \Cup \Phi_{t_3}) \dot{-} \Phi'_{t_1})$, where Φ_{t_1}, Φ_{t_2}, and Φ_{t_3} are the smallest set require for t_1, t_2, and t_3 respectively. Term $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ always requires the effect set required by the term t_1, and in order to type both branches, we need to provide the maximum amount of privileges that the branches are going to require. Rule (SMasc) defines that the smallest set for ascription $t :: T$ is the smallest set Φ required to typecheck t.

We now show some examples of the function $\rightsquigarrow$. In order to write the following examples more concisely, we use notation $t_1; t_2$ to represent the second projection of the pair expression $\{\, t_1, t_2 \,\}.2$:

$$f : \mathsf{Unit}^{\{\, a(1,1) \,\}} \to \mathsf{Unit}^{\{\,\}} \qquad\qquad g : \mathsf{Unit}^{\{\,\}} \to \mathsf{Unit}^{\{\, a(1,1) \,\}}$$

The function f receives a Unit as argument, it requires the effect set $\{\, a(1,1) \,\}$, returns Unit, and it does not produce any effects. Function g receives a Unit as argument, it does not require any effect in order to be applied, returns Unit, and produces the effect set $\{\, a(1,1) \,\}$.

$$\Gamma \vdash (g\ \mathsf{unit}); (f\ \mathsf{unit}) \rightsquigarrow \{\,\}$$

The example shows that the minimum effect set for applying f to unit after applying g to unit is empty, because the application of function g produces the effect set $\{\, a(1,1) \,\}$, which is fully consumed by the application of f. Thus, this expression does not require any effect in order to type.

For the next example we assume that we have another function f'.

$$f' : \mathsf{Unit}^{\{\, a(1,1) \,\}} \to (\mathsf{Unit}^{\{\, a(1,1) \,\}} \to \mathsf{Unit}^{\{\,\}})^{\{\,\}}$$

Function f' receives a Unit as argument, requires the effect set $\{\, a(1,1) \,\}$, returns another function, and does not produce any effects. The function that return f' is a function that receives a Unit as argument, requires the effect set $\{\, a(1,1) \,\}$, returns Unit, and does not produce any effect set.

$$\Gamma \vdash ((f'\ \mathsf{unit})\ \mathsf{unit}) \rightsquigarrow \{\, a(2,2) \,\}$$

The smallest effect set for f' is $\{\, a(1,1) \,\}$, and $(f'\mathsf{unit})$ produces a function whose minimal set is $a(1,1)$; thus the minimal effect set of the expression is the sum of both effect sets, i.e. the effect set $a(2,2)$.

Let us assume that we have functions f'' and g'', both functions receive a Unit as argument, returns Unit, and neither produces any effect. f'' requires effect set $a(1,1)$ and g'' requires $b(1,1)$.

$$f'' : \mathsf{Unit}^{\{\, a(1,1) \,\}} \to \mathsf{Unit}^{\{\,\}} \qquad\qquad g'' : \mathsf{Unit}^{\{\, b(1,1) \,\}} \to \mathsf{Unit}^{\{\,\}}$$

$$\Gamma \vdash \mathsf{if\ true\ then}\ (f''\ \mathsf{unit})\ \mathsf{else}\ (g''\ \mathsf{unit}) \rightsquigarrow \{\, a(0,1), b(0,1) \,\}$$

The smallest set for $(f''\ \mathsf{unit})$ is $\{\, a(1,1) \,\}$ and for $(g''\ \mathsf{unit})$ is $\{\, b(1,1) \,\}$; thus the smallest effect set for the if expression is the union of privileges and intersection of obligations: $a(1,1) \Cup b(1,1) = \{\, a(0,1), b(0,1) \,\}$.

3.2 Dynamic Semantics

The small-step operational semantics of λ_{mme} is defined in Figure 3.10 and 3.11. Judgment $\langle t, \Phi \rangle \mapsto \langle t', \Phi' \rangle$ denotes that a configuration composed of a term t and an effect set Φ takes one step of evaluation to a configuration with term t' and effect set Φ'. Figure 3.10 presents

$$\text{(Context)}\frac{\langle t, \Phi\rangle \to \langle t', \Phi'\rangle}{\langle E[t], \Phi\rangle \mapsto \langle E[t'], \Phi'\rangle}$$

Figure 3.10: Term reduction

$$\boxed{\langle t, \Phi\rangle \to \langle t', \Phi'\rangle}$$

$$\text{(E-Pair)}\frac{}{\langle \{\, u_1 :: T_1, u_2 :: T_2 \,\}, \Phi\rangle \to \langle \{\, u_1, u_2 \,\} :: T_1 \times T_2, \Phi\rangle}$$

$$\text{(E-Proj1)}\frac{}{\langle \{\, u_1, u_2 \,\}.1, \Phi\rangle \to \langle u_1, \Phi\rangle} \qquad \text{(E-Proj2)}\frac{}{\langle \{\, u_1, u_2 \,\}.2, \Phi\rangle \to \langle u_2, \Phi\rangle}$$

$$\text{(E-IfT)}\frac{}{\langle \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_t, \Phi\rangle} \qquad \text{(E-IfF)}\frac{}{\langle \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_f, \Phi\rangle}$$

$$\text{(E-Asct)}\frac{}{\langle u :: T' :: T, \Phi\rangle \to \langle u :: T, \Phi\rangle} \qquad \text{(E-App)}\frac{}{\langle (\lambda x : T_1\ .\ t)\ v, \Phi\rangle \to \langle t[x \to v], \Phi\rangle}$$

$$\text{(E-Op)}\frac{\begin{array}{c}\mathrm{op}\ v_1 \to_\zeta v_2 \\ \zeta(\mathrm{op}) = {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}\end{array}}{\langle \mathrm{op}\ v_1, \Phi\rangle \to \langle v_2, \Phi \dot{-} \Phi_c + \Phi_p\rangle}$$

Figure 3.11: Notion of term reduction

the term reduction, rule (Context) defines that if the configuration $\langle t, \Phi\rangle$ can take a step to $\langle t', \Phi'\rangle$ using the notions of term reduction, then configuration $\langle E[t], \Phi\rangle$ can take a step with effect set Φ to the configuration $\langle E[t'], \Phi'\rangle$ for any E.

Only the application of operations op are effectful, the application of operations are the only expression that modifies the dynamic effect set.

Figure 3.11 presents the notion of term reduction. Rule (E-pair) reduces the expression $\{\, u_1 :: T_1, u_2 :: T_2 \,\}$ by *extracting* the ascriptions of the raw values, without changing the effect set. Rule (E-proj1) reduces pair $\{\, u_1, u_2 \,\}$ by extracting the first element u_1 , without changing the effect set (the ascription of the raw values is subsumed). Similar to the previous rule, rule (Tproj2) reduces pair $\{\, u_1, u_2 \,\}$ by extracting the second element u_2, without changing the effect set. Rule (E-IfT) reduces the if expression $\mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f$ by selecting the then branch t_t. Similarly to the previous rule, rule (E-IfF) reduces the if expression $\mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f$ by selecting the else branch t_f. Rule (E-Asct) reduces the expression $u :: T' :: T$ by *swallowing* the internal ascription, producing the value $v :: T$. Rule (E-App) reduces the function application $(\lambda x : T_1\ .\ t)\ v$ by applying the substitution $t[x \to v]$. Rule (E-Op) reduces the expression op v_1 by applying the relation op $v_1 \to_\zeta v_2$ [2] and modifying the effect set Φ by subtracting the effect set Φ_c and adding the set Φ_p, this is the only rule that modifies the dynamic effect set.

[2]Defined in subsection 2.2.2 from the previous chapter.

Next, we show an example of reduction of an application. For illustration purposes we define the operation print and the effect IO.

The operation print represents the primitive that prints its argument on screen. The type of print is defined as $\zeta(\mathsf{print}) = \mathsf{Nat}^{\{\, IO(1,1) \,\}} \to \mathsf{Unit}^{\cdot}$, where the type of print is a function that receives a Nat, requires the effect set $\{\, IO(1,1) \,\}$, returns Unit, and does not produce any effect set. The reduction of print is defined as $\mathsf{print}\ n \to_{\zeta} \mathsf{unit}$, where for the primitive print for any natural number as argument, the expression produces a unit. Consider the following program $(\lambda x : \mathsf{Nat}\ .\ \mathsf{print}\ x))\ 5$. The reduction of this program with initial set $\{\, IO(1,1) \,\}$ is the following:

$$\langle (\lambda x : \mathsf{Nat}\ .\ \mathsf{print}\ x))\ 5, \{\, IO(1,1) \,\}\rangle \mapsto$$

$$\langle (\text{print } 5), \{\, IO(1,1) \,\}\rangle \mapsto$$

$$\langle \text{unit}, \{\ \}\rangle$$

The application does not change the effect set, because the effectful term is the body of the function, not the function itself [3].

3.3 Properties

In this chapter we define the soundness property for λ_{mme}. Similar to the previous chapter, we define soundness as follows: a sound effect system is one that does a sound approximation of the effects, i.e. a sound effect system guarantees that when a well typed program executes, the predicted effect set contains at least the privileges that the program actually requires and at most the obligations that the program actually performs. Formally, soundness is defined as:

Theorem 3.1. *Soundness*

$$\Phi; \cdot \vdash t : T; \Phi_s \text{ and } \langle t, \Phi\rangle \mapsto^* \langle v, \Phi_{\text{d}}\rangle \text{ then } \Phi_s \sqsubseteq \Phi_{\text{d}}$$

The effect soundness property of λ_{mme}, like SFLE, states that if a term t is well typed given an effect set Φ_s, and the pair: term t and effect set Φ_s reduces to a pair of value v and effect set Φ_{d}, [4] then effect set Φ_s is contained in Φ_{d}. Thus, if the type system of λ_{mme} determines statically that a program fulfill its obligations with a certain effect set, then we can guarantee that the program, with the same effect set, will fulfill its obligations when executed. The proof of this property is show in the following chapter, with an extended version of λ_{mme}. A complete proof is provided in appendix.

[3] The previous program cannot start with an empty effect set, otherwise the program would not be well typed. We could add the condition $\Phi_c \leq_p \Phi$ to the rule (E-App), thus the program would get stuck if there are not enough privileges.

[4] Like SFLE, λ_{mme} cannot diverge and the soundness property is more broad that λ_{mme}, the property also accepts program that diverges.

3.4 Summary

In this Chapter we have formally presented λ_{mme}, a language with may-must effects. In subsection 3.1.1 we presented the syntax for λ_{mme}. As we showed in the previous chapter, λ_{mme} has a similar syntax to SFLE, but with a different interpretation of effects; λ_{mme} represents effects as privileges and obligations. Subsection 3.1.4 defines subtyping for types. Subsection 3.1.2 defines effects with privileges and obligations, and its operators. Subsection 3.1.3 defines well formedness of types and effect sets. In subsection 3.1.5 we provide the typing rules for λ_{mme}, and provided some examples. We separate the typing rules in two: typing rules for values and typing rules for terms. In subsection 3.1.6 we provided an algorithm that calculates the minimum effect set that a term required in order to type. In section 3.2 we defined the dynamic semantics for λ_{mme}. The dynamic evaluation has a configuration composed by the term and the effect set. It is necessary to keep track of the dynamic effect set for the proof of soundness (presented in appendix) described in section 3.3.

Chapter 4

Lambda May-Must With Effect Polymorphism

In this chapter we present λ^{p}_{mme} , an extension of λ_{mme} with effect polymorphism. Effect polymorphism allows the programmer the abstraction of effectful programs that share a certain logic. For example, a simple API of a database requires that all operations (such as create, read, update or delete) have to open and close the connection to the database. With effect polymorphism it is possible to write the function performAction that has the following type:

$$\mathsf{performAction} :< \alpha > \{\, \mathit{open}(1,1), \alpha, \mathit{close}(1,1) \,\} \, (\{\, \alpha \,\} \, \mathsf{String} \to \mathsf{Unit} \, \{\ \}) \to \mathsf{Unit} \, \{\ \}$$

Where the polymorphic effect, in this case α, is written between the symbols $<$ and $>$. The function receives as argument an effectful function that receives as argument a query of type String, performs the effect α and returns a Unit. performAction opens the connection, applies the function received as argument and closes the connection.

This chapter is organized as follows: first, we provide the static semantics in Section 4.1. Second, we present the dynamic semantics in Section 4.2. Finally in Section 4.3, we present the soundness property of the language.

4.1 Static Semantics

In this section we present the static semantics of λ^{p}_{mme} , an extension of λ_{mme} with support for effect polymorphism. In sub-section 4.1.1, we introduce the new syntactic elements in order to introduce effect polymorphism to the language. In sub-section 4.1.2 we show the operators needed in order to work with polymorphism. In sub-section 4.1.3 we prove well formedness for polymorphic effects. In sub-section 4.1.4 we define sub-typing in presence of effect polymorphism. In sub-section 4.1.5 we present the type rules of effect polymorphism. In sub-section 4.1.6 we present the extended version of the minimum effect set function ($\rightsquigarrow$).

$$
\begin{array}{rcll}
 & & \alpha \in \textsc{Effect Var}, & \\
\mathrm{e} & ::= & \varepsilon(n,n) \mid n\alpha & \text{(effects)} \\
T & ::= & \ldots \mid \forall\alpha.\,{}^{\Phi}T^{\Phi} & \text{(types)} \\
u & ::= & \ldots \mid (\Lambda\alpha\ .\ t) & \text{(raw values)} \\
t & ::= & \ldots \mid t\ [\mathrm{e}] & \text{(terms)} \\
E & ::= & \ldots \mid \Box\ [\mathrm{e}] & \text{(evaluation context)} \\
\Phi & ::= & \cdot \mid \Phi, \mathrm{e} & \text{(effect set)} \\
\Delta & ::= & \cdot \mid \Delta, \alpha & \text{(effect context)}
\end{array}
$$

Figure 4.1: Syntax

$$
\begin{array}{rcll}
\{\ \}[\alpha \to \mathrm{e}] & = & [\![\{\ \}]\!] & \\
\Phi, n\alpha[\alpha \to \mathrm{e}] & = & [\![\Phi[\alpha \to \mathrm{e}], n\mathrm{e}]\!] & \\
\Phi, n\beta[\alpha \to \mathrm{e}] & = & [\![\Phi[\alpha \to \mathrm{e}], n\beta]\!] & \alpha \neq \beta
\end{array}
$$

Figure 4.2: Substitution on effect set

4.1.1 Syntax

The syntax for λ^{p}_{mme} is presented in the Figure 4.1. λ^{p}_{mme} extends the syntax of λ_{mme} by adding effects variables α. Effects e can be a concrete effect $\varepsilon(n,n)$ or a scaled effect variable $n\alpha$. Number n represents the amount of times the effect *appears* in the effect set. α is syntactic sugar for 1α. 0α means that the effect variable is not present in the set.

A polymorphic type $\forall\alpha.\,{}^{\Phi_1}T^{\Phi_2}$ binds the effect variable α in the type T and both effect sets Φ_1 and Φ_2, i.e. the effect variable α can appear in T or in either effect sets Φ_1 or Φ_2.

Raw values u are extended with effect abstraction $(\Lambda\alpha\ .\ t)$. Terms t are extended with effect instantiation $t\ [\mathrm{e}]$. Evaluation context E is extended with effect instantiation with a hole $\Box\ [\mathrm{e}]$.

Substitution of effect variables is defined as usual on terms and types; we apply the substitution on each component of the term (or type). Figure 4.2 defines effect substitution on effect sets, where the substitution on the empty effect set, is the empty effect set. The effect substitution of α for e in an effect set that contains $n\alpha$ is $n\mathrm{e}$. After performing a substitution, we normalize the resulting effect set.

