Université Paris Diderot (Paris 7)
École doctorale 386 : Sciences Mathématiques de Paris Centre Équipe Gallium, Inria

Doctorat
Informatique

Type Systems as Coercions

Julien Cretin

(version for review generated on November 6, 2013)

Thèse dirigée par Didier Rémy
et soutenue le _/_/_ devant le Jury composé de :

Président	
Rapporteurs	Mariangiola Dezani-Ciancaglini
	Stephanie Weirich
Examinateurs	
Directeur	Didier Rémy

Abstract

Functional programming languages, like OCaml or Haskell, rely on the λ-calculus for their core language. Although they have different reduction strategies and type system features, their proof of soundness and normalization (in the absence of recursion) should be factorizable. This thesis does such a factorization for theoretical type systems featuring recursive types, subtyping, bounded polymorphism, and constraint polymorphism. Interestingly, soundness and normalization for strong reduction imply soundness and normalization for all usual strategies. Our observation is that a generalization of existing coercions permits to describe all type system features stated above in an erasable and composable way. We illustrate this by proposing two concrete type systems: first, an explicit type system with a restricted form of coercion abstraction to express subtyping and bounded polymorphism; and an implicit type system with unrestricted coercion abstraction that generalizes the explicit type system with recursive types and constraint polymorphism—but without the subject reduction property. A side technical result is an adaptation of the step-indexed proof technique for type-soundness to calculi equipped with a strong notion of reduction.

Résumé

Remerciements

Contents

Chapter 1

Introduction

In this thesis, we make a distinction between programming languages and type systems. A *type system* has exactly one underlying programming language. It classifies its programs in order to reject those that may go wrong. Sometimes, it also require programs to terminate. Type systems do so by defining abstractions of program behaviors, which are verified by the type checker at compile-time. Thus, type systems also play a role for code documentation. Here, we restrict our attention to type systems for functional programming languages. A programming language is functional if its programs can take a function as argument or return a function as a result.

Type systems usually come with some *inference* mechanism. Inference permits the programmer to write fewer typing annotations. OCaml and Haskell have a powerful inference mechanism, while Coq requires more guidance from the user. This difference comes from the fact that Coq has a more expressive type system than these two languages. We call a *surface type system* a type system defined for the programmer. Surface type systems usually have a powerful inference mechanism, which makes them easier to program with. We call a *kernel type system* a type system defined for the language designer. They are usually more concise and built to have fewer, more fundamental features. Sometimes the kernel type system of a language is simply the fully-explicit version of the surface type system, in which case the differences are mainly syntactical. But sometimes there is some desugaring and term must be elaborated from the surface type system to the kernel type system. In this case, the kernel type system usually has fewer constructs, which makes it shorter to formalize. However, the kernel type system has to be more expressive than the surface one, which on the opposite usually makes it harder to formalize than the surface language. For instance, the GHC implementation of Haskell has a surface type system, Haskell with extensions (GADTs, type families, type classes, etc.), and a kernel type system, called FC (System F with equality coercions and coercion abstraction). Type classes, GADTs, and type families are surface type system features, and do not need to be proven sound in FC. However, FC has to deal with equality coercions and coercion abstraction. It is less fastidious to prove soundness in FC, although it may be more involved conceptually.

The untyped *λ-calculus* with constants (pairs, integers, etc.) is the simplest functional programming language. It has by definition no type system, and is thus a programming language in this sense. The λ-calculus is the underlying language of type systems such as Coq, Haskell, and OCaml for example. Each of them has its own practical implementation and its own type system. Although their underlying programming language share the λ-

calculus, none of them is exactly the λ-calculus. For example, Haskell and OCaml provide side effects and use a particular reduction strategy. Both use weak reduction, but Haskell uses a call-by-need strategy while OCaml uses a call-by-value strategy. Coq is *pure* (without side effects), uses strong reduction, and its type system also ensures strong normalization.

Although these type systems have their own particularities, they still share the λ-calculus for their core language. As such, it is interesting to study the possibility for the λ-calculus to have a type system that would ensure the soundness and strong normalization of other more exiting type systems. The soundness and normalization for strong reduction imply the soundness and normalization for all usual strategies. As a consequence, studying soundness and normalization in a strong reduction setting is enough. Studying only the pure λ-calculus (without side effects) still gives a result for languages with side effects when the type system keeps track of side effects with a monadic encoding as it is the case in Haskell. Although dependent type systems are a desire, their design is still a research topic in itself, so we do not include them here.

To define such general type system for the λ-calculus with strong reduction, we generalize existing ideas: several type systems use subtyping, containments, or coercions to express some of their features. For instance, System $\mathsf{F}_{<:}$ uses subtyping to retype a term or to constrain type abstraction. GHC uses equality coercions to implement GADTs and type families from its surface type system to its kernel type system.

In this document we introduce a framework for defining and studying type systems based on *typing coercions*. We define typing coercions (which we simply call coercions) as composable and erasable typing transformations. We instantiate the framework on two type systems where all features are expressed as coercions on top of the STLC. In particular, we make a distinction between *computational types* (usually called simple types) and *erasable types*. Computational types are related to the reduction and terms, while erasable types are related to typing. Coercions introduce and eliminate erasable types, while they can only provide congruence rules for computational types. Congruence rules for erasable types are derivable from their introduction and elimination rules. Moreover, we exhibit and explain one fundamental feature behind all of these type systems: coercion abstraction.

The first type system we define, called System F_ι^p, subsumes a few existing functional type systems: System $\mathsf{F}_{<:}$ [4], MLF [17], and System F_η [22]. System $\mathsf{F}_{<:}$ features upper bounded polymorphism and arrow congruence. MLF extends System F with lower bounded polymorphism to feature a complete inference mechanism as long as term variables used polymorphically are annotated. System F_η extends System F with η-expansion: a term is well-typed in System F_η if one of its η-expansion is well-typed in System F. The main particularity of System F_η, compared to usual type systems with subtyping, is the distributivity rule $\forall\alpha\,(\tau \to \sigma) \leq (\forall\alpha\,\tau) \to \forall\alpha\,\sigma$. Our type system F_ι^p combines the following coercion features: polymorphism, lower bounded polymorphism, upper bounded polymorphism, and η-expansion. These features seen as coercions permit to derive the distributivity rule and the polymorphism congruence rule of System F_η, the upper bounded polymorphism congruence rule of the most expressive version of System $\mathsf{F}_{<:}$, and the lower bounded polymorphism congruence rules of MLF. Lower and upper bounded polymorphism can be seen as restricted forms of coercion abstraction. We call this restriction parametric because the abstract coercion has to be parametric on either its argument type (upper bounded polymorphism) or its result type (lower bounded polymorphism). This restriction ensures that coercion abstractions are coherent, *i.e.* that they do not introduce inconsistencies to the current environment.

The second type system we define, called System L^c, subsumes additional type systems:

System $\mathsf{F}_{\mathsf{rec}}$ and $\mathsf{Constraint\ ML}$ [24]. System $\mathsf{F}_{\mathsf{rec}}$ extends System F with general equi-recursive types. $\mathsf{Constraint\ ML}$ extends ML (prenex polymorphism with complete inference) with subtyping and still provides type inference. As a kernel type system the main feature of $\mathsf{Constraint\ ML}$ is constraint abstraction, which is similar to coercion abstraction. System L^{c} combines coherent polymorphism, η-expansion, and recursive types as coercions. It can also constrain a set of types to the types satisfying a proposition. Since coercions are one sort of propositions, this constrained kind feature and polymorphism permit unrestricted coercion abstraction.

Although System L^{c} subsumes System F_{ι}^{p}, it is not better on every point. System F_{ι}^{p} has an explicit version where explicit terms and explicit coercions capture the essence of typing derivations. The proof of subject reduction is thus done by reduction of the explicit term witnessing the derivation. System L^{c} has only an implicit version without a subject reduction property, even though the type system is shown to be sound. This difference comes from the fact that an explicit version of System L^{c} with subject reduction would need coercion decomposition and a push reduction rule. When we have a coercion from $\tau' \to \sigma'$ to $\tau \to \sigma$ in-between a redex, we have to decompose it into two coercions in order to progress: one coercion from σ' to σ and one from τ to τ'. The main difficulty with this extension is the proof of consistency. We managed to show one side of the decomposition, however we do not know whether the other side holds in our semantics. Moving polymorphism from the coercion language to the term language may be a solution to restore subject reduction at the cost of losing deep type and coercion abstraction and instantiation.

Both System F_{ι}^{p} and System L^{c} can be seen as kernel type systems. The type systems they subsume can then be seen as associated surface type systems. For instance, type systems with inference such as MLF or $\mathsf{Constraint\ ML}$ are surface type systems for System L^{c}.

In addition to the definition of the coercion framework, the main contributions of this thesis are the study of coercion abstraction, the definition of a general η-expansion rule, and a step-indexed semantics for strong reduction. All this work is done in a strong reduction setting in order for the soundness result to be applicable to all usual strategies (those included in strong reduction, like weak reduction and its call-by-value and call-by-need restrictions). Stating the soundness property in a strong reduction setting also permits to capture more of the information contained in the type system. For instance, when a term admits a typing derivation, it is usually sound under term abstractions while the soundness property in weak reduction does not imply it, since reduction may not look under lambdas. This thesis, by defining a common framework describing several type systems, permits to easily compare the features of these systems. A common underlying programming language is highly valuable when comparing two type systems. It is more interesting to say that a type system subsumes another one, than to say that it encodes the other one. In the first case, the same term can be used from one type system to the other. Finally, a first contribution of System F_{ι}^{p} was to prove the strong normalization of MLF.

My thesis is that defining erasable type system features as coercions, *i.e.* composable inclusions between invariants, naturally gives the most out of a type system feature and its combination with other coercion features. For instance, combining η-expansion and upper bounded polymorphism naturally gives a type system more expressive than the most expressive version of System $\mathsf{F}_{<:}$. Once the type system is defined as coercions, adding coercion abstraction as a coercion extends the expressivity even further. Finally, the study of composable type systems in a strong reduction setting gives more information than with a particular strategy. A composable type system does not distinguish between top-level expressions and expressions under arbitrary environments. In particular, it does not have specialized rules for typing top-level closed terms. As a consequence, results for well-typed closed terms are likely to also hold for well-typed open terms. This is, for example, the case for soundness and normalization.

Chapter 2

The λ-calculus

On the one hand, programming languages such as C, C++, or Java, have functions defined at top-level. In other words, functions are statements and not expressions. For instance, it is not possible to pass a function as argument to another function without using pointers or wrapper classes. A typical need for this kind of feature is the *map function*. The map function takes a transformation function and a collection as arguments. It returns a collection similar to the original collection where all elements have been transformed using the transformation function.

On the other hand, *functional programming languages* use the paradigm of first-class functions. In other words, functions are expressions and not statements. It is possible to pass a function as argument to another function and thus it is possible to define a map function for all kinds of collections. It is also possible to define an anonymous function for local use, for example, inside an arithmetical expression.

The λ-calculus is a functional programming language invented by Alonzo Church. It is at the same time, simple to explain and reason about, and able to describe high-level programming. It is actually used as the foundation of several general-purpose programming languages, such as OCaml [23] or Haskell [15]. These languages are also functional programming languages.

All type systems, existing or new, exposed in this document are based on the λ-calculus. This chapter describes the syntax, the reduction rules, and some properties of the λ-calculus. We only present the properties we refer to in subsequent chapters, since this calculus satisfies numerous properties. Moreover, relying on fewer properties shows that our framework for type systems applies to more calculi. For instance, our results do not rely on the determinism, confluence, or standardization of the calculus (see 6.1.2 where standardization may become necessary).

2.1 Terminology

In the λ-calculus, programs are called *terms*. So I use programs to refer to programs of arbitrary languages, and terms to refer to programs of the λ-calculus. As in mathematics we use the letter x for variables. We use the letters a and b for terms. The notion of execution is called *reduction*. The result of a good execution is called a *value*. The definition of a function is called the *body* of the function.

2.2 Syntax

The λ-calculus main characteristic is to place functions at the center of the language by making them usual programs: functions are terms. In particular in the pure λ-calculus all terms are functions. However the λ-calculus may be easily extended with additional terms. For instance we can add pairs, tags, or user-defined data-types.

What is a function and how do we use it? A function in programming languages can be seen, as in mathematics, as a relation between an input and an output, with the property that each input is in relation with exactly one output. To use a function we need to provide an input. This input will be substituted in the function body for the input variable.

For instance, f defined by $f(x) = x + 1$ is a mathematical function between natural numbers. The input natural number x is in relation with a unique output which in this case is its successor. To use it we need to say with which natural number we want to evaluate it. To do so, we replace x by the chosen natural number. For instance, $f(1) = 2$ and $f(2) = 3$.

But we can also see functions, as in programming languages, as subroutines: a function is a program where some parts are not yet defined. We say that the function abstracts over these parts. To use such a function we supply it with concrete parts for the abstract parts. It is actually enough to study functions that abstract over exactly one term using curryfication, see 2.5.2.

For instance, f defined by $f(x) = x + 1$ is a function between expressions. The input expression x is in relation with a unique output which in this case is the expression $x + 1$. To use it we need to say with which expression we want to substitute. To do so, we replace x by the chosen expression. For instance, $f(3 \times 4) = 3 \times 4 + 1$ and $f(f(1)) = f(1) + 1$.

According to these definitions, the three notions we need to create and use functions are:

- The notion of term abstraction which we write $\lambda x\, a$ where a is the definition, called body, of the function and x is its abstract part. For instance, $\lambda x\, x + 1$ would be the definition of the function returning $x + 1$ for x.

- The notion of term application which we write $a\, b$ where a is the function and b is the argument. See below for more details.

- The notion of term variable which we write x. It designates the places where the concrete argument will be substituted.

It may at first seem strange why application is written $a\, b$ instead of $x(a)$ as it would be the case in mathematics or other programming languages, where x is the name of the function and a its argument. In non-functional programming languages and in mathematics, functions are first named, and then used. It is not possible to use a function anonymously, without giving it a name. In the λ-calculus, since functions are terms and not only definitions, they don't have names by default. We will see in the next paragraph how we can give names to terms. Since names (variables) are also terms, when we say that applications take a term to be used as a function we allow both names and functions to be used. See Section 2.3 to see that we can use even more terms as functions. Finally, we do not need parentheses by default around the argument. It is only required to add parentheses when something is ambiguous. Notice that we define a notion of construct precedence in order to have fewer ambiguous terms (see the paragraph about construct precedence).

$$
\begin{array}{llr}
x, y & & \text{Term variables} \\
a, b & ::= \; x \mid \lambda x\, a \mid a\, a \mid \langle a, a \rangle \mid \mathsf{fst}\, a \mid \mathsf{snd}\, a & \text{Terms}
\end{array}
$$

Figure 2.1: Syntax of the λ-calculus

Giving names In mathematics and programming languages we can name objects and functions. The same effect is already possible in the pure λ-calculus using only the three constructions (abstraction, application, and variable) we defined. The construction $\mathsf{let}\ x = b\ \mathsf{in}\ a$ can be defined as syntactic sugar for $(\lambda x\, a)\, b$, the application of the argument term b to the function term $\lambda x\, a$. To understand why it gives b the name x in a, let's look at its definition. We apply the argument b to the function abstracting over x in a, so b is the concrete term for the abstract term x in a, which is exactly what we wanted.

Construct precedence Terms can be given either by their abstract syntax tree or as text. In this latter case some ambiguity might appear. Take the following term $a_1\, a_2\, a_3$ for instance. It is not clear whether it is a_1 applied to a_2 and a_3, namely $(a_1\, a_2)\, a_3$, or a_1 applied to a_2 applied to a_3, namely $a_1\, (a_2\, a_3)$. To resolve this ambiguity we let the application constructor be left associative: the first form is the right one in the absence of parentheses. Similarly the term $\lambda x\, a\, b$ can either be $\lambda x\, (a\, b)$ or $(\lambda x\, a)\, b$. To resolve this last ambiguity we give precedence to the application: the first form is the right one in the absence of parentheses.

Pairs The soundness property defined in Section 2.5.3 is only meaningful we extend the pure λ-calculus with other constructors, since otherwise it trivially holds. To keep things simple, we add pairs to the λ-calculus, but other constructors could be added as well: records, variants, etc. To construct the pair of a and b we write $\langle a, b \rangle$. To extract the first component from a pair a we write $\mathsf{fst}\, a$ and to extract its second component we write $\mathsf{snd}\, a$.

Summary All these definitions are summarized in Figure 2.1. We write x and y for variables and a and b for terms. The set of terms is inductively defined. Terms can be either variables x, abstractions $\lambda x\, a$, applications $a\, a$, pairs $\langle a, a \rangle$, or projections $\mathsf{fst}\, a$ or $\mathsf{snd}\, a$.

2.3 Reduction rules

Programs can be executed. This either returns a result or loops indefinitely. The same mechanism applies for the λ-calculus, even if we do not study side-effects. The notion of execution is called *reduction*. Reduction is a relation between terms, written $a \rightsquigarrow b$, which means that the term a reduces on the term b in one step. This relation is not a function, which means reduction is a priori non-deterministic.

In mathematics expressions are evaluated. For instance, to evaluate $f(2)$, we look for the definition of f, say $f(x) = x + 1$, and we substitute x in the definition by its value. We get $2 + 1$ which in turn evaluates to 3. In programming languages programs are executed. Most of the time it is a list of instructions to be executed sequentially. When an instruction is a call to a subroutine, the list of instructions in the subroutines are prepended to the remaining instructions. The same mechanism applies to the λ-calculus. When a function is applied to an argument as in $(\lambda x\, a)\, b$, it can reduce the body of the function, namely a, after the variable x has been substituted by the argument b. We use the notation $a[x/b]$ to denote the substitution

$$\frac{\overset{\textsc{RedCtx}}{a \rightsquigarrow b}}{E[a] \rightsquigarrow E[b]} \qquad \overset{\textsc{RedApp}}{(\lambda x\, a)\, b \rightsquigarrow a[x/b]} \qquad \overset{\textsc{RedFst}}{\mathsf{fst}\,\langle a, b\rangle \rightsquigarrow a} \qquad \overset{\textsc{RedSnd}}{\mathsf{snd}\,\langle a, b\rangle \rightsquigarrow b}$$

Figure 2.2: Reduction relation

$$\begin{array}{rcll} E & ::= & \lambda x\,[] \mid []\,a \mid a\,[] \mid \langle [], a\rangle \mid \langle a, []\rangle \mid \mathsf{fst}\,[] \mid \mathsf{snd}\,[] & \text{Evaluation contexts} \\ p & ::= & x \mid p\,v \mid \mathsf{fst}\,p \mid \mathsf{snd}\,p & \text{Prevalues} \\ v & ::= & p \mid \lambda x\,v \mid \langle v, v\rangle & \text{Values} \\ r & ::= & E[r] \mid \langle a, a\rangle\,a \mid \mathsf{fst}\,(\lambda x\,a) \mid \mathsf{snd}\,(\lambda x\,a) & \text{Errors} \end{array}$$

Figure 2.3: Notations

of x by b in a. This reduction rule can then be written $(\lambda x\, a)\, b \rightsquigarrow a[x/b]$. We call this rule RedApp and give it in Figure 2.3.

Notice that the substitution to make sense, has to be capture avoiding. This means that if the argument b contains free variables (variables not bounded by an abstraction), they should remain free after the substitution $a[x/b]$. Concretely, if a is $\lambda y\, y\, x$ and b is y, the substitution $(\lambda y\, y\, x)[x/y]$ should return $\lambda y_1\, y_1\, y$ and not $\lambda y\, y\, y$. We renamed the variable y with the fresh variable y_1 in a. This operation is called α-conversion. It does not modify the meaning of the term and is always possible. We say that $\lambda y\, y\, x$ is α-equivalent to $\lambda y_1\, y_1\, x$. In the following we treat terms up to α-equivalence.

We also have two rules for pairs: one for the first projection of a pair and one for the second one. The first projection of a pair, namely $\mathsf{fst}\,\langle a, b\rangle$, reduces on the first component of the pair, namely a. We write this reduction rule $\mathsf{fst}\,\langle a, b\rangle \rightsquigarrow a$ and name it RedFst. We have a similar rule for the second projection.

Finally, we want reduction to occur anywhere in the term. This is called strong reduction. Properties that hold for strong reduction also holds for reduction considering less evaluation contexts. In particular the soundness property for smaller reduction relation is a consequence of the soundness property for strong reduction. Evaluation contexts are defined in Figure 2.3. They are terms where exactly one subterm was replaced by a hole, written $[]$.

We can now give all reduction rules in Figure 2.2. Rule RedCtx is the context reduction rule. If a subterm a reduces to b under the evaluation context E, then the term $E[a]$, where the hole of the evaluation context E was replaced by the subterm a, reduces to $E[b]$. We write $a \rightsquigarrow^{+} a$ the transitive closure of the reduction relation $a \rightsquigarrow a$. And we write $a \rightsquigarrow^{\star} a$ its reflexive and transitive closure. We call a redex the left-hand side of rules RedApp, RedFst, and RedSnd.

In the absence of side effects, the goal of reduction is to reach a value. For instance, in mathematics, when we evaluate an expression, we want the evaluation to terminate and return a result. We will talk about termination in Section 2.5.4 but we can already talk about results in the λ-calculus. When the reduction of a term terminates, it reaches an irreducible term. Irreducible terms are terms that cannot reduce. Irreducible terms are results of reduction, but they are not necessarily values. Some irreducible terms are values, some are errors.

Errors can either be immediate errors or errors under an evaluation context. An immediate error looks like a redex but it is not a redex. To define what looks like a redex we need to define the notions of constructors and destructors. A constructor is either an abstraction or a pair, while a destructor is either an application context or a projection context. Redexes are usually a destructor applied to an associated constructor. For instance, the redex $(\lambda x\, a)\, b$ applies the constructor $\lambda x\, a$ to the destructor $[]\, b$. The term $\langle a_1, a_2\rangle\, b$ looks like a redex because it applies the constructor $\langle a_1, a_2\rangle$ to the destructor $[]\, b$. However pairs are not associated to applications and the given term cannot be reduced: it is an immediate error. Similarly, abstractions are not associated to projections and $\mathsf{fst}\,(\lambda x\, a)$ and $\mathsf{snd}\,(\lambda x\, a)$ are immediate errors too. We write r for errors and Ω for the set of errors. Errors are given in Figure 2.3. We call valid, a term which is not an error. We write $\mho$ the set of valid terms. It is by definition the complement of Ω.

Values are valid irreducible terms. Because they are valid they should not contain errors. And because they are irreducible, they should not contain redexes or they would not be irreducible. As a consequence, values are terms that do not contain errors or redexes. Because errors and redexes are constructors applied to destructors, we isolate the values that do not start with a constructor and call them prevalues. We define the set of values and prevalues inductively in Figure 2.3. We write p for prevalues and v for values. We call neutrals terms that do not start with a constructor and head normal forms those that do. These two sets of terms are complement of each other. So prevalues are neutral values.

Prevalues contain variables, because they are valid irreducible neutral terms. Prevalues also contain applications of a value to a prevalue $p\, v$, because both p and v do not contain errors or redexes by induction, and the application itself is not an error or a redex as p is neutral. Finally, prevalues contain projections of prevalues $\mathsf{fst}\, p$ and $\mathsf{snd}\, p$, because p does not contain redexes of any sort and the projection itself is not an error or a redex as p is neutral. Values contain prevalues by definition and constructors applied to values: $\lambda x\, v$ and $\langle v, v\rangle$. We can unfold these definitions and say that values are a series of constructors followed by a series of destructors applied to variables.