The normalization function $[\![-]\!]$ is defined in Figure 4.3. The function is defined recursively by cases. We assume that the elements of the effect set are sorted alphabetically. The normalization of an empty effect set is the empty effect set. An effect variable with a scale of 0 is removed. The effect variable $n(m\alpha)$ is normalized by multiplying n with m. Similarly, the normalization of $n\varepsilon(n_1,n_2)$ is multiplying n with the privileges n_1 and obligations n_2. The normalization of an effect variable that appears twice is the sum between the scaling n and m. If the concrete effect ε appears twice, then the normalization is the sum of the privileges and obligations. If an effect e does not appears again in Φ and does not match with any other case, then we just apply normalization to the remaining effect set. Let us

$$
\begin{array}{rcll}
[\![\{\ \}]\!] & = & \{\ \} & \\
[\![\Phi, 0\alpha)]\!] & = & [\![\Phi]\!] & \\
[\![\Phi, n(m\alpha)]\!] & = & [\![\Phi, (n * m)\alpha)]\!] & \\
[\![\Phi, n\varepsilon(n_1, n_2)]\!] & = & [\![\Phi, \varepsilon(n_1 * n, n_2 * n)]\!] & \\
[\![\Phi, n\alpha, m\alpha]\!] & = & [\![\Phi, (n + m)\alpha]\!] & \\
[\![\Phi, \varepsilon(n_1, m_1), \varepsilon(n_2, m_2)]\!] & = & [\![\Phi, \varepsilon(n_1 + n_2, m_1 + m_2)]\!] & \\
[\![\Phi, \mathrm{e}]\!] & = & [\![\Phi]\!], \mathrm{e} & \mathrm{e} \notin \Phi
\end{array}
$$

Figure 4.3: Normalization of effect sets

$$
\text{(PolyVarP)}\ \frac{\ldots \quad \Phi_1 \leq_p \Phi_2 \qquad n < m}{\Phi_1, n\alpha \leq_p \Phi_2, m\alpha} \qquad\qquad \text{(PolyVarO)}\ \frac{\ldots \quad \Phi_1 \leq_o \Phi_2 \qquad n < m}{\Phi_1, n\alpha \leq_o \Phi_2, m\alpha}
$$

Figure 4.4: Effect set containment

consider the following examples for the normalization function $[\![-]\!]$:

$$\{\ 2\alpha, 3\beta\ \}[\alpha \to 4\beta] = [\![\{\ 2(4\beta), 3\beta\ \}]\!] = [\![\{\ 8\beta, 3\beta\ \}]\!] = \{\ 11\beta\ \}$$

After performing the substitution of α by 4β, we obtain the effect set $[\![\{\ 2(4\beta), 3\beta\ \}]\!]$. We have to apply normalization twice on the effect set. The first time we multiply 2 with 4, and the second time we join the effect variables β. Similar to the previous example:

$$\{\ 2\alpha, IO(1,1)\ \}[\alpha \to IO(2,3)] = [\![\{\ 2IO(2,3), IO(1,1)\ \}]\!] =$$

$$[\![\{\ IO(4,6), IO(1,1)\ \}]\!] = \{\ IO(5,7)\ \}$$

The substitution of α with $IO(1,1)$ produces the effect set $[\![\{\ 2IO(2,3), IO(1,1)\ \}]\!]$. We apply normalization twice. We multiply 2 with the privileges and obligations of $IO(2,3)$, then we join the effect IO that appears twice.

The effect context Δ contains the effect variables used to check that there are no free effect variables in types or effect sets. The effect set Δ does not contain the scale of the effect variable.

4.1.2 Effect Sets

The operators $\leq_p$ (privilege ordering) and $\leq_o$ (obligation ordering) are extended with one rule each, as shown in Figure 4.4. Rule (PolyVarP) defines that $\Phi_1, n\alpha$ has less privileges than $\Phi_2, m\alpha$, if Φ_1 has less privileges than Φ_2, n is less than m, and effect variable α appears in both effects sets. Rule (PolyVarO) is defined analogously to (PolyVarP). Effect set containment $\sqsubseteq$ is defined as it was defined in the previous chapter: $\Phi_1 \sqsubseteq \Phi_2 = (\Phi_1 \leq_p \Phi_2)$ and $(\Phi_2 \leq_o \Phi_1)$.

$$
\begin{aligned}
\Phi_1 \Cap \Phi_2 = \;& \ldots \\
& \cup \\
& \{\text{MIN}(n,m)\alpha \mid n\alpha \in \Phi_1 \wedge m\alpha \in \Phi_2\}
\end{aligned}
$$

$$
\begin{aligned}
\Phi_1 \Cup \Phi_2 = \;& \ldots \\
& \cup \\
& \{\text{MAX}(n,m)\alpha \mid n\alpha \in \Phi_1 \wedge m\alpha \in \Phi_2\} \\
& \cup \\
& \{\, n\alpha \mid n\alpha \in \Phi_1 \bigoplus n\alpha \in \Phi_2 \,\}
\end{aligned}
$$

$$
\begin{aligned}
\Phi_1 \dot{-} \Phi_2 = \;& \ldots \\
& \cup \\
& \{\, (n-m)\alpha \mid n\alpha \in \Phi_1 \wedge m\alpha \in \Phi_2 \,\}
\end{aligned}
$$

$$
\begin{aligned}
\Phi_1 + \Phi_2 = \;& \ldots \\
& \cup \\
& \{\, (n+m)\alpha \mid n\alpha \in \Phi_1 \wedge m\alpha \in \Phi_2 \,\} \\
& \cup \\
& \{\, n\alpha \mid n\alpha \in \Phi_1 \bigoplus n\alpha \in \Phi_2 \,\}
\end{aligned}
$$

Figure 4.5: Algebra of may-must effects

Figure 4.5 defines the algebra of effect sets and effects. The operators have the same definitions as before, with the addition of the cases for effect variables. The intersection of effect sets $\Cap$ adds the case for intersection of effect variables, where the intersection is the minimum amount of effect variable α presented in any of the effect set. The union of effect sets $\Cup$ adds two cases for effect variable: the first case is when both effect sets contain the effect variable α, then the union is the maximum amount of effect variables on any of the effect set; the second case is when the effect variable α appears only in one effect set, in this case the union is just the variable (with the amount of times it appears). The subtraction of effect sets $\dot{-}$ also considers the case for effect variables. The subtraction of effect variables is the subtraction (of naturals) [1]. The addition between effect sets adds two cases: the first one, the effect variable α appears in both effect set, in this case the resulting effect set is the *addition* of the quantities of the effect variable. The second case is when the effect variable α appears only in one effect set; in this case, the addition is just the effect variable with its scale.

4.1.3 Well formedness

The judgment for well formedness of types $\Delta \vdash T$ (and effect sets $\Delta \vdash \Phi$) defines that given the effect context Δ, the type T (and the effect set Φ) is well formed.

[1] After applying a subtraction, if the resulting effect is 0α, then it is removed from the effect set.

$$\boxed{\Delta \vdash T}$$

...

$$\frac{\Delta, \alpha \vdash T \qquad \Delta, \alpha \vdash \Phi_c \qquad \Delta, \alpha \vdash \Phi_p}{\Delta \vdash \forall \alpha.\, {}^{\Phi_c} T^{\Phi_p}}$$

Figure 4.6: Well-formedness of types

$$\boxed{\Delta \vdash \Phi}$$

...

$$\frac{\Delta \vdash \Phi \qquad n \in \textsc{Nat} \qquad \alpha \in \Delta}{\Delta \vdash \Phi, n\alpha}$$

Figure 4.7: Well-formedness of effect sets

Figure 4.6 presents the extension of well-formedness of types. Polymorphic type $\forall \alpha.\, {}^{\Phi_c} T^{\Phi_p}$ is well formed if type T, the effect sets Φ_c, and Φ_p are all well formed in an extended effect context Δ, α .

The extension of the rules of well formedness is presented in Figure 4.7. An effect set Φ is well formed if the effect variable α belongs to the effect set Δ and n belong to the naturals.

4.1.4 Subtyping

Figure 4.8 presents the extension of subtyping ($<:$). Rule (Spoly) defines that type $\forall \alpha.\, {}^{\Phi_{c1}} T_1 {}^{\Phi_{p2}}$ is subtype of type $\forall \alpha.\, {}^{\Phi_{c2}} T_2 {}^{\Phi_{p2}}$ if the effect set Φ_{c2} is contained in effect set Φ_{c1}, type T_1 is subtype of type T_2, and effect set Φ_{p_1} is contained in effect set Φ_{p_2}.

4.1.5 Typing Rules

We extend the typing judgments with the effect context Δ. The typing rules are extended with rules for effect polymorphism. The rule for effect abstraction is presented in the Figure 4.9, and effect instantiation is presented in Figure 4.10. The typing judgment for terms $\Phi; \Gamma; \Delta \vdash t : T; \Phi'$ means that term t has type T under type context Γ and effect context Δ; and consumes effect set Φ and produces effect set Φ'.

Rule (Teffabs) assigns to $(\Lambda \alpha\ .\ t)$ type $\forall \alpha.\, {}^{\Phi_c} T^{\Phi_p}$ if using the minimum effect set function ($\rightsquigarrow$), which calculates the effect set Φ_c and using that effect set to typecheck the body of the effect abstraction, then the body has type T.

...

$$\text{(Spoly)}\frac{\begin{array}{c}\Phi_{c1} \sqsubseteq \Phi_{c2} \\ T_1 <: T_2 \qquad \Phi_{p_2} \sqsubseteq \Phi_{p_1}\end{array}}{\forall\alpha.\,^{\Phi_{c1}}T_1{}^{\Phi_{p_2}} <: \forall\alpha.\,^{\Phi_{c2}}T_2{}^{\Phi_{p_2}}}$$

Figure 4.8: Subtyping

$$\boxed{\Gamma;\Delta \vdash_u u : T}$$

...

$$\text{(Teffabs)}\frac{\begin{array}{c}\Gamma,\Delta,\alpha \vdash t \rightsquigarrow \Phi_c \\ \Phi_c;\Gamma;\Delta,\alpha \vdash t : T;\Phi_p\end{array}}{\Gamma;\Delta \vdash_u (\Lambda\alpha\ .\ t) : \forall\alpha.\,^{\Phi_c}T^{\Phi_p}}$$

Figure 4.9: Typing for base value

$$\boxed{\Phi;\Gamma;\Delta \vdash t : T;\Phi'}$$

...

$$\text{(Teffins)}\frac{\begin{array}{c}\mathrm{e} \in \mathrm{Eff} \qquad \Phi;\Gamma;\Delta \vdash t : \forall\alpha.\,^{\Phi_c}T^{\Phi_p};\Phi_1 \\ \Phi_c[\alpha \to \mathrm{e}] \leq_p \Phi_1\end{array}}{\Phi;\Gamma;\Delta \vdash t\ [\mathrm{e}] : T[\alpha \to \mathrm{e}];\Phi_1 \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]}$$

Figure 4.10: Typing rules for terms

The intuition behind an effect abstraction $(\Lambda\alpha\ .\ t)$ is that a term t may contain effect variable α in its type T, or in the effect set that it consumes Φ_c or produces Φ_p, after being instantiated. Thus, when we do an instantiation with effect e we have to do a substitution of the effect variable. For example, let us consider the following type for a term t:

$$t : \forall\alpha.\,^{\{\,\alpha\,\}}\mathsf{Unit}^{\{\,\alpha\,\}} \to \mathsf{Unit}^{\{\,\}\{\,\}}$$

where t has type of an effect abstraction of a function that consumes effect variable α. Thus, the instantiation of term t with an effect e should have the same type of a function that consumes effect e:

$$t\ [\mathrm{e}] : \mathsf{Unit}^{\{\,\mathrm{e}\,\}} \to \mathsf{Unit}^{\{\,\}}$$

Rule (Teffins) assigns to $t\ [\mathrm{e}]$ the type $T[\alpha \to \mathrm{e}]$ and produces the effect set $\Phi_1 \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]$ if e belong to the set Eff, t has type $\forall\alpha.\,^{\Phi_c}T^{\Phi_p}$, and produces the effect set Φ_1, and the amount of privileges needed $\Phi_c[\alpha \to \mathrm{e}]$ is lower than the amount of privileges available at the moment of the application Φ_1. An effect instantiation is similar to an application. The *body* of an effect abstraction may be a term, therefore it does not reduce until it is instantiated. Effect set Φ_1 represents the effect set of the term t when it reduces to an effect

$$\boxed{\Gamma, \Delta \vdash t \rightsquigarrow \Phi}$$

...

$$\text{SMins}\frac{\begin{array}{c}\Gamma, \Delta \vdash t \rightsquigarrow \Phi_t \\ \Phi_t; \Gamma; \Delta \vdash t : \forall\alpha.\, {}^{\Phi_c}T^{\Phi_p}; \Phi'_t\end{array}}{\Gamma, \Delta \vdash (t\ [\mathrm{e}]) \rightsquigarrow \Phi_t + \Phi_c[\alpha \to \mathrm{e}] \dot{-} \Phi'_t}$$

Figure 4.11: Minimum effect set

abstraction ($\Lambda\alpha\ .\ t'$). Effect set Φ_c represents the effect set that the term t' will consume when reduced, and effect set Φ_p represents the effect set that the term t' will produce. Thus, the output effect $\Phi_1 \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]$ represents the difference between the consumed and the produced effects after reducing the term t'.

Let us present an example of effect instantiation. We assume that we have the following effect polymorphic function:

$$f : \mathsf{Unit}^{\{\,\alpha\,\}} \to \mathsf{Unit}^{\{\,\}}$$

function f consumes the effect α. The type derivation of applying the function f to unit, inside an effect abstraction, instantiated with effect $IO(1,1)$ is the following:

$$\frac{IO(1,1) \in \textsc{Eff} \qquad \dfrac{\overset{\ldots}{\{\,IO(2,2)\,\}; \cdot; \cdot \vdash (\Lambda\alpha\ .\ f\ \mathsf{unit}) : \forall\alpha.\, {}^{\{\,\alpha\,\}}\mathsf{Unit}^{\{\,\}}; \{\,IO(2,2)\,\}} \qquad \{\,\alpha\,\}[\alpha \to IO(1,1)] \leq_p \{\,IO(2,2)\,\}}{\{\,IO(2,2)\,\}; \cdot; \cdot \vdash (\Lambda\alpha\ .\ f\ \mathsf{unit})\ [IO(1,1)] : \mathsf{Unit}; \Phi'}$$

where:

$$\{\,\alpha\,\}[\alpha \to IO(1,1)] = \{\,IO(1,1)\,\}$$

and, effect set Φ' is defined as:

$$\Phi' = \{\,IO(2,2)\,\} \dot{-} \{\,IO(1,1)\,\} + \{\,\} = \{\,IO(1,1)\,\}$$

therefore, the instantiation is going to consume the effect $IO(1,1)$:

$$\{\,IO(2,2)\,\}; \cdot; \cdot \vdash (\Lambda\alpha\ .\ f\ \mathsf{unit})\ [IO(1,1)] : \mathsf{Unit}; \{\,IO(1,1)\,\}$$

The term $f\ \mathsf{unit}$ is going to consume the effect set $\{\,IO(1,1)\,\}$ after been instantiated with the effect $IO(1,1)$, and it is not going to produce any effect set. The resulting effect set of the instantiation is $\{\,IO(1,1)\,\}$.

4.1.6 Minimum Effect Set

Figure 4.11 presents the extension of the minimum effect set function ($\rightsquigarrow$) where the effect context Δ is added to the judgment $\Gamma, \Delta \vdash t \rightsquigarrow \Phi$.

Like any other value, an effect abstraction does not require any effect in order to be typed[2]. Rule (SMins) defines the smallest set for effect instantiation $t\ [\mathrm{e}]$, Φ_t is the smallest set that

[2]Defined by the rule (SMv) in Chapter 3.1.

...

$$\text{(E-Poly)}\frac{}{\langle(\Lambda\alpha\ .\ t)\ [\mathrm{e}], \Phi\rangle \to \langle t[\alpha \to \mathrm{e}], \Phi\rangle}$$

Figure 4.12: Notions of reductions

requires t, $\Phi_c[\alpha \to \mathrm{e}]$ is the smallest set that requires the body of the effect abstraction t, where it is instantiated with the effect e, and Φ'_t is the set produced by typing t with the smallest set.

4.2 Dynamic Semantics

We extend the dynamic semantics of λ_{mme} by adding a rule for reduction of effect instantiation.

Rule (E-Poly), defined in Figure 4.12, reduces the expression $(\Lambda\alpha\ .\ t)\ [\mathrm{e}]$ by applying the substitution $t[\alpha \to \mathrm{e}]$.

Consider the expression defined in the previous section:

$$(\Lambda\alpha\ .\ f\ \mathsf{unit})\ [IO(1,1)]$$

Where the reduction of f is defined as $f\ \mathsf{unit} \to_\zeta \mathsf{unit}$ [3], where f applied to unit reduces to unit. The reduction of the expression with initial effect set $\{\ IO(2,2)\ \}$ is:

$$\langle(\Lambda\alpha\ .\ f\ \mathsf{unit})\ [IO(1,1)], \{\ IO(2,2)\ \}\rangle \mapsto$$

$$\langle f\ \mathsf{unit}, \{\ IO(2,2)\ \}\rangle \mapsto$$

$$\langle \mathsf{unit}, \{\ IO(1,1)\ \}\rangle$$

4.3 Properties

In this chapter we define the soundness property for λ^p_{mme} . Like λ_{mme}, we defined soundness as a sound approximation of the effects of a well-typed program. Thus a program that typechecks is going to consume less privileges and fulfill more obligations than the type system can calculate. Formally, soundness is defined as:

Theorem 4.1. *Soundness*

$$\Phi; \cdot; \cdot \vdash t : T; \Phi_s \text{ and } \langle t, \Phi\rangle \mapsto^* \langle v, \Phi_{\mathrm{d}}\rangle \text{ then } \Phi_s \sqsubseteq \Phi_{\mathrm{d}}$$

[3] The function $\to_\zeta$, introduced in Chapter 2, defines how a primitive reduces.

We express the soundness property by using a logical predicate. Logical Predicates are special cases of *logical relations* where the property of interest is unary [10]. The logical predicate is presented in Figure 4.15. We define the logical predicate using two mutually defined predicates: predicates for values $\mathcal{V}[\![T]\!]$ and a predicate for terms $\mathcal{E}[\![T]\!]$.

Predicate $\mathcal{V}[\![T]\!]$ is defined by cases on the type. A Value v satisfies the predicate $v : \mathcal{V}[\![B]\!]$ if value v, under any effect set, has type B. Value $\{\, u_1, u_2 \,\} :: T_1 \times T_2$ satisfies the predicate $(\{\, u_1, u_2 \,\} :: T_1 \times T_2) : \mathcal{V}[\![T_1 \times T_2]\!]$ if the first projection u_1 ascribed to T_1 satisfies the predicate $(u_1 :: T_1) : \mathcal{V}[\![T_1]\!]$ and the second projection $u.2$ ascribed to T_2 satisfies the predicate $(u_2 :: T_2) : \mathcal{V}[\![T_2]\!]$. A value v satisfies predicate $v : \mathcal{V}[\![{T_1}^{\Phi_c} \to {T_2}^{\Phi_p}]\!]$ if value v, under any effect set, has type ${T_1}^{\Phi_c} \to {T_2}^{\Phi_p}$ and for any value v' that satisfies the predicate $v' : \mathcal{V}[\![T_1]\!]$, term $v\ v'$, under effect set Φ_c, satisfies predicate $\mathcal{E}[\![T_2]\!]$. The value v satisfies the predicate $v : \mathcal{V}[\![\forall\alpha.\, {}^{\Phi_c}T^{\Phi_p}]\!]$ if value v, under any effect set has type $\forall\alpha.\, {}^{\Phi_c}T^{\Phi_p}$, and for any effect e that belongs to Eff (defined on subsection 3.1.1), term v [e] (instantiation of v' with the effect e), under the substituted effect set $\Phi_c[\alpha \to \mathrm{e}]$, satisfies the predicate $\mathcal{E}[\![T_2[\alpha \to \mathrm{e}]]\!]$.

The predicate for term $\Phi \vdash t : \mathcal{E}[\![T]\!]$ defines that under the effect context Φ, the term t satisfies the predicate for the type T if for any effect set Φ' that has more privileges than Φ, term t starting with effect set Φ', has type T and produces the effect set Φ_s as output. The configuration of term t and effect set Φ' after zero or more steps reaches the configuration: value v and effect set Φ_d. In addition, the value v satisfies the predicate $\mathcal{V}[\![T]\!]$ and the effect set Φ_d contains the static effect set Φ_s, i.e. Φ_s is a conservative approximation of Φ_d. Using the logical predicate, we define semantic typing as:

Definition 4.1 (Semantic Typing)**.**

$$\Phi; \Gamma; \Delta \models t : T \stackrel{\text{def}}{=} \forall\gamma \in \mathcal{G}[\![\Gamma]\!] \text{ and } \forall\delta \in \mathcal{G}[\![\Delta]\!]$$
$$\text{then } \delta(\Phi) \vdash \gamma(\delta(t)) : \mathcal{E}[\![\delta(T)]\!]$$

The definition states that under any *good* substitutions of term variables and effect variables, term t satisfies the predicate $\mathcal{E}[\![T]\!]$ given the effect set Φ.

We use for the definition of semantic typing γ as a substitution function for term variables and δ as a substitution function for effect variables. We require that these functions belong to the interpretations for term context Γ and effect context Δ, respectively. We define the interpretation of Γ as follows:

Definition 4.2 (Term Context Interpretation)**.**

$$\mathcal{G}[\![\cdot]\!] = \{\, \emptyset \,\}$$
$$\mathcal{G}[\![\Gamma, x : T]\!] = \{\, \gamma[x \to v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \quad \text{and } v : \mathcal{V}[\![T]\!] \,\}$$

The definition has two cases: for the empty context the substitution function is empty; and for a non-empty context $\Gamma, (x : T)$, the extension of the substitution function γ belongs to the substitution interpretation $\gamma[x \to v] \in \mathcal{G}[\![\Gamma, (x : T)]\!]$ if the value v satisfies the predicate $\mathcal{V}[\![T]\!]$ and γ belongs to the interpretation of the previous context $\mathcal{G}[\![\Gamma]\!]$. i.e. a substitution function γ maps only variables of type T to values that satisfy the predicate $\mathcal{V}[\![T]\!]$.

$$\frac{}{\mathsf{noObligations}(\{\ \})} \qquad \frac{\mathsf{noObligations}(\Phi)}{\mathsf{noObligations}(\Phi, \mathrm{e}(0, n))}$$

Figure 4.13: No obligations

We define effect context interpretation as follows:

Definition 4.3 (Effect Context Interpretation)**.**

$$\mathcal{G}[\![\cdot]\!] = \{\, \emptyset \,\}$$
$$\mathcal{G}[\![\Delta, \alpha]\!] = \{\, \delta[\alpha \to \mathrm{e}] \mid \delta \in \mathcal{G}[\![\Delta]\!] \text{ and } \mathrm{e} \in \textsc{Eff} \,\}$$

Effect context interpretation is defined analogously to the term context interpretation. We define effect context interpretation as follows: a substitution function δ maps only effect variables to effects that are in the set Eff(defined on subsection 3.1.1).

We now define the fundamental property:

Proposition 4.1 (Fundamental Property)**.**

$$\text{If } \ \Phi; \Gamma; \Delta \vdash t : T; \Phi' \ \text{ then } \ \Phi; \Gamma; \Delta \models t : T$$

The fundamental property says that: if a term t statically type-checks then the term t also semantically type-checks. Semantically type-checking means that the term, under any good substitution of its contexts, satisfies the logical predicate. Additionally, satisfying the logical predicates implies that the term t is sound. Therefore, if a term t statically typechecks then the term t is sound.

Using soundness we can define two corollaries:

Corollary 4.1 (Dutiful Programs)**.**

$$\text{If } \ \Phi; \Gamma; \Delta \vdash t : T; \Phi_s \text{ and } \mathsf{noObligations}(\Phi_s) \ \text{ and } \ \langle t, \Phi \rangle \mapsto^* \langle t', \Phi_{\mathrm{d}} \rangle \ \text{ and } \mathsf{irred}(t') \text{ then } \mathsf{noObligations}(\Phi_{\mathrm{d}})$$

The predicate $\mathsf{noObligations}(\Phi)$, defined in Figure. 4.13, states that all obligations from Φ are equal to 0. $\mathsf{irred}(t')$, defined in Figure. 4.14, states that is not possible for t' to do another step. The first corollary is defined as follows: if a term t typechecks and fulfills all its obligation statically, then after zero or more steps the program will reach an irreducible term t', and that term fulfills all its obligations dynamically.

Corollary 4.2 (Type Safety)**.**

$$\text{If } \ \Phi; \Gamma; \Delta \vdash t : T; \Phi_s \ \text{ and } \ \langle t, \Phi \rangle \mapsto^* \langle v, \Phi_{\mathrm{d}} \rangle \ \text{ then } \{\ \}; \Gamma; \Delta \vdash v : T; \{\ \}$$

$$\mathsf{irred}(t) = \nexists t'.t \mapsto t'$$

Figure 4.14: Irreducible

$$\begin{array}{rcl}
v : \mathcal{V}[\![B]\!] & \iff & \forall \Phi \,.\, \Phi; \cdot; \cdot \vdash v : B; \Phi \\
(\{\, u_1, u_2 \,\} :: T_1 \times T_2) : \mathcal{V}[\![T_1 \times T_2]\!] & \iff & (u_1 :: T_1) : \mathcal{V}[\![T_1]\!] \text{ and } (u_2 :: T_2) : \mathcal{V}[\![T_2]\!] \\
v : \mathcal{V}[\![{T_1}^{\Phi_c} \to {T_2}^{\Phi_p}]\!] & \iff & \forall \Phi \,.\, \Phi; \cdot; \cdot \vdash v : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi \text{ and} \\
 & & \forall\, v' \,.\, v' : \mathcal{V}[\![T_1]\!] \,. \\
 & & \Phi_c \vdash v\ v' : \mathcal{E}[\![T_2]\!] \\
v : \mathcal{V}[\![\forall \alpha.\, {}^{\Phi_c}T^{\Phi_p}]\!] & \iff & \Phi \,.\, \Phi; \cdot; \cdot \vdash v : \forall \alpha.\, {}^{\Phi_c}T^{\Phi_p}; \Phi \text{ and} \\
 & & \forall \mathrm{e} \in \mathrm{Eff} \,.\, \Phi_c[\alpha \to \mathrm{e}] \vdash v\ [\mathrm{e}] : \mathcal{E}[\![T[\alpha \to \mathrm{e}]]\!] \\
\Phi \vdash t : \mathcal{E}[\![T]\!] & \iff & \forall \Phi' \,.\, \Phi \leq_p \Phi' \,. \\
 & & \exists \Phi_s \,.\, \Phi'; \cdot; \cdot \vdash t : T; \Phi_s \text{ and} \\
 & & \exists \Phi_{\mathrm{d}}, \exists v \,.\, \langle t, \Phi' \rangle \mapsto^* \langle v, \Phi_{\mathrm{d}} \rangle \text{ then} \\
 & & v : \mathcal{V}[\![T]\!] \text{ and } \Phi_s \sqsubseteq \Phi_{\mathrm{d}}
\end{array}$$

Figure 4.15: Logical predicate

The second corollary is defined as follows: if a term t typechecks then after zero or more steps the program will reach a value v of type T. This corollary is type safety, i.e. a well-typed program cannot get stuck.

4.4 Summary

In this chapter we presented λ^p_{mme}, an extension of λ_{mme} with effect polymorphism. In Subsection 4.1.1 we presented the syntax for λ^p_{mme}. Subsection 4.1.3 defines well-formedness of types and effect set. Subsection 4.1.4 extends subtyping of λ_{mme} with effect polymorphic types. Subsection 4.1.2 extends the definition of effects and its operators by adding rules for effect variables. Subsection 4.1.5 provides the typing rules for λ^p_{mme} and examples. Subsection 4.1.6 extends the algorithm with effect instantiations. In Section 4.2 we defined the evaluation of effect instantiation. Section 4.3 presents soundness, the proof sketch, and some corollaries derived by the soundness property.

Chapter 5

Related Work

The study of side effects has had two main approaches: type-and-effect systems and monads. An example of implementation of a type-and-effect system is the exception system of Java [12, 14], where exception is the only side effect that can be captured by the type-and-effect system. Monads [19] are used in Haskell (and Haskell-like languages) [33, 22, 6, 30] and in some foundational work on side effects [31, 24, 25, 15]. For example, the following type of the Haskell's function putStrLn:

$$\mathsf{putStrLn} :: \mathsf{String} \to IO\ ()$$

receives a string and returns an action of type $IO\ ()$ [1]. The type $IO\ ()$ represents an action IO with a value of type (). It is possible to represent a type-and effect-system as a monadic system [34]. In the following section, we discuss different effect systems that allow a stronger level of specification than a classical effect system, and we compare these effect systems to λ^{p}_{mme} .

5.1 Linear Haskell

Linear Haskell [4] is an extension of Haskell with linear types. Linear Haskell supports the linear arrow ($\multimap$) from linear logic [13]. A linear arrow defines linear functions. In Linear Haskell, monads can have an extra parameter, the multiplicity, which represents the amount of times the result can be used. There are only two possibles multiplicities: 1 or ω. The multiplicity 1 means that the result is linear, i.e. the result has to be used only once [2]. The multiplicity ω means that the result is unrestricted, it can be used as many times as necessary.

The paper shows how to implement a simple I/O protocol using linear types. The linear

[1] The type () represents the type Unit in Haskell.

[2] The authors argued and defined what it means to *use* the argument once.

I/O protocol is defined as follows:

$$
\begin{aligned}
&\mathsf{type}\ \mathrm{File}\\
&\mathsf{openFile} :: \mathsf{FilePath} \to IO_l\ 1\ \mathsf{File}\\
&\mathsf{readLine} :: \mathsf{File} \multimap IO_l\ 1\ (\mathsf{File}, \mathsf{UnrestrictedByteString})\\
&\mathsf{closeFile} :: \mathsf{File} \multimap IO_l\ \omega\ ()
\end{aligned}
$$

They define the monad IO_l as the linear version of the IO monad. openFile receives a path to a file and returns a file that must be used. readLine receives a file and returns a linear file and an unrestricted string. closeFile receives a file and returns unit. This protocol offers some guarantees that are not present in a similar protocol implemented in Haskell. For example, this protocol guarantees that the file must be closed at the end of the program, and that it is not possible to read a file after closing it.