Examples Let's consider the following example: $\mathsf{let}\ y = \lambda x\, x\ \mathsf{in}\ y\, y$. We give the name y to the term $\lambda x\, x$, and then we apply it to itself. In order to reduce this term we first need to unfold the let sugar. We look at its definition. The expression $\mathsf{let}\ x = b\ \mathsf{in}\ a$ stands for $(\lambda x\, a)\, b$, which reduces to $a[x/b]$. So we have that $\mathsf{let}\ x = b\ \mathsf{in}\ a$ reduces to $a[x/b]$. So our example reduces to $(\lambda x\, x)\,(\lambda x\, x)$ which itself reduces to $\lambda x\, x$. We observe that $\lambda x\, x$ always returns its argument, we call this term the identity function and use id as sugar for it:

$$\mathsf{id} \triangleq \lambda x\, x$$

Another interesting term is the looping combinator omega defined as sugar:

$$\mathsf{omega} \triangleq \mathsf{let}\ y = \lambda x\, x\, x\ \mathsf{in}\ y\, y$$

Let's see how it reduces. We first unfold its definition, then unfold the let-definition. We can now reduce it by rule RedApp. We α-rename the first abstraction from x to y. We can now fold the let-definition and omega-definition:

$$\begin{aligned}\mathsf{omega} \triangleq \mathsf{let}\ y = \lambda x\, x\, x\ \mathsf{in}\ y\, y \triangleq{} & (\lambda y\, y\, y)\,(\lambda x\, x\, x)\\ \rightsquigarrow{} & (\lambda x\, x\, x)\,(\lambda x\, x\, x)\\ \overset{\alpha}{=}{} & (\lambda y\, y\, y)\,(\lambda x\, x\, x) \triangleq \mathsf{let}\ y = \lambda x\, x\, x\ \mathsf{in}\ y\, y \triangleq \mathsf{omega}\end{aligned}$$

As a conclusion we have $\mathsf{omega} \rightsquigarrow \mathsf{omega}$, so the looping combinator reduces on itself in one step and this is the only reduction it can do. This is an example of a non-terminating term: it has no result (good or bad). But it never reaches an error either.

Strategies It is possible to choose a subrelation of the reduction relation that would be a partial function. This is the case for most real-life programming languages. Almost all programming languages use weak reduction, which consists in removing the abstraction context $\lambda x\,[]$. They also reduce pairs from left to right by modifying the evaluation context $\langle a, []\rangle$ to $\langle v, []\rangle$. Then most programming languages use what is called a call-by-value strategy, while some languages use a call-by-name (or its call-by-need optimization) strategy. Each time evaluation contexts or the reduction relation are modified, the set of values has to be changed too.

In call-by-value we modify the evaluation context $a\,[]$ to $(\lambda x\, a)\,[]$. In other words we only reduce the argument of an application if its function is an abstraction. We also restrain rule RedApp to values, which is why we call this strategy call-by-value. The rule becomes $(\lambda x\, a)\, v \rightsquigarrow a[x/v]$. Notice that the set of values has changed. It contains abstractions $\lambda x\, a$, and pairs of values $\langle v, v\rangle$.

In call-by-name we remove the evaluation context $a\,[]$. The set of values is the same as those of call-by-value.

2.4 Encodings

Although we can extend the pure λ-calculus with pairs, naturals, booleans, etc., we can already reason with them with encodings. These encodings are not as fast as their analog extensions (the typical example is for natural subtraction), but they can simulate the computations. Another difference is that extensions of the pure λ-calculus may contain errors and terms that reduce on errors, while the pure λ-calculus contain no errors. This is why we study the λ-calculus with pairs instead of the pure λ-calculus, when we are interested in the soundness property. The pure λ-calculus contain no errors because there is only one sort of constructor and one sort of destructor, and they are associated.

2.4.1 Booleans

Booleans come with two constructors, namely true and false, and one destructor, namely the if-statement, written $\mathsf{if}\ a\ \mathsf{then}\ a\ \mathsf{else}\ a$. In the expression $\mathsf{if}\ b\ \mathsf{then}\ a_1\ \mathsf{else}\ a_2$, we say that b is the conditional and it should evaluate to a boolean. The term a_1 is the term to evaluate if the conditional is true, while a_1 is evaluated when the conditional is false. The two reduction rules are thus:

RedTrue
$$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ a\ \mathsf{else}\ b \rightsquigarrow a$$

RedFalse
$$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ a\ \mathsf{else}\ b \rightsquigarrow b$$

We can encode these three constructs in the pure λ-calculus as follows:

- $\mathsf{true} \triangleq \lambda x\, \lambda y\, x$
- $\mathsf{false} \triangleq \lambda x\, \lambda y\, y$
- $\mathsf{if}\ b\ \mathsf{then}\ a_1\ \mathsf{else}\ a_2 \triangleq b\, a_1\, a_2$

Let's verify that the two expected reduction rules work:

- $\mathsf{if\ true\ then}\ a\ \mathsf{else}\ b \triangleq (\lambda x\, \lambda y\, x)\, a\, b \leadsto (\lambda y\, a)\, b \leadsto a$
- $\mathsf{if\ false\ then}\ a\ \mathsf{else}\ b \triangleq (\lambda x\, \lambda y\, y)\, a\, b \leadsto (\lambda y\, y)\, b \leadsto b$

We can now define additional functions:

- $\mathsf{not} \triangleq \lambda x\, \mathsf{if}\ x\ \mathsf{then\ false\ else\ true}$
- $\mathsf{and} \triangleq \lambda x\, \lambda y\, \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else\ false}$
- $\mathsf{or} \triangleq \lambda x\, \lambda y\, \mathsf{if}\ x\ \mathsf{then\ true\ else}\ y$

2.4.2 Pairs

Pairs can also be encoded in the pure λ-calculus as follows:

- $\langle a, b\rangle \triangleq \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ a\ \mathsf{else}\ b$
- $\mathsf{fst}\ a \triangleq a\, \mathsf{true}$
- $\mathsf{snd}\ a \triangleq a\, \mathsf{false}$

Let's verify that the two expected reduction rules work:

$$\begin{aligned}
\mathsf{fst}\ \langle a, b\rangle &\triangleq (\lambda x\, x\, \mathsf{true})\, ((\lambda x_1\, \lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\, a\, b) \\
&\leadsto (\lambda x_1\, \lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\, a\, b\, \mathsf{true} \\
&\leadsto (\lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ a\ \mathsf{else}\ x_2)\, b\, \mathsf{true} \\
&\leadsto (\lambda y\, \mathsf{if}\ y\ \mathsf{then}\ a\ \mathsf{else}\ b)\, \mathsf{true} \\
&\leadsto \mathsf{if\ true\ then}\ a\ \mathsf{else}\ b \\
&\leadsto a
\end{aligned}$$

$$\begin{aligned}
\mathsf{snd}\ \langle a, b\rangle &\triangleq (\lambda x\, x\, \mathsf{false})\, ((\lambda x_1\, \lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\, a\, b) \\
&\leadsto (\lambda x_1\, \lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ x_1\ \mathsf{else}\ x_2)\, a\, b\, \mathsf{false} \\
&\leadsto (\lambda x_2\, \lambda y\, \mathsf{if}\ y\ \mathsf{then}\ a\ \mathsf{else}\ x_2)\, b\, \mathsf{false} \\
&\leadsto (\lambda y\, \mathsf{if}\ y\ \mathsf{then}\ a\ \mathsf{else}\ b)\, \mathsf{false} \\
&\leadsto \mathsf{if\ false\ then}\ a\ \mathsf{else}\ b \\
&\leadsto b
\end{aligned}$$

2.4.3 Sums

Sums come with two constructors, namely the injections, and one destructor, namely the case operator. The left injection is written $\mathsf{inl}\, a$ and the right injection is written $\mathsf{inr}\, a$. The case operator is written $\mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \mid \mathsf{inr}\, x_2 \mapsto b_2\}$. If the argument a is of the form $\mathsf{inl}\, a_1$, then the first branch b_1 is evaluated with x_1 substituted with a_1. But if a is of the form $\mathsf{inr}\, a_2$, a similar evaluation happens with the second branch b_2. There two reduction rules are thus:

$$\textsc{RedInl}\quad \mathsf{case}\ \mathsf{inl}\, a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \,|\, \mathsf{inr}\, x_2 \mapsto b_2\} \rightsquigarrow b_1[x_1/a]$$

$$\textsc{RedInr}\quad \mathsf{case}\ \mathsf{inr}\, a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \,|\, \mathsf{inr}\, x_2 \mapsto b_2\} \rightsquigarrow b_2[x_2/a]$$

We can encode these constructs in the pure λ-calculus as follows:

- $\mathsf{inl}\, a \triangleq \lambda y_1\, \lambda y_2\, y_1\, a$
- $\mathsf{inr}\, a \triangleq \lambda y_1\, \lambda y_2\, y_2\, a$
- $\mathsf{case}\ a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \,|\, \mathsf{inr}\, x_2 \mapsto b_2\} \triangleq a\, (\lambda x_1\, b_1)\, (\lambda x_2\, b_2)$

Let's verify that the two expected reduction rules work:

$$\begin{aligned}
\mathsf{case}\ \mathsf{inl}\, a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \,|\, \mathsf{inr}\, x_2 \mapsto b_2\} &\triangleq (\lambda y_1\, \lambda y_2\, y_1\, a)\, (\lambda x_1\, b_1)\, (\lambda x_2\, b_2) \\
&\rightsquigarrow (\lambda y_2\, (\lambda x_1\, b_1)\, a)\, (\lambda x_2\, b_2) \\
&\rightsquigarrow (\lambda x_1\, b_1)\, a \\
&\rightsquigarrow b_1[x_1/a]
\end{aligned}$$

$$\begin{aligned}
\mathsf{case}\ \mathsf{inr}\, a\ \mathsf{of}\ \{\mathsf{inl}\, x_1 \mapsto b_1 \,|\, \mathsf{inr}\, x_2 \mapsto b_2\} &\triangleq (\lambda y_1\, \lambda y_2\, y_2\, a)\, (\lambda x_1\, b_1)\, (\lambda x_2\, b_2) \\
&\rightsquigarrow (\lambda y_2\, y_2\, a)\, (\lambda x_2\, b_2) \\
&\rightsquigarrow (\lambda x_2\, b_2)\, a \\
&\rightsquigarrow b_2[x_2/a]
\end{aligned}$$

2.5 Properties

We present some properties of the λ-calculus, like confluence and curryfication. These properties are for the whole programming language. On another side, some properties are about terms, and all terms of the λ-calculus do not satisfy these properties. For example, soundness and termination are not true for all terms. Notice however that the pure λ-calculus is sound: all its term are sound. This comes from the fact that there is only one sort of constructor and one sort of destructor, and they are associated. There is syntactically no errors in the pure λ-calculus.

2.5.1 Confluence

A reduction relation is confluent if any two reduction paths starting from the same term can be joined. More precisely if a reduces in zero or more steps to a_1 and if it also reduces in zero or more steps to a_2, then there is a term b such that a_1 and a_2 reduce in zero or more steps to b.

Definition 1 (Confluence)**.** *A relation $a \rightsquigarrow a$ is confluent if it satisfies the following property. If $a \rightsquigarrow^\star a_1$ and $a \rightsquigarrow^\star a_2$ hold, then there is b such that $a_1 \rightsquigarrow^\star b$ and $a_2 \rightsquigarrow^\star b$ hold.*

The λ-calculus with pairs is confluent. Other extensions of the λ-calculus are also confluent, but we focus on the λ-calculus extended with pairs in this document. The λ-calculus is also locally confluent. Local confluence is similar to confluence but its hypotheses do exactly one reduction step.

Definition 2 (Local confluence)**.** *A relation $a \rightsquigarrow a$ is locally confluent if it satisfies the following property. If $a \rightsquigarrow a_1$ and $a \rightsquigarrow a_2$ hold, then there is b such that $a_1 \rightsquigarrow^\star b$ and $a_2 \rightsquigarrow^\star b$ hold.*

2.5.2 Curryfication

In the λ-calculus, functions take exactly one argument and return exactly one result. However using pairs, it is possible to take several arguments and return several results. For instance, a swap function that takes two arguments at the same time and returns them swapped, could be written:

$$\mathsf{swap} \triangleq \lambda x \, \langle \mathsf{snd}\, x, \mathsf{fst}\, x \rangle$$

It is however more convenient to use functions over several arguments like $\lambda(x, y)\, \langle y, x \rangle$ although we still have to return a pair. Functions taking several arguments can also be curryfied. This means that instead of abstracting over a series of arguments, the term abstracts the first component of the series and returns an abstraction over the second component of the series, and so on. More precisely $\lambda(x_1, \dots, x_n)\, a$ can be curryfied as $\lambda x_1 \dots \lambda x_n\, a$ and reciprocally for uncurryfication. A similar mechanism applies for multi-application. The application of a series of argument $a\,(b_1, \dots, b_n)$ becomes $(a\, b_1) \dots b_n$. The parentheses are not necessary.

2.5.3 Soundness

Any powerful enough programming language allows somehow to write programs that go wrong. For instance, null pointer exceptions or segmentation faults are consequences of programs that went wrong. In the pure λ-calculus, programs cannot go wrong since no errors can appear syntactically. However when we extend it, errors may appear, which is why we added pairs.

A term a is sound if and only if none of its reduction paths yield to an error (or all its reduction paths yield to valid terms):

$$\forall b,\ a \rightsquigarrow^\star b \Rightarrow b \in \mho$$

2.5.4 Termination

Any powerful enough programming language allows some kind of looping mechanism (while-loops, for-loops, recursive functions, etc.) hence non-termination. The λ-calculus does not differ on this point, for instance the omega term loops indefinitely on itself. When we use a program to implement an algorithm, we are usually interested in its termination to obtain the resulting value. So we are interested into the termination of its reduction paths.

A term a is strongly normalizing if all of its reduction paths are finite.

Part I

Type Systems as Usual

Chapter 3

Existing Type Systems

In this chapter, we describe some existing type systems: one type system per section. All these type systems will be unified and subsumed by our final type system in Chapter 5, namely System L^{c}. Understanding all these type systems in detail is not necessary, however it is important to notice the differences in their definition, presentation, and set of features. At a first glance, it is not obvious that they may be unified and that the features of one type system are compatible with the features of another type system.

Since this chapter presents a series of existing type systems, it may be boring to read for readers familiar with these notions. Each type system can be skipped independently, whether the reader know them or not. The only new content, which may thus be of interested to experienced readers, is the explicit version of System F_η in Section 3.4. However this extension comes without surprises.

In the previous chapter we saw that for powerful enough programming languages, some interesting properties about terms, like soundness and termination, do not hold for every term. We want a way to know when they hold to avoid buggy programs or looping algorithms. The first approach is to mathematically prove the properties we are interested in each time we consider a term. If we write a lot of programs this can quickly become not tractable. Moreover, this is neither modular, nor resilient to code changes. The second approach would be to use more automation. One way would be to use an automatic prover and help it when needed. But another successful approach is to use type systems. Besides, as said in the introduction, type systems are also a good starting point for code documentation.

A type system is a small syntactic language which objects can be interpreted as proofs. The main object is the term judgment, which is usually of the form $\Gamma \vdash a : \tau$ and tells that a is sound and in some type systems that it also terminates. These objects can either be completely written by the programmer, partially inferred or totally inferred. In the present document we do not discuss the inference issue and only look at the type system once all inference is done. Said otherwise we only study kernel type systems.

Type systems can either be implicit or explicit. Explicit type systems have an additional syntactical class called explicit terms. They are partial typing derivations and contain all necessary annotations for the typing derivations to be unique under some conditions. Moreover, explicit type systems define a notion of reduction over these explicit terms that bisimulates the term reduction, up to erasable steps. This reduction is directly used to show the subject reduction property by reduction of the typing derivation. This is a very strong property that all type systems do not necessarily have. On the other side, implicit type systems contain no

$$
\begin{array}{rcll}
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
\tau, \sigma, \rho & ::= & \tau \to \tau \mid \tau \times \tau & \text{Types} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) & \text{Environments}
\end{array}
$$

Figure 3.1: STLC syntax

explicit terms and thus no type annotations and no explicit reduction. Type systems always have an implicit version since it can be deduced from the explicit version by erasing type annotations. As a consequence, we assume the definition of the explicit version by default. Terms will refer to explicit terms, reduction to explicit reduction, and so on. In case of ambiguity, we will specify whether we are referring to the implicit or explicit version.

In this chapter, we give as much as possible the explicit and implicit version of the considered type systems. For System F_η, there is no explicit version in the literature, however we define one very naturally. For Constraint ML the case is worse, an explicit version would be very hard to write since the typing rules are not local: the consistency of constraints is checked at top level. An explicit version of Constraint ML would need to reduce the consistency proof of the top-level set of constraints.

When defining type systems, we factorize both versions by writing the judgments in the following form $E \Rightarrow J$ where E is an explicit element witnessing the judgment J. The witness E is a syntactic representation of the derivation of J. For instance, the explicit term judgment is of the form $M \Rightarrow \Gamma \vdash a : \tau$ where M is an explicit term which is a partial proof that the term a has type τ under environment Γ. It is a partial proof because it only contains enough information to rebuild the term a, and enough information to rebuild the type τ given an environment Γ. Explicit judgments are equivalent to implicit judgments. For instance, $M \Rightarrow \Gamma \vdash a : \tau$ holds for some M if and only if $\Gamma \vdash a : \tau$ holds.

3.1 The STLC

The Simply Typed Lambda Calculus (abbreviated as STLC) is the most simple type system. It provides both soundness and termination and illustrates all there is to know about the λ-calculus.

3.1.1 Definition

The STLC defines syntactical types, described on Figure 3.1. Types are written τ, σ, or ρ. They contain arrows $\tau \to \tau$ and products $\tau \times \tau$. Arrow types are used to type functions, while product types are used to type pairs. A term of type $\tau \to \sigma$ is a function with a domain represented by τ and a range represented by σ. While a term of type $\tau \times \sigma$ is a pair of a term of type τ and a term of type σ. Base types (leaves for the syntax tree of types), like `int` or `bool`, are necessary to write any type at all, but since they are neither necessary nor interesting to the theory, we leave them aside and use meta-variables in a generic way.

In order to type functions we need the notion of environments, written Γ. They assign a type to each free term variable. They are lists of bindings of the form $(x : \tau)$. The order does not matter yet, but it will in later type systems.

Terms of the STLC mainly correspond to the terms of the λ-calculus and are given on Figure 3.1. They are written M or N and contain variables x, abstractions $\lambda(x : \tau)\, M$

$$
\begin{array}{rcll}
E & ::= & \lambda(x:\tau)\,[] \mid []\,M \mid M\,[] \mid \langle [], M\rangle \mid \langle M, []\rangle \mid \mathsf{fst}\,[] \mid \mathsf{snd}\,[] & \text{Evaluation contexts} \\
p & ::= & x \mid p\,v \mid \mathsf{fst}\,p \mid \mathsf{snd}\,p & \text{Prevalues} \\
v & ::= & p \mid \lambda(x:\tau)\,v \mid \langle v, v\rangle & \text{Values}
\end{array}
$$

Figure 3.2: STLC notations

$$
\textsc{RedCtx}\ \frac{M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}
\qquad
\textsc{RedApp}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_\beta M[x/N]
\qquad
\textsc{RedFst}\ \mathsf{fst}\,\langle M, N\rangle \rightsquigarrow_\beta M
\qquad
\textsc{RedSnd}\ \mathsf{snd}\,\langle M, N\rangle \rightsquigarrow_\beta N
$$

Figure 3.3: STLC reduction relation

where the term variable x is annotated with its type τ, applications $M\,M$, pairs $\langle M, M\rangle$, and projections $\mathsf{fst}\,M$ and $\mathsf{snd}\,M$.

We also have to define some notations, which are given on Figure 3.2. These notations are about the reduction of terms. Since the λ-calculus uses strong reduction, we need to use strong reduction for explicit terms too. And thus we define evaluation contexts, prevalues, and values for explicit terms in strong reduction.

Evaluation contexts, using the overloaded notation E, resemble those of the λ-calculus modulo the type annotation for term abstraction. They are one-hole contexts of depth one. Prevalues, again using the overloaded notation p, are variables or destructors applied to values but where a constructor is expected, in which case only a prevalue is applied. And values, using the overloaded notation v, are prevalues or constructors applied to values.

We define the reduction rules of explicit terms. These are given on Figure 3.3. We write $M \rightsquigarrow_\beta N$ to say that M reduces on N. Reduction rules resemble those of the λ-calculus modulo type annotations, which are ignored. Rule RedCtx is the context rule of strong reduction. Rule RedApp tells that the application of a term N to an abstraction $\lambda(x:\tau)\,M$ reduces to the substitution of x by N in M, which we write $M[x/N]$. Finally, rules RedFst and RedSnd tell that first and second projections of a pair reduce to the first and second elements of the pair respectively.

The implicit judgment $\Gamma \vdash a : \tau$ tells that the term a has type τ under environment Γ. The rules to derive this judgment are given on Figure 3.4. In the explicit version we add the term M witnessing the derivation: $M \Rightarrow \Gamma \vdash a : \tau$. This notation is actually coherent with our more general notion of explicit judgments written $E \Rightarrow J$ where E is a witness of the derivation of the judgment J. We reuse this notation later for coercions $G \Rightarrow \Gamma \vdash \tau \triangleright \sigma$ where the coercion G witnesses the coercion judgment $\Gamma \vdash \tau \triangleright \sigma$.

Rule TermVar tells that the term variable x has type τ under the well-formed environment Γ if the variable x is associated to the type τ in the environment Γ. The term witnessing this proof is the term variable x itself. Rule TermLam gives the function $\lambda x\,a$ the type $\tau \to \sigma$ under environment Γ, if the term a has type σ under the extended environment $\Gamma, (x:\tau)$. The environment is extended because x is now free in a and we need to say how it behaves. If we name M the term for $\Gamma, (x:\tau) \vdash a : \sigma$, then the term we use for the conclusion is $\lambda(x:\tau)\,M$ since the type τ cannot be guessed by looking at the term M only. Rule TermApp tells that if a is a function of type $\tau \to \sigma$ under environment Γ and b is a term of type τ under environment Γ, then the application of b to a is of type σ under Γ. The environments are

$$\frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow \Gamma \vdash x:\tau}\ \textsc{TermVar} \qquad \frac{M \Rightarrow \Gamma,(x:\tau)\vdash a:\sigma}{\lambda(x:\tau)\,M \Rightarrow \Gamma \vdash \lambda x\,a:\tau\to\sigma}\ \textsc{TermLam}$$

$$\frac{M \Rightarrow \Gamma \vdash a:\tau\to\sigma \qquad N \Rightarrow \Gamma\vdash b:\tau}{M\,N \Rightarrow \Gamma\vdash a\,b:\sigma}\ \textsc{TermApp} \qquad \frac{M \Rightarrow \Gamma\vdash a:\tau \qquad N\Rightarrow\Gamma\vdash b:\sigma}{\langle M,N\rangle \Rightarrow \Gamma\vdash\langle a,b\rangle:\tau\times\sigma}\ \textsc{TermPair}$$

$$\frac{M \Rightarrow \Gamma\vdash a:\tau\times\sigma}{\mathsf{fst}\,M \Rightarrow \Gamma\vdash \mathsf{fst}\,a:\tau}\ \textsc{TermFst} \qquad \frac{M \Rightarrow \Gamma\vdash a:\tau\times\sigma}{\mathsf{snd}\,M \Rightarrow \Gamma\vdash \mathsf{snd}\,a:\sigma}\ \textsc{TermSnd}$$

Figure 3.4: STLC term judgment relation

$$\frac{\Gamma\vdash\tau\ \mathsf{type} \qquad \Gamma\vdash\sigma\ \mathsf{type}}{\Gamma\vdash\tau\to\sigma\ \mathsf{type}}\ \textsc{TypeArr} \qquad \frac{\Gamma\vdash\tau\ \mathsf{type} \qquad \Gamma\vdash\sigma\ \mathsf{type}}{\Gamma\vdash\tau\times\sigma\ \mathsf{type}}\ \textsc{TypeProd}$$

$$\frac{}{\varnothing\ \mathsf{env}}\ \textsc{EnvEmpty} \qquad \frac{x\notin\mathsf{dom}(\Gamma) \qquad \Gamma\vdash\tau\ \mathsf{type}}{\Gamma,(x:\tau)\ \mathsf{env}}\ \textsc{EnvTerm}$$

Figure 3.5: STLC well formedness relations

unchanged since the free variables of $a\,b$ contain those of a and b. The term we use is $M\,N$, given the terms for the premises were M for the function and N for the argument.

The next three rules are about pairs and the environment does not play a role since no abstractions take place. Rule TermPair tells that the pair $\langle a,b\rangle$ has type $\tau\times\sigma$ if a has type τ and b has type σ. The term is similar to the lambda term. Rule TermFst (resp. TermSnd) tells that the first (resp. second) projection $\mathsf{fst}\,a$ (resp. $\mathsf{snd}\,a$) of a has type τ (resp. σ) if a has type $\tau\times\sigma$. The term is similar to the lambda term.

The judgment $\Gamma\vdash\tau\ \mathsf{type}$ tells that the type τ is well-formed under environment Γ. And judgment $\Gamma\ \mathsf{env}$ tells that the environment Γ is well-formed. The rules to derive these judgments are given on Figure 3.5. A type is always well-formed and an environment is well-formed if it binds its variable at most once and their types are well-formed.

Rule TypeArr (resp. TypeProd) tells that $\tau\to\sigma$ (resp. $\tau\times\sigma$) is well-formed under environment Γ if both types τ and σ are well-formed under Γ. The empty environment $\varnothing$ is well-formed by EnvEmpty and the extension of the environment Γ with the binding $(x:\tau)$ is well-formed, by rule EnvTerm, if x is not already bound in Γ and τ is well-formed under Γ.

3.1.2 Properties

In an explicit type system, a judgment has to be unique according to its explicit entity. For instance, when the explicit term judgment $M\Rightarrow\Gamma\vdash a:\tau$ holds, we know that a is determined by M only and τ is a function of the term M and the environment Γ.

$$
\begin{array}{rclrcl}
\lfloor x \rfloor &=& x & \lfloor \langle M, N \rangle \rfloor &=& \langle \lfloor M \rfloor, \lfloor N \rfloor \rangle \\
\lfloor \lambda(x:\tau)\, M \rfloor &=& \lambda x\, \lfloor M \rfloor & \lfloor \mathsf{fst}\, M \rfloor &=& \mathsf{fst}\, \lfloor M \rfloor \\
\lfloor M\, N \rfloor &=& \lfloor M \rfloor\, \lfloor N \rfloor & \lfloor \mathsf{snd}\, M \rfloor &=& \mathsf{snd}\, \lfloor M \rfloor
\end{array}
$$

Figure 3.6: STLC drop function

Lemma 3 (Uniqueness). *The following assertions hold.*

- *If* $M \Rightarrow \Gamma_1 \vdash a_1 : \tau_1$ *and* $M \Rightarrow \Gamma_2 \vdash a_2 : \tau_2$ *hold, then* $a_1 = a_2$ *holds.*
- *If* $M \Rightarrow \Gamma \vdash a : \tau_1$ *and* $M \Rightarrow \Gamma \vdash a : \tau_2$ *hold, then* $\tau_1 = \tau_2$ *hold.*

Actually a is a function of M even if M is not well-typed. We call this function the *drop* function and we write it $\lfloor M \rfloor$. It is simply defined by dropping all type decorations in M. The formal definition is given on Figure 3.6. This lemma explains why we usually omit a in explicit term judgments.

Lemma 4. *If* $M \Rightarrow \Gamma \vdash a : \tau$ *holds, then* $a = \lfloor M \rfloor$ *holds.*

Although we can write $\Gamma \vdash M : \tau$ instead of $M \Rightarrow \Gamma \vdash a : \tau$, we will not do so in order to keep similar notations between all type systems. In particular, some type systems have other explicit judgments than the term judgment and the notation $E \Rightarrow J$ where E is an explicit entity and J is an implicit judgment will be used for them.

The next lemma gives the equivalence between the implicit and explicit version of the type system. It tells that M is actually a function of the implicit typing derivation. In other words, we can extract from a derivation of $\Gamma \vdash a : \tau$ the term M such that $M \Rightarrow \Gamma \vdash a : \tau$ holds. And reciprocally that the term M is only a decoration and the validity of the judgment does not depend on it.

Lemma 5 (Equivalence). $\Gamma \vdash a : \tau$ *holds if and only if* $M \Rightarrow \Gamma \vdash a : \tau$ *holds for some* M.

We have some simulation properties about the explicit term reduction. If an explicit term can reduce, then its dropped lambda term can do the same reduction. And reciprocally, if the dropped lambda term of an explicit term can reduce, then the explicit term can do it too.

Lemma 6 (Bisimulation). *The following assertions hold.*

- *If* $M \rightsquigarrow_\beta N$ *holds, then* $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$ *holds too.*
- *If* $\lfloor M \rfloor \rightsquigarrow b$ *holds, then there is an* N *such that* $M \rightsquigarrow_\beta N$ *and* $b = \lfloor N \rfloor$ *hold.*
- *If* M *is a value, then* $\lfloor M \rfloor$ *is a value too.*

Contrary to the untyped reduction relation of the λ-calculus, the explicit reduction relation is strongly normalizing for well-typed terms. And by bisimulation, well-typed lambda terms strongly normalize too.

Lemma 7 (Termination). *If* $M \Rightarrow \Gamma \vdash a : \tau$ *holds, then* M *strongly normalizes.*

Corollary 8. *If* $\Gamma \vdash a : \tau$ *holds, then* a *strongly normalizes.*

Similarly to the reduction relation of the λ-calculus, the explicit reduction relation is confluent. If an explicit term can reduce in two manners, then there is a term that joins these two reduction paths.

Lemma 9 (Confluence)**.** *The explicit reduction is confluent.*

The STLC type system obeys some usual syntactical properties. The first property is called weakening. It tells that a well-formed type (resp. a well-typed term) under the environment Γ is also well-formed (resp. well-typed with the same type) under a well-formed extended environment Γ'. This property is in part used to prove the substitution lemma.

Lemma 10 (Weakening)**.** *If $\Gamma \subseteq \Gamma'$ and Γ' env hold, then the following assertions hold:*

- *If $\Gamma \vdash \tau$ type holds, then $\Gamma' \vdash \tau$ type holds.*
- *If $M \Rightarrow \Gamma \vdash a : \tau$ holds, then $M \Rightarrow \Gamma' \vdash a : \tau$ holds.*

Corollary 11. *If Γ env and $(x : \tau) \in \Gamma$ hold, then $\Gamma \vdash \tau$ type holds.*

The second property is called substitution. It tells that the substitution $a[x/b]$ of x by b in a has type σ under environment Γ extended with Γ' if the argument b has type τ under Γ and the body a has type σ under Γ extended with x associated to τ and Γ'. This lemma is used to prove the subject reduction property for the RedApp case.

Lemma 12 (Substitution)**.** *If $M \Rightarrow \Gamma, (x : \tau), \Gamma' \vdash a : \sigma$ and $N \Rightarrow \Gamma \vdash b : \tau$ hold, then $M[x/N] \Rightarrow \Gamma, \Gamma' \vdash a[x/b] : \sigma$ holds.*

The next property is called extraction. It tells that the subcomponents of a judgment are well-formed. We can extract the well-formedness of the type and environment of a term judgment, and we can extract the well-formedness of the environment of a type judgment.

Lemma 13 (Extraction)**.** *The following assertions hold:*

- *If $\Gamma \vdash \tau$ type holds, then Γ env holds.*
- *If $\Gamma \vdash a : \tau$ holds, then $\Gamma \vdash \tau$ type holds.*

The explicit reduction preserves typing. When a term a has type τ under environment Γ with witness M which reduces to N, then the term N actually witnesses that $\lfloor N \rfloor$ has type τ under Γ. By bisimulation we deduce the subject reduction property for the well-typed lambda terms. If the term a is well-typed and reduces to b, then it is still well-typed with the same typing (environment and type).

Lemma 14 (Explicit subject reduction)**.** *If $M \Rightarrow \Gamma \vdash a : \tau$ and $M \rightsquigarrow_\beta N$ hold, then $N \Rightarrow \Gamma \vdash b : \tau$ holds where $b = \lfloor N \rfloor$.*

Corollary 15 (Subject reduction)**.** *If $\Gamma \vdash a : \tau$ and $a \rightsquigarrow b$ hold, then $\Gamma \vdash b : \tau$ holds.*

The subject reduction property usually goes with the progress property. Both properties together show type soundness. Progress tells that if a term a is well-typed then it either reduces or it is a value. The same property holds for the explicit language.

Lemma 16 (Explicit progress)**.** *If $M \Rightarrow \Gamma \vdash a : \tau$ holds and M is not a value, then there is an term N such that $M \rightsquigarrow_\beta N$ holds.*

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M \rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
 & \mid & \Lambda\alpha\, M \mid M[\tau] & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha\, \tau & \text{Types} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha & \text{Environments}
\end{array}
$$

Figure 3.7: System F syntax

Lemma 17 (Progress)**.** *If $\Gamma \vdash a : \tau$ holds and a is not a value, then there is a term b such that $a \rightsquigarrow b$ holds.*

And finally, the STLC type system obeys the soundness property: well-typed terms are sound. A term is sound if all its reductions do not encounter an error. This proposition relies on subject reduction and the fact that well-typed term are valid terms.

Proposition 18 (Soundness)**.** *If $\Gamma \vdash a : \tau$ holds, then a is sound.*

3.2 System F

The STLC type system is sound but quite limited. For instance, the encodings we used in Section 2.4 cannot be given a generic type in the STLC, but they admit one in System F. A more simple example is the following sound and terminating term, which is typable in System F but not in the STLC:

$$\mathsf{let}\ x = \lambda y\, y\ \mathsf{in}\ x\, x$$

According to the body of the let-construct, the type of x has to be an arrow type and the left-hand side of the arrow type has to be again the type of x. As a consequence we would need some sort of infinite type. One solution is to use recursive types as in System $\mathsf{F}_{\mathsf{rec}}$ (see Section 3.3), while another one is to use polymorphism as described in this section.

Types are used to show particular properties (like soundness, termination, etc.) but they can also be used by the programmer to describe the behavior of a term, as documentation. For instance a term of type $\tau \to \sigma$ takes any term of type τ and returns a term of type σ. The type syntax of the STLC is quite poor to be used as documentation, we might want to extend it. A useful and powerful extension is polymorphism. The STLC extended with polymorphism is called System F.

Let's consider the identity term written id which definition is $\lambda x\, x$. The most we can say is that it takes a term and returns it as is. In the STLC we can say that for some type τ, if it takes a term of type τ, it returns a term of type τ. However it is true for any type τ but we cannot say it in the syntax of types. We need to introduce a new type constructor that does not have a computational content but only a typing (and documentation) content.

3.2.1 Definition

System F extends the syntax of the STLC with a class of type variables, written α or β. Syntactical types are also extended with type variables α and polymorphic types $\forall\alpha\, \tau$. Bindings are extended with type bindings α. Terms are extended with type abstractions $\Lambda\alpha\, M$ and type applications $M[\tau]$. All these changes are formally given on Figure 3.7.

$$
\begin{array}{rcll}
E & ::= & \lambda(x:\tau)\,[] \mid []\,M \mid M\,[] \mid \langle [], M\rangle \mid \langle M, []\rangle \mid \mathsf{fst}\,[] \mid \mathsf{snd}\,[] & \text{Evaluation contexts} \\
 & \mid & \Lambda\alpha\,[] \mid [][\tau] & \\
p & ::= & x \mid p\,v \mid \mathsf{fst}\,p \mid \mathsf{snd}\,p \mid p[\tau] & \text{Prevalues} \\
v & ::= & p \mid \lambda(x:\tau)\,v \mid \langle v, v\rangle \mid \Lambda\alpha\,v & \text{Values}
\end{array}
$$

Figure 3.8: System F notations

$$
\textsc{RedCtx}\ \frac{M \rightsquigarrow_{\beta\iota} N}{E[M] \rightsquigarrow_{\beta\iota} E[N]}
\qquad
\textsc{RedApp}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_{\beta} M[x/N]
\qquad
\textsc{RedFst}\ \mathsf{fst}\,\langle M, N\rangle \rightsquigarrow_{\beta} M
\qquad
\textsc{RedSnd}\ \mathsf{snd}\,\langle M, N\rangle \rightsquigarrow_{\beta} N
$$

$$
\textsc{RedFor}\ (\Lambda\alpha\,M)[\tau] \rightsquigarrow_{\iota} M[\alpha/\tau]
$$

Figure 3.9: System F reduction rules

Notations for System F follow the same logic as the λ-calculus or the STLC. They are given on Figure 3.8. Evaluation contexts are all one-hole terms of depth one and correspond to strong reduction. Prevalues are variables or destructors applied to values but where a constructor is expected, in which case only a prevalue is applied. And values are prevalues or constructors applied to values. The new prevalue is $p[\tau]$ since type application is a destructor and the new value is $\Lambda\alpha\,v$ since type abstraction is a constructor.

Reduction rules for System F are given on Figure 3.9. We write $M \rightsquigarrow_\beta N$ when M reduces to N with a computational content and $M \rightsquigarrow_\iota N$ when M reduces on N without changing the computational content of M. We use the meta-variable $\beta\iota$ to stand for β or ι. This labeling is used in the bisimulation property when linking the explicit reduction and the λ-calculus reduction.

The first four rules are those of the STLC. The last rule, namely RedFor, is new and about polymorphism. Hence it is a ι-reduction rule while the STLC reduction rules were β-reduction rules. When a type application follows a type abstraction, reduction substitutes the term to which they are applied replacing the type variable with the type argument.

The term judgment relation is defined on Figure 3.10. The first six rules are those of the STLC. The last two rules are about polymorphism. Rule TermGen tells that if a has type τ under the environment Γ extended with the type variable α, then it can also be typed $\forall\alpha\,\tau$ under Γ. The witness used for this typing rule is $\Lambda\alpha M$ when M is the witness for the unique premise. Rule TermInst does the contrary, it instantiates a type variable. If a has polymorphic type $\forall\alpha\,\tau$ and σ is a well-formed type, then a also has type τ where all free occurrences of α are substituted with σ, namely $\tau[\alpha/\sigma]$. The witness of the rule is $M[\sigma]$ where M is the witness of the unique term judgment premise.

Finally the well-formedness relations are given on Figure 3.11. Rule TypeVar tells that a type variable is well-formed if it is bound in the environment, which has to be well-formed. Rule TypeArr (resp. TypeProd) tells that $\tau \to \sigma$ (resp. $\tau \times \sigma$) is well-formed if both τ and σ are well-formed. The polymorphic type $\forall\alpha\,\tau$ is well-formed if its body τ is well-formed in the same environment extended with α, according to rule TypeFor.

The empty environment is always well-formed, rule EnvEmpty. An environment extended

$$\frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow \Gamma \vdash x:\tau}\ \textsc{TermVar}
\qquad
\frac{M \Rightarrow \Gamma,(x:\tau) \vdash a:\sigma}{\lambda(x:\tau)\,M \Rightarrow \Gamma \vdash \lambda x\,a : \tau\to\sigma}\ \textsc{TermLam}$$

$$\frac{M \Rightarrow \Gamma \vdash a:\tau\to\sigma \qquad N \Rightarrow \Gamma \vdash b:\tau}{M\,N \Rightarrow \Gamma \vdash a\,b:\sigma}\ \textsc{TermApp}
\qquad
\frac{M \Rightarrow \Gamma \vdash a:\tau \qquad N \Rightarrow \Gamma \vdash b:\sigma}{\langle M,N\rangle \Rightarrow \Gamma \vdash \langle a,b\rangle : \tau\times\sigma}\ \textsc{TermPair}$$

$$\frac{M \Rightarrow \Gamma \vdash a:\tau\times\sigma}{\mathsf{fst}\ M \Rightarrow \Gamma \vdash \mathsf{fst}\ a:\tau}\ \textsc{TermFst}
\qquad
\frac{M \Rightarrow \Gamma \vdash a:\tau\times\sigma}{\mathsf{snd}\ M \Rightarrow \Gamma \vdash \mathsf{snd}\ a:\sigma}\ \textsc{TermSnd}$$

$$\frac{M \Rightarrow \Gamma,\alpha \vdash a:\tau}{\Lambda\alpha\,M \Rightarrow \Gamma \vdash a:\forall\alpha\,\tau}\ \textsc{TermGen}
\qquad
\frac{M \Rightarrow \Gamma \vdash a:\forall\alpha\,\tau \qquad \Gamma\vdash\sigma\ \mathsf{type}}{M[\sigma] \Rightarrow \Gamma \vdash a:\tau[\alpha/\sigma]}\ \textsc{TermInst}$$

Figure 3.10: System F term judgment relation

$$\frac{\Gamma\ \mathsf{env} \qquad \alpha\in\Gamma}{\Gamma\vdash\alpha\ \mathsf{type}}\ \textsc{TypeVar}
\qquad
\frac{\Gamma\vdash\tau\ \mathsf{type} \qquad \Gamma\vdash\sigma\ \mathsf{type}}{\Gamma\vdash\tau\to\sigma\ \mathsf{type}}\ \textsc{TypeArr}
\qquad
\frac{\Gamma\vdash\tau\ \mathsf{type} \qquad \Gamma\vdash\sigma\ \mathsf{type}}{\Gamma\vdash\tau\times\sigma\ \mathsf{type}}\ \textsc{TypeProd}
\qquad
\frac{\Gamma,\alpha\vdash\tau\ \mathsf{type}}{\Gamma\vdash\forall\alpha\,\tau\ \mathsf{type}}\ \textsc{TypeFor}$$

$$\textsc{EnvEmpty}\ \ \varnothing\ \mathsf{env}
\qquad
\frac{x\notin\mathsf{dom}(\Gamma) \qquad \Gamma\vdash\tau\ \mathsf{type}}{\Gamma,(x:\tau)\ \mathsf{env}}\ \textsc{EnvTerm}
\qquad
\frac{\alpha\notin\mathsf{dom}(\Gamma) \qquad \Gamma\ \mathsf{env}}{\Gamma,\alpha\ \mathsf{env}}\ \textsc{EnvType}$$

Figure 3.11: System F well-formedness relations

with a term binding is well-formed, by rule EnvTerm, if its term variable is not already bound and its type is well-formed. A well-formed environment extended with a type binding is well-formed, by rule EnvType, if its type variable is not already bound.

3.2.2 Properties

System F obeys almost the same properties as the STLC. The explicit term assures the uniqueness of the implicit term and its type when the term is well-typed. Concretely, if $M \Rightarrow \Gamma \vdash a : \tau$ holds, then a is a function of M only and τ is a function of both M and Γ.

Lemma 19 (Uniqueness)**.** *The following assertions hold.*

- *If $M \Rightarrow \Gamma_1 \vdash a_1 : \tau_1$ and $M \Rightarrow \Gamma_2 \vdash a_2 : \tau_2$ hold, then $a_1 = a_2$ holds.*
- *If $M \Rightarrow \Gamma \vdash a : \tau_1$ and $M \Rightarrow \Gamma \vdash a : \tau_2$ hold, then $\tau_1 = \tau_2$ hold.*

Similarly to the STLC, a is a function of M even if M is not well-typed. We define the drop function of System F on Figure 3.12. The drop function drops the typing annotations. Type abstractions and type applications are erased. This lemma explain why we usually omit a in explicit term judgments.

$$\begin{array}{rcl} \lfloor x \rfloor &=& x \\ \lfloor \lambda(x:\tau)\,M \rfloor &=& \lambda x\,\lfloor M \rfloor \\ \lfloor M\,N \rfloor &=& \lfloor M \rfloor\,\lfloor N \rfloor \end{array} \qquad \begin{array}{rcl} \lfloor \langle M, N\rangle \rfloor &=& \langle \lfloor M \rfloor, \lfloor N \rfloor \rangle \\ \lfloor \mathsf{fst}\,M \rfloor &=& \mathsf{fst}\,\lfloor M \rfloor \\ \lfloor \mathsf{snd}\,M \rfloor &=& \mathsf{snd}\,\lfloor M \rfloor \end{array} \qquad \begin{array}{rcl} \lfloor \Lambda\alpha\,M \rfloor &=& \lfloor M \rfloor \\ \lfloor M[\tau] \rfloor &=& \lfloor M \rfloor \end{array}$$

Figure 3.12: System F drop function

Lemma 20. *If $M \Rightarrow \Gamma \vdash a : \tau$ holds, then $a = \lfloor M \rfloor$ holds.*

System F also comes with an equivalence lemma as the STLC. The term M is a function of the implicit typing derivation. And the judgment holds also in absence of the explicit term.

Lemma 21 (Equivalence)**.** *$\Gamma \vdash a : \tau$ holds if and only if $M \Rightarrow \Gamma \vdash a : \tau$ holds for some M.*

Lemma 22 (ι-termination)**.** *The ι-reduction terminates.*

Proof. The number of explicit term nodes strictly decreases at each ι-reduction step. Only the types get bigger. □

The bisimulation property is now more interesting. If a term can reduce with a β-step, then its dropped lambda term can do the same reduction. If a term can reduce with a ι-step, then its dropped term remains the same. Reciprocally, if a dropped lambda term can reduce, the term can do a series of ι-steps to do the same β-step. However, the term now has to be well-typed since we need a classification lemma about ι-normal forms.

Lemma 23 (Bisimulation)**.** *The following assertions hold.*

- *If $M \rightsquigarrow_\beta N$ holds, then $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$ holds too.*
- *If $M \rightsquigarrow_\iota N$ holds, then $\lfloor M \rfloor = \lfloor N \rfloor$ holds.*
- *If $M \Rightarrow \Gamma \vdash a : \tau$ and $a \rightsquigarrow b$ holds, then there is an N such that $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ and $b = \lfloor N \rfloor$ hold.*
- *If M is an explicit value (see Figure 3.8), then $\lfloor M \rfloor$ is an implicit value.*

System F is strongly normalizing. The proof of this result is quite involved and initially due Girard [14]. Our proof of soundness for System $\mathsf{L^c}$ in Chapter 5 is actually based on the ideas of this proof. The proof is semantic and interprets types as sets of strongly normalizing terms. If a term is well-typed, it is in its semantic type, and thus strongly normalizes.

Lemma 24 (Termination)**.** *If $\Gamma \vdash a : \tau$ holds, then a strongly normalizes.*

Corollary 25. *If $M \Rightarrow \Gamma \vdash a : \tau$ holds, then M strongly normalizes.*

The explicit reduction of System F is also confluent. The proof can be done using strong normalization and local confluence, which holds since there are no critical pairs.

Lemma 26 (Confluence)**.** *The explicit reduction is confluent.*

System F also has a weakening lemma. It tells that a well-formed type (resp. a well-typed term) under Γ is also well-formed (resp. well-typed with the same type) under an extended environment Γ'.

Lemma 27 (Weakening). *If* $\Gamma \subseteq \Gamma'$ *and* Γ' env *hold, then the following assertions hold.*

- *If* $\Gamma \vdash \tau$ type *holds, then* $\Gamma' \vdash \tau$ type *holds.*
- *If* $M \Rightarrow \Gamma \vdash a : \tau$ *holds, then* $M \Rightarrow \Gamma' \vdash a : \tau$ *holds.*

The term substitution lemma tells that the substitution $a[x/b]$ has type σ under Γ if the argument b has type τ under Γ and the body a has type b under Γ extended with x associated to τ.

Lemma 28 (Term substitution). *If* $N \Rightarrow \Gamma \vdash b : \tau$ *and* $M \Rightarrow \Gamma, (x : \tau), \Gamma' \vdash a : \sigma$ *hold, then* $M[x/N] \Rightarrow \Gamma, \Gamma' \vdash a[x/b] : \sigma$ *holds.*

There is a similar property for types, called type substitution. If a type σ is well formed under environment Γ, then well-typed terms (resp. well-formed types) under Γ, α, Γ' are well-typed (resp. well-formed) under $\Gamma, \Gamma'[\alpha/\sigma]$ after type substitution replacing occurrences of α by type σ. Notice that we have to substitute in the remaining environment too as type variable α may appear there too.

Lemma 29 (Type substitution). *If* $\Gamma \vdash \sigma$ type *holds, then the following assertions hold.*

- *If* $\Gamma, \alpha, \Gamma' \vdash \tau$ type *holds, then* $\Gamma, \Gamma'[\alpha/\sigma] \vdash \tau[\alpha/\sigma]$ type *holds.*
- *If* $M \Rightarrow \Gamma, \alpha, \Gamma' \vdash a : \tau$ *holds, then* $M[\alpha/\sigma] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash a : \tau[\alpha/\sigma]$ *holds.*

The extraction property tells that the subcomponents of a judgment are well-formed. This is similar to the STLC. We can extract environment well-formedness from type well-formedness, and type well-formedness from term derivations.

Lemma 30 (Extraction). *The following assertions hold:*

- *If* $\Gamma \vdash \tau$ type *holds, then* Γ env *holds.*
- *If* $\Gamma \vdash a : \tau$ *holds, then* $\Gamma \vdash \tau$ type *holds.*

System F has also an explicit and implicit subject reduction properties which tell that if a term M has type τ under environment Γ and reduces on term N, then N also has type τ under Γ. Similarly for implicit terms when a reduces on b.