Like λ^p_{mme} , the effect system of Linear Haskell allows the programmer to specify stronger specifications than a standard type system. λ^p_{mme} , unlike Linear Haskell, makes the distinction between privileges and obligations, thus allowing λ^p_{mme} to express stronger specifications than Linear Haskell. The main advantage of Linear Haskell over λ^p_{mme} is that it has a real implementation while λ^p_{mme} is a formal calculus.

5.2 Idris

Idris is a programming language with dependent types [5]. Idris deals with side effects in two ways: the first one is using monads, like Haskell [3]. The second one is using the library Effects [6]. Effects uses the algebraic representation of effects [3], where effects are viewed as resulting from operations and dealt with using effect handlers [26, 27]. An effect handler is similar to a catch expression from the Java language. Let us consider the type of a function that generates a random number in Effects:

$$\mathsf{rndInt} : \mathsf{Integer} \to \mathsf{Integer} \to \{[\mathsf{RND}]\}\ \mathsf{Eff}\ \mathsf{Integer}$$

The function rndInt receives the lower and upper bounds, returns a random integer between these bounds, and the function produces the effect RND.

Idris has dependent types. These types allow Idris to also support dependent effects, which can express operations that can change the available effects. An example of the use of dependent effects are resource protocols. For example, the effect FILE_IO is parameterized over the state of the file handler which is either empty, open for reading, or open for writing. Therefore, the function close has type:

$$\mathsf{close} : \mathsf{String} \to \{[\mathsf{FILE\_IO}(\mathsf{OpenFile})] \Longrightarrow [\mathsf{FILE\_IO}()]\}\ \mathsf{Eff}\ ()$$

close receives the name of the file, returns unit and the initial state of the file is open, and after executing the function the state of the file is empty, which denotes a closed file. The

[3] Idris was developed in Haskell.

type enforces that when the function is applied, it changes the state of the file handler from *open* to *close*. Therefore it would not be possible to read the file again after the file had been closed.

With Idris it is possible to express and verify usage protocols by associating resources with each effect. This is similar to typestates [1, 28], which allow tracking states of resources in types. Idris allows us to ensure that if a function starts in an initial state, it will finish in a given final state. That is different from what the type-and-effect system of λ^{p}_{mme} can ensure. The type-and-effect system of λ^{p}_{mme} ensures that a function will fulfill its obligations, but cannot ensure the initial or final state of the resource. For example, a function that has the obligation of opening and closing a file, may first close an open file and then open the file, thus leaving the file open. Idris cannot ensure that an action is actually performed. Also Idris having dependent effects, it can express stronger specification than are currently possible with λ^{p}_{mme} .

5.3 First-Class Effects

Yuheng Long et al. defined λ_{fe} a calculus with first class effects [16]. λ_{fe} has an hybrid type and effect system, meaning that it uses run-time information to improve the precision of the type system. λ_{fe} calculates the may-effects and the must-effects: the may-effects are the over approximation of effects that a program at most performs, and the must-effects are the lower-approximation of effects that a program at least performs. In λ_{fe} the effects are only region accesses and have the form π_{ρ}, where π represents region access: *allocation*, *read*, and *write*. ρ represents the regions [4]. For instance, the following effect:

$$wr_{a}$$

represents write effect in region a. The type judgment of λ_{fe} is defined as $\Gamma \vdash \text{e} : T, \sigma \sim \sigma'$, where under context Γ the expression e has type T, must effects σ and may effects σ'. The may-must effects of λ_{fe} are similar to the ones of λ^{p}_{mme} : they represent the privileges and obligations of the program, but the interpretation of effects of both languages are different: in λ_{fe} effects are capabilities, they only represent the capacity of writing or reading a region of memory, but in λ^{p}_{mme} the effect represent any effectful actions.

First-class effects allow the programmer to define programs that are not expressible in λ^{p}_{mme} because effects are values. Therefore it is possible for the programmer to assign, compare, or even pattern-matching effects. λ^{p}_{mme} , instead, allows to keep track of the number of times an action may or must be performed; that possibility is not possible in λ_{fe}. λ_{fe} has hybrid typing, i.e. the language performs some type-checking at run time. The type-and-effect system of λ^{p}_{mme} is fully static: the dynamic checks in the dynamic semantic of λ^{p}_{mme} are only used in the soundness proof, and can be erased in an implementation.

Another relation between both systems is the soundness property. λ_{fe} defines the concept of *trace consistency*, where that definition is similar to the idea of sound approximation of effects of λ^{p}_{mme} : the static effects are a sound approximation of the dynamic effects.

[4] In the literature, there are works that use type-and-effect systems as capabilities [7, 11, 32]

5.4 Summary

In this chapter we discussed some effect systems that also allow to specify stronger properties than a simple effect system.

Section 5.1 presented Linear Haskell, an extension of Haskell with linear types. Linear Haskell can be used to implement a file protocol, similar to the one presented in the introduction chapter of this thesis. The protocol that can be implemented with Linear Haskell ensures different properties than the one implemented in λ^{p}_{mme} .

Section 5.2 showed Effects, an library for algebraic effect on the programming language Idris. Like Linear Haskel, with Effects it is possible to write a file protocol, but that file protocol gives stronger guarantees than the one implemented in λ_{mme} (or the one implemented in Linear Haskell).

Section 5.3 presented λ_{fe}, a language with first-class effects. λ_{fe} shares some concepts with λ^{p}_{mme} , like may-must effects or the definition of effect soundness. Both languages use different design approaches. λ_{fe} is designed in order to support first-class effects, while λ^{p}_{mme} takes inspiration from linear type systems in order to check the obligations in the type-and-effect system.

Chapter 6

Conclusions

6.1 Contributions

In this work we have designed and formalized λ^{p}_{mme} , a type and effect system with polymorphic and may-must effects. First, we introduced SFL, a simple functional language. We introduced its type system, dynamic semantics, properties and its limitations. Second, we showed an extension with side effects (SFLE), where SFLE allows the programmer to annotate the effects that may occur when the program executes. Third, we defined λ_{mme} whose type-and-effect system supports may-must effects. May-must effects are different from the effects presented in SLFE. Finally, we extended λ_{mme} with effect polymorphism; we called that language λ^{p}_{mme} . Effect polymorphism allows the programmer to abstract effectful programs that share some logic.

We defined and proved soundness of the language. We defined soundness as a sound approximation of the effects of a well-typed program. Therefore, a program that typechecks is going to consume less privileges and fulfill more obligations than predicted statically. For the proof of soundness we use logical predicates.

The type-and-effect system of λ^{p}_{mme} allows the programmer to enforce stronger specifications than a simple type and effect system (like SFLE). We presented an application of may-must effect that guarantees that a program is going to perform a given effectful operation.

The type-and-effect system of λ^{p}_{mme} , as any static type system, will detect errors before executing a 'faulty' program, but this type-and-effect system is not yet useful for a programmer. The language has limitations that prevent its use in the real world.

6.2 Limitations and Future Work

A limitation of λ^{p}_{mme} is that the type system cannot enforce the order in which effects are consumed or produced. For example, if a program has the privileges to open and close a file, λ^{p}_{mme} cannot enforce that the program will first open the file and then close it.

Another limitation is that λ^{p}_{mme} lacks recursion. In order to add recursion to the language, we should add the fix operator. The type, in simple types, for the fix operator is defined as follows:

$$\text{Tfix}\frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \text{fix}[t] : T}$$

Extending λ^{p}_{mme} with fix raises some questions that we do not know how to answer yet: What are the input and output effects for t? What are the output effects of $\mathsf{fix}[t]$?

The type-and-effect system of λ^{p}_{mme} also lacks an alias analysis. Without an alias analysis we cannot express typestates. For example, with λ^{p}_{mme} we cannot talk about the state of the file, we can only enforce that certain operations have to be performed (or not). We cannot ensure that the file is closed before opening it.

Extending λ^{p}_{mme} with dependency [2, 8], even limited dependency [29] where types can dependent only on values of a certain domain, would allow us to enforce stronger specifications. For example, it could be possible to keep track of the usage of effects on lists. For example, Let us consider the following type $\mathsf{Vector}\ T\ [n]$: a vector is a list of elements of type T and size n. We could introduce the dependent lambda, $(\lambda_{\text{dep}}\ n\ .\ t)$, for binding a natural number to the type and effect sets. The type rule for dependent lambda would be:

$$(\text{T}\lambda_{\text{d}}p)\frac{\begin{array}{c}\Gamma, (n : \mathsf{Nat}), \Delta \vdash t \rightsquigarrow \Phi_c \\ \Phi_c; \Gamma, (n : \mathsf{Nat}); \Delta \vdash t : T; \Phi_p\end{array}}{\Gamma; \Delta \vdash_u (\lambda_{\text{dep}}\ n\ .\ t) : \Pi n : \mathsf{Nat}\ .\ {}^{\Phi_c}T^{\Phi_p}},$$

Where n can appear in t, type T, and effect sets Φ_c or Φ_p. It would then be possible to type the map function, which given an effectful function of type ${T_1}^{\{\alpha\}} \to {T_2}^{\{\beta\}}$ and a vector of size n, applies the function to all elements of the vector:

$$\mathsf{map} : \forall \alpha, \beta.\ ({T_1}^{\{\alpha\}} \to {T_2}^{\{\beta\}}) \to (\Pi n : \mathsf{Nat}\ .\ \mathsf{Vector}\ T_1\ [n]^{\{n\alpha\}} \to \mathsf{Vector}\ T_2\ [n]^{\{n\beta\}})$$

For readability we omit empty effect sets from the type and we assume that the polymorphic lambda supports multiple arguments. map after instantiating the polymorphic variables α and β, receives an effectful function and returns a dependent lambda. The dependent lambda requires a natural number that binds the body and its type. The body of the polymorphic lambda is an effectful function that receives a vector whose elements have type T_1 and size n, consumes the effect α n times (one for each element of the vector), produces the effect β n times and returns a vector whose elements have type T_2 and size n.

This approach has limitations, it is not possible to guarantee that map only apply the function to all the elements of the vector, map may apply the function only to the first element n times.

Bibliography

[1] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022. ACM, 2009.

[2] David Aspinall and Martin Hofmann. Dependent types. *Advanced Topics in Types and Programming Languages*, pages 45–86, 2005.

[3] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[4] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear haskell: practical linearity in a higher-order polymorphic language. *arXiv preprint arXiv:1710.09756*, 2017.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[6] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *ACM SIGPLAN Notices*, volume 48, pages 133–144. ACM, 2013.

[7] Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM Sigplan Notices*, volume 43, pages 213–224. ACM, 2008.

[8] Adam Chlipala. Certified programming with dependent types, 2011.

[9] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, and Helmut Veith. *Model checking*. Cyber-Physical Systems, 2018.

[10] Karl Crary. Logical relations and a case study in equivalence checking. *Advanced Topics in Types and Programming Languages*, pages 223–244, 2005.

[11] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM, 1999.

[12] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. *ACM SIGPLAN Notices*, 36(5):59–69, 2001.

[13] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

[14] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. Addison-Wesley Professional, 2000.

[15] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In *International Conference on Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999.

[16] Yuheng Long, Yu David Liu, and Hridesh Rajan. First-class effect reflection for effect-guided programming. In *ACM SIGPLAN Notices*, volume 51, pages 820–837. ACM, 2016.

[17] John M Lucassen and David K Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–57. ACM, 1988.

[18] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[19] Eugenio Moggi. Computational lambda-calculus and monads. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 14–23, 1989.

[20] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[21] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.

[22] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[23] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[24] Gordon Plotkin and John Power. Computational effects and operations: An overview. 2002.

[25] Gordon Plotkin and John Power. Notions of computation determine monads. In *International Conference on Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002.

[26] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

[27] Matija Pretnar. Logic and handling of algebraic effects. 2010.

[28] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1):157–171,

1986.

[29] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ACM SIGPLAN Notices*, volume 46, pages 266–278. ACM, 2011.

[30] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the Dijkstra monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[31] Ross Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[32] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, 1997.

[33] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[34] Philip Wadler. The marriage of effects and monads. *SIGPLAN Not.*, 34(1):63–74, September 1998.

List of Lemmas, Propositions, Theorems, and Definitions

Appendix A

Proof

Lemma A.1 (Canonical Forms)**.**

- If u is a core value of type Bool, then u is true or false.
- If u is a core value of type Unit, then u is unit.
- If u is a core value of type $T_1 \times T_2$, then u is a pair $\{ u_1, u_2 \}$.
- If u is a core value of type ${T_1}^{\Phi_c} \to {T_2}^{\Phi_p}$, then u is $(\lambda^{\Phi_c} x : T_1 \,.\, t)$ or it is op.
- If u is a core value of type $\forall\alpha.\, {}^{\Phi_c}T^{\Phi_p}$, then u is $(\Lambda\alpha \,.\, t)$.

Proof. Direct by the syntax defined in the Figure 4.1 □

Proposition A.1 (Monotonicity of the Typing)**.**

$$\text{If } \Phi_a; \Gamma; \Delta \vdash t : T; \Phi'_a \text{ and } \forall \Phi_b \,.\, \Phi_a \leq_p \Phi_b \text{ then } \Phi_b; \Gamma; \Delta \vdash t : T; \Phi'_b \text{ and } \Phi'_a \leq_p \Phi'_b$$

Proof. By induction on the typing rules

Case (Values and Variables). By Lemma.A.3 (Typing values with any effect set) we know that the effect set is preserved. The same reasoning applies to variables.