Lemma 31 (Explicit subject reduction). *If* $M \Rightarrow \Gamma \vdash a : \tau$ *and* $M \leadsto_{\beta\iota} N$ *hold, then* $N \Rightarrow \Gamma \vdash b : \tau$ *holds where* $b = \lfloor N \rfloor$*.*

Lemma 32 (Subject reduction). *If* $\Gamma \vdash a : \tau$ *and* $a \leadsto b$ *hold, then* $\Gamma \vdash b : \tau$ *holds.*

This property usually goes with the progress property. Both properties together show type soundness. Progress tells that if a term a is well-typed then it either reduces or is a value. This property holds for both the explicit and implicit version.

Lemma 33 (Explicit progress). *If* $M \Rightarrow \Gamma \vdash a : \tau$ *holds and* M *is not a value, then there is a term* N *such that* $M \leadsto N$ *holds.*

Lemma 34 (Progress). *If* $\Gamma \vdash a : \tau$ *holds and* a *is not a value, then there is a term* b *such that* $a \leadsto b$ *holds.*

And finally, System F obeys the soundness property. We recall that a type system is sound if all its well-typed terms are sound. And a term is sound if it cannot reduce to an error as defined in Figure 2.3.

Proposition 35 (Soundness). *If* $\Gamma \vdash a : \tau$ *holds, then* a *is sound.*

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
& \mid & \Lambda\alpha\, M \mid M[\tau] \mid \mathsf{fold}_{\mu\alpha\,\tau}\, M \mid \mathsf{unfold}\, M & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha\, \tau \mid \mu\alpha\, \tau & \text{Types} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha & \text{Environments} \\
\mathsf{rec} & ::= & \mathsf{NE} \mid \mathsf{WF} & \text{Well-foundness}
\end{array}
$$

Figure 3.13: System $\mathsf{F}_{\mathsf{rec}}$ syntax

$$
\begin{array}{rcll}
E & ::= & \lambda(x:\tau)\, [] \mid []\, M \mid M\, [] \mid \langle [], M\rangle \mid \langle M, []\rangle \mid \mathsf{fst}\, [] \mid \mathsf{snd}\, [] & \text{Evaluation contexts} \\
& \mid & \Lambda\alpha\, [] \mid [][\tau] \mid \mathsf{fold}_{\mu\alpha\,\tau}\, [] \mid \mathsf{unfold}\, [] & \\
p & ::= & x \mid p\, v \mid \mathsf{fst}\, p \mid \mathsf{snd}\, p \mid p[\tau] \mid \mathsf{unfold}\, p & \text{Prevalues} \\
v & ::= & p \mid \lambda(x:\tau)\, v \mid \langle v, v\rangle \mid \Lambda\alpha\, v \mid \mathsf{fold}_{\mu\alpha\,\tau}\, v & \text{Values}
\end{array}
$$

Figure 3.14: System $\mathsf{F}_{\mathsf{rec}}$ notations

3.3 System $\mathsf{F}_{\mathsf{rec}}$

System F can be extended with recursive types (Chapter 21 of [26]). For instance the type $\mu\alpha\,\tau \times \alpha$, which corresponds to the infinite type $\tau \times (\tau \times \ldots)$, contains terms that behave as streams of type τ. However one precaution has to be taken when dealing with recursive types: all recursive types do not necessarily have a meaning. For instance $\mu\alpha\,\alpha$ has no structure, it is an ill-formed recursive definition. Recursive type $\mu\alpha\,\tau$ has a meaning only if the functor associating α to τ is well-founded, *i.e.* occurrences of α occur only under a computational type. Computational types, defined in the introduction, are related to the reduction and terms, while erasable types are related to typing. Computational types classify head normal forms and are thus those of the STLC.

3.3.1 Definition

System $\mathsf{F}_{\mathsf{rec}}$ extends the syntactical types of System F with recursive types $\mu\alpha\,\tau$. The syntax of terms is also extended with foldings $\mathsf{fold}_{\mu\alpha\,\tau}\, a$ and unfoldings $\mathsf{unfold}\, a$. Finally, we add a syntactic class for well-foundness: NE are for non-expansive functors, and WF for well-founded ones. All these are formally given on Figure 3.13.

Notations for System $\mathsf{F}_{\mathsf{rec}}$ follow the same logic as for System F. They are given on Figure 3.14. Evaluation contexts are all one-hole terms of depth one and correspond to strong reduction. Prevalues are variables or destructors applied to values but where a constructor is expected, in which case only a prevalue is applied. And values are prevalues or constructors applied to values. The new prevalue is $\mathsf{unfold}\, p$ since unfolding is a destructing operation, while $\mathsf{fold}_{\mu\alpha.\tau} v$ is the new value since it is a constructor (see the reduction rule RedRec).

Reduction rules for System $\mathsf{F}_{\mathsf{rec}}$ are given on Figure 3.15. The first five rules are those of System F. The last rule, namely Rule RedRec, is new and about recursive types. When a term has its type folded and unfolded, it is like nothing happened. It is a ι-reduction rule since it does not change the computational content of its term.

The term judgment relation is defined on Figure 3.16. The first eight rules are those of System F. The last two rules are about recursive types. They allow to fold and unfold the

$$\frac{\overset{\text{RedCtx}}{M \leadsto_{\beta\iota} N}}{E[M] \leadsto_{\beta\iota} E[N]} \qquad \overset{\text{RedApp}}{(\lambda(x:\tau)\,M)\,N \leadsto_\beta M[x/N]} \qquad \overset{\text{RedFst}}{\mathsf{fst}\,\langle M, N\rangle \leadsto_\beta M} \qquad \overset{\text{RedSnd}}{\mathsf{snd}\,\langle M, N\rangle \leadsto_\beta N}$$

$$\overset{\text{RedFor}}{(\Lambda\alpha\,M)[\tau] \leadsto_\iota M[\alpha/\tau]} \qquad \overset{\text{RedRec}}{\mathsf{unfold}\,(\mathsf{fold}_{\mu\alpha\,\tau}\,M) \leadsto_\iota M}$$

Figure 3.15: System $\mathsf{F_{rec}}$ reduction rules

$$\frac{\overset{\text{TermVar}}{\Gamma\ \mathsf{env} \qquad (x:\tau) \in \Gamma}}{x \Rightarrow \Gamma \vdash x : \tau} \qquad \frac{\overset{\text{TermLam}}{M \Rightarrow \Gamma, (x:\tau) \vdash a : \sigma}}{\lambda(x:\tau)\,M \Rightarrow \Gamma \vdash \lambda x\,a : \tau \to \sigma}$$

$$\frac{\overset{\text{TermApp}}{M \Rightarrow \Gamma \vdash a : \tau \to \sigma \qquad N \Rightarrow \Gamma \vdash b : \tau}}{M\,N \Rightarrow \Gamma \vdash a\,b : \sigma} \qquad \frac{\overset{\text{TermPair}}{M \Rightarrow \Gamma \vdash a : \tau \qquad N \Rightarrow \Gamma \vdash b : \sigma}}{\langle M, N\rangle \Rightarrow \Gamma \vdash \langle a, b\rangle : \tau \times \sigma}$$

$$\frac{\overset{\text{TermFst}}{M \Rightarrow \Gamma \vdash a : \tau \times \sigma}}{\mathsf{fst}\,M \Rightarrow \Gamma \vdash \mathsf{fst}\,a : \tau} \qquad \frac{\overset{\text{TermSnd}}{M \Rightarrow \Gamma \vdash a : \tau \times \sigma}}{\mathsf{snd}\,M \Rightarrow \Gamma \vdash \mathsf{snd}\,a : \sigma} \qquad \frac{\overset{\text{TermGen}}{M \Rightarrow \Gamma, \alpha \vdash a : \tau}}{\Lambda\alpha\,M \Rightarrow \Gamma \vdash a : \forall\alpha\,\tau}$$

$$\frac{\overset{\text{TermInst}}{M \Rightarrow \Gamma \vdash a : \forall\alpha\,\tau \qquad \Gamma \vdash \sigma\ \mathsf{type}}}{M[\sigma] \Rightarrow \Gamma \vdash a : \tau[\alpha/\sigma]} \qquad \frac{\overset{\text{TermFold}}{M \Rightarrow \Gamma \vdash a : \tau[\alpha/\mu\alpha\,\tau]}}{\mathsf{fold}_{\mu\alpha\,\tau}\,M \Rightarrow \Gamma \vdash a : \mu\alpha\,\tau}$$

$$\frac{\overset{\text{TermUnfold}}{M \Rightarrow \Gamma \vdash a : \mu\alpha\,\tau}}{\mathsf{unfold}\,M \Rightarrow \Gamma \vdash a : \tau[\alpha/\mu\alpha\,\tau]}$$

Figure 3.16: System $\mathsf{F_{rec}}$ term judgment relation

definition of a recursive type $\mu\alpha\,\tau$. As for polymorphism these rules are implicit since they do not change the computational content of the term they are typing, but only its invariant. Rule TermFold tells that a term a with type $\tau[\alpha/\mu\alpha\,\tau]$ has also type $\mu\alpha\,\tau$ by folding the recursive definition. Notice that the explicit version of the term needs a type annotation to know which recursive type it has to fold. Rule TermUnfold does the opposite: a term a of type $\mu\alpha\,\tau$ has also type $\tau[\alpha/\mu\alpha\,\tau]$.

Compared to System F, we need a judgment about the well-foundness of functors, which we use to tell when a recursive type is well-formed. This judgment is written $\alpha \mapsto \tau : \mathsf{rec}$ and means that the functor associating the type variable α to the type τ is non-expansive (resp. well-founded) if rec is NE (resp. WF). Intuitively, a well-founded functor uses its variable only under computational types, while a non-expansive functor may use its variable anywhere. The well-foundness relation is given on Figure 3.17.

Rule RecVar tells that the identity functor is non-expansive. Rule RecCst tells that constant functors are well-founded. They do not use their variable, which means that they

$$\textsc{RecVar}\quad \alpha \mapsto \alpha : \mathsf{NE} \qquad \textsc{RecCst}\quad \frac{\alpha \notin \mathsf{fv}(\tau)}{\alpha \mapsto \tau : \mathsf{WF}} \qquad \textsc{RecSub}\quad \frac{\alpha \mapsto \tau : \mathsf{WF}}{\alpha \mapsto \tau : \mathsf{NE}} \qquad \textsc{RecArr}\quad \frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \mapsto \sigma : \mathsf{NE}}{\alpha \mapsto \tau \to \sigma : \mathsf{WF}}$$

$$\textsc{RecProd}\quad \frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \mapsto \sigma : \mathsf{NE}}{\alpha \mapsto \tau \times \sigma : \mathsf{WF}} \qquad \textsc{RecFor}\quad \frac{\alpha \mapsto \tau : \mathsf{rec}}{\alpha \mapsto \forall \beta\, \tau : \mathsf{rec}} \qquad \textsc{RecMu}\quad \frac{\beta \mapsto \tau : \mathsf{WF} \qquad \alpha \mapsto \tau : \mathsf{rec}}{\alpha \mapsto \mu \beta\, \tau : \mathsf{rec}}$$

Figure 3.17: System $\mathsf{F}_{\mathsf{rec}}$ well-foundness judgment relation

$$\textsc{TypeVar}\quad \frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}} \qquad \textsc{TypeArr}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}} \qquad \textsc{TypeProd}\quad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}$$

$$\textsc{TypeFor}\quad \frac{\Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma \vdash \forall \alpha\, \tau\ \mathsf{type}} \qquad \textsc{TypeMu}\quad \frac{\Gamma, \alpha \vdash \tau\ \mathsf{type} \qquad \alpha \mapsto \tau : \mathsf{WF}}{\Gamma \vdash \mu \alpha\, \tau\ \mathsf{type}}$$

$$\textsc{EnvEmpty}\quad \varnothing\ \mathsf{env} \qquad \textsc{EnvTerm}\quad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}} \qquad \textsc{EnvType}\quad \frac{\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma\ \mathsf{env}}{\Gamma, \alpha\ \mathsf{env}}$$

Figure 3.18: System $\mathsf{F}_{\mathsf{rec}}$ well-formedness relations

do not have a role in the recursive definition containing them. Rule RecSub tells that well-founded functors are also non-expansive: we can forget their well-foundness. Rules RecArr and RecProd tell that functors, which head type constructors are arrows and products, are well-founded if their subfunctors are non-expansive. This comes from the fact that arrows and products are computational types. Finally, rules RecFor and RecMu tell that functors, which head type constructors are polymorphic types or recursive types, are non-expansive (resp. well-founded) if their subfunctor is non-expansive (resp. well-founded). These type constructors are erasable and not computational. Additionally for the functor ending with a recursive type, an additional premise is necessary for the recursive type to be also well-founded.

Finally the well-formedness relations are given on Figure 3.18. Rule TypeVar tells that a type variable is well-formedness if it is bound in the environment, which has to be well-formed. Rule TypeArr (resp. TypeProd) tells that $\tau \to \sigma$ (resp. $\tau \times \sigma$) is well-formed if both τ and σ are well-formed. The polymorphic type $\forall \alpha\, \tau$ is well-formed if τ is well-formed in the same environment extended with α, according to rule TypeFor. Rule TypeMu tells that a recursive type is well-formed if its functor is well-founded and its body is well-formed under its environment extended with its type variable.

The empty environment is always well-formed, rule EnvEmpty. An environment extended with a term binding is well-formed, by rule EnvTerm, if its term variable is not already bound and its type is well-formed. A well-formed environment extended with a type binding is well-formed, by rule EnvType, if its type variable is not already bound.

3.3.2 Properties

System $\mathsf{F}_{\mathsf{rec}}$ obeys almost the same properties as System F. The main difference is about termination. While every well-typed terms of System F strongly normalize, some well-typed terms of System $\mathsf{F}_{\mathsf{rec}}$ may loop indefinitely. However these terms remain sound. One example is the omega term $\mathsf{omega} \stackrel{\mathsf{def}}{=} (\lambda y\, y\, y)\, (\lambda x\, x\, x)$. For any well-formed type τ under environment Γ, we define the term N as $\lambda(y : \mu\alpha\, (\alpha \to \tau))\, (\mathsf{unfold}\, y)\, y$ and the term M as $N\, (\mathsf{fold}_{\mu\alpha\, (\alpha\to\tau)}\, N)$. We have $\lfloor M \rfloor = \mathsf{omega}$. We can easily see that the term M defines a derivation for $\Gamma \vdash \mathsf{omega} : \tau$. We did not only prove that omega is well-typed, but that it accepts any well-formed type, in particular $\forall\alpha\, \alpha$, which we usually call the bottom type.

Despite the non-termination of some terms, the ι-reduction still strongly normalizes for all terms even for ill-typed ones. This property is useful to prove the backward simulation property, which can be used to prove subject reduction for the explicit version.

Lemma 36. *The ι-reduction strongly normalizes.*

Proof. The number of explicit term nodes strictly decreases during reduction. □

It has been shown that with restricted forms of recursion [20], System $\mathsf{F}_{\mathsf{rec}}$ may recover strong normalization. The most simple restriction is to consider positive recursion, where recursive type variables may only be used in covariant occurrences.

Besides the different restrictions on recursive types to recover strong normalization, there is another notion about recursive types which is subject to variants. System $\mathsf{F}_{\mathsf{rec}}$ can be presented with equi-recursive types or iso-recursive types. The version of this section has iso-recursive types: recursive types are isomorphic to their unfolding. With equi-recursive types, recursive types are equal to their unfolding. This last notion is more general and subsumes the first notion. See Section 5.4.4 for a discussion about equi- and iso-recursive types.

3.4 System F_η

As types are approximations of terms and some approximations are finer than others (fewer terms satisfy them), we may want to have an order between types related to the order between approximations. Giving a syntax in the type system for this ordering is called subtyping or type containment. This mechanism was studied along polymorphism in System F_η by Mitchell [22]. Since System F_η was described implicitly, we adapt the presentation for our explicit version. The main difference is for the distributivity rule which was $\forall\alpha\, (\tau \to \sigma) \triangleright (\forall\alpha\, \tau) \to \forall\alpha\, \sigma$ and is now $\forall\alpha\, (\tau \to \sigma) \triangleright \tau \to \forall\alpha\, \sigma$, given that α is not free in τ. The original rule is derivable is our setting. We also add product types.

We want our ordering to satisfy the fact that, if a type τ is smaller than σ under Γ with proof G, written $G \Rightarrow \Gamma \vdash \tau \triangleright \sigma$, then all terms of type τ under Γ also have type σ under Γ. In the explicit version the term M has to be annotated with the proof G, written $G\langle M\rangle$, when such retyping occurs.

Mitchell proved that the implicit version of this type system is equivalent to a version without the containment relation and the containment typing rule, but with an additional typing rule about η-expansion. This typing rule allows to type a function by actually typing its η-expansion. We can extend this idea to product types as well. In other words, a also has type $\tau \to \sigma$ under Γ, if its η-expansion $\lambda x\, a\, x$ has the same type $\tau \to \sigma$ under the same

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
& \mid & \Lambda\alpha\, M \mid M[\tau] \mid G\langle M\rangle & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha\, \tau & \text{Types} \\
G & ::= & \Diamond \mid G \circ G \mid \Lambda\alpha \mid \cdot\tau \mid G \xrightarrow{\tau} G \mid G \times G \mid \forall\alpha\, G \mid \mathsf{dist} & \text{Explicit containments} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha & \text{Environments}
\end{array}
$$

Figure 3.19: System F_η syntax

$$
\begin{array}{rcll}
E & ::= & \lambda(x:\tau)\, [] \mid []\, M \mid M\, [] \mid \langle [], M\rangle \mid \langle M, []\rangle \mid \mathsf{fst}\, [] \mid \mathsf{snd}\, [] & \text{Evaluation contexts} \\
& \mid & \Lambda\alpha\, [] \mid [][\tau] \mid G\langle []\rangle & \\
p & ::= & x \mid p\, v \mid \mathsf{fst}\, p \mid \mathsf{snd}\, p \mid p[\tau] \mid (\forall\alpha\, G)\langle p\rangle & \text{Prevalues} \\
& \mid & \mathsf{dist}\langle p\rangle \mid \mathsf{dist}\langle \Lambda\alpha\, p\rangle \mid (G \xrightarrow{\tau} G)\langle p\rangle \mid (G \times G)\langle p\rangle & \\
v & ::= & p \mid \lambda(x:\tau)\, v \mid \langle v, v\rangle \mid \Lambda\alpha\, v & \text{Values}
\end{array}
$$

Figure 3.20: System F_η notations

environment Γ. The typing rule follows:

$$
\frac{\Gamma \vdash \lambda x\, a\, x : \tau \to \sigma \qquad x \notin \mathsf{fv}(a)}{\Gamma \vdash a : \tau \to \sigma}\ \textsc{TermEtaArr}
\qquad\qquad
\frac{\Gamma \vdash \langle \mathsf{fst}\, a, \mathsf{snd}\, a\rangle : \tau \times \sigma}{\Gamma \vdash a : \tau \times \sigma}\ \textsc{TermEtaProd}
$$

This version actually explains the name of System F_η. This rule implies that a term a has type τ under Γ in System F_η, if there is a term b of type τ under Γ in System F such that b η-reduces to a.

3.4.1 Definition

System F_η extends System F . The syntax of terms is extended with containment applications $G\langle M\rangle$. A new syntactical class for containment proofs is added. Containment proofs are written G and contain the reflexivity proof $\Diamond$, transitivity proofs $G \circ G$, free generalizations $\Lambda\alpha$, type applications $\cdot\tau$, arrow subtypings $G \xrightarrow{\tau} G$, product subtypings $G \times G$, polymorphism congruences $\forall\alpha\, G$, and distributivity proofs dist. These changes with respect to System F are formally given on Figure 3.19.

The syntax of evaluation contexts, prevalues, and values for System F_η follow the same schema as System F. They are given on Figure 3.20. Evaluation contexts are all one-hole terms of depth one and define a strong notion of reduction. Prevalues are variables or destructors applied to prevalues. And values are prevalues or constructors applied to values. There are only new prevalues since containments proofs are only destructors in this setting. The polymorphism congruence waits for a type abstraction to reduce, so $(\forall\alpha\, G)\langle p\rangle$ is a prevalue. The distributivity proof waits a type abstraction followed by a computational constructor (like a term abstraction or a pair). So both $\mathsf{dist}\langle p\rangle$ and $\mathsf{dist}\langle \Lambda\alpha\, p\rangle$ may no reduce and are prevalues. Finally computational subtypings wait a computational constructor, so both $(G \xrightarrow{\tau} G)\langle p\rangle$ and $(G \times G)\langle p\rangle$ are prevalues.

Reduction rules for System F_η are given on Figure 3.21. The first five rules are those of System F. The next rules are new: one for each containment proof. The distributivity proof

$$\frac{M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}\ \textsc{RedCtx} \qquad \textsc{RedApp}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_\beta M[x/N] \qquad \textsc{RedFst}\ \mathsf{fst}\,\langle M,N\rangle \rightsquigarrow_\beta M \qquad \textsc{RedSnd}\ \mathsf{snd}\,\langle M,N\rangle \rightsquigarrow_\beta N$$

$$\textsc{RedFor}\ (\Lambda\alpha\,M)[\tau] \rightsquigarrow_\iota M[\alpha/\tau] \qquad \textsc{RedRefl}\ \Diamond\langle M\rangle \rightsquigarrow_\iota M \qquad \textsc{RedTrans}\ (G_2 \circ G_1)\langle M\rangle \rightsquigarrow_\iota G_2\langle G_1\langle M\rangle\rangle$$

$$\textsc{RedTLam}\ \Lambda\alpha\langle M\rangle \rightsquigarrow_\iota \Lambda\alpha\,M \qquad \textsc{RedTApp}\ \bullet\tau\langle M\rangle \rightsquigarrow_\iota M[\tau]$$

$$\textsc{RedArr}\ (G_1 \xrightarrow{\tau} G_2)\langle\lambda(x:\tau')\,M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\,G_2\langle M[x/G_1\langle x\rangle]\rangle$$

$$\textsc{RedProd}\ (G_1 \times G_2)\langle\langle M_1, M_2\rangle\rangle \rightsquigarrow_\iota \langle G_1\langle M_1\rangle, G_2\langle M_2\rangle\rangle \qquad \textsc{RedCongr}\ (\forall\alpha\,G)\langle\Lambda\alpha\,M\rangle \rightsquigarrow_\iota \Lambda\alpha\,G\langle M\rangle$$

$$\textsc{RedDistArr}\ \mathsf{dist}\langle\Lambda\alpha\,\lambda(x:\tau)\,M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\,\Lambda\alpha\,M \qquad \textsc{RedDistProd}\ \mathsf{dist}\langle\Lambda\alpha\,\langle M_1, M_2\rangle\rangle \rightsquigarrow_\iota \langle\Lambda\alpha\,M_1, \Lambda\alpha\,M_2\rangle$$

Figure 3.21: System F_η reduction rules

has two reduction rules since it has one reduction rule per computational type: arrow and product types in our setting. Rule RedRefl shows that the reflexivity proof does not modify a term and hence its typing. Rule RedTrans shows that coercing the term M with $G_2 \circ G_1$ is like coercing M with G_1 and then coercing the result with G_2. The typing modifications are done in sequence. Rules RedTLam and RedTApp simply unfold the definition of the generalization and instantiation containment proofs.

Rule RedArr expects a function and applies G_2 to its body and G_1 to its arguments. In doing so the type annotation of the abstraction changes from τ' to τ, which is why we had to annotate the arrow subtyping coercion. Rule RedProd expects a pair and applies G_1 to its first component and G_2 to its second one. Finally, rule RedCongr applies its subcontainment proof to the body of the type abstraction it is applied to. Rule RedDistArr swaps type and term abstractions. And rule RedDistProd swaps type abstraction and pairs.

The term judgment relation is defined on Figure 3.22. The first eight rules are those of System F. The new rule is the last one about containment, named TermCont. It tells that if a term a has type τ under Γ with witness M, then it also has type σ under the same environment Γ with witness $G\langle M\rangle$, given that G witnesses the containment proof that τ is smaller than σ under Γ.

This containment relation is actually a particularity of System F_η. Its judgment is written $G \Rightarrow \Gamma \vdash \tau \triangleright \sigma$ and means that G is a containment proof that the type τ is smaller than the type σ under the environment Γ. The rules are given on Figure 3.23. Notice that the extraction property for this judgment (Lemma 39) is quite particular, not only the well-formedness of Γ is an input, but also the well-formedness of $\Gamma \vdash \tau\ \mathsf{type}$ (which implies the well-formedness of Γ) is given as an input. This explains why premises about the well-formedness of τ are absent. This extraction property also explains how we can factorize the distributivity proofs for the arrow and product types in one proof: if the input type is an arrow then dist stands for the arrow distributivity rule, and if the input type is a product then dist stands for the product

$$\frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow \Gamma \vdash x:\tau}\ \textsc{TermVar} \qquad \frac{M \Rightarrow \Gamma,(x:\tau)\vdash a:\sigma}{\lambda(x:\tau)\,M \Rightarrow \Gamma\vdash \lambda x\,a:\tau\to\sigma}\ \textsc{TermLam}$$

$$\frac{M \Rightarrow \Gamma\vdash a:\tau\to\sigma \qquad N\Rightarrow\Gamma\vdash b:\tau}{M\,N\Rightarrow\Gamma\vdash a\,b:\sigma}\ \textsc{TermApp} \qquad \frac{M\Rightarrow\Gamma\vdash a:\tau \qquad N\Rightarrow\Gamma\vdash b:\sigma}{\langle M,N\rangle\Rightarrow\Gamma\vdash\langle a,b\rangle:\tau\times\sigma}\ \textsc{TermPair}$$

$$\frac{M\Rightarrow\Gamma\vdash a:\tau\times\sigma}{\mathsf{fst}\,M\Rightarrow\Gamma\vdash\mathsf{fst}\,a:\tau}\ \textsc{TermFst} \qquad \frac{M\Rightarrow\Gamma\vdash a:\tau\times\sigma}{\mathsf{snd}\,M\Rightarrow\Gamma\vdash\mathsf{snd}\,a:\sigma}\ \textsc{TermSnd} \qquad \frac{M\Rightarrow\Gamma,\alpha\vdash a:\tau}{\Lambda\alpha\,M\Rightarrow\Gamma\vdash a:\forall\alpha\,\tau}\ \textsc{TermGen}$$

$$\frac{M\Rightarrow\Gamma\vdash a:\forall\alpha\,\tau \qquad \Gamma\vdash\sigma\ \mathsf{type}}{M[\sigma]\Rightarrow\Gamma\vdash a:\tau[\alpha/\sigma]}\ \textsc{TermInst} \qquad \frac{M\Rightarrow\Gamma\vdash a:\tau \qquad G\Rightarrow\Gamma\vdash\tau\rhd\sigma}{G\langle M\rangle\Rightarrow\Gamma\vdash a:\sigma}\ \textsc{TermCont}$$

Figure 3.22: System F_η term judgment relation

distributivity rule. This factorization permits to have fewer type annotations on coercions and thus on terms.

Rule ContRefl defines $\Diamond$ as the containment proof that any type is contained in itself. Similarly rule ContTrans defines $G_2 \circ G_1$ as the proof for containment transitivity. If G_1 proves that τ_2 contains τ_1 and G_2 proves that τ_3 contains τ_2, then $G_2 \circ G_1$ proves that τ_3 contains τ_1 by transitivity.

Rule ContTLam tells that we can add a forall quantifier on any well-formed type as long as the type variable was not already bound. Said otherwise, we can quantify over any fresh type variable. This rule resemble TermGen with the main difference that, in TermGen, the type variable may be bound prior to quantification, whereas the containment rule only quantifies over fresh type variables. This difference will actually be removed in type systems System F_ι^p and System L^c (see Part II). Rule CoerTLam in Figure 4.11 has no restriction on the generalized type variable α, and is thus equivalent to rule TermGen. This is a crucial point if we want to define type systems as coercions, which is the technique we use in order to unify all the type systems of this chapter. Rule TermGen is about an erasable type, namely the polymorphic type, and should thus be in the coercion judgment and not in the typing judgment for terms.

Rule ContTApp is the analog of rule TermInst. It adds the instantiation typing rule in the containment relation and this time there is no loss of power. A polymorphic type is smaller than any of its instantiations as long as the argument type is well-formed.

Rule ContArr is a standard rule in type systems with subtyping. It is the arrow congruence rule, which means that it tells when an arrow type is smaller than another arrow type. Type $\tau' \to \sigma'$ is smaller than $\tau \to \sigma$ under Γ if τ is smaller than τ' under Γ and σ' is smaller than σ under Γ. Notice the inversion of inclusion for the domain types, this comes from the contravariance of the domain of the arrow type. This rule needs an additional premise for well-formedness: type τ has to be well-formed under Γ. Notice that the containment proof for this rule needs a type annotation. This annotation is needed for reduction to rebuild the abstraction type annotation in rule RedArr. Rule ContProd is similar. Type $\tau' \times \sigma'$ is

$$\frac{\textsc{ContRefl}}{\Diamond \Rightarrow \Gamma \vdash \tau \rhd \tau} \qquad \textsc{ContTrans}\ \frac{G_1 \Rightarrow \Gamma \vdash \tau_1 \rhd \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash \tau_2 \rhd \tau_3}{G_2 \circ G_1 \Rightarrow \Gamma \vdash \tau_1 \rhd \tau_3} \qquad \textsc{ContTLam}\ \frac{\alpha \notin \mathsf{dom}(\Gamma)}{\Lambda\alpha \Rightarrow \Gamma \vdash \tau \rhd \forall\alpha\,\tau}$$

$$\textsc{ContTApp}\ \frac{\Gamma \vdash \sigma\ \mathsf{type}}{\bullet\sigma \Rightarrow \Gamma \vdash \forall\alpha\,\tau \rhd \tau[\alpha/\sigma]} \qquad \textsc{ContArr}\ \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad G_1 \Rightarrow \Gamma \vdash \tau \rhd \tau' \qquad G_2 \Rightarrow \Gamma \vdash \sigma' \rhd \sigma}{G_1 \xrightarrow{\tau} G_2 \Rightarrow \Gamma \vdash \tau' \to \sigma' \rhd \tau \to \sigma}$$

$$\textsc{ContProd}\ \frac{G_1 \Rightarrow \Gamma \vdash \tau' \rhd \tau \qquad G_2 \Rightarrow \Gamma \vdash \sigma' \rhd \sigma}{G_1 \times G_2 \Rightarrow \Gamma \vdash \tau' \times \sigma' \rhd \tau \times \sigma} \qquad \textsc{ContCongr}\ \frac{G \Rightarrow \Gamma, \alpha \vdash \tau \rhd \sigma}{\forall\alpha\, G \Rightarrow \Gamma \vdash \forall\alpha\,\tau \rhd \forall\alpha\,\sigma}$$

$$\textsc{ContDistArr}\ \frac{\Gamma \vdash \tau\ \mathsf{type}}{\mathsf{dist} \Rightarrow \Gamma \vdash \forall\alpha\,(\tau \to \sigma) \rhd \tau \to \forall\alpha\,\sigma} \qquad \frac{\textsc{ContDistProd}}{\mathsf{dist} \Rightarrow \Gamma \vdash \forall\alpha\,(\tau \times \sigma) \rhd \forall\alpha\,\tau \times \forall\alpha\,\sigma}$$

Figure 3.23: System F_η containment judgment relation

smaller than $\tau \times \sigma$ under Γ if τ' is smaller than τ under Γ and σ' is smaller than σ under Γ.

Rule ContCongr is a congruence rule for the polymorphic type. If G is a proof that τ is contained in σ under the extended environment Γ, α, then $\forall\alpha\, G$ is a proof that $\forall\alpha\,\tau$ is contained in $\forall\alpha\,\sigma$ under Γ. This rule is the only rule binding a variable for a subproof.

The distributivity rule ContDistArr is quite particular to System F_η. Most type systems with subtyping do not have this rule. It tells that we can permute type abstraction and term abstraction as long as the term abstraction type does not depend on the abstract type. This rule, if applied to a term of type $\forall\alpha\,\tau \to \sigma$ under Γ, can actually be seen as a weakening lemma to add the term binding $(x : \tau)$ and the type binding α to get environment $\Gamma, (x : \tau), \alpha$, followed by a type instantiation with α to get type $\tau \to \sigma$, followed by an application rule on term x to get type σ, followed by a generalization over α to get type $\forall\alpha\,\sigma$ under $\Gamma, (x : \tau)$, and finally followed by an abstraction rule over x of type τ to get $\tau \to \forall\alpha\,\sigma$ under Γ as expected by the containment rule. These explanations can be summed up in the following typing derivation. We see in the final term $\lambda(x : \tau)\,\Lambda\alpha[]\,\alpha\,x$ that we permute the term and the type abstraction.

$$\dfrac{\dfrac{\dfrac{\dfrac{[] \Rightarrow \Gamma, (x : \tau), \alpha \vdash [] : \forall\alpha\,\tau \to \sigma}{[]\,\alpha \Rightarrow \Gamma, (x : \tau), \alpha \vdash [] : \tau \to \sigma} \qquad x \Rightarrow \Gamma, (x : \tau), \alpha \vdash x : \tau}{[]\,\alpha\,x \Rightarrow \Gamma, (x : \tau), \alpha \vdash []\,x : \sigma}}{\Lambda\alpha[]\,\alpha\,x \Rightarrow \Gamma, (x : \tau) \vdash []\,x : \forall\alpha\,\sigma}}{\lambda(x : \tau)\,\Lambda\alpha[]\,\alpha\,x \Rightarrow \Gamma \vdash \lambda x\,[]\,x : \tau \to \forall\alpha\,\sigma}$$

Rule ContDistProd is similar but for the product type. It allows to permute a type abstraction and a pair. A polymorphic pair of expressions can be seen as a pair of polymorphic expressions. These two distribution rules are derivable in our type systems as a consequence of our rule CoerTLam which generalizes rule ContTLam by generalizing over bound type variable and not only fresh type variables.

$$\frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \textsc{TypeVar} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}}\ \textsc{TypeArr} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}\ \textsc{TypeProd} \qquad \frac{\Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma \vdash \forall \alpha\, \tau\ \mathsf{type}}\ \textsc{TypeFor}$$

$$\textsc{EnvEmpty}\ \ \varnothing\ \mathsf{env} \qquad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}}\ \textsc{EnvTerm} \qquad \frac{\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma\ \mathsf{env}}{\Gamma, \alpha\ \mathsf{env}}\ \textsc{EnvType}$$

Figure 3.24: System F_η well-formedness relations

Finally the well-formedness relations are given on Figure 3.24. These rules are exactly the same as those of System F.

3.4.2 Properties

The properties of System F_η are similar to those of System F. However a few lemma have to be extended to the new containment judgment. The first lemma is about the uniqueness of the implicit judgment according to its explicit entity. Notice that only the type on the right-hand side of the containment judgment is unique, the rest of the judgment has to be given along the proof G. This explains why a single distributivity proof dist is sufficient.

Lemma 37 (Uniqueness)**.** *The following assertions hold.*

- *If* $M \Rightarrow \Gamma_1 \vdash a_1 : \tau_1$ *and* $M \Rightarrow \Gamma_2 \vdash a_2 : \tau_2$ *hold, then* $a_1 = a_2$ *holds.*
- *If* $M \Rightarrow \Gamma \vdash a : \tau_1$ *and* $M \Rightarrow \Gamma \vdash a : \tau_2$ *hold, then* $\tau_1 = \tau_2$ *holds.*
- *If* $G \Rightarrow \Gamma \vdash \tau \rhd \sigma_1$ *and* $G \Rightarrow \Gamma \vdash \tau \rhd \sigma_2$ *hold, then* $\sigma_1 = \sigma_2$ *holds.*

The equivalence lemma also has to be extended. However, this extension is much more natural, because it follows our schema about explicit entities witnessing judgments. From an implicit containment judgment, we can create a proof G witnessing this judgment.

Lemma 38 (Equivalence)**.** *The following assertions hold.*

- $\Gamma \vdash a : \tau$ *holds if and only if* $M \Rightarrow \Gamma \vdash a : \tau$ *holds for some* M*.*
- $\Gamma \vdash \tau \rhd \sigma$ *holds if and only if* $G \Rightarrow \Gamma \vdash \tau \rhd \sigma$ *holds for some* G*.*

Finally, the extraction lemma is extended for containments. A containment derivation can transform the well-formedness of its left-hand type to the well-formedness of its right-hand type.

Lemma 39 (Extraction)**.** *The following assertions hold:*

- *If* $\Gamma \vdash \tau\ \mathsf{type}$ *holds, then* $\Gamma\ \mathsf{env}$ *holds.*
- *If* $M \Rightarrow \Gamma \vdash a : \tau$ *holds, then* $\Gamma \vdash \tau\ \mathsf{type}$ *holds.*
- *If* $G \Rightarrow \Gamma \vdash \tau \rhd \sigma$ *and* $\Gamma \vdash \tau\ \mathsf{type}$ *hold, then* $\Gamma \vdash \sigma\ \mathsf{type}$ *holds.*

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
c_\alpha & & & \text{Instance variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
& \mid & \Lambda(\alpha \triangleleft c : \tau)\, M \mid G\langle M\rangle & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall(\alpha \triangleleft \tau)\, \tau \mid \bot & \text{Types} \\
G & ::= & \bot\tau \mid c_\alpha \mid \forall(\triangleleft\, G) \mid \forall(\alpha \triangleleft c :)\, G \mid \& \mid \invamp \mid G \circ G \mid \Diamond & \text{Instantiations} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha \mid \Gamma, (c_\alpha : \tau \triangleright \tau) & \text{Environments}
\end{array}
$$

Figure 3.25: MLF syntax

3.5 MLF

An important feature for surface type systems, which are type systems for the programmer, is inference. Inference allows programmers to avoid writing all the typing derivations but only parts of them. Ideally and without any documentation consideration, we might want the programmer to write only the lambda term he wants to run. And at most we want him to write the explicit term. While previous type systems (but the STLC) have no complete inference, MLF which is more expressive than System F has a complete inference as long as the function parameters that are used polymorphically are annotated. However, we do not present MLF in its surface type system version[16], but in its kernel version[29], which is posterior.

3.5.1 Definition

MLF extends the STLC with instance-bounded polymorphism (which we may also refer to as lower bounded polymorphism) and bottom type. The polymorphic type of MLF abstracts over a type variable that is an instance of a lower bound type, instead of just abstracting over a type variable as it is the case in System F. We recover the polymorphic type $\forall\alpha\, \tau$ of System F by using an instance-bounded polymorphic type $\forall(\alpha \triangleleft \bot)\, \tau$ with a particular lower bound called bottom. All types are instance of the bottom type, and thus we can instantiate a lower bounded polymorphic type with bound bottom $\forall(\alpha \triangleleft \bot)\, \tau$ with any type σ as we can do in System F, because we have $\bot \triangleright \sigma$. So MLF is an extension of System F by its features and an extension of the STLC by its syntax.

MLF adds type and instance variables to the STLC. Type variables are written α or β, while instance variables are written c_α. All instance variables are linked to a unique type variable. Terms are extended with lower bounded abstraction $\Lambda(\alpha \triangleleft c : \tau)\, M$ and instance application $G\langle M\rangle$. Lower bounded abstraction abstracts over both the type variable α and the instance variable c of instance type $\tau \triangleright \alpha$. The instance variable c is associated to the type variable α, and we refer to it as c_α to enhance this association. The instance application uses an instance proof G of type $\tau \triangleright \sigma$ to retype the term M from type τ to σ. Types are extended with type variables α, lower bounded polymorphic types $\forall(\alpha \triangleleft \tau)\, \tau$, and the bottom type $\bot$. A syntactic class for instantiation proofs is added and written with the letter G. They contain bottom instantiations $\bot\tau$, variables c_α, inside instantiations $\forall(\triangleleft\, G)$, under instantiations $\forall(\alpha \triangleleft c :)\, G$, elimination instantiations $\&$, introduction instantiations $\invamp$, and the usual transitivity $G \circ G$ and reflexivity $\Diamond$. Finally, we extend environments with type and instance bindings. Type bindings are similar to System F, while instance bindings are new. In the implicit version, we remember that type σ is an instance of type τ by writing $(\tau \triangleright \sigma)$.

$$
\begin{array}{rcll}
E & ::= & \lambda(x:\tau)\,[] \mid []\,M \mid M\,[] \mid \langle [], M\rangle \mid \langle M, []\rangle \mid \mathsf{fst}\,[] \mid \mathsf{snd}\,[] & \text{Evaluation contexts} \\
 & \mid & \Lambda(\alpha \triangleleft c:\tau)\,[] \mid G\langle [] \rangle & \\
p & ::= & x \mid p\,v \mid \mathsf{fst}\,p \mid \mathsf{snd}\,p \mid \&\langle p\rangle & \text{Prevalues} \\
 & \mid & (\forall(\alpha \triangleleft c:)\,G)\langle p\rangle \mid (\forall(\triangleleft\, G))\langle p\rangle \mid \bot\tau\langle p\rangle \mid c_\alpha\langle v\rangle & \\
v & ::= & p \mid \lambda(x:\tau)\,v \mid \langle v, v\rangle \mid \Lambda(\alpha \triangleleft c:\tau)\,v & \text{Values}
\end{array}
$$

Figure 3.26: MLF notations

RedCtx
$$\frac{M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}$$

RedApp
$$(\lambda(x:\tau)\,M)\,N \rightsquigarrow_\beta M[x/N]$$

RedFst
$$\mathsf{fst}\,\langle M, N\rangle \rightsquigarrow_\beta M$$

RedSnd
$$\mathsf{snd}\,\langle M, N\rangle \rightsquigarrow_\beta N$$

RedId
$$\Diamond\langle M\rangle \rightsquigarrow_\iota M$$

RedSeq
$$(G_2 \circ G_1)\langle M\rangle \rightsquigarrow_\iota G_2\langle G_1\langle M\rangle\rangle$$

RedIntro
$$\frac{c_\alpha, \alpha \notin \mathsf{fv}(M)}{\parr\langle M\rangle \rightsquigarrow_\iota \Lambda(\alpha \triangleleft c:\bot)\,M}$$

RedElim
$$\&\langle \Lambda(\alpha \triangleleft c:\tau)\,M\rangle \rightsquigarrow_\iota M[\alpha/\tau][c_\alpha/\Diamond]$$

RedUnder
$$(\forall(\alpha \triangleleft c:)\,G)\langle \Lambda(\alpha \triangleleft c:\tau)\,M\rangle \rightsquigarrow_\iota \Lambda(\alpha \triangleleft c:\tau)\,G\langle M\rangle$$

RedInside
$$(\forall(\triangleleft\, G))\langle \Lambda(\alpha \triangleleft c:\tau)\,M\rangle \rightsquigarrow_\iota \Lambda(\alpha \triangleleft c:G(\tau))\,M[c_\alpha/c_\alpha \circ G]$$

Figure 3.27: MLF reduction rules

However, for the explicit version, we need to give a name to the proof that σ is an instance of τ, so we write $(c_\alpha : \tau \triangleright \sigma)$. All these changes are summed up on Figure 3.25.

Evaluation contexts, prevalues, and values for MLF are given in Figure 3.26. Prevalues are extended with instance applications to a prevalue of the elimination instantiation, the under instantiation, the inside instantiation, the bottom instantiation, and instantiation applications to a value of a instance variable. Values are extended with lower bounded abstraction of a value.

Reduction rules for MLF are given on Figure 3.27. The first four rules are those of the STLC. The next rules are new: one for each instantiation proof. Notice however that rules RedRefl and RedTrans are exactly like those of System F_η. All the new rules are ι-reduction rules since they do not modify the computational content of the term to which they apply.

Rule RedIntro is similar in spirit to rule RedTLam of System F_η. However, in System F_η, the type variable is syntactically present, while in MLF it has to be generated. In both cases the variable has to be fresh. In MLF it is generated fresh, while in System F_η it is fresh by typing. So an abstraction with a bottom lower bound can be freely added to a term.

Rule RedElim is again similar to the rule RedFor of System F_η: both are instantiation rules. However, there are two main differences. In System F_η, the instantiation argument is given separately, while in MLF it defaults to the bound τ. The second difference is that the instance variable has also to be substituted in MLF. Since the instance variable is taken equal to the bound, the reflexivity proof witnesses that τ is an instance of τ.

Rule RedUnder is very close to rule RedCongr of System F_η: both are congruence rules

$$
\begin{array}{rr}
c_\alpha(\tau) = \alpha & \mathfrak{P}(\tau) = \forall(\alpha \triangleleft \bot)\, \tau \text{ with } \alpha \notin \mathsf{fv}(\tau) \\
(\bot\tau)(\bot) = \tau & \&(\forall(\alpha \triangleleft \tau)\, \sigma) = \sigma[\alpha/\tau] \\
\Diamond(\tau) = \tau & (\forall(\triangleleft\, G))(\forall(\alpha \triangleleft \tau)\, \sigma) = \forall(\alpha \triangleleft G(\tau))\, \sigma \\
(G_2 \circ G_1)(\tau) = G_2(G_1(\tau)) & (\forall(\alpha \triangleleft c :)\, G)(\forall(\alpha \triangleleft \tau)\, \sigma) = \forall(\alpha \triangleleft \tau)\, G(\sigma)
\end{array}
$$

Figure 3.28: MLF update function

$$
\textsc{TermVar}\ \frac{\Gamma\ \mathsf{env} \qquad (x : \tau) \in \Gamma}{x \Rightarrow \Gamma \vdash x : \tau}
\qquad
\textsc{TermLam}\ \frac{M \Rightarrow \Gamma, (x : \tau) \vdash a : \sigma}{\lambda(x : \tau)\, M \Rightarrow \Gamma \vdash \lambda x\, a : \tau \to \sigma}
$$

$$
\textsc{TermApp}\ \frac{M \Rightarrow \Gamma \vdash a : \tau \to \sigma \qquad N \Rightarrow \Gamma \vdash b : \tau}{M\, N \Rightarrow \Gamma \vdash a\, b : \sigma}
\qquad
\textsc{TermPair}\ \frac{M \Rightarrow \Gamma \vdash a : \tau \qquad N \Rightarrow \Gamma \vdash b : \sigma}{\langle M, N \rangle \Rightarrow \Gamma \vdash \langle a, b \rangle : \tau \times \sigma}
$$

$$
\textsc{TermFst}\ \frac{M \Rightarrow \Gamma \vdash a : \tau \times \sigma}{\mathsf{fst}\, M \Rightarrow \Gamma \vdash \mathsf{fst}\, a : \tau}
\qquad
\textsc{TermSnd}\ \frac{M \Rightarrow \Gamma \vdash a : \tau \times \sigma}{\mathsf{snd}\, M \Rightarrow \Gamma \vdash \mathsf{snd}\, a : \sigma}
\qquad
\textsc{TermTAbs}\ \frac{M \Rightarrow \Gamma, \alpha, (c_\alpha : \tau \triangleright \alpha) \vdash a : \sigma}{\Lambda(\alpha \triangleleft c : \tau)\, M \Rightarrow \Gamma \vdash a : \forall(\alpha \triangleleft \tau)\, \sigma}
$$

$$
\textsc{TermTApp}\ \frac{M \Rightarrow \Gamma \vdash a : \tau \qquad G \Rightarrow \Gamma \vdash \tau \triangleright \sigma}{G\langle M \rangle \Rightarrow \Gamma \vdash a : \sigma}
$$

Figure 3.29: MLF term judgment relation

for the polymorphic type. The only difference is anecdotal: MLF handles a lower bound while System F_η does not. Notice that, similarly to System F_η, the two instance variables have to be the same and the two type variables have to be the same too. If this is not the case, one of them must be renamed before the reduction can occur. The first two variables bind in the instance proof G, while the last two variables bind in the term M.

Finally, rule RedInside has no counter-part in System F_η since it is about the bound. If a lower bound abstraction $\Lambda(\alpha \triangleleft c : \tau)M$ encounters an inside instantiation $\forall(\triangleleft\, G)$, the instantiation proof G modifies the lower bound according to the previous bound τ, written $G(\tau)$ and define in Figure 3.28, and the instance variable c_α is updated in order to apply the instantiation G before itself. Together with rule RedElim it allows arbitrary instantiations with any instance of the lower bound, since it applies a first instantiation followed by an application of the resulting bound.