Case (Pair). We have to show that:

$$\begin{gathered}\text{If } \Phi_a; \Gamma; \Delta \vdash \{ t_1, t_2 \} : T_1 \times T_2; \Phi''_a \\ \text{then } \Phi_b; \Gamma; \Delta \vdash \{ t_1, t_2 \} : T_1 \times T_2; \Phi''_b \text{ and } \Phi''_a \leq_p \Phi''_b\end{gathered}$$

Our induction hypothesis are:

$$\begin{gathered}\Phi_a; \Gamma; \Delta \vdash t_1 : T_1; \Phi'_a \\ \Phi'_a; \Gamma; \Delta \vdash t_2 : T_2; \Phi''_a\end{gathered}$$

Thus for any Φ_b that has more privileges than Φ_a, applying the induction hypothesis, we know that:

$$\begin{gathered}\Phi_b; \Gamma; \Delta \vdash t_1 : T_1; \Phi'_b \text{ and } \Phi'_a \leq_p \Phi'_b \\ \Phi'_b; \Gamma; \Delta \vdash t_2 : T_2; \Phi''_b \text{ and } \Phi''_a \leq_p \Phi''_b\end{gathered}$$

Therefore:

$$\Phi_b; \Gamma; \Delta \vdash \{\, t_1, t_2 \,\} : T_1 \times T_2; \Phi''_b$$

Case (Application). We have to show that:

$$\begin{gathered}\text{If } \Phi_a; \Gamma; \Delta \vdash t_1\ t_2 : T_2; \Phi'''_a \\ \text{then } \Phi_b; \Gamma; \Delta \vdash t_1\ t_2 : T_2; \Phi'''_b \text{ and } \Phi'''_a \leq_p \Phi'''_b\end{gathered}$$

Our induction hypothesis are:

$$\begin{gathered}\Phi_a; \Gamma; \Delta \vdash t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi'_a \\ \Phi'_a; \Gamma; \Delta \vdash t_2 : T_1; \Phi''_a \\ \Phi'''_a = \Phi''_a \dot{-} \Phi_c + \Phi_p\end{gathered}$$

Thus for any Φ_b that has more privileges than Φ_a, applying the induction hypothesis, we know that:

$$\begin{gathered}\Phi_b; \Gamma; \Delta \vdash t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi'_b \text{ and } \Phi'_a \leq_p \Phi'_b \\ \Phi'_b; \Gamma; \Delta \vdash t_2 : T_2; \Phi''_b \text{ and } \Phi''_a \leq_p \Phi''_b\end{gathered}$$

And because the algebraic operations are monotonous, we know that:

$$\begin{gathered}\Phi''_a \leq_p \Phi''_b \\ \Phi''_a \dot{-} \Phi_c + \Phi_p \leq_p \Phi''_b \dot{-} \Phi_c + \Phi_p \\ \Phi'''_a \leq_p \Phi'''_b\end{gathered}$$

Therefore:

$$\Phi_b; \Gamma; \Delta \vdash t_1\ t_2 : T_2; \Phi'''_b \text{ and } \Phi'''_a \leq_p \Phi'''_b$$

Case (If). We have to show that:

$$\begin{gathered}\text{If } \Phi_a; \Gamma; \Delta \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T; \Phi_{2a} \Cap \Phi_{3a} \\ \text{then } \Phi_b; \Gamma; \Delta \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T; \Phi_{2b} \Cap \Phi_{3b} \text{ and } (\Phi_{2a} \Cap \Phi_{3a}) \leq_p (\Phi_{2b} \Cap \Phi_{3b})\end{gathered}$$

Our induction hypothesis are:

$$\begin{gathered}\Phi_a; \Gamma; \Delta \vdash t_1 : \mathsf{Bool}; \Phi_{1a} \\ \Phi_{1a}; \Gamma; \Delta \vdash t_2 : T; \Phi_{2a} \\ \Phi_{1a}; \Gamma; \Delta \vdash t_2 : T; \Phi_{3a}\end{gathered}$$

Thus for any Φ_b that has more privileges than Φ_a, applying the induction hypothesis, we know that:

$$\begin{gathered}\Phi_{1b}; \Gamma; \Delta \vdash t_1 : T_1; \Phi_{1b} \text{ and } \Phi_{1a} \leq_p \Phi_{1b} \\ \Phi_{1b}; \Gamma; \Delta \vdash t_2 : T_2; \Phi_{2b} \text{ and } \Phi_{2a} \leq_p \Phi_{2b} \\ \Phi_{1b}; \Gamma; \Delta \vdash t_2 : T; \Phi_{3b} \text{ and } \Phi_{3a} \leq_p \Phi_{3b}\end{gathered}$$

We have to show that:

$$(\Phi_{2a} \Cap \Phi_{3a}) \leq_p (\Phi_{2b} \Cap \Phi_{3b})$$

Which is true, because $\Cap$ is defined as the minimum of privileges between two sets, and the minimum of privileges between two sets that contain less privileges than the set that contain more.

Therefore:

$$\Phi_b; \Gamma; \Delta \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T; \Phi_{2b} \Cap \Phi_{3b} \text{ and } (\Phi_{2a} \Cap \Phi_{3a}) \leq_p (\Phi_{2b} \Cap \Phi_{3b})$$

The rest of the cases are direct by induction hypothesis. □

Lemma A.2 (Intersection Relation Components)**.**

$$\text{If } \Phi_1 \Cap \Phi_2 \text{ then } \Phi_1 \Cap \Phi_2 \sqsubseteq \Phi_1 \text{ and } \Phi_1 \Cap \Phi_2 \sqsubseteq \Phi_2$$

Proof. By definition of intersection □

Theorem A.1 (Soundness)**.**

$$\Phi; \cdot; \cdot \vdash t : T; \Phi_s \text{ and } \langle t, \Phi \rangle \mapsto^* \langle v, \Phi_{\mathrm{d}} \rangle \text{ then } \Phi_s \sqsubseteq \Phi_{\mathrm{d}}$$

Proof. By the theorem A.2 (Fundamental Property), where we instantiate Γ and Δ as empty. □

Definition A.1 (Logical Predicate)

$$\begin{array}{rcl}
v : \mathcal{V}[\![B]\!] & \iff & \forall \Phi\ .\ \Phi; \cdot; \cdot \vdash v : B; \Phi \\
(\{\, u_1, u_2 \,\} :: T_1 \times T_2) : \mathcal{V}[\![T_1 \times T_2]\!] & \iff & (u_1 :: T_1) : \mathcal{V}[\![T_1]\!] \text{ and } (u_2 :: T_2) : \mathcal{V}[\![T_2]\!] \\
v : \mathcal{V}[\![{T_1}^{\Phi_c} \to {T_2}^{\Phi_p}]\!] & \iff & \forall \Phi\ .\ \Phi; \cdot; \cdot \vdash v : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi \text{ and} \\
 & & \forall\ v'\ .\ v' : \mathcal{V}[\![T_1]\!]\ . \\
 & & \Phi_c \vdash v\ v' : \mathcal{E}[\![T_2]\!] \\
v : \mathcal{V}[\![\forall \alpha.\, {}^{\Phi_c} T^{\Phi_p}]\!] & \iff & \Phi\ .\ \Phi; \cdot; \cdot \vdash v : \forall \alpha.\, {}^{\Phi_c} T^{\Phi_p}; \Phi \text{ and} \\
 & & \forall \mathrm{e} \in \mathit{Eff}\ .\ \Phi_c[\alpha \to \mathrm{e}] \vdash v\ [\mathrm{e}] : \mathcal{E}[\![T[\alpha \to \mathrm{e}]]\!] \\
\Phi \vdash t : \mathcal{E}[\![T]\!] & \iff & \forall \Phi'\ .\ \Phi \leq_p \Phi'\ . \\
 & & \exists \Phi_s\ .\ \Phi'; \cdot; \cdot \vdash t : T; \Phi_s \text{ and} \\
 & & \exists \Phi_{\mathrm{d}}, \exists v\ .\ \langle t, \Phi' \rangle \mapsto^* \langle v, \Phi_{\mathrm{d}} \rangle \text{ then} \\
 & & v : \mathcal{V}[\![T]\!] \text{ and } \Phi_s \sqsubseteq \Phi_{\mathrm{d}}
\end{array}$$

Proposition A.2 (Preservation)**.**

$$\text{If } \Phi_a; \Gamma; \Delta \vdash t : T; \Phi'_a \text{ and } \langle t, \Phi_a \rangle \to \langle t', \Phi_b \rangle \text{ then}$$
$$\Phi_b; \Gamma; \Delta \vdash t' : T; \Phi'_b \text{ and } (\Phi'_a \sqsubseteq \Phi'_b)$$

Proof. By case on the operational semantics:

Case (E-APP).

$$\text{(E-App)}\frac{}{\langle(\lambda x : T_1 \,.\, t)\ v, \Phi\rangle \to \langle t[x \to v], \Phi\rangle}$$

We know that:

$$\Phi_a; \Gamma; \Delta \vdash (\lambda x : T_1 \,.\, t)\ v : T_2; \Phi_a \dot{-} \Phi_c + \Phi_p$$
$$\Phi_c \leq_p \Phi_a$$
$$\langle(\lambda x : T_1 \,.\, t)\ v, \Phi_a\rangle \to \langle t[x \to v], \Phi_a\rangle$$

We need to show that:

$$\Phi_a; \Gamma; \Delta \vdash t[x \to v] : T_2; \Phi'_a$$
$$(\Phi_a \dot{-} \Phi_c + \Phi_p) \sqsubseteq \Phi'_a$$

We know that:

$$\Phi_c; \Gamma, (x : T_1); \Delta \vdash t : T_2; \Phi_p$$

By Lemma A.5 (Substitution Lemma), we know that:

$$\Phi_c; \Gamma; \Delta \vdash t[x \to v] : T_2; \Phi_p$$

Because we know that Φ_a has more privileges than Φ_c, and by Proposition A.1 (Monotonicity of the Typing) we know that:

$$\Phi_a; \Gamma; \Delta \vdash t[x \to v] : T_2; \Phi'_a$$

By definition, we know that t is going to consume Φ_c and is going to produce Φ_p. Therefore, we know that:

$$\Phi'_a = (\Phi_a \dot{-} \Phi_c + \Phi_p)$$

Thus:

$$(\Phi_a \dot{-} \Phi_c + \Phi_p) \sqsubseteq (\Phi_a \dot{-} \Phi_c + \Phi_p)$$

Case (E-IFT).

$$\text{(E-IfT)}\frac{}{\langle \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_t, \Phi\rangle}$$

We know that:

$$\Phi_a; \Gamma; \Delta \vdash \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f : T; \Phi'_a$$
$$\langle \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f, \Phi_a\rangle \to \langle t_t, \Phi_a\rangle$$

We need to show that:

$$\Phi_a; \Gamma; \Delta \vdash t_t : T'; \Phi_t$$
$$\Phi'_a \sqsubseteq \Phi_t$$

By the premise, we know that:

$$\frac{\begin{array}{c}\Phi_a; \Gamma; \Delta \vdash \mathsf{true} : \mathsf{Bool}; \Phi_a \\ \Phi_a; \Gamma; \Delta \vdash t_t : T; \Phi_{1a} \\ \Phi_a; \Gamma; \Delta \vdash t_f : T; \Phi_{2a}\end{array}}{\Phi_a; \Gamma; \Delta \vdash \mathsf{if\ true\ then}\ t_t\ \mathsf{else}\ t_f : T; \Phi_{1a} \Cap \Phi_{2a}}$$

Where:

$$\Phi'_a \equiv \Phi_{1a} \Cap \Phi_{2a}$$

Therefore, we know that:

$$\Phi_a; \Gamma; \Delta \vdash t_t : T; \Phi_{1a}$$

thus:

$$\Phi_t \equiv \Phi_{1a}$$

By Lemma. A.2 we know that:

$$(\Phi_{1a} \Cap \Phi_{2a}) \sqsubseteq \Phi_{1a}$$

Case (E-IFF).

$$\text{(E-IfF)}\frac{}{\langle \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f, \Phi\rangle \to \langle t_f, \Phi\rangle}$$

Homologous to the case before, we know that:

$$\begin{array}{c}\Phi_a; \Gamma; \Delta \vdash \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f : T; \Phi'_a \\ \langle \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f, \Phi_a\rangle \to \langle t_f, \Phi_a\rangle\end{array}$$

We need to show that:

$$\begin{array}{c}\Phi_a; \Gamma; \Delta \vdash t_f : T'; \Phi_f \\ \Phi'_a \sqsubseteq \Phi_f\end{array}$$

By the premise, we know that:

$$\frac{\begin{array}{c}\Phi_a; \Gamma; \Delta \vdash \mathsf{false} : \mathsf{Bool}; \Phi_a \\ \Phi_a; \Gamma; \Delta \vdash t_t : T; \Phi_{1a} \\ \Phi_a; \Gamma; \Delta \vdash t_f : T; \Phi_{2a}\end{array}}{\Phi_a; \Gamma; \Delta \vdash \mathsf{if\ false\ then}\ t_t\ \mathsf{else}\ t_f : T; \Phi_{1a} \Cap \Phi_{2a}}$$

Where:

$$\Phi'_a \equiv \Phi_{1a} \Cap \Phi_{2a}$$

Therefore, we know that:

$$\Phi_a; \Gamma; \Delta \vdash t_f : T; \Phi_{2a}$$

thus:

$$\Phi_f \equiv \Phi_{2a}$$

By Lemma. A.2 we know that:

$$(\Phi_{1a} \Cap \Phi_{2a}) \sqsubseteq \Phi_{2a}$$

Case (E-OP).

$$\text{(E-Op)} \frac{\begin{array}{c}\text{op } v_1 \to_\zeta v_2 \\ \zeta(\text{op}) = {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}\end{array}}{\langle \text{op } v_1, \Phi \rangle \to \langle v_2, \Phi \dot{-} \Phi_c + \Phi_p \rangle}$$

By definition we know that v_2 types, and because it is a value, can type with any effect set:

$$\Phi'; \Gamma; \Delta \vdash v_2 : T_2; \Phi'$$

We have to show that:

$$\Phi \dot{-} \Phi_c + \Phi_p \sqsubseteq \Phi \dot{-} \Phi_c + \Phi_p$$

Which it is true.
Case (E-Proj1).

$$\text{(E-Proj1)} \frac{}{\langle \{\, u_1, u_2 \,\}.1, \Phi \rangle \to \langle u_1, \Phi \rangle}$$

We know that:

$$\begin{array}{c}\Phi_a; \Gamma; \Delta \vdash (\{\, u_1, u_2 \,\} :: T_1 \times T_2).1 : T_1; \Phi_a \\ \langle (\{\, u_1, u_2 \,\} :: T_1 \times T_2).1, \Phi_a \rangle \to \langle u_1 :: T_1, \Phi_a \rangle\end{array}$$

Let us define v_1 as $u_1 :: T_1$. We need to show that:

$$\Phi_a; \Gamma; \Delta \vdash v_1 : T_1; \Phi'_a$$

Because we know that by the premise that the pair types, we know that v_1 types, and we know that v_1 is a value, thus:

$$\begin{array}{c}\Phi_a = \Phi'_a \\ \Phi_a; \Gamma; \Delta \vdash v_1 : T_1; \Phi_a\end{array}$$

Case (E-Proj2).

$$\text{(E-Proj2)}\frac{}{\langle \{\, u_1, u_2 \,\}.2, \Phi\rangle \to \langle u_2, \Phi\rangle}$$

We apply the same reasoning of the previous case:

$$\Phi_a; \Gamma; \Delta \vdash (\{\, u_1, u_2 \,\} :: T_1 \times T_2).2 : T_2; \Phi_a$$
$$\langle (\{\, u_1, v_2 \,\} :: T_1 \times T_2).2, \Phi_a\rangle \to \langle u_2 :: T_2, \Phi_a\rangle$$

Case (E-Asc.).

$$\text{(E-Asct)}\frac{}{\langle v :: T_1 :: T_2, \Phi\rangle \to \langle v :: T_2, \Phi\rangle}$$

We know that:

$$\Phi_a; \Gamma; \Delta \vdash v :: T_1 :: T_2 : T_2; \Phi_a$$
$$\langle v :: T_1 :: T_2, \Phi_a\rangle \to \langle v :: T_2, \Phi_a\rangle$$

We need to show that:

$$\Phi_a; \Gamma; \Delta \vdash v :: T_2 : T_2; \Phi_a$$

By the premise, we know that:

$$\frac{\Phi_a; \Gamma; \Delta \vdash v :: T_1 : T_1; \Phi_a \quad \Delta \vdash T_1 \qquad T_1 <: T_2}{\Phi_a; \Gamma; \Delta \vdash v :: T_1 :: T_2 : T_2; \Phi_a}$$

Thus we know that:

$$\Phi_a; \Gamma; \Delta \vdash v : T; \Phi_a$$
$$T <: T_2$$

Therefore:

$$\frac{\Phi_a; \Gamma; \Delta \vdash v : T; \Phi_a \quad \Delta \vdash T \qquad T <: T_2}{\Phi_a; \Gamma; \Delta \vdash v :: T_2 : T_2; \Phi_a}$$

Case (E-Poly).

$$\text{(E-Poly)}\frac{}{\langle (\Lambda\alpha\ .\ t)\ [\mathrm{e}], \Phi\rangle \to \langle t[\alpha \to \mathrm{e}], \Phi\rangle}$$

We know that:

$$\Phi_a; \Gamma; \Delta \vdash (\Lambda\alpha\ .\ t)\ [\mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]$$
$$\Phi_c[\alpha \to \mathrm{e}] \leq_p \Phi_a$$

This case is similar to the application.