The term judgment relation is defined on Figure 3.29. The first six rules are those of the STLC. The last two rules are new. However the last one, named TermTApp, resembles the rule TermCont of System F_η. It explains how an instantiation is used: when a term a has type τ with proof M and σ is an instance of τ with proof G, then the term a has also type σ with proof $G\langle M \rangle$. All of this happens under the environment Γ because no bindings occur.

Rule TermTAbs is the abstraction rule for lower bounded polymorphism. If a term a has type σ under the environment Γ extended with the type variable α and its associated instance variable c_α witnessing that α is an instance of τ, then it also has type $\forall(\alpha \triangleleft \tau)\, \sigma$ under environment Γ. The term witnessing this rule is $\Lambda(\alpha \triangleleft c : \tau)M$.

$$\textsc{InstBot}\quad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\bot\tau \Rightarrow \Gamma \vdash \bot \triangleright \tau} \qquad \textsc{InstUnder}\quad \frac{G \Rightarrow \Gamma, \alpha, (c_\alpha : \tau \triangleright \alpha) \vdash \sigma_1 \triangleright \sigma_2}{\forall(\alpha \triangleleft c :)\, G \Rightarrow \Gamma \vdash \forall(\alpha \triangleleft \tau)\, \sigma_1 \triangleright \forall(\alpha \triangleleft \tau)\, \sigma_2} \qquad \textsc{InstAbstr}\quad \frac{(c_\alpha : \tau \triangleright \alpha) \in \Gamma}{c_\alpha \Rightarrow \Gamma \vdash \tau \triangleright \alpha}$$

$$\textsc{InstInside}\quad \frac{G \Rightarrow \Gamma \vdash \tau_1 \triangleright \tau_2}{\forall(\ \triangleleft G) \Rightarrow \Gamma \vdash \forall(\alpha \triangleleft \tau_1)\, \sigma \triangleright \forall(\alpha \triangleleft \tau_2)\, \sigma} \qquad \textsc{InstIntro}\quad \frac{\alpha \notin \mathsf{dom}(\Gamma)}{⅋ \Rightarrow \Gamma \vdash \tau \triangleright \forall(\alpha \triangleleft \bot)\, \tau}$$

$$\textsc{InstComp}\quad \frac{G_1 \Rightarrow \Gamma \vdash \tau_1 \triangleright \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash \tau_2 \triangleright \tau_3}{G_2 \circ G_1 \Rightarrow \Gamma \vdash \tau_1 \triangleright \tau_3} \qquad \textsc{InstElim}\quad \& \Rightarrow \Gamma \vdash \forall(\alpha \triangleleft \tau)\, \sigma \triangleright \sigma[\alpha/\tau] \qquad \textsc{InstId}\quad \Diamond \Rightarrow \Gamma \vdash \tau \triangleright \tau$$

Figure 3.30: MLF instance judgment relation

The instance relation of MLF is similar in some points to the containment relation of System F_η. Its judgment is written $G \Rightarrow \Gamma \vdash \tau \triangleright \sigma$, as in System F_η, and means that G is a proof that the type σ is an instance of the type τ under the environment Γ. The rules are given on Figure 3.30. Similarly to System F_η, the extraction property for this judgment takes the well-formedness of its left-hand type as an input. Rules InstId and InstComp are exactly the rules ContRefl and ContTrans of System F_η. Rule InstIntro is very similar to rule ContTLam of System F_η since it generalizes over a free type variable. The difference is about the bound, but since the bound is the bottom type, it behaves exactly as a polymorphic type.

Rule InstElim resembles rule ContTApp of System F_η since it is about instantiation. The difference is that the instance type argument is not provided but defaults to the lower bound τ. We see in this rule that MLF does not allow recursive bounds. With recursive bounds α would be free in τ and the substitution $\sigma[\alpha/\tau]$ would be ill-formed. With recursive bounds the solution would be to instantiate with the recursive type $\mu\alpha\,\tau$ and the proof that $\mu\alpha\,\tau$ is an instance of $\tau[\alpha/\mu\alpha\,\tau]$ would be the unfolding coercion (see rule CoerUnfold in Figure 5.6).

Rule InstAbstr looks for an instance hypothesis in the environment. If the type variable α is an instance of the type τ by hypothesis in environment Γ with name c_α, then c_α is a proof that α is an instance of τ under Γ. By rule InstBot, any well-formed type τ is an instance of the bottom type $\bot$ with proof $\bot\tau$.

Rule InstUnder is the body congruence rule for the lower bounded polymorphic type, while rule InstInside is the bound congruence rule. A lower bounded polymorphic type is an instance of another lower bounded polymorphic type if they have the same bound and the body of the first is an instance of the body of the second under the same environment extended with the assumption of the mutual bound. Said otherwise: if the type σ_2 is an instance of type σ_1 under the environment Γ extended with the type variable α and the assumption that α is an instance of τ, then the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau)\, \sigma_2$ is an instance of the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau)\, \sigma_1$. Similarly for the bound congruence, two lower bounded polymorphic types are in the instance relation if their bounds are also in the instance relation in the same order. If the bound τ_2 is an instance of the bound τ_1 under the environment Γ (because recursive bounds are not allowed), then the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau_2)\, \sigma$ is an instance of the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau_1)\, \sigma$ under the same environment Γ.

Finally the well-formedness relations are given on Figure 3.31. The first three rules for type

$$\frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \text{TypeVar} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}}\ \text{TypeArr} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}\ \text{TypeProd}$$

$$\frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma, \alpha, (c_\alpha : \tau \rhd \alpha) \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \forall(\alpha \lhd \tau)\, \sigma\ \mathsf{type}}\ \text{TypeFor} \qquad \frac{\Gamma\ \mathsf{env}}{\Gamma \vdash \bot\ \mathsf{type}}\ \text{TypeBot}$$

$$\frac{}{\varnothing\ \mathsf{env}}\ \text{EnvEmpty} \qquad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}}\ \text{EnvTerm} \qquad \frac{\alpha, c_\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, \alpha, (c_\alpha : \tau \rhd \alpha)\ \mathsf{env}}\ \text{EnvType}$$

Figure 3.31: MLF well-formedness relations

well-formedness are the same as those for System F. The next two rules are new. Rule TypeFor is an adaptation of the System F rule of the same name. A lower bounded polymorphic type is well-formed if its body type is well-formed under the environment extended with its binding, namely the type variable α and the instance assumption that α is an instance of the lower well-formed bound τ. Rule TypeBot tells that the bottom type is well-formed under any well-formed environment.

The first two rules of the environment well-formedness are the same as those of the STLC. The last rule, namely EnvType, is new. The extended environment $\Gamma, \alpha, (c_\alpha : \tau \rhd \alpha)$ is well-formed if τ is well-formed under the environment Γ. Besides, the instance variable c_α and the type variable α must not already be bound in Γ.

3.5.2 Properties

The properties of MLF are similar to those of System F with some modifications about the bounds. The type substitution lemma of System F is replaced by a lower bound substitution lemma.

Lemma 40 (Lower bound substitution)**.** *If $\Gamma \vdash \sigma$ type and $G \Rightarrow \Gamma \vdash \tau \rhd \sigma$ hold, then the following assertions hold.*

- *If $\Gamma, \alpha, (c_\alpha : \tau \rhd \alpha), \Gamma' \vdash \rho$ type holds, then $\Gamma, \Gamma'[\alpha/\sigma] \vdash \rho[\alpha/\sigma]$ type holds.*
- *If $M \Rightarrow \Gamma, \alpha, (c_\alpha : \tau \rhd \alpha), \Gamma' \vdash a : \rho$ holds, then $M[\alpha/\sigma][c_\alpha/G] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash a : \rho[\alpha/\sigma]$ holds.*
- *If $G' \Rightarrow \Gamma, \alpha, (c_\alpha : \tau \rhd \alpha), \Gamma' \vdash \rho \rhd \rho'$ holds, then $G'[\alpha/\sigma][c_\alpha/G] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash \rho[\alpha/\sigma] \rhd \rho'[\alpha/\sigma]$ holds.*

3.6 System $\mathsf{F}_{<:}$

System $\mathsf{F}_{<:}$ is another extension of System F dealing with the notion of subtyping. However, there are two main differences with System F_η. On the one hand, System $\mathsf{F}_{<:}$ lacks the distributivity rule. However on the other hand, System $\mathsf{F}_{<:}$ enjoys upper bounded polymorphism,

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
c & & & \text{Subtyping variables} \\
M, N & ::= & x \mid \lambda(x : \tau)\, M \mid M\, M \mid \langle M, M \rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M & \text{Explicit terms} \\
& \mid & \Lambda(\alpha \triangleright c : \tau)\, M \mid M[\tau \triangleright G] \mid G\langle M \rangle & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall(\alpha \triangleright \tau)\, \tau \mid \top & \text{Types} \\
G & ::= & \Diamond \mid G \circ G \mid c \mid \top \mid G \xrightarrow{\tau} G \mid G \times G \mid \forall(\alpha \triangleright c : \tau)[G, G] & \text{Explicit subtypings} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x : \tau) \mid \Gamma, \alpha \mid \Gamma, (c : \tau \triangleright \tau) & \text{Environments}
\end{array}
$$

Figure 3.32: System $\mathsf{F}_{<:}$ syntax

$$
\begin{array}{rcll}
E & ::= & \lambda(x : \tau)\, [] \mid []\, M \mid M\, [] \mid \langle [], M \rangle \mid \langle M, [] \rangle \mid \mathsf{fst}\, [] \mid \mathsf{snd}\, [] & \text{Evaluation contexts} \\
& \mid & \Lambda(\alpha \triangleright c : \tau)\, [] \mid [][\tau \triangleright G] \mid G\langle [] \rangle & \\
p & ::= & x \mid p\, v \mid \mathsf{fst}\, p \mid \mathsf{snd}\, p \mid p[\tau \triangleright G] \mid (\forall(\alpha \triangleright c : \tau)[G, G])\langle p \rangle & \text{Prevalues} \\
& \mid & (G \xrightarrow{\tau} G)\langle p \rangle \mid (G \times G)\langle p \rangle \mid c\langle v \rangle & \\
v & ::= & p \mid \lambda(x : \tau)\, v \mid \langle v, v \rangle \mid \Lambda(\alpha \triangleright c : \tau)\, v \mid \top\langle v \rangle & \text{Values}
\end{array}
$$

Figure 3.33: System $\mathsf{F}_{<:}$ notations

which can be seen as an analog of the lower bounded polymorphism of MLF. The analogy only holds for the syntax of types: lower bounded polymorphism in MLF is fully composable, while upper bounded polymorphism is shallow in $\mathsf{F}_{<:}$. We detail these ideas with coercions in Section 4.5.4. The polymorphic type of System $\mathsf{F}_{<:}$ abstracts over a type variable smaller than an upper bound type. We recover the polymorphic type of System F by using a particular upper bound called top. All types are smaller than the top type, and thus we can instantiate an upper bounded polymorphic type with bound top with any type we want as we can already do in System F. This is why System $\mathsf{F}_{<:}$ is an extension of System F in terms of features. But we actually prefer to see it as a syntactical extension of the STLC with subtyping, the top type, and upper bounded polymorphism. We will see in Section 4.5.4 that System $\mathsf{F}_{<:}$ is actually not complete according to how it extends the STLC.

3.6.1 Definition

System $\mathsf{F}_{<:}$ extends the STLC syntax with type variables, written α or β, and subtyping variables, written c. Subtyping variables are only necessary in the explicit version of the type system. Types are extended with type variables α, upper bounded polymorphic types $\forall(\alpha \triangleright \tau)\, \tau$, and the top type $\top$. Terms are extended with upper bounded type abstractions $\Lambda(\alpha \triangleright c : \tau)\, M$, upper bounded type applications $M[\tau \triangleright G]$, and subtyping applications $G\langle M \rangle$. A new class for subtypings is added. Subtypings are written G and contain reflexivity $\Diamond$, transitivity $G \circ G$, variables c, top subtypings $\top$, arrow congruence $G \xrightarrow{\tau} G$, product congruence $G \times G$, and upper bounded congruence $\forall(\alpha \triangleright c : \tau)[G, G]$. Finally, we extend environments with type and subtyping bindings. Type bindings are similar to System F, while subtyping bindings are new. In the implicit version, we remember that type τ is a subtype of type σ by writing $(\tau \triangleright \sigma)$. However, for the explicit version, we need to give a name to the proof that τ is a subtype of σ, so we write $(c : \tau \triangleright \sigma)$. These modifications can be found on Figure 3.32.

$$\frac{M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}\ \textsc{RedCtx} \qquad \textsc{RedApp}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_\beta M[x/N] \qquad \textsc{RedFst}\ \mathsf{fst}\,\langle M, N\rangle \rightsquigarrow_\beta M \qquad \textsc{RedSnd}\ \mathsf{snd}\,\langle M, N\rangle \rightsquigarrow_\beta N$$

$$\textsc{RedFor}\ (\Lambda(\alpha \triangleright c:\tau)\,M)[\sigma \triangleright G] \rightsquigarrow_\iota M[\alpha/\sigma][c/G] \qquad \textsc{RedRefl}\ \Diamond\langle M\rangle \rightsquigarrow_\iota M \qquad \textsc{RedTrans}\ (G_2 \circ G_1)\langle M\rangle \rightsquigarrow_\iota G_2\langle G_1\langle M\rangle\rangle$$

$$\textsc{RedArr}\ (G_1 \xrightarrow{\tau} G_2)\langle\lambda(x:\tau')\,M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\,G_2\langle M[x/G_1\langle x\rangle]\rangle$$

$$\textsc{RedProd}\ (G_1 \times G_2)\langle\langle M_1, M_2\rangle\rangle \rightsquigarrow_\iota \langle G_1\langle M_1\rangle, G_2\langle M_2\rangle\rangle$$

$$\textsc{RedCongr}\ (\forall(\alpha \triangleright c:\tau')[G_1, G_2])\langle\Lambda(\alpha \triangleright c:\tau)\,M\rangle \rightsquigarrow_\iota \Lambda(\alpha \triangleright c:\tau')\,G_2\langle M[c/G_1]\rangle$$

Figure 3.34: System $\mathsf{F}_{<:}$ reduction rules

Notations for System $\mathsf{F}_{<:}$ follow the same logic as the STLC. They are given on Figure 3.33. Evaluation contexts are all one-hole terms of depth one and correspond to strong reduction. Prevalues are variables or destructors applied to values but where a constructor is expected, in which case only a prevalue is applied. And values are prevalues or constructors applied to values. The new prevalues are: upper bounded application since it waits an upper bounded abstraction, congruences because they wait their constructor (abstraction or pair), and subtyping variables applied to a value. The new values are upper bounded abstraction, and the top subtyping.

Reduction rules for System $\mathsf{F}_{<:}$ are given on Figure 3.34. The first four rules are those of the STLC. The next rules are new: one for each subtyping proof. Notice however that rules RedRefl, RedTrans, RedArr, and RedProd are exactly like those of System F_η. So we only focus on the two new rules RedFor and RedCongr, which are actually ameliorations of the System F_η rules of the same name. All these rules are ι-reduction rules since they do not modify the computational content of the term on which they apply.

Rule RedFor matches an upper bounded application over an upper bounded abstraction. This is a regular scheme for abstraction constructs. It reduces to a substitution. In the current case, we not only substitutes the type variable α with the argument type σ as it is done in the System F rule, but we also substitute the subtyping variable c with the argument subtyping proof G.

Finally, rule RedCongr applies a subtyping proof under an upper bounded abstraction. This is more technical than in System F_η since now the bounds may be modified too. The binding $(c : \alpha \triangleright \tau')$ is the new binding, G_2 is the subproof to apply on the term body and G_1 is the subtyping proof to update the bound from τ to τ' and replaces the old subtyping variable c.

The term judgment relation is defined on Figure 3.35. The first six rules are those of the STLC. The last three rules are new. However the last one, named TermSub, resembles the rule TermCont of System F_η. It explains how a subtyping proof is used: when a term a has type τ with proof M and τ is a subtype of σ with proof G, then the term a has also type σ

$$\frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow \Gamma \vdash x:\tau}\ \textsc{TermVar} \qquad \frac{M \Rightarrow \Gamma,(x:\tau)\vdash a:\sigma}{\lambda(x:\tau)\,M \Rightarrow \Gamma \vdash \lambda x\,a:\tau\to\sigma}\ \textsc{TermLam}$$

$$\frac{M \Rightarrow \Gamma \vdash a:\tau\to\sigma \qquad N \Rightarrow \Gamma\vdash b:\tau}{M\,N \Rightarrow \Gamma\vdash a\,b:\sigma}\ \textsc{TermApp} \qquad \frac{M \Rightarrow \Gamma\vdash a:\tau \qquad N\Rightarrow\Gamma\vdash b:\sigma}{\langle M,N\rangle \Rightarrow \Gamma\vdash\langle a,b\rangle:\tau\times\sigma}\ \textsc{TermPair}$$

$$\frac{M\Rightarrow\Gamma\vdash a:\tau\times\sigma}{\mathsf{fst}\,M\Rightarrow\Gamma\vdash\mathsf{fst}\,a:\tau}\ \textsc{TermFst} \qquad \frac{M\Rightarrow\Gamma\vdash a:\tau\times\sigma}{\mathsf{snd}\,M\Rightarrow\Gamma\vdash\mathsf{snd}\,a:\sigma}\ \textsc{TermSnd} \qquad \frac{M\Rightarrow\Gamma,\alpha,(c:\alpha\rhd\tau)\vdash a:\rho}{\Lambda(\alpha\rhd c:\tau)\,M\Rightarrow\Gamma\vdash a:\forall(\alpha\rhd\tau)\,\rho}\ \textsc{TermGen}$$

$$\frac{M\Rightarrow\Gamma\vdash a:\forall(\alpha\rhd\tau)\,\rho \qquad \Gamma\vdash\sigma\ \mathsf{type} \qquad G\Rightarrow\Gamma\vdash\sigma\rhd\tau[\alpha/\sigma]}{M[\sigma\rhd G]\Rightarrow\Gamma\vdash a:\rho[\alpha/\sigma]}\ \textsc{TermInst}$$

$$\frac{M\Rightarrow\Gamma\vdash a:\tau \qquad G\Rightarrow\Gamma\vdash\tau\rhd\sigma}{G\langle M\rangle\Rightarrow\Gamma\vdash a:\sigma}\ \textsc{TermSub}$$

Figure 3.35: System $\mathsf{F}_{<:}$ term judgment relation

with proof $G\langle M\rangle$. All of this happens under the environment Γ because no bindings occur.

Rules TermGen and TermInst are ameliorations of the System F rules of the same name. The abstraction and application are now for upper bounded types. If a term a has type ρ under an extended environment $\Gamma,\alpha,(c:\alpha\rhd\tau)$, where c is a subtyping proof that α is a subtype of τ, with witness M, then it also has type $\forall(\alpha\rhd\tau)\,\rho$ under environment Γ with witness $\Lambda(\alpha\rhd c:\tau)\,M$. This witness binds c in M and α in τ and M. As a consequence, unlike for MLF, upper bounds are recursive in System $\mathsf{F}_{<:}$. If a term a has type $\forall(\alpha\rhd\tau)\,\rho$ under environment Γ with witness M, then it also has type $\rho[\alpha/\sigma]$ under environment Γ by instantiation with the well-formed type σ and the proof G that σ is a subtype of $\tau[\alpha/\sigma]$. The type variable α may appear in the type τ, since we allow recursive type. This is why the right-hand type of the subtyping proof is not simply τ.

The subtyping relation of System $\mathsf{F}_{<:}$ is similar in some points to the containment relation of System F_η. Its judgment is written $G\Rightarrow\Gamma\vdash\tau\rhd\sigma$, as in System F_η, and means that G is a proof that the type τ is a subtype of the type σ under the environment Γ. The rules are given on Figure 3.36. Similarly to System F_η, the extraction property for this judgment takes the well-formedness of its left-hand type as an input. Rule SubRefl, SubTrans, SubArr, and SubProd are exactly the rules ContRefl, ContTrans, ContArr, and ContProd of System F_η.

Rule SubVar looks a subtyping hypothesis in the environment. If the type τ is a subtype of the type σ by hypothesis in environment Γ with name c, then c is a proof that τ is a subtype of σ under Γ. By rule SubTop, any type τ is a subtype of the top type $\top$ with proof $\top$.

Rule SubCongr is a congruence rule for the upper bounded polymorphic type. It is similar to rule ContCongr of System F_η, but it is more technical and complicated since it has to handle the bound. If, assuming that type variable α is a subtype of type τ', we can show that it is also

$$\frac{\textsc{SubRefl}}{\Diamond \Rightarrow \Gamma \vdash \tau \triangleright \tau} \qquad \frac{G_1 \Rightarrow \Gamma \vdash \tau_1 \triangleright \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash \tau_2 \triangleright \tau_3}{G_2 \circ G_1 \Rightarrow \Gamma \vdash \tau_1 \triangleright \tau_3}\,\textsc{SubTrans} \qquad \frac{(c : \tau \triangleright \sigma) \in \Gamma}{c \Rightarrow \Gamma \vdash \tau \triangleright \sigma}\,\textsc{SubVar}$$

$$\textsc{SubTop}\quad \top \Rightarrow \Gamma \vdash \tau \triangleright \top \qquad \frac{\Gamma \vdash \tau \ \mathsf{type} \qquad G_1 \Rightarrow \Gamma \vdash \tau \triangleright \tau' \qquad G_2 \Rightarrow \Gamma \vdash \sigma' \triangleright \sigma}{G_1 \xrightarrow{\tau} G_2 \Rightarrow \Gamma \vdash \tau' \to \sigma' \triangleright \tau \to \sigma}\,\textsc{SubArr}$$

$$\frac{G_1 \Rightarrow \Gamma \vdash \tau' \triangleright \tau \qquad G_2 \Rightarrow \Gamma \vdash \sigma' \triangleright \sigma}{G_1 \times G_2 \Rightarrow \Gamma \vdash \tau' \times \sigma' \triangleright \tau \times \sigma}\,\textsc{SubProd}$$

$$\frac{\Gamma, \alpha \vdash \tau' \ \mathsf{type} \qquad G_1 \Rightarrow \Gamma, \alpha, (c : \alpha \triangleright \tau') \vdash \alpha \triangleright \tau \qquad G_2 \Rightarrow \Gamma, \alpha, (c : \alpha \triangleright \tau') \vdash \sigma \triangleright \sigma'}{\forall(\alpha \triangleright c : \tau')[G_1, G_2] \Rightarrow \Gamma \vdash \forall(\alpha \triangleright \tau)\, \sigma \triangleright \forall(\alpha \triangleright \tau')\, \sigma'}\,\textsc{SubCongr}$$

Figure 3.36: System $\mathsf{F}_{<:}$ subtyping judgment relation

a subtype of τ and we can also show that type σ is a subtype of σ' then we have that $\forall(\alpha \triangleright \tau)\, \sigma$ is a subtype of $\forall(\alpha \triangleright \tau')\, \sigma'$. The first premise about the well-formedness of τ' is necessary to prove the extraction lemma. We can wonder why we need this particular subtyping premise for the bound. The body subtyping seems natural. Rule SubCongr actually differs depending on the variant of System $\mathsf{F}_{<:}$ we consider. We present here two other variants:

$$\frac{\Gamma, \alpha, (\alpha \triangleright \tau) \vdash \sigma \triangleright \sigma'}{\Gamma \vdash \forall(\alpha \triangleright \tau)\, \sigma \triangleright \forall(\alpha \triangleright \tau)\, \sigma'}\,\textsc{Kernel-Fsub} \qquad \frac{\Gamma \vdash \tau' \triangleright \tau \qquad \Gamma, \alpha, (\alpha \triangleright \tau') \vdash \sigma \triangleright \sigma'}{\Gamma \vdash \forall(\alpha \triangleright \tau)\, \sigma \triangleright \forall(\alpha \triangleright \tau')\, \sigma'}\,\textsc{Full-Fsub}$$

In Kernel $\mathsf{F}_{<:}$, the bounds τ and τ' must be equal, whereas Full $\mathsf{F}_{<:}$ only requires the bound τ' to be a subtype of the bound τ. Moreover, α cannot appear free in the bounds τ or τ' in Kernel or Full $\mathsf{F}_{<:}$, while the presented variant, called F-bounded, allows this form of recursion. The most general assumption, $\Gamma, \alpha, (\alpha \triangleright \tau') \vdash \alpha \triangleright \tau$ is that of F-bounded. Perhaps surprisingly, this is a slightly more general rule [4] than the more intuitive one $\Gamma, \alpha, (\alpha \triangleright \tau') \vdash \tau' \triangleright \tau$. In summary, we have Kernel $\mathsf{F}_{<:} \subset$ Full $\mathsf{F}_{<:} \subset$ F-bounded where all inclusions are strict.