We need to show:

$$\Phi_a; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi'_a$$
$$(\Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]) \sqsubseteq \Phi'_a$$

By the induction hypothesis and the lemma. A.6 (Substitution on effects) we know that:

$$\Phi_c; \Gamma; \Delta, \alpha \vdash t : T; \Phi_p \Rightarrow \Phi_c[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi_p[\alpha \to \mathrm{e}]$$

Because we know that Φ_a has more privileges than $\Phi_c[\alpha \to \mathrm{e}]$, and by Proposition A.1 (Monotonicity of the Typing) we know that:

$$\Phi_a; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi'_a$$

By definition, we know that t is going to consume $\Phi_c[\alpha \to \mathrm{e}]$ and is going to produce $(\Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}])$. Therefore, we know that:

$$\Phi'_a = (\Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}])$$

Thus:

$$(\Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]) \sqsubseteq (\Phi_a \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}])$$

Case (Pair).

$$\text{(E-Pair)} \frac{}{\langle \{\, u_1 :: T_1, u_2 :: T_2 \,\}, \Phi \rangle \to \langle \{\, u_1, u_2 \,\} :: T_1 \times T_2, \Phi \rangle}$$

We know that:

$$\Phi_a; \Gamma; \Delta \vdash \{\, u_1 :: T_1, u_2 :: T_2 \,\} : T_1 \times T_2; \Phi_a$$
$$\langle \{\, u_1 :: T_1, u_2 :: T_2 \,\}, \Phi_a \rangle \to \langle \{\, u_1, u_2 \,\} :: T_1 \times T_2, \Phi_a \rangle$$

We need to show that:

$$\Phi_a; \Gamma; \Delta \vdash \{\, u_1, u_2 \,\} :: T_1 \times T_2 : T_1 \times T_2; \Phi'_a$$

Which it is true, because $\{\, u_1, u_2 \,\} :: T_1 \times T_2$ is a value.

□

Lemma A.3 (Typing values with any effect set)**.**

$$\text{If } \Phi; \Gamma; \Delta \vdash v : T; \Phi \text{ then } \forall \Phi' \,.\, \Phi'; \Gamma; \Delta \vdash v : T; \Phi'$$

Proof. Direct by the typing rules of the values □

Lemma A.4 (Ascription and Relation)**.**

$$\text{If } \Phi \vdash t : \mathcal{E}[\![T']\!] \text{ and } T' <: T \text{ then } \Phi \vdash t :: T : \mathcal{E}[\![T]\!]$$

Proof. By the operational semantics, we know that the type of the ascription is preserved, and we know that t types types, therefore $t :: T$ also types. □

Proposition A.3 (Preservation of the logical relation backwards)**.**

$$\begin{array}{lll} \text{If} & (\Phi_1; \cdot; \cdot \vdash t : T; \Phi_{1s} \text{ and } \langle t, \Phi_1 \rangle \to \langle t', \Phi_2 \rangle \text{ and} & \\ & \qquad \Phi_2 \vdash t' : \mathcal{E}[\![T]\!]) & \text{then} \quad \Phi_1 \vdash t : \mathcal{E}[\![T]\!] \end{array}$$

Proof. We need to show that:

$$\Phi_1 \vdash t : \mathcal{E}[\![T]\!]$$

Unfolding the definition of the logical relation of the premise, we know that:

$$\begin{array}{lll} \Phi_2 \vdash t' : \mathcal{E}[\![T]\!] & \iff & \forall {\Phi_2}' \,.\, \Phi_2 \leq_p {\Phi_2}' \,. \\ & & \exists \Phi_{2s} \,.\, {\Phi_2}'; \cdot; \cdot \vdash t' : T; \Phi_{2s} \text{ and} \\ & & \exists \Phi_{2\mathrm{d}}, \exists v \,.\, \langle t', {\Phi_2}' \rangle \mapsto^* \langle v, \Phi_{2\mathrm{d}} \rangle \text{ then} \\ & & v : \mathcal{V}[\![T]\!] \text{ and } \Phi_{2s} \sqsubseteq \Phi_{2\mathrm{d}} \end{array}$$

Unfolding the logical relation:

$$\begin{array}{lll} \Phi_1 \vdash t : \mathcal{E}[\![T]\!] & \iff & \forall {\Phi_1}' \,.\, \Phi_1 \leq_p {\Phi_1}' \,. \\ & & \exists \Phi_{1s} \,.\, {\Phi_1}'; \cdot; \cdot \vdash t : T; \Phi_{1s} \text{ and} \\ & & \exists \Phi_{1\mathrm{d}}, \exists v' \,.\, \langle t, {\Phi_1}' \rangle \mapsto^* \langle v', \Phi_{1\mathrm{d}} \rangle \text{ then} \\ & & v' : \mathcal{V}[\![T]\!] \text{ and } \Phi_{1s} \sqsubseteq \Phi_{1\mathrm{d}} \end{array}$$

We only have to show the existence of a value v' such that

$$v' : \mathcal{V}[\![T]\!] \text{ and } \Phi_{1s} \sqsubseteq \Phi_{1\mathrm{d}}$$

By the premise, we know that:

$$\forall\ \Phi_1'\ \exists \Phi_{1s} \,.\, \Phi_1 \leq_p \Phi_1' \,.\, \Phi_1'; \cdot; \cdot \vdash t : T; \Phi_{1s}$$

And:

$$\langle t, \Phi_1 \rangle \to \langle t', \Phi_2 \rangle \wedge \langle t', \Phi_2 \rangle \mapsto^* \langle v', \Phi_{2\mathrm{d}} \rangle$$

Therefore:

$$\begin{array}{c} v \equiv v' \\ \Phi_{2\mathrm{d}} \equiv \Phi_{1\mathrm{d}} \end{array}$$

Thus:

$$\langle t, \Phi_1 \rangle \mapsto^* \langle v, \Phi_{2\mathrm{d}} \rangle$$

By the premise we know that:

$$v : \mathcal{V}[\![T]\!]$$

We know by the premise that:

$$\Phi_{2s} \sqsubseteq \Phi_{2\mathrm{d}}$$

And by the Prop. A.2 (Preservation) we know that:

$$\Phi_{1s} \sqsubseteq \Phi_{2s}$$

Therefore:

$$\Phi_{1s} \sqsubseteq \Phi_{2\mathrm{d}}$$

□

Lemma A.5 (Term Substitution Lemma)**.**

$$\text{If } \Phi; \Gamma, (x : T_a); \Delta \vdash t : T_b; \Phi' \text{ and } \Phi; \Gamma; \Delta \vdash v : T_a; \Phi \text{ then}$$
$$\Phi; \Gamma; \Delta \vdash t[x \to v] : T_b; \Phi'$$

Proof. By induction on $\Phi; \Gamma, (x : T_a); \Delta \vdash t : T_b; \Phi'$:
Case (Variable). We have:

$$\Phi; \Gamma, (x : T_a); \Delta \vdash y : T_b; \Phi \text{ and } \Phi; \Gamma; \Delta \vdash v : T_a; \Phi$$

We have two cases:
Case $(x \neq y)$. This case is trivial, the substitution of the variable, is the same variable:

$$y[x \to v] = y$$

Thus, we already know that the variable types:
Case $(x = y)$. Then we know that $T_a = T_b$, thus by property of the term context (Weakening):

$$\Phi; \cdot; \Delta \vdash v : T_a; \Phi \Rightarrow \Phi; \Gamma; \Delta \vdash v : T_a; \Phi$$

The rest of the cases follows the induction hypothesis. □

Lemma A.6 (Effect Substitution Lemma)**.**

$$\text{If } \Phi; \Gamma; \Delta, \alpha \vdash t : T; \Phi' \text{ and } \mathrm{e} \in \mathit{Eff} \text{ then}$$
$$\Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi'[\alpha \to \mathrm{e}]$$

Proof. By induction on $\Phi; \Gamma; \Delta, \alpha \vdash t : T; \Phi'$:

Case (Variable and Values). We know:

$$\Phi; \Gamma; \Delta, \alpha \vdash v : T; \Phi$$

We have to show that:

$$\Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash v[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi[\alpha \to \mathrm{e}]$$

Because we know that a term variable, cannot be an effect variable, the substitution is empty for variables. We can apply the same argument for values (in the case of function and effect abstraction we can apply induction hypothesis on the body):

$$\Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash v : T[\alpha \to \mathrm{e}]; \Phi[\alpha \to \mathrm{e}]$$

We know that a value can be typed with any effect set.

Case (Effect Instantiation).

$$\begin{gathered}\text{If } \Phi; \Gamma; \Delta, \alpha \vdash t\ [\mathrm{e}'] : T; \Phi' \text{ and } \mathrm{e} \in \mathrm{Eff} \text{ then} \\ \Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash (t\ [\mathrm{e}'])[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi'[\alpha \to \mathrm{e}]\end{gathered}$$

By induction hypothesis we know that t types under substitution:

$$\Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : \forall \beta.\ ^{\Phi_c}T^{\Phi_p}[\alpha \to \mathrm{e}]; \Phi'[\alpha \to \mathrm{e}]$$

In order for the instantiation to type, we have to show that:

$$\Phi_c[\beta \to \mathrm{e}'][\alpha \to \mathrm{e}] \leq_p \Phi'[\alpha \to \mathrm{e}]$$

We know that:

$$\Phi_c[\beta \to \mathrm{e}'] \leq_p \Phi'$$

Therefore, by Lemma A.8, we know that the substitution of an effect variable with an effect does not alter the relation between the effect sets.

Case (Ascription).

$$\begin{gathered}\text{If } \Phi; \Gamma; \Delta, \alpha \vdash t :: T : T; \Phi' \text{ and } \mathrm{e} \in \mathrm{Eff} \text{ then} \\ \Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash (t :: T)[\alpha \to \mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi'[\alpha \to \mathrm{e}]\end{gathered}$$

Similar to the previous case, we know by induction hypothesis we know that t types under substitution:

$$\Phi[\alpha \to \mathrm{e}]; \Gamma; \Delta \vdash t[\alpha \to \mathrm{e}] : T'[\alpha \to \mathrm{e}]; \Phi'[\alpha \to \mathrm{e}]$$

In order for the ascription to type, we have to show that:

$$T'[\alpha \to \mathrm{e}] <: T[\alpha \to \mathrm{e}]$$

We know that:

$$T' <: T$$

By Lemma A.9 we know that the sub-typing relation under effect substitution is preserved.

The rest of the cases are direct by induction hypothesis. □

Lemma A.7 (Monotonicity of the Logical relation)**.**

$$\text{If } \Phi_a \vdash t : \mathcal{E}[\![T]\!] \text{ and } \Phi_a \leq_p \Phi_b \text{ then } \Phi_b \vdash t : \mathcal{E}[\![T]\!]$$

Proof. By the definition of the logical relation. □

Definition A.2 (Term Environment Interpretation)

$$\mathcal{G}[\![\cdot]\!] = \{\, \emptyset \,\}$$
$$\mathcal{G}[\![\Gamma, x : T]\!] = \{\, \gamma[x \to v] \mid \gamma \in \mathcal{G}[\![\Gamma]\!] \text{ and } v : \mathcal{V}[\![T]\!] \,\}$$

Definition A.3 (Effect Environment Interpretation)

$$\mathcal{G}[\![\cdot]\!] = \{\, \emptyset \,\}$$
$$\mathcal{G}[\![\Delta, \alpha]\!] = \{\, \delta[\alpha \to \mathrm{e}] \mid \delta \in \mathcal{G}[\![\Delta]\!] \text{ and } \mathrm{e} \in \mathit{Eff} \}$$

Definition A.4 (Semantic Type Soundness)

$$\Phi; \Gamma; \Delta \models t : T \stackrel{\text{def}}{=} \forall \gamma \in \mathcal{G}[\![\Gamma]\!] \text{ and } \forall \delta \in \mathcal{G}[\![\Delta]\!]$$
$$\text{then } \delta(\Phi) \vdash \gamma(\delta(t)) : \mathcal{E}[\![\delta(T)]\!]$$

Lemma A.8 (Effect Order Preserve Under Substitution)**.**

$$\text{If } \Phi_1 \leq_p \Phi_2 \text{ then } \Phi_1[\alpha \to \mathrm{e}] \leq_p \Phi_2[\alpha \to \mathrm{e}]$$

Proof. By induction on $\leq_p$:
Case (Empty Left Effect Set).

$$(\text{EmptyP})\frac{}{\{\ \} \leq_p \Phi}$$

We have to show that $\{\ \} \leq_p \Phi[\alpha \to \mathrm{e}]$, which is true by definition, any set has more privileges than the empty set.
Case (Concrete Effect).

$$(\text{NumberP})\frac{\Phi_1 \leq_p \Phi_2 \qquad n_2 \leq n_2'}{\Phi_1, \varepsilon(n_1, n_2) \leq_p \Phi_2, \varepsilon(n_1', n_2')}$$

We have to show that $\Phi_1, \varepsilon(n_1, n_2)[\alpha \to \mathrm{e}] \leq_p \Phi_2, \varepsilon(n_1', n_2')[\alpha \to \mathrm{e}]$, which is true, because:

$$\Phi_1, \varepsilon(n_1, n_2)[\alpha \to \mathrm{e}] = \Phi_1[\alpha \to \mathrm{e}], \varepsilon(n_1, n_2)$$
$$\Phi_2, \varepsilon(n_1', n_2')[\alpha \to \mathrm{e}] = \Phi_2[\alpha \to \mathrm{e}], \varepsilon(n_1', n_2')$$

and, by induction hypothesis we know that $\Phi_1[\alpha \to \mathrm{e}] \leq_p \Phi_2[\alpha \to \mathrm{e}]$.
Case (Effect Variable).

$$(\text{PolyVarP})\frac{\Phi_1 \leq_p \Phi_2 \qquad n < m}{\Phi_1, n\alpha \leq_p \Phi_2, m\alpha}$$

We have to show that $\Phi_1, n\alpha[\alpha' \to \mathrm{e}] \leq_p \Phi_2, m\alpha[\alpha' \to \mathrm{e}]$, there are two cases:

Case ($\alpha \neq \alpha'$). Therefore:

$$\Phi_1, n\alpha[\alpha' \to \mathrm{e}] = \Phi_1[\alpha' \to \mathrm{e}], n\alpha$$
$$\Phi_2, m\alpha[\alpha' \to \mathrm{e}] = \Phi_2[\alpha' \to \mathrm{e}], m\alpha$$

By induction hypothesis, we know that $\Phi_1[\alpha' \to \mathrm{e}] \leq_p \Phi_2[\alpha' \to \mathrm{e}]$.
Case ($\alpha = \alpha'$). Therefore:

$$\Phi_1, n\alpha[\alpha \to \mathrm{e}] = \Phi_1[\alpha \to \mathrm{e}], n\mathrm{e}$$
$$\Phi_2, m\alpha[\alpha \to \mathrm{e}] = \Phi_2[\alpha \to \mathrm{e}], m\mathrm{e}$$

We know that $m\mathrm{e}$ contains more privileges than $n\mathrm{e}$, because we know that $n < m$ and by induction hypothesis, we know that $\Phi_1[\alpha' \to \mathrm{e}] \leq_p \Phi_2[\alpha' \to \mathrm{e}]$ [1].