Finally the well-formedness relations are given on Figure 3.37. The first three rules for type well-formedness are the same as those for System F. The next two rules are new. Rule TypeFor is an adaptation of the System F rule of the same name. An upper bounded polymorphic type is well-formed if its body type is well-formed under the environment extended with its binding. Rule TypeTop tells that the top type is well-formed under any well-formed environment. The first two rules of the environment well-formedness are the same as those of the STLC. The last rule, namely EnvType, is new. The extended environment $\Gamma, \alpha, (c : \alpha \triangleright \tau)$ is well-formed if τ is well-formed under environment Γ extended with α. Besides, the subtyping variable c and the type variable α must not be already bound in Γ.

3.6.2 Properties

The properties of System $\mathsf{F}_{<:}$ are similar to those of System F with some modifications about the bounds. The type substitution lemma of System F is replaced by an upper bound substi-

$$\textsc{TypeVar}\ \frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}} \qquad \textsc{TypeArr}\ \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}} \qquad \textsc{TypeProd}\ \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}$$

$$\textsc{TypeFor}\ \frac{\Gamma, \alpha \vdash \tau\ \mathsf{type} \qquad \Gamma, \alpha, (c : \alpha \triangleright \tau) \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \forall(\alpha \triangleright \tau)\, \sigma\ \mathsf{type}} \qquad \textsc{TypeTop}\ \frac{\Gamma\ \mathsf{env}}{\Gamma \vdash \top\ \mathsf{type}}$$

$$\textsc{EnvEmpty}\ \varnothing\ \mathsf{env} \qquad \textsc{EnvTerm}\ \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}} \qquad \textsc{EnvType}\ \frac{\alpha, c \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma, \alpha, (c : \alpha \triangleright \tau)\ \mathsf{env}}$$

Figure 3.37: System $\mathsf{F}_{<:}$ well-formedness relations

α, β		Type variables
τ	$::= \alpha \mid \tau \to \tau \mid \tau \times \tau$	Types
σ	$::= \tau \mid \forall \overline{\alpha}.C \Rightarrow \sigma$	Type schemes
Γ	$::= \varnothing \mid \Gamma, (x : \sigma)$	Environments

Figure 3.38: Constraint ML syntax

tution lemma.

Lemma 41 (Upper bound substitution)**.** *If $\Gamma \vdash \sigma$ type and $G \Rightarrow \Gamma \vdash \sigma \triangleright \tau[\alpha/\sigma]$ hold, then the following assertions hold.*

- *If $\Gamma, \alpha, (c : \alpha \triangleright \tau), \Gamma' \vdash \rho$ type holds, then $\Gamma, \Gamma'[\alpha/\sigma] \vdash \rho[\alpha/\sigma]$ type holds.*
- *If $M \Rightarrow \Gamma, \alpha, (c : \alpha \triangleright \tau), \Gamma' \vdash a : \rho$ holds, then $M[\alpha/\sigma][c/G] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash a : \rho[\alpha/\sigma]$ holds.*
- *If $G' \Rightarrow \Gamma, \alpha, (c : \alpha \triangleright \tau), \Gamma' \vdash \rho \triangleright \rho'$ holds, then $G'[\alpha/\sigma][c/G] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash \rho[\alpha/\sigma] \triangleright \rho'[\alpha/\sigma]$ holds.*

3.7 Constraint ML

Constraint ML is an inference type system with constraints in the tradition of the Hindley/Milner type system, where syntactical types are split between simple types and type schemes. In Constraint ML, type schemes are extended to constrained type schemes. The presentation we give is inspired from [24]. The reason to choose this presentation rather than the more general setting of Chapter 10 of [27] is to keep the presentation lighter. We only define the implicit version of the type system since it does not have an explicit version.

The syntax of Constraint ML is defined on Figure 3.38. It extends the STLC with type variables α or β. Types contain type variables α, arrow types $\tau \to \tau$, and product types $\tau \times \tau$. Type schemes contain types τ and constrained type schemes $\forall \overline{\alpha}.C \Rightarrow \sigma$. The constrained type scheme $\forall \overline{\alpha}.C \Rightarrow \sigma$ quantifies over the type variables $\overline{\alpha}$ that satisfy the constraint C, which we handle in an abstract way. The type variables $\overline{\alpha}$ are bound in the constraint C and the body

$$\dfrac{(x:\sigma)\in\Gamma}{C;\Gamma\vdash x:\sigma}\;\textsc{TermVar}\qquad \dfrac{C;\Gamma\vdash a:\tau\qquad C\vdash\tau\rhd\tau'}{C;\Gamma\vdash a:\tau'}\;\textsc{TermSub}\qquad \dfrac{C;\Gamma,(x:\tau)\vdash a:\tau'}{C;\Gamma\vdash\lambda x\,a:\tau\to\tau'}\;\textsc{TermAbs}$$

$$\dfrac{C;\Gamma\vdash a_1:\tau_1\to\tau_2\qquad C;\Gamma\vdash a_2:\tau_1}{C;\Gamma\vdash a_1\,a_2:\tau_2}\;\textsc{TermApp}\qquad \dfrac{C;\Gamma\vdash a_1:\tau_1\qquad C;\Gamma\vdash a_2:\tau_2}{C;\Gamma\vdash\langle a_1,a_2\rangle:\tau_1\times\tau_2}\;\textsc{TermPair}$$

$$\dfrac{C;\Gamma\vdash a:\tau_1\times\tau_2}{C;\Gamma\vdash\mathsf{fst}\,a:\tau_1}\;\textsc{TermFst}\qquad \dfrac{C;\Gamma\vdash a:\tau_1\times\tau_2}{C;\Gamma\vdash\mathsf{snd}\,a:\tau_2}\;\textsc{TermSnd}\qquad \dfrac{C;\Gamma\vdash a:\sigma\qquad C;\Gamma,(x:\sigma)\vdash a':\tau'}{C;\Gamma\vdash(\lambda x\,a')\,a:\tau'}\;\textsc{TermLet}$$

$$\dfrac{C\wedge D;\Gamma\vdash a:\tau\qquad \overline{\alpha}\notin\mathsf{fv}(C,\Gamma)}{C\wedge\exists\overline{\alpha}.D;\Gamma\vdash a:\forall\overline{\alpha}.D\Rightarrow\tau}\;\textsc{TermIntro}\qquad \dfrac{C;\Gamma\vdash a:\forall\overline{\alpha}.D\Rightarrow\tau'\qquad C\vdash D\overline{[\alpha\leftarrow\tau]}}{C;\Gamma\vdash a:\tau'\overline{[\alpha\leftarrow\tau]}}\;\textsc{TermElim}$$

Figure 3.39: Constraint ML term judgment relation

of the type scheme σ. In terms of expressivity, this is the most interesting part of Constraint ML: constraint abstraction. Of course the inference part is interesting, but since our main concern are kernel type systems, we only focus on expressivity. Finally, environments are lists of term bindings of the form $(x : \sigma)$ binding a term variable x to its type scheme σ.

The term judgment is written $C; \Gamma \vdash a : \sigma$ meaning that the term a has type scheme σ under environment Γ given the constraint C holds. The rules are given on Figure 3.39. Rule TermVar looks up in the environment to find the type scheme of a term variable. Rule TermSub allows to change the type of a term from τ to τ' under environment Γ with respect to the constraint C, provided $\tau \rhd \tau'$ is a valid constraint with respect to C. We deliberately leave the notion of constraint abstract, since we want to focus on the mechanisms.

Rule TermAbs and TermApp are the usual typing rules for the arrow type. If a term a has type τ' under an environment extended with $(x : \tau)$ then the term $\lambda x\, a$ has type $\tau \to \tau'$. And if the term a_1 has type $\tau_1 \to \tau_2$ and the term a_2 has type τ_1, then the application $a_1\, a_2$ has type τ_2. Rules TermPair, TermFst, and TermSnd are similar to their STLC alternatives.

Rule TermLet explains how to type the let-binding: $(\lambda x\, a')\, a$. This rule is not simply derivable from rules TermAbs and TermApp since the term variable x gets associated to a type scheme and not simply a type. This comes from the fact that Constraint ML does inference. Since the abstraction is directly applied, its argument type can be generalized.

Rules TermIntro and TermElim are the two interesting rules. To generalize a type τ over the type variables $\overline{\alpha}$, the usual condition that the type variables $\overline{\alpha}$ should not be free in the environment Γ has to be satisfied. But this is not enough. The generalized type has to gather enough constraints, denoted as D, in order for the remaining constraint C not to mention the type variables $\overline{\alpha}$. Finally, the conclusion constraint to be satisfied is not simply C, but $C \wedge \exists\overline{\alpha}.D$. This additional condition is present to make sure that the erasable abstraction is inhabited: there is an instantiation for the type variables $\overline{\alpha}$ such that the constraint D holds for this instantiation. Finally, a quantified type $\forall\overline{\alpha}.D \Rightarrow \tau'$ can be instantiated with types $\overline{\tau}$, as long as the constraint $D\overline{[\alpha \leftarrow \tau]}$ holds under the constraint C.

3.8 Existing Coercions

We define coercions as erasable and composable typing transformations. In particular, coercions do not modify the computational content of the term they are applied to. As a consequence, coercions are only visible in the explicit version of type systems. This definition is closely related to subtyping, since subtyping defines a relation on types which is composable and do not affect computational content. Actually, subtyping is a particular case of coercions: those that do not alter the typing environment.

Since coercions are composable typing transformations, we may wonder if it would be possible to have all typing transformations of a type system to be expressed as coercions and thus be composable. This is our approach in Part II. The introduction and elimination rules of erasable types are all expressed as coercions. This naturally gives fully access to the concerned type feature: erasable and composable introduction and elimination rule. The typical counter-example is System $\mathsf{F}_{<:}$ which features upper bounded polymorphism only partially: it is an erasable type feature, but there is no composable introduction and elimination rule (see Section 4.5.4).

Some existing type systems already use coercions as a tool, even if their whole type system is not expressed as coercions. System F_η, which we described in Section 3.4 uses containments as coercions. System $\mathsf{F}_{<:}$, which we defined in Section 3.6 uses subtyping as coercions. MLF, defined in Section 3.5, uses instantiations as coercions. However, all three type systems are not fully expressed as coercions. In particular, erasable abstractions (introduction rules of erasable types) are not included in the coercion judgment but only accessible to the term judgment. As a consequence, η-expansion is not sufficient for deep generalization, and distributivity rules are necessary, like in System F_η. Erasable abstractions are: type abstraction for System F_η, upper bounded abstraction for System $\mathsf{F}_{<:}$, and lower bounded abstraction for MLF. The case for $\mathsf{Constraint\ ML}$ is worse, because erasable quantifiers are not even part of types, but only accessible in type schemes. So there is no hope to use η-expansion to have deep generalization, because such constraint would have no syntax.

The type systems defined in this chapter have distinct sets of features, but it looks like they rely on more general ideas. As such, we may wonder if we can unify these type systems in a single framework. This factorization would ease syntactical comparison of these type systems and give results about the compatibility and orthogonality of their features (Section 4.5.5). It would also help to understand which features of one type system are missing to another one. For instance, MLF and System $\mathsf{F}_{<:}$ looks similar and different at the same time and we may wonder whether one can be extended to contain the other one. Both type systems are similar because they feature bounded polymorphism: upper bounded polymorphism for System $\mathsf{F}_{<:}$ and lower bounded polymorphism for MLF. However, MLF features deep instantiation but misses subtyping, while System $\mathsf{F}_{<:}$ features subtyping but misses deep instantiation. And finally, the soundness and strong normalization of this unified type system would imply the soundness and strong normalization of all the subsumed type systems. For instance, the strong normalization of MLF has been proved with this approach.

Part II

Type Systems as Coercions

Chapter 4

An explicit calculus of coercions: System F_ι^p

System $\mathsf{F}_{<:}$ and MLF are extensions of System F in somewhat dual ways. They both include bounded polymorphism, which is a form of coercion abstraction: $\mathsf{F}_{<:}$ has upper bounded polymorphism while MLF has lower bounded polymorphism. However, they have incompatible definitions. The same argument holds for all pairs of existing type systems described in Chapter 3. The goal of this chapter is to define a framework where the features of the type systems described in Chapter 3 are expressed in a uniform way, are orthogonal, and can be freely combined. We shall see in Section 4.6 why this may not be completely possible in the explicit version.

A type system can be seen as the following: a syntax for invariants Φ (what we used to call approximations), typing rules for the language constructs $a : \Phi$, and coercions for invariants $\Phi_1 \triangleright \Phi_2$, which tells us that $a : \Phi_1$ implies $a : \Phi_2$ for all a, or that Φ_1 is a better approximation than Φ_2. Semantically, the invariants Φ are interpreted as sets of terms, typing rules as proofs of memberships, and coercions as proofs of inclusion. Actually, coercions could be extended to be any kind of proofs and inclusions would just be a particular kind of proofs, see Chapter 5.

Similarly to Chapter 3, we simultaneously define the implicit and explicit version of the type system. According to this view of type systems, the explicit version gives witnesses to the two judgments: the term judgment $a : \Phi$ becomes $M \Rightarrow a : \Phi$ and the coercion judgment $\Phi_1 \triangleright \Phi_2$ becomes $G \Rightarrow \Phi_1 \triangleright \Phi_2$. The term M and the coercion proof G witness the implicit judgments.

As before, we consider strong reduction for two reasons. First, because the soundness of strong reduction implies the soundness of all other reduction strategies. Only the correspondence between syntactical values and semantic values, which is simple to prove, has to be rechecked for each reduction strategy. Then, because strong reduction gives a better insight to the language. Understanding strong reduction is a guaranty that no corner cases have been forgotten.

The type system containing all the features of this chapter, called System F_ι^p, is given in Section 4.3. The first section describes the bare framework without any additional features and thus corresponds to the STLC. The next section describes extensions as pairwise orthogonal features that can be combined at will. We then describe System F_ι^p, give its properties, and discuss its expressiveness. In this framework, invariants Φ are typings of the form $\Gamma \vdash \tau$ where Γ is a list of bindings describing the invariants of the environment in which the term can be

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
c & & & \text{Coercion variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M \rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid G\langle M \rangle & \text{Explicit terms} \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau & \text{Types} \\
G & ::= & c \mid \Diamond \mid G \circ G \mid *G & \text{Coercions} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha \mid \Gamma, (c : \tau \triangleright \tau) & \text{Environments}
\end{array}
$$

Figure 4.1: Base system syntax

$$
\begin{array}{rcll}
\Sigma & ::= & \varnothing \mid \Sigma, \alpha \mid \Sigma, (c : \tau \triangleright \tau) & \text{Erasable environments} \\
E & ::= & \lambda(x:\tau)\, [] \mid []\, M \mid M\, [] \mid \langle [], M \rangle \mid \langle M, [] \rangle \mid \mathsf{fst}\, [] \mid \mathsf{snd}\, [] \mid G\langle [] \rangle & \text{Evaluation contexts} \\
p & ::= & x \mid p\, v \mid \mathsf{fst}\, p \mid \mathsf{snd}\, p \mid c\langle v \rangle & \text{Prevalues} \\
v & ::= & p \mid \lambda(x:\tau)\, v \mid \langle v, v \rangle & \text{Values}
\end{array}
$$

Figure 4.2: Base system notations

considered and τ is a type describing the invariant of the term.

4.1 Base system

The base system for a non-dependent type system[1] on the λ-calculus is the STLC. So, the framework we define is equivalent to the STLC, although its presentation differs a little to better fit in our framework.

The syntax for the base type system is given on Figure 4.1. It corresponds to the STLC with two small differences. First, we added variables to all syntactic classes in order to be able to abstract over them later. Notice that we also added the corresponding binders. Type variables are written α or β, while coercion variables are written c. Type bindings are written α and coercion bindings are written $(c : \tau \triangleright \tau)$. Notice that coercion variables only exist in the explicit version: they are always implicit in the implicit version. The second difference is the new syntactical class of *coercions*. Currently, it only contains coercion variables c, weakening $*G$, and operations to close coercions by reflexivity $\Diamond$ and transitivity $G \circ G$.

Terms, written M or N, contain variables x, abstractions $\lambda(x : \tau)\, M$ where the term variable x is annotated with its type τ, applications $M\, M$, pairs $\langle M, M\rangle$, projections $\mathsf{fst}\, M$ and $\mathsf{snd}\, M$, and the new coercion construct $G\langle M\rangle$. This construct is used to change the invariant (called a typing in our framework) of the term M according to the coercion G, interpreted as a proof of inclusion between invariants.

Types, written τ, σ, or ρ, contain variables α, arrow types $\tau \to \tau$, and product types $\tau \times \tau$. Coercions, written G, contain variables c, reflexivity $\Diamond$, transitivity $G \circ G$, and weakening $*G$. Environments, written Γ, are lists of bindings. Bindings contain term bindings $(x : \tau)$, type bindings α, and coercion bindings $(c : \tau \triangleright \tau)$. The coercion variable in the coercion binding is only necessary in the explicit version of the type system. In the implicit version, coercion bindings can be written $(\tau \triangleright \tau)$, omitting the coercion variable.

We define erasable environments, evaluation contexts, and values in Figure 4.2. We write Σ the subset of environments that are erasable: they contain type and coercion bindings (all bindings but the computational ones: term bindings). Erasable environments are used for typing coercions because coercions may change environments as part of typings.

Evaluation contexts E contain all possible one hole contexts of depth one. We overload the notation from evaluation contexts of the λ-calculus and make the distinction clear when necessary.

Prevalues and values are defined as usual for strong reduction. Prevalues are either variables or destructors applied to prevalues. For instance the application is a destructor expecting a constructor on its left-hand side, so its corresponding prevalue is $p\, v$. The case for the coercion construct is special since it can be either a constructor or a destructor depending on its left-hand side. And depending on this left-hand side, it will expect or not a constructor on its right-hand side. Reflexivity, transitivity, and coercion variables do not expect a constructor for their coerced term. However, since reflexivity and transitivity always reduce, they are not prevalues. And since coercion variables never reduce, they are prevalues. Finally, values are either prevalues or constructors applied to values. Notice that prevalues are neutral values, *i.e.* values that do not start with a constructor. We overload the notation from prevalues and values of the λ-calculus, but we make the distinction clear when necessary.

Reduction rules are given in Figure 4.3. We label the reduction with two annotations: β for computational steps and ι for typing steps. Computational steps are those of the λ-calculus, while typing steps are new and only have to do with typings. We write $\beta\iota$ for either β or ι.

[1] see Section 6.1.7 for discussion about dependent type systems

$$\textsc{RedCtx}\ \frac{M \rightsquigarrow_{\beta\iota} N}{E[M] \rightsquigarrow_{\beta\iota} E[N]} \qquad \textsc{RedApp}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_{\beta} M[x/N] \qquad \textsc{RedFst}\ \mathsf{fst}\ \langle M,N\rangle \rightsquigarrow_{\beta} M \qquad \textsc{RedSnd}\ \mathsf{snd}\ \langle M,N\rangle \rightsquigarrow_{\beta} N$$

$$\textsc{RedRefl}\ \Diamond\langle M\rangle \rightsquigarrow_{\iota} M \qquad \textsc{RedTrans}\ (G_2 \circ G_1)\langle M\rangle \rightsquigarrow_{\iota} G_2\langle G_1\langle M\rangle\rangle \qquad \textsc{RedWeak}\ (*G)\langle M\rangle \rightsquigarrow_{\iota} G\langle M\rangle$$

Figure 4.3: Base system reduction rules

$$\textsc{TermVar}\ \frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow x : \Gamma \vdash \tau} \qquad \textsc{TermLam}\ \frac{M \Rightarrow a : \Gamma,(x:\tau) \vdash \sigma}{\lambda(x:\tau)\,M \Rightarrow \lambda x\,a : \Gamma \vdash \tau \to \sigma}$$

$$\textsc{TermApp}\ \frac{M \Rightarrow a : \Gamma \vdash \tau \to \sigma \qquad N \Rightarrow b : \Gamma \vdash \tau}{M\,N \Rightarrow a\,b : \Gamma \vdash \sigma} \qquad \textsc{TermPair}\ \frac{M \Rightarrow a : \Gamma \vdash \tau \qquad N \Rightarrow b : \Gamma \vdash \sigma}{\langle M,N\rangle \Rightarrow \langle a,b\rangle : \Gamma \vdash \tau \times \sigma}$$

$$\textsc{TermFst}\ \frac{M \Rightarrow a : \Gamma \vdash \tau \times \sigma}{\mathsf{fst}\ M \Rightarrow \mathsf{fst}\ a : \Gamma \vdash \tau} \qquad \textsc{TermSnd}\ \frac{M \Rightarrow a : \Gamma \vdash \tau \times \sigma}{\mathsf{snd}\ M \Rightarrow \mathsf{snd}\ a : \Gamma \vdash \sigma}$$

$$\textsc{TermCoer}\ \frac{M \Rightarrow a : \Gamma,\Sigma \vdash \tau \qquad G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma}{G\langle M\rangle \Rightarrow a : \Gamma \vdash \sigma}$$

Figure 4.4: Base system term judgment relation

The judgment for our reduction is thus written: $M \rightsquigarrow_{\beta\iota} M$.

Rule RedCtx is the context rule for strong reduction. If a term M can do a β (resp. ι) step to N, then the depth-one evaluation context E filled with M, namely $E[M]$, can also do a β (resp. ι) step to $E[N]$. Rules RedApp, RedFst, and RedSnd are similar to those of the λ-calculus, modulo the β annotation and the type annotation for the explicit abstraction.

Coercions never modify the computational content, and their reduction rules are thus ι-steps. Rule RedRefl shows that the reflexivity coercion does not modify a term and hence its typing. Rule RedTrans shows that coercing the term M with $G_2 \circ G_1$ is like coercing M with G_1 and then coercing the result with G_2, applying both coercions in sequence. Finally, rule RedWeak simply removes the weakening coercion making the subcoercion accessible.

The term judgment relation, given in Figure 4.4, is of the form $M \Rightarrow a : \Gamma \vdash \tau$ to match the requested form $M \Rightarrow a : \Phi$ since invariants Φ are typings $\Gamma \vdash \tau$. The left part $M \Rightarrow$ is used for the explicit version only. The first six rules are those of the STLC modulo the different notation (see Section 3.1). The last rule, named TermCoer, is new. It tells that a term of invariant $\Gamma, \Sigma \vdash \tau$ can be seen with invariant $\Gamma \vdash \sigma$, given a coercion from $\Gamma, \Sigma \vdash \tau$ to $\Gamma \vdash \sigma$; this is written $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma$.