□

Lemma A.9 (Subtyping Preserve Under Effect Substitution)**.**

$$\text{If } T \leq_p T' \text{ then } T[\alpha \to \mathrm{e}] \leq_p T'[\alpha \to \mathrm{e}]$$

Proof. We apply the substitution on each component of T and T', if the component does not have effect sets, then the substitution is empty, otherwise, we know by Lemma A.8 that effect set preserve their relation under effect substitution. □

Theorem A.2 (Fundamental Property)**.**

$$\text{If } \forall op\,.\, op \in OP\,.\, op : \mathcal{V}[\![T]\!] \text{ then (If } \Phi; \Gamma; \Delta \vdash t : T; \Phi' \text{ then } \Phi; \Gamma; \Delta \models t : T)$$

Proof. First we are going to assume that all primitives are in the logical relation. Thus we are going to proof that:

$$\text{If } \Phi; \Gamma; \Delta \vdash t : T; \Phi' \text{ then } \Phi; \Gamma; \Delta \models t : T$$

Proof by induction on the typing judgment:
Case (Values). In this case, $t = u :: T$, therefore its typing derivation is:

$$(\mathrm{Tv})\frac{\Gamma; \Delta \vdash_u u : T' \quad T' <: T}{\Phi; \Gamma; \Delta \vdash u :: T : T; \Phi}$$

We know that values can do not do a step, thus we only need to show:

$$(\gamma(\delta(u)) :: \gamma(\delta(T))) : \mathcal{V}[\![\gamma(\delta(T))]\!]$$

The only interesting cases are the function and the effect abstraction. In the case of the other base values, we know that the substitution is empty. Thus we have to show that:

$$v : \mathcal{V}[\![T]\!]$$

Values are in the relation if they type, and we assumed that t types.

For the following raw values cases, we are going to subsume ascription in order to improve readability.

[1] If Φ_1 contains $n\alpha$ then Φ_2 also has to contain at least $n\alpha$, because Φ_2 has more privileges than Φ_1.

Case (λ). In this case, $u = (\lambda x : T_1 \,.\, t)$, therefore its typing derivation is:

$$(\mathrm{T}\lambda)\frac{\begin{array}{c}\Gamma, (x : T_1) \vdash t \rightsquigarrow \Phi_c \\ \Phi_c; \Gamma, (x : T_1) \ \vdash t : T_2; \Phi_p\end{array}}{\Gamma \vdash_u (\lambda x : T_1 \,.\, t) : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}}$$

We need to show that:

$$\gamma(\delta(\lambda x : T_1 \,.\, t)) : \mathcal{V}[\![\delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p})]\!]$$

Unfolding we need to show:

$$\begin{array}{l}\forall \Phi \,.\, \Phi; \cdot; \cdot \vdash \gamma(\delta(\lambda x : T_1 \,.\, t)) : \delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p}); \Phi \ \text{ and} \\ \forall\ v \,.\, v : \mathcal{V}[\![\delta(T_1)]\!] \,. \\ \delta(\Phi_c) \vdash \gamma(\delta(\lambda x : T_1 \,.\, t))\ v : \mathcal{E}[\![\delta(T_2)]\!]\end{array}$$

By Lemma. A.3(Typing values with any effect set) we know that the function type with any effect set.

Let us take $v : \mathcal{V}[\![\delta(T_1)]\!]$; we have to show that:

$$\delta(\Phi_c) \vdash \gamma(\delta(\lambda x : T_1 \,.\, t))\ v : \mathcal{E}[\![\delta(T_2)]\!]$$

Unfolding:

$$\begin{array}{l}\forall \Phi \,.\, \delta(\Phi_c) \leq_p \Phi \,. \\ \exists \Phi_{fs} \,.\, \Phi; \cdot; \cdot \vdash \gamma(\delta(\lambda x : T_1 \,.\, t))\ v : \delta(T_2); \Phi_{fs} \text{ and} \\ \exists \Phi_{\mathrm{fd}}, \exists v_f \,.\, \langle \gamma(\delta(\lambda x : T_1 \,.\, t))\ v', \Phi\rangle \mapsto^* \langle v_f, \Phi_{\mathrm{fd}}\rangle \ \text{ then} \\ v_f : \mathcal{V}[\![\delta(T_2)]\!] \text{ and } \Phi_{fs} \sqsubseteq \Phi_{\mathrm{fd}}\end{array}$$

Let's assume that $\gamma(\delta(\lambda x : T_1 \,.\, t))\ v$ types and eventually reach to a value, otherwise trivially it is in the relation.

We know that:

$$\gamma(\delta(\lambda x : T_1 \,.\, t)) = (\lambda x : T_1 \,.\, \gamma(\delta(t)))$$

Thus we know that $\gamma(\delta((\lambda x : T_1 \,.\, t)))\ v$ can do a step:

$$\langle (\lambda x : T_1 \,.\, \gamma(\delta(t)))\ v, \Phi\rangle \to \langle \gamma(\delta(t))[x \to v], \Phi\rangle$$

We also know that the body of the function types:

$$\Phi_c; \Gamma, (x : T_1); \Delta \vdash t : T_2; \Phi_p$$

By induction hypothesis we know the following:

$$\Phi_c; \Gamma, (x : T_1); \Delta \models t : T_2$$

We assumed that $v : \mathcal{V}[\![T_1]\!]$ and $\gamma \in \mathcal{G}[\![\Gamma]\!]$ thus by the definition of multi-substitution, we can extend γ as $\gamma[x \to v]$:

$$\delta(\Phi_c) \vdash \delta(\gamma[x \to v](t)) : \mathcal{E}[\![\delta(T_2)]\!]$$

Because applying the multi-substitutions and then do another substitution is the same that do all the substitution. Therefore, we know:

$$\delta(\gamma[x \to v](t)) = \gamma(\delta(t))[x \to v]$$

Thus we know:

$$\delta(\Phi_c) \vdash \gamma(\delta(t))[x \to v] : \mathcal{E}[\![\delta(T_2)]\!]$$

Therefore by the Prop. A.3(Preservation of the logical relation backwards) we have:

$$\delta(\Phi_c) \vdash (\lambda x : T_1 \,.\, \gamma(\delta(t)))\ v : \mathcal{E}[\![\delta(T_2)]\!]$$

Case (Operation). . This case, $u = \text{op}$ we assumed that all operation are in the relation.
Case (Function of effects). In this case, $u = (\Lambda\alpha \,.\, t)$, therefore its typing derivation is:

$$(\text{Teffabs})\frac{\begin{array}{c}\Gamma, \Delta, \alpha \vdash t \rightsquigarrow \Phi_c \\ \Phi_c; \Gamma; \Delta, \alpha \vdash t : T; \Phi_p\end{array}}{\Gamma; \Delta \vdash_u (\Lambda\alpha \,.\, t) : \forall\alpha.\, {}^{\Phi_c}T^{\Phi_p}}$$

We need to show that:

$$\gamma(\delta(\Lambda\alpha \,.\, t)) : \mathcal{V}[\![\delta(\forall\alpha.\, {}^{\Phi_c}T^{\Phi_p})]\!]$$

We know that $\gamma(\delta(\Lambda\alpha \,.\, t)) = (\Lambda\alpha \,.\, \gamma(\delta(t)))$. For readability we are going to use the next definition:

$$v' \stackrel{\text{def}}{=} (\Lambda\alpha \,.\, \gamma(\delta(t)))$$

Unfolding we need to show:

$$\begin{array}{l}\forall\Phi \,.\, \Phi; \cdot; \cdot \vdash v' : \gamma(\delta(\forall\alpha.\, {}^{\Phi_c}T^{\Phi_p})); \Phi \text{ and} \\ \forall \text{e} \in \text{Eff} \,.\, \delta(\Phi_c)[\alpha \to \text{e}] \vdash v'\ [\text{e}] : \mathcal{E}[\![\delta(T)[\alpha \to \text{e}]]\!]\end{array}$$

By Lemma.A.3 (Typing values with any effect set) we know that the abstraction of effect type with any value. Lets take an $\text{e} \in \text{Eff}$ We have to show:

$$\delta(\Phi_c)[\alpha \to \text{e}] \vdash v'\ [\text{e}] : \mathcal{E}[\![\delta(T)[\alpha \to \text{e}]]\!]$$

Unfolding:

$$\begin{array}{l}\forall\Phi \,.\, \delta(\Phi_c)[\alpha \to \text{e}] \leq_p \Phi \,. \\ \exists\Phi_{fs} \,.\, \Phi; \cdot; \cdot \vdash v'\ [\text{e}] : \delta(T)[\alpha \to \text{e}]; \Phi_{fs} \text{ and} \\ \exists\Phi_{f\text{d}}, \exists v_f \,.\, \langle v'\ [\text{e}], \Phi\rangle \mapsto^* \langle v_f, \Phi_{f\text{d}}\rangle \text{ then} \\ v_f : \mathcal{V}[\![\delta(T)[\alpha \to \text{e}]]\!] \text{ and } \Phi_{fs} \sqsubseteq \Phi_{f\text{d}}\end{array}$$

Similar to the case of the lambda; lets assume that $(v'\ [\text{e}])$ types and eventually reach to a value. Thus we know that $(v'\ [\text{e}])$ can do a step and do not produce a error:

$$\langle v'\ [\text{e}], \Phi\rangle \Rightarrow \langle t[\alpha \to \text{e}], \Phi\rangle$$

Also we know that the body of the function types:

$$\Phi_c; \Gamma; \Delta, \alpha \vdash t : T; \Phi_p$$

By induction hypothesis we know the following:

$$\Phi_c; \Gamma; \Delta, \alpha \models t : T$$

We assumed that e ∈ Eff and $\delta \in \mathcal{G}[\![\Delta]\!]$ thus by the definition of multi-substitution, we can extend δ as $\delta[\alpha \to \mathrm{e}]$ thus we know:

$$\delta[\alpha \to \mathrm{e}](\Phi_c) \vdash \delta[\alpha \to \mathrm{e}](\gamma(t)) : \mathcal{E}[\![\delta[\alpha \to \mathrm{e}](T)]\!]$$

Because applying the multi-substitutions and then do another substitution is the same that do all the substitution, then we know:

$$\begin{gathered}\delta[\alpha \to \mathrm{e}](\gamma(t)) = \gamma(\delta(t))[\alpha \to \mathrm{e}] \\ \delta[\alpha \to \mathrm{e}](\gamma(\Phi_c)) = \gamma(\delta(\Phi_c))[\alpha \to \mathrm{e}] \\ \delta[\alpha \to \mathrm{e}](\gamma(T)) = \gamma(\delta(T))[\alpha \to \mathrm{e}]\end{gathered}$$

Therefore:

$$\delta(\Phi_c)[\alpha \to \mathrm{e}] \vdash \gamma(\delta(t))[\alpha \to \mathrm{e}] : \mathcal{E}[\![\delta(T)[\alpha \to \mathrm{e}]]\!]$$

By the Prop. A.3(Preservation of the logical relation backwards) we know:

$$\delta(\Phi_c)[\alpha \to \mathrm{e}] \vdash v'\ [\mathrm{e}] : \mathcal{E}[\![\delta(T))[\alpha \to \mathrm{e}]]\!]$$

Case (Variable). In this case, $t = x$, therefore its typing derivation is:

$$(\mathrm{Tx})\frac{x : T \in \Gamma}{\Phi; \Gamma \vdash x : T; \Phi}$$

We need to show $\Phi; \Gamma; \Delta \models x : T$. Following the definition of the fundamental property:

$$\begin{gathered}\forall \gamma \in \mathcal{G}[\![\Gamma]\!] \text{ and } \forall \delta \in \mathcal{G}[\![\Delta]\!] \text{ then} \\ \delta(\Phi) \vdash \gamma(\delta(x)) : \mathcal{E}[\![\delta(T)]\!]\end{gathered}$$

Assuming that $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and because x is well-typed therefore x is in the domain of Γ, we have $\Gamma(x) = T$, hence $\delta(\Phi) \vdash \gamma(\delta(x)) : \mathcal{E}[\![\delta(T)]\!]$.
Case (Pair). In this case, $t = \{\, t_1, t_2 \,\}$, therefore its typing derivation is:

$$(\mathrm{Tpair})\frac{\Phi; \Gamma; \Delta \vdash t_1 : T_1; \Phi_1 \qquad \Phi_1; \Gamma; \Delta \vdash t_2 : T_2; \Phi_2}{\Phi; \Gamma; \Delta \vdash \{\, t_1, t_2 \,\} : T_1 \times T_2; \Phi_2}$$

We need to show $\Phi; \Gamma; \Delta \models \{\, t_1, t_2 \,\} : T_1 \times T_2$. First let assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\delta \in \mathcal{G}[\![\Delta]\!]$ and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash \delta(\gamma(\{\, t_1, t_2 \,\})) : \mathcal{E}[\![\delta(T_1 \times T_2)]\!]$$

Simplifying and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash \{\, \delta(\gamma(t_1)), \delta(\gamma(t_2)) \,\} : \mathcal{E}[\![\delta(T_1) \times \delta(T_2)]\!]$$

By induction hypothesis we know that:

$$\begin{array}{rcl} \Phi;\Gamma;\Delta \models t_1 : T_1 & \Rightarrow & \delta(\Phi) \vdash \delta(\gamma(t_1)) : \mathcal{E}[\![\delta(T_1)]\!] \\ \Phi_1;\Gamma;\Delta \models t_2 : T_2 & \Rightarrow & \delta(\Phi_1) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T_2)]\!] \end{array}$$

Unfolding both definitions, we have:

$$\begin{array}{rcl} \delta(\Phi) \vdash \delta(\gamma(t_1)) : \mathcal{E}[\![\delta(T_1)]\!] & \Rightarrow & \forall \Phi \,.\, \delta(\Phi) \leq_p \Phi' \,. \\ & & \exists \Phi_{s1} \,.\, \Phi';\cdot;\cdot \vdash \delta(\gamma(t_1)) : \delta(T_1); \Phi_{s1} \text{ and} \\ & & \exists \Phi_{\mathrm{d}1}, \exists v_1 \,.\, \langle \delta(\gamma(t_1)), \Phi' \rangle \mapsto^* \langle v_1, \Phi_{\mathrm{d}} \rangle \text{ then} \\ & & v_1 : \mathcal{V}[\![\delta(T_1)]\!] \text{ and } \Phi_{s1} \sqsubseteq \Phi_{\mathrm{d}1} \\ \delta(\Phi_1) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T_2)]\!] & \Rightarrow & \Phi'_1 \,.\, \gamma(\Phi_1) \leq_p \Phi'_1 \,. \\ & & \exists \Phi_{s2} \,.\, \Phi;\cdot;\cdot \vdash \delta(\gamma(t_2)) : \delta(T_2); \Phi_{s2} \text{ and} \\ & & \exists \Phi_{\mathrm{d}2}, \exists v_2 \,.\, \langle \delta(\gamma(t_2)), \Phi'_1 \rangle \mapsto^* \langle v_2, \Phi_{\mathrm{d}2} \rangle \text{ then} \\ & & v_2 : \mathcal{V}[\![\delta(T_2)]\!] \text{ and } \Phi_{s2} \sqsubseteq \Phi_{\mathrm{d}2} \end{array}$$

Thus we know that $\{\, \delta(\gamma(t_1)), \delta(\gamma(t_2)) \,\}$ reduces :

$$\exists \Phi'_{\mathrm{d}} \,.\, \langle \{\, \delta(\gamma(t_1)), \delta(\gamma(t_2)) \,\}, \Phi' \rangle \mapsto^* \langle \{\, v_1, v_2 \,\}, \Phi'_{\mathrm{d}} \rangle$$

By definition, we know that $v_1 = u_1 :: \delta(T_1)$ and $v_2 = u_2 :: \delta(T_2)$, therefore we know that:

$$\langle \{\, v_1, v_2 \,\}, \Phi'_{\mathrm{d}} \rangle \to \langle \{\, u_1, u_2 \,\} :: \delta(T_1) \times \delta(T_2), \Phi'_{\mathrm{d}} \rangle$$

Because v_1 and v_2 are in the relation, then:

$$(\{\, u_1, u_2 \,\} :: \delta(T_1) \times \delta(T_2)) : \mathcal{V}[\![\delta(T_1) \times \delta(T_2)]\!]$$

Therefore:

$$\delta(\Phi) \vdash \{\, \delta(\gamma(t_1)), \delta(\gamma(t_2)) \,\} : \mathcal{E}[\![\delta(T_1) \times \delta(T_2)]\!]$$

Case (Application). In this case, $t = t_1\ t_2$, therefore its typing derivation is:

$$(\text{Tapp}) \frac{\begin{array}{c} \Phi;\Gamma;\Delta \vdash t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p}; \Phi_1 \\ \Phi_1;\Gamma;\Delta \vdash t_2 : T_1; \Phi_2 \qquad \Phi_c \leq_p \Phi_2 \end{array}}{\Phi;\Gamma;\Delta \vdash t_1\ t_2 : T_2; \Phi_2 \dot{-} \Phi_c + \Phi_p}$$

We need to show $\Phi;\Gamma;\Delta \models t_1\ t_2 : T_2$. First let assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\delta \in \mathcal{G}[\![\Delta]\!]$ and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash \delta(\gamma(t_1\ t_2)) : \mathcal{E}[\![\delta(T_2)]\!]$$

Simplifying and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash (\delta(\gamma(t_1))\ \delta(\gamma(t_2))) : \mathcal{E}[\![\delta(T_2)]\!]$$

By induction hypothesis we know that:

$$\begin{array}{rcl} \Phi;\Gamma;\Delta \models t_1 : {T_1}^{\Phi_c} \to {T_2}^{\Phi_p} & \Rightarrow & \delta(\Phi) \vdash \delta(\gamma(t_1)) : \mathcal{E}[\![\delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p})]\!] \\ \Phi_1;\Gamma;\Delta \models t_2 : T_1 & \Rightarrow & \delta(\Phi_1) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T_1)]\!] \end{array}$$

Unfolding both definitions, we have:

$$\begin{array}{rcl}
\delta(\Phi) \vdash \delta(\gamma(t_1)) : \mathcal{E}[\![\delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p})]\!] & \Rightarrow & \forall \Phi \,.\, \delta(\Phi) \leq_p \Phi' \,. \\
& & \exists \Phi_{s1} \,.\, \Phi'; \cdot; \cdot \vdash \delta(\gamma(t_1)) : \delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p}); \Phi_{s1} \text{ and} \\
& & \exists \Phi_{\mathrm{d}1}, \exists v_1 \,.\, \langle \delta(\gamma(t_1)), \Phi' \rangle \mapsto^* \langle v_1, \Phi_{\mathrm{d}} \rangle \text{ then} \\
& & v_1 : \mathcal{V}[\![\delta({T_1}^{\Phi_c} \to {T_2}^{\Phi_p})]\!] \text{ and } \Phi_{s1} \sqsubseteq \Phi_{\mathrm{d}1} \\
\delta(\Phi_1) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T_1)]\!] & \Rightarrow & \Phi_1' \,.\, \gamma(\Phi_1) \leq_p \Phi_1' \,. \\
& & \exists \Phi_{s2} \,.\, \Phi; \cdot; \cdot \vdash \delta(\gamma(t_2)) : \delta(T_1); \Phi_{s2} \text{ and} \\
& & \exists \Phi_{\mathrm{d}2}, \exists v_2 \,.\, \langle \delta(\gamma(t_2)), \Phi_1' \rangle \mapsto^* \langle v_2, \Phi_{\mathrm{d}2} \rangle \text{ then} \\
& & v_2 : \mathcal{V}[\![\delta(T_1)]\!] \text{ and } \Phi_{s2} \sqsubseteq \Phi_{\mathrm{d}2}
\end{array}$$

Thus we know that $(\delta(\gamma(t_1))\ \delta(\gamma(t_2)))$ reduces :

$$\exists \Phi'_{\mathrm{d}} \,.\, \langle (\delta(\gamma(t_1))\ \delta(\gamma(t_2))), \Phi' \rangle \mapsto^* \langle (v_1\ v_2), \Phi'_{\mathrm{d}} \rangle$$

Because we know that $\delta(\gamma(v_2)) : \mathcal{V}[\![\delta(\gamma(T_1))]\!]$ by definition of the relation of the lambda, and we know that:

$$\delta(\Phi_c) \vdash v_1\ v_2 : \mathcal{E}[\![\delta(T_2)]\!]$$

Thus by the Proposition .A.3 (Preservation of the logical relation backwards). we know that:

$$\delta(\Phi_c) \vdash (\delta(\gamma(t_1))\ \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T_2)]\!]$$

Because we know that the expression types, and we know that $\delta(\Phi_c) \leq_p \delta(\Phi)$ By Lemma. A.7 (Monotonicity of the Logical relation) we know:

$$\delta(\Phi) \vdash v_1\ v_2 : \mathcal{E}[\![\delta(T_2)]\!]$$

Case (Effect instance). In this case, $t = t'$ [e], therefore its typing derivation is:

$$(\text{Teffins})\frac{\begin{array}{c}\mathrm{e} \in \mathrm{Eff} \qquad \Phi; \Gamma; \Delta \vdash t : \forall \alpha. {}^{\Phi_c}T^{\Phi_p}; \Phi_1 \\ \Phi_c[\alpha \to \mathrm{e}] \leq_p \Phi_1\end{array}}{\Phi; \Gamma; \Delta \vdash t\ [\mathrm{e}] : T[\alpha \to \mathrm{e}]; \Phi_1 \dot{-} \Phi_c[\alpha \to \mathrm{e}] + \Phi_p[\alpha \to \mathrm{e}]}$$

We need to show $\Phi; \Gamma; \Delta \models t'\ [\mathrm{e}] : T[\alpha \to \mathrm{e}]$. First let assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\delta \in \mathcal{G}[\![\Delta]\!]$ and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash \delta(\gamma(t'\ [\mathrm{e}])) : \mathcal{E}[\![\delta(T[\alpha \to \mathrm{e}])]\!]$$

Simplifying and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash (\delta(\gamma(t'))\ [\mathrm{e}]) : \mathcal{E}[\![\delta(T[\alpha \to \mathrm{e}])]\!]$$

By induction hypothesis we know that:

$$\Phi; \Gamma; \Delta \models t' : \forall \alpha. {}^{\Phi_c}T^{\Phi_p} \quad \Rightarrow \quad \delta(\Phi) \vdash \delta(\gamma(t')) : \mathcal{E}[\![\delta(\forall \alpha. {}^{\Phi_c}T^{\Phi_p})]\!]$$

Unfolding the definition, we have:

$$\begin{array}{rcl}
\delta(\Phi) \vdash \delta(\gamma(t')) : \mathcal{E}[\![\delta(\forall \alpha. {}^{\Phi_c}T^{\Phi_p})]\!] & \Rightarrow & \forall \Phi \,.\, \delta(\Phi) \leq_p \Phi' \,. \\
& & \exists \Phi_s \,.\, \Phi'; \cdot; \cdot \vdash \delta(\gamma(t')) : \delta(\forall \alpha. {}^{\Phi_c}T^{\Phi_p}); \Phi_s \text{ and} \\
& & \exists \Phi_{\mathrm{d}}, \exists v \,.\, \langle \delta(\gamma(t')), \Phi' \rangle \mapsto^* \langle v, \Phi_{\mathrm{d}} \rangle \text{ then} \\
& & v : \mathcal{V}[\![\delta(\forall \alpha. {}^{\Phi_c}T^{\Phi_p})]\!] \text{ and } \Phi_s \sqsubseteq \Phi_{\mathrm{d}}
\end{array}$$

Thus we know that $(\delta(\gamma(t'))\ [\mathrm{e}])$ reduces:

$$\exists \Phi'_{\mathrm{d}}\ .\ \langle (\delta(\gamma(t'))\ [\mathrm{e}]), \Phi' \rangle \mapsto^* \langle (v\ [\mathrm{e}]), \Phi'_{\mathrm{d}} \rangle$$

Because we know that $\mathrm{e} \in \mathrm{Eff}$ by definition of the relation of abstraction of effects, we know that:

$$\delta(\Phi_c[\alpha \to \mathrm{e}]) \vdash (v\ [\mathrm{e}]) : \mathcal{E}[\![\delta(T[\alpha \to \mathrm{e}])]\!]$$

Thus by the Proposition .A.3 (Preservation of the logical relation backwards). We know that:

$$\delta(\Phi_c[\alpha \to \mathrm{e}]) \vdash (\delta(\gamma(t'))\ [\mathrm{e}]) : \mathcal{E}[\![\delta(T[\alpha \to \mathrm{e}])]\!]$$

Because we know that the expression types, we know $\delta(\Phi_c[\alpha \to \mathrm{e}])) \leq_p \delta(\Phi)$ By Lemma. A.7 (Monotonicity of the Logical relation) we know:

$$\gamma(\delta(\Phi)) \vdash (\delta(\gamma(t'))\ [\mathrm{e}]) : \mathcal{E}[\![\delta(T[\alpha \to \mathrm{e}])]\!]$$

Case (If). In this case, $t = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$, therefore its typing derivation is:

$$(\mathrm{Tif})\frac{\begin{array}{c}\Phi; \Gamma; \Delta \vdash t_1 : \mathsf{Bool}; \Phi_1 \\ \Phi_1; \Gamma; \Delta \vdash t_2 : T; \Phi_2 \\ \Phi_1; \Gamma; \Delta \vdash t_3 : T; \Phi_3\end{array}}{\Phi; \Gamma; \Delta \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T; \Phi_2 \Cap \Phi_3}$$

We need to show $\Phi; \Gamma; \Delta \models \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T$. First, let assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\delta \in \mathcal{G}[\![\Delta]\!]$ and following the definition the fundamental property, we have to prove that:

$$\delta(\gamma(\Phi)) \vdash \delta(\gamma(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3)) : \mathcal{E}[\![\delta(\gamma(T))]\!]$$

Simplifying and following the definition the fundamental property, we have to prove that:

$$\delta(\gamma(\Phi)) \vdash \mathsf{if}\ \delta(\gamma(t_1))\ \mathsf{then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3)) : \mathcal{E}[\![\delta(\gamma(T))]\!]$$

By induction hypothesis we know that:

$$\begin{array}{rcl}\Phi; \Gamma; \Delta \models t_1 : \mathsf{Bool} & \Rightarrow & \delta(\Phi) \vdash \delta(\gamma(t_1)) : \mathcal{E}[\![\delta(\mathsf{Bool})]\!] \\ \Phi_1; \Gamma; \Delta \models t_2 : T & \Rightarrow & \delta(\Phi_1) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T)]\!] \\ \Phi_1; \Gamma; \Delta \models t_3 : T & \Rightarrow & \delta(\Phi_1) \vdash \delta(\gamma(t_3)) : \mathcal{E}[\![\delta(T)]\!]\end{array}$$

Also we know that the type Bool can not have variables; thus the substitution is empty:

$$\delta(\mathsf{Bool}) = \mathsf{Bool}$$

Unfolding the definition, we have:

$$\begin{array}{rcl}\Phi; \Gamma; \Delta \models t_1 : \mathsf{Bool} & \Rightarrow & \forall \Phi\ .\ \delta(\Phi) \leq_p \Phi'\ . \\ & & \exists \Phi_s\ .\ \Phi'; \cdot; \cdot \vdash \delta(\gamma(t_1)) : \mathsf{Bool}; \Phi_{s1} \text{ and} \\ & & \exists \Phi_{\mathrm{d}1}, \exists v'\ .\ \langle \delta(\gamma(t_1)), \Phi' \rangle \mapsto^* \langle v', \Phi_{\mathrm{d}} \rangle \text{ then} \\ & & v' : \mathcal{V}[\![\mathsf{Bool}]\!] \text{ and } \Phi_{s1} \sqsubseteq \Phi_{\mathrm{d}1}\end{array}$$

Thus we know that :

$$\exists \Phi_{\mathrm{d}1}\ .\ \langle (\mathsf{if}\ \delta(\gamma(t_1))\ \mathsf{then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3))), \Phi' \rangle \mapsto^* \langle (\mathsf{if}\ v'\ \mathsf{then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3))), \Phi_{\mathrm{d}1} \rangle$$

Because we know that v' is a boolean; thus there are two cases:

Case ($v' = \mathsf{true}$).

$$\langle(\mathsf{if\ true\ then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3))), \Phi_{\mathrm{d1}}\rangle \Longrightarrow \langle\delta(\gamma(t_2)), \Phi_{\mathrm{d1}}\rangle$$

Case ($v' = \mathsf{false}$).

$$\langle(\mathsf{if\ false\ then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3))), \Phi_{\mathrm{d1}}\rangle \Longrightarrow \langle\delta(\gamma(t_3)), \Phi_{\mathrm{d1}}\rangle$$

For both cases, by induction hypothesis we know that $\delta(\gamma(t_2))$ and $\delta(\gamma(t_3))$ are in the relation with Φ_1, by our induction hypothesis we know that $\Phi_1 \sqsubseteq \Phi_{\mathrm{d1}}$, thus we that $\Phi_1 \leq_p \Phi_{\mathrm{d1}}$, thus by the lemma. A.7 (Monotonicity of the Logical Relation) we know that:

$$\delta(\Phi_{\mathrm{d1}}) \vdash \delta(\gamma(t_2)) : \mathcal{E}[\![\delta(T)]\!]$$
$$\delta(\Phi_{\mathrm{d1}}) \vdash \delta(\gamma(t_3)) : \mathcal{E}[\![\delta(T)]\!]$$

Thus by the proposition. A.3 (Preservation of the logical relation backwards) :

$$\delta(\Phi) \vdash \mathsf{if}\ \delta(\gamma(t_1))\ \mathsf{then}\ \delta(\gamma(t_2))\ \mathsf{else}\ \delta(\gamma(t_3)) : \mathcal{E}[\![\delta(T)]\!]$$

Case (Ascription). In this case, $t = (t' :: T)$, therefore its typing derivation is:

$$(\text{Tascribe})\frac{\begin{array}{c}\Delta \vdash T \\ \Phi; \Gamma; \Delta \vdash t : T'; \Phi_1 \qquad T' <: T\end{array}}{\Phi; \Gamma; \Delta \vdash t :: T : T; \Phi_1}$$

We need to show $\Phi; \Gamma; \Delta \models (t' :: T) : T$. First let assume $\gamma \in \mathcal{G}[\![\Gamma]\!]$ and $\delta \in \mathcal{G}[\![\Delta]\!]$ and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash \delta(\gamma(t' :: T)) : \mathcal{E}[\![\delta(T)]\!]$$

Simplifying and following the definition the fundamental property, we have to prove that:

$$\delta(\Phi) \vdash (\delta(\gamma(t')) :: \delta(T)) : \mathcal{E}[\![\delta(T)]\!]$$

By induction hypothesis we know that:

$$\Phi; \Gamma; \Delta \models t' : T' \quad \begin{array}{c}T' <: T \\ \Rightarrow\end{array} \quad \delta(\Phi) \vdash \delta(t') : \mathcal{E}[\![\delta(T')]\!]$$

We know that substituting effect variables on types does not alter their relation:

$$\delta(T') <: \delta(T)$$

Thus by Lemma A.4 (Ascription and Relation) :

$$\delta(\gamma(\Phi)) \vdash (\delta(\gamma(t')) :: \delta(\gamma(T))) : \mathcal{E}[\![\delta(\gamma(T))]\!]$$

□