The coercion judgment relation is written $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma$ and defined in Figure 4.5. When the erasable environment Σ is empty we may write $G \Rightarrow \Gamma \vdash \tau \rhd \sigma$ instead of $G \Rightarrow \Gamma \vdash$

$$\textsc{CoerRefl}\quad \Diamond \Rightarrow \Gamma \vdash \tau \triangleright \tau \qquad\qquad \textsc{CoerTrans}\quad \frac{G_1 \Rightarrow \Gamma, \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \triangleright \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_3}{G_2 \circ G_1 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \triangleright \tau_3}$$

$$\textsc{CoerVar}\quad \frac{(c : \tau \triangleright \sigma) \in \Gamma}{c \Rightarrow \Gamma \vdash \tau \triangleright \sigma} \qquad\qquad \textsc{CoerWeak}\quad \frac{\Gamma, \Sigma\ \mathsf{env} \qquad G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma}{*G \Rightarrow \Gamma \vdash \tau \triangleright \sigma}$$

Figure 4.5: Base system coercion judgment relation

$(\varnothing \vdash \tau) \triangleright \sigma$. The intuitive interpretation of $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ is that for all term a (and explicit term M) of typing $\Gamma, \Sigma \vdash \tau$, the term a (and explicit term $G\langle M\rangle$) has typing $\Gamma \vdash \sigma$. This interpretation comes from the fact that the typing rules of our considered extensions are of the form:

$$\frac{a : \Gamma, \Sigma \vdash \tau}{a : \Gamma \vdash \sigma}$$

For example, the typing rule for type abstraction is:

$$\frac{a : \Gamma, \alpha \vdash \tau}{a : \Gamma \vdash \forall \alpha\, \tau}$$

One may wonder why the coercion judgment is not written $\Gamma_1 \vdash \tau_1 \triangleright \Gamma_2 \vdash \tau_2$ as one could expect from our notion of coercion invariants $\Phi_1 \triangleright \Phi_2$. The reason is that we don't know which part of the environment is modified by the coercion, which is however necessary for weakening and substitution lemmas, for instance. For example, consider the coercion $(\Gamma, \alpha \vdash \tau) \triangleright (\Gamma, \alpha \vdash \forall \alpha\, \tau)$ which is sound, as we may generalize from $\Gamma, \alpha \vdash \tau$ to $\Gamma \vdash \forall \alpha\, \tau$ and weaken to $\Gamma, \alpha \vdash \forall \alpha\, \tau$. What if we weakened the coercion from Γ, α to $\Gamma, \alpha, (x : \alpha)$. It seems correct, but $(\Gamma, \alpha, (x : \alpha) \vdash \tau) \triangleright (\Gamma, \alpha, (x : \alpha) \vdash \forall \alpha\, \tau)$ is not. What we actually wanted to write was $\Gamma \vdash (\alpha \vdash \tau) \triangleright (\alpha \vdash \forall \alpha\, \tau)$ specifying that the environment Γ is not modified by the coercion. However, this notation $\Gamma \vdash (\Sigma_1 \vdash \tau_1) \triangleright (\Sigma_2 \vdash \tau_2)$ is subsumed by our current notation $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ using inverse modifications (see Section 6.1.11).

Rule CoerRefl defines the reflexivity coercion $\Diamond$ as the proof that typing $\Gamma \vdash \tau$ is included in itself without modifying the environment, which corresponds to the reflexivity closure of the inclusion relation. We remind that typings are to be interpreted as set of terms and coercions as inclusion proofs between these sets of terms.

Similarly, rule CoerTrans defines the transitivity coercion $G_2 \circ G_1$ as the proof for inclusion transitivity. If G_1 proves that $\Gamma, \Sigma_2, \Sigma_1 \vdash \tau_1$ is included in $\Gamma, \Sigma_2 \vdash \tau_2$ by extending Σ_1, and if G_2 proves that $\Gamma, \Sigma_2 \vdash \tau_2$ is included in $\Gamma \vdash \tau_3$ by extending Σ_2, then $G_2 \circ G_1$ proves that $\Gamma, \Sigma_2, \Sigma_1 \vdash \tau_1$ is included in $\Gamma \vdash \tau_3$ by transitivity and extending Σ_2, Σ_1.

Finally, rule CoerVar defines the coercion variable c as the proof that τ is included in σ, given that the environment Γ contains an hypothesis, named c, that τ is included in σ. And rule CoerWeak defines the weakening coercion $*G$ as the proof that τ is smaller than σ without any environment extension, whenever τ is smaller than σ with erasable environment Σ and witness G. For the rule to be well-formed, the extended environment Γ, Σ has to be well-formed.

$$\frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \textsc{TypeVar} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}}\ \textsc{TypeArr} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}\ \textsc{TypeProd}$$

$$\frac{}{\varnothing\ \mathsf{env}}\ \textsc{EnvEmpty} \qquad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}}\ \textsc{EnvTerm}$$

Figure 4.6: Base system well-formedness relations

Type and environment well-formedness relations are given on Figure 4.6. We write $\Gamma \vdash \tau\ \mathsf{type}$ when type τ is well-formed under environment Γ. And environment well-formedness is written $\Gamma\ \mathsf{env}$. Type variable α is well-formed under environment Γ, by rule TypeVar, if it is bound in Γ and Γ is well-formed. Rules TypeArr and TypeProd tell that $\tau \to \sigma$ and $\tau \times \sigma$ are well-formed under Γ if τ and σ are well-formed under Γ.

The empty environment is well-formed according to rule EnvEmpty. Rule EnvTerm tells that the extended environment $\Gamma, (x : \tau)$ is well-formed if x is not already bound and τ is well-formed under Γ. Notice that there is an environment well-formedness rule for each kind of abstraction and not for each kind of variable.

A global note about the syntax of this base system is that features never *change* any kind of rules (typing rules, reduction rules, or well-formedness rules). Features may only *add* types (with their well-formedness rules), coercions (with their typing, ι-reduction, and environment rules), prevalues, and values. Finally they do not add more explicit terms, environments, erasable environments, or evaluation contexts. This non-modification property explains that the following features can be combined in an orthogonal way: all features rely on the base system only and no features alter the base system. The base system is sufficient for all extensions but contains only the expressivity of the STLC.

4.2 Features

Features usually define new invariants in the form of erasable types, like polymorphism which adds the polymorphic type $\forall\alpha\,\tau$. We make a distinction between *computational types* and *erasable types*. On the one hand, computational types (or simple types) are defined in the base system which corresponds to the STLC. We call them computational types, because their head constructor describe a computational behavior. The arrow type constructor classifies term abstractions, while the product type constructor classifies pairs of terms. Computational types are introduced and eliminated through term typing rules and each rule is associated to a term construct. These types and rules are already defined in the base system and no features will need a new one. Notice that we only consider type system features: a type system feature extends the type system but not the programming language. Adding integers to the programming language is not a type system extension, but a programming language extension. On the other hand, erasable types are related to typing and are independent from computation. They are introduced and eliminated through coercions and are thus erasable.

All features define an erasable type and its associated coercions. The only feature that does not add an erasable type is the η-expansion. This feature defines coercion rules for the congruence of computational types. As such, η-expansion could actually be considered in the base system. But since it adds subtyping, we decided to see it as an extension feature.

Congruence rules are actually only necessary for computational types. Erasable types have derivable congruence rules from their introduction and elimination rules. The typical case is using transitivity to eliminate the existing erasable constructor, then using the subcoercion, and finally introducing the erasable constructor back. In particular, congruence rules for polymorphic types and recursive types are derivable. For example, see Section 5.4.4 to see how to derive the congruence rule for recursive types.

4.2.1 Polymorphism

Polymorphism is the possibility to abstract over types in typing derivations. We make this new invariant apparent in the syntax with an erasable type. In the implicit version the only syntactical change is that we add the polymorphic type $\forall\alpha\,\tau$ to the syntax of types. For the explicit version, besides the polymorphic type, we need to add two coercions: one for type abstraction $\Lambda\alpha$ and one for type application $\bullet\,\tau$. We also need to extend prevalues and values accordingly.

$$
\begin{array}{rcll}
\tau, \sigma, \rho & ::= & \ldots \mid \forall\alpha\,\tau & \text{Types} \\
G & ::= & \ldots \mid \Lambda\alpha \mid \bullet\,\tau & \text{Coercions} \\
p & ::= & \ldots \mid \bullet\,\tau\langle p\rangle & \text{Prevalues} \\
v & ::= & \ldots \mid \Lambda\alpha\langle v\rangle & \text{Values}
\end{array}
$$

Reduction rules have to be extended with one additional rule called RedPoly. If a type application $\bullet\,\tau$ follows a type abstraction $\Lambda\alpha$, then we can remove these nodes and substitute α for τ in the term.

$$
\textsc{RedPoly} \quad \bullet\,\tau\langle\Lambda\alpha\langle M\rangle\rangle \rightsquigarrow_\iota M[\alpha/\tau]
$$

We add two coercion rules to the existing ones: one for type abstraction and one for type application. Rule CoerTLam is the most interesting one since it illustrates what there is to say about the coercion judgment. It can be read with rule TermCoer under hand. If the term a has typing $\Gamma, \alpha \vdash \tau$ with witness M, then it also has typing $\Gamma \vdash \forall\alpha\,\tau$ with witness $\Lambda\alpha\langle M\rangle$. We see that both the type and the environment have been modified by the coercion. The environment has been extended with the type binding α, while the type gets closed by type abstraction. Similarly, rule CoerTApp tells that we can instantiate a polymorphic type $\forall\alpha\,\tau$ with a well-formed type σ. In this case, only the type is modified, the environment is not extended or modified in another way.

$$
\textsc{CoerTLam} \quad \Lambda\alpha \Rightarrow \Gamma \vdash (\alpha \vdash \tau) \triangleright \forall\alpha\,\tau
\qquad\qquad
\textsc{CoerTApp} \quad \frac{\Gamma \vdash \sigma \ \mathsf{type}}{\bullet\,\sigma \Rightarrow \Gamma \vdash \forall\alpha\,\tau \triangleright \tau[\alpha/\sigma]}
$$

We also add rules to describe when a polymorphic type is well-formed and when an environment extended with a type variable is well-formed. A polymorphic type is well-formed if its body is well-formed under the environment extended with its type binding. An environment extended with a type binding is well-formed if its prefix is well-formed and the type variable is not already bound.

$$\frac{\Gamma, \alpha \vdash \tau \ \mathsf{type}}{\Gamma \vdash \forall \alpha\, \tau \ \mathsf{type}}\ \textsc{TypeFor} \qquad \frac{\alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma \ \mathsf{env}}{\Gamma, \alpha \ \mathsf{env}}\ \textsc{EnvType}$$

4.2.2 Eta-expansion

We call η-expansion the extension of coercions with computational type congruence. The reason why we use this name is because η-expansion derivations give the congruence rules of computational types. This feature is actually more expressive than the usual congruence rules we can find in type systems with subtyping. Because in our setting, coercion are not only between types, but between typings. So η-expansion allows us to do subtyping between typings and not only types. We will explain in a few paragraph how η-expansion works.

This extension does not need any syntactic changes in the implicit version. However, we need to add two coercions in the explicit version. Actually, we need to add as many coercions as we have computational types. In our case, we have just two computational types: the arrow type and the product type. So we have two η-expansion coercions, one for each.

$$G \ ::= \ \dots \mid G \xrightarrow{\tau} G \mid G \times G \qquad \text{Coercions}$$

These coercions can be intuitively understood as $\lambda(x : \tau)\, G_2\langle [ ]\, (G_1\langle x\rangle)\rangle$ for the arrow η-expansion coercion $G_1 \xrightarrow{\tau} G_2$, and $\langle G_1\langle \mathsf{fst}\, [ ]\rangle, G_2\langle \mathsf{snd}\, [ ]\rangle\rangle$ for the product η-expansion coercion $G_1 \times G_2$. These are called η-expansion because when we take their image by the drop function erasing types and coercions, we get the η-expansions $\lambda x\, [ ]\, x$ for the arrow type and $\langle \mathsf{fst}\, [ ], \mathsf{snd}\, [ ]\rangle$ for the product type. This gives an idea of why these coercions are erasable and do not modify the computational content of their hole.

This intuition could be used as an encoding: whenever we want to want to retype a function, we can η-expand it. However, the resulting term would only be η-equivalent to the initial term, and this disagrees with our wish of coercions being truly erasable.

Before we look at the associated coercion rules, we need to extend the prevalues. Several choices can be done depending on the reduction rules we want and this is discussed in Section 4.3 when describing the reductions rules of F_ι^p. In our case, η-expansions are only destructors, so they are prevalues.

$$p \ ::= \ \dots \mid (G \xrightarrow{\tau} G)\langle p\rangle \mid (G \times G)\langle p\rangle \qquad \text{Prevalues}$$

Since the η-expansion coercions are destructors, they reduce when they are applied to their associated constructor in accordance with their intuitive definition. In other words, we take their η-expansion view as terms and fill the hole with their argument (starting with the correct constructor) and reduce the created redex at the hole. For instance, for rule RedEtaArr, the left-hand side $(G_1 \xrightarrow{\tau'} G_2)\langle \lambda(x : \tau)\, M\rangle$ can be seen as $\lambda(x : \tau')\, G_2\langle (\lambda(x : \tau)\, M)\, (G_1\langle x\rangle)\rangle$ which reduces to the right-hand side $\lambda(x : \tau')\, G_2\langle M[x/G_1\langle x\rangle]\rangle$.

RedEtaArr

$$(G_1 \xrightarrow{\tau'} G_2)\langle \lambda(x : \tau)\, M\rangle \rightsquigarrow_\iota \lambda(x : \tau')\, G_2\langle M[x/G_1\langle x\rangle]\rangle$$

RedEtaProd

$$(G_1 \times G_2)\langle\langle M_1, M_2\rangle\rangle \rightsquigarrow_\iota \langle G_1\langle M_1\rangle, G_2\langle M_2\rangle\rangle$$

When adding a new feature, the most important part of the definition is always the extension of the coercion relation. Here, the rules follow the intuition of the η-expansion view. Rule CoerEtaArr tells that an arrow type $\tau \to \sigma$ under environment Γ, Σ can be coerced into the arrow type $\tau' \to \sigma'$ under the environment Γ, given that τ' can be coerced to τ under the extended environment Γ, Σ and that σ can be coerced to σ' under Γ and binding the erasable environment Σ. Notice the contravariance for the left-hand type of the arrow constructor. For well-formedness reasons, the type τ' has to be well-formed under environment Γ in order for the type variables bound in Σ to not be used by τ'. One may wonder why G_2 binds Σ in G_1 and why G_1 does not bind anything. This can be understood by looking at the erasable context view $\lambda(x : \tau)\, G_2\langle[]\,(G_1\langle x\rangle)\rangle$. We see that G_1 is actually under the scope of G_2, and that the only thing under the scope of G_1 is the term variable x, which does not use anything from its environment but itself.

Rule CoerEtaProd is easier since the product type is covariant on both arguments. If the typing $\Gamma, \Sigma \vdash \tau$ can be coerced to the typing $\Gamma \vdash \tau'$ and similarly the typing $\Gamma, \Sigma \vdash \sigma$ can be coerced to $\Gamma \vdash \sigma'$, then the product type $\tau \times \sigma$ can be coerced to the product $\tau' \times \sigma'$ under the environment Γ and extending with the erasable environment Σ.

CoerEtaArr

$$\frac{\Gamma \vdash \tau' \ \mathsf{type} \qquad G_1 \Rightarrow \Gamma, \Sigma \vdash (\varnothing \vdash \tau') \triangleright \tau \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \sigma) \triangleright \sigma'}{G_1 \overset{\tau'}{\to} G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau \to \sigma) \triangleright \tau' \to \sigma'}$$

CoerEtaProd

$$\frac{G_1 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \tau' \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \sigma) \triangleright \sigma'}{G_1 \times G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau \times \sigma) \triangleright \tau' \times \sigma'}$$

Notice that when adding η-expansion and polymorphism to the base system, we get exactly System F_η. This will be shown in Section 4.5.2. However, this presentation differs from the original presentation in [22] by factorizing free generalization, type instantiation, arrow congruence, distributivity, and polymorphic congruence into type generalization, type instantiation, and arrow η-expansion. This presentation is more economical, which has been made possible by extending coercions from types to typings.

4.2.3 Bottom

This extension closes the hierarchy of types with a minimum, called bottom and written $\bot$. This extension is useful later when dealing with lower bounded polymorphism. There is also a similar extension to close the hierarchy of types with a maximum, called top and written $\top$ (see the next section). We extend the syntax of types with the bottom type $\bot$. We extend coercions with the absurd coercion $\bot\tau$ and prevalues with $(\bot\tau)\langle p\rangle$.

$$\begin{array}{rcll} \tau, \sigma, \rho & ::= & \ldots \mid \bot & \text{Types} \\ G & ::= & \ldots \mid \bot\tau & \text{Coercions} \\ p & ::= & \ldots \mid (\bot\tau)\langle p\rangle & \text{Prevalues} \end{array}$$

There are no reduction rules associated to the bottom type because there is no constructor of type bottom. But there is a coercion rule CoerBot telling that from the bottom type we can go to any type. So $\bot$ is actually a minimum of all well-formed types in any environment, according to the coercion relation we are defining.

$$\frac{\Gamma \vdash \tau \ \mathsf{type}}{\bot\tau \Rightarrow \Gamma \vdash \bot \triangleright \tau}\ \textsc{CoerBot}$$

Finally, we need to give the well-formedness rule for the bottom type, namely TypeBot. The bottom type is well-formed under any well-formed environment.

$$\frac{\Gamma \ \mathsf{env}}{\Gamma \vdash \bot \ \mathsf{type}}\ \textsc{TypeBot}$$

4.2.4 Top

This extension is similar to the preceding one and closes the hierarchy of types with a maximum, called top and written $\top$. This extension is useful later, and in particular when dealing with upper bounded polymorphism. We extend the syntax of types with the top type $\top$. We extend coercions with the forget coercion $\top$ and values with the blind value $\top\langle v\rangle$.

$$\begin{array}{rcll} \tau,\sigma,\rho & ::= & \ldots \mid \top & \text{Types} \\ G & ::= & \ldots \mid \top & \text{Coercions} \\ v & ::= & \ldots \mid \top\langle v\rangle & \text{Values} \end{array}$$

There are no reduction rules associated to the top type because there are no associated destructors. But there is a coercion rule CoerTop that tells that any type can be coerced to top. So $\top$ is actually a maximum of all well-formed types in any environment.

$$\textsc{CoerTop}\quad \top \Rightarrow \Gamma \vdash \tau \triangleright \top$$

Finally, we need to give the well-formedness rule for the top type, namely TypeTop. The top type is well-formed under any well-formed environment.

$$\frac{\Gamma \ \mathsf{env}}{\Gamma \vdash \top \ \mathsf{type}}\ \textsc{TypeTop}$$

4.2.5 Lower Bounded polymorphism

Lower bounded polymorphism, when paired with the bottom type on top of the base system, corresponds to MLF . We show in detail how in Section 4.5.3. A small difference lies in the absence of recursive bounds in MLF which are possible in this extension.

This extension resembles the extension of the STLC with polymorphism but instead of abstracting over a type, we abstract over a type greater than another one. In other words, we abstract over instances of a type, called the lower bound. We say that the bound is recursive when it may contain occurrences of the abstract type. We extend the syntax of types with the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau)\,\tau$. We add two coercions: one for abstraction $\Lambda(\alpha \triangleleft c : \tau)$ and one for application $\cdot[\tau \triangleleft G]$. Notice that in $\Lambda(\alpha \triangleleft c : \tau)$ we simultaneously abstract over c and α. We extend the prevalues and values accordingly.

$$
\begin{array}{rcll}
\tau, \sigma, \rho & ::= & \ldots \mid \forall(\alpha \triangleleft \tau)\, \tau & \text{Types} \\
G & ::= & \ldots \mid \Lambda(\alpha \triangleleft c : \tau) \mid \cdot[\tau \triangleleft G] & \text{Coercions} \\
p & ::= & \ldots \mid (\cdot[\tau \triangleleft G])\langle p \rangle & \text{Prevalues} \\
v & ::= & \ldots \mid \Lambda(\alpha \triangleleft c : \tau)\langle v \rangle & \text{Values}
\end{array}
$$

We extend the reduction rules with rule RedPolyL which is similar to RedPoly; it takes a lower bounded type abstraction followed by a lower bounded type application, and it reduces it into a type substitution followed by a coercion substitution.

$$
\textsc{RedPolyL} \qquad \cdot[\sigma \triangleleft G]\langle \Lambda(\alpha \triangleleft c : \tau)\langle M \rangle \rangle \rightsquigarrow_\iota M[\alpha/\sigma][c/G]
$$

The coercion typing rules need also to be extended. Rule CoerTLamL tells that if we can type a term with ρ under an environment extended with type variable α and coercion variable c of coercion type $\tau \triangleright \alpha$, then this term also has type $\forall(\alpha \triangleleft \tau)\, \rho$ under the non-extended environment. Notice that this abstraction extends the environment with two binders and, although we write $\alpha \triangleleft \tau$ in the syntax of types, we still write $\tau \triangleright \alpha$ in the coercion binding. Rule CoerTAppL permits to view a term of type $\forall(\alpha \triangleleft \tau)\, \rho$ with type $\rho[\alpha/\sigma]$ whenever σ is well-formed and an instance of $\tau[\alpha/\sigma]$. Here, we must substitute α by σ in τ since we allow recursive bounds.

$$
\textsc{CoerTLamL} \quad \Lambda(\alpha \triangleleft c : \tau) \Rightarrow \Gamma \vdash (\alpha, (c : \tau \triangleright \alpha) \vdash \rho) \triangleright \forall(\alpha \triangleleft \tau)\, \rho
\qquad
\textsc{CoerTAppL} \quad \frac{\Gamma \vdash \sigma \;\mathsf{type} \qquad G \Rightarrow \Gamma \vdash \tau[\alpha/\sigma] \triangleright \sigma}{\cdot[\sigma \triangleleft G] \Rightarrow \Gamma \vdash \forall(\alpha \triangleleft \tau)\, \rho \triangleright \rho[\alpha/\sigma]}
$$

Finally, we extend the well-formedness rules. By rule TypeForL, the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau)\, \rho$ is well-formed if its body ρ is well-formed under the environment extended with the type binding α and coercion binding $(c : \tau \triangleright \alpha)$. So α is bound in ρ and we know that it is an instance of τ. By rule EnvTypeL, the extended environment $\Gamma, \alpha, (c : \tau \triangleright \alpha)$ is well-formed if the bound variables are not already bound in environment Γ and τ is well-formed under Γ, α which allows τ to mention α and thus permits recursive bounds.

$$
\textsc{TypeForL} \quad \frac{\Gamma, \alpha, (c : \tau \triangleright \alpha) \vdash \rho \;\mathsf{type}}{\Gamma \vdash \forall(\alpha \triangleleft \tau)\, \rho \;\mathsf{type}}
\qquad
\textsc{EnvTypeL} \quad \frac{c, \alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \alpha \vdash \tau \;\mathsf{type}}{\Gamma, \alpha, (c : \tau \triangleright \alpha) \;\mathsf{env}}
$$

4.2.6 Upper Bounded polymorphism

Upper bounded polymorphism, when paired with top type and η-expansion on top of the base system, gives a type system more expressive than the most expressive version of System $\mathsf{F}_{<:}$. We show in detail how in Section 4.5.4.

This extension resembles lower bounded polymorphism but instead of abstracting over a type greater than another one, it abstracts over a type smaller than another one. We call this last type the upper bound of the abstract type. We say that the bound is recursive when it may contain occurrences of the abstract type. We extend the syntax of types with the upper bounded polymorphic type $\forall(\alpha \triangleright \tau)\, \tau$. We add two coercions: one for abstraction $\Lambda(\alpha \triangleright c : \tau)$ and one for application $\cdot[\tau \triangleright G]$. We also extend the prevalues and values accordingly and similarly to the previous extension. Notice that only the triangles are inverted from $\triangleleft$ to $\triangleright$ in a way to make what were previously lower bounds upper bounds.

$$
\begin{array}{rcll}
\tau, \sigma, \rho & ::= & \ldots \mid \forall(\alpha \triangleright \tau)\, \tau & \text{Types} \\
G & ::= & \ldots \mid \Lambda(\alpha \triangleright c : \tau) \mid \cdot[\tau \triangleright G] & \text{Coercions} \\
p & ::= & \ldots \mid (\cdot[\tau \triangleright G])\langle p \rangle & \text{Prevalues} \\
v & ::= & \ldots \mid \Lambda(\alpha \triangleright c : \tau)\langle v \rangle & \text{Values}
\end{array}
$$

We extend the reduction rules with rule RedPolyU which is similar to RedPolyL since it reduces an upper bounded type abstraction followed by an upper bounded type application into a type substitution followed by a coercion substitution.

$$
\textsc{RedPolyU} \quad \cdot[\sigma \triangleright G]\langle \Lambda(\alpha \triangleright c : \tau)\langle M \rangle \rangle \rightsquigarrow_\iota M[\alpha/\sigma][c/G]
$$

The coercion typing rules need also to be extended. Rule CoerTLamU says that if we can type a term with ρ under an environment extended with type variable α and coercion variable c of coercion type $\alpha \triangleright \tau$, then this term also has type $\forall(\alpha \triangleright \tau)\, \rho$ under the non-extended environment. Notice that this abstraction extend the environment with two binders similarly to rule CoerTLamL. Rule CoerTAppU permits to view a term of type $\forall(\alpha \triangleright \tau)\, \rho$ with type $\rho[\alpha/\sigma]$ whenever σ is well-formed and smaller than $\tau[\alpha/\sigma]$. We substituted α by σ in τ since we allow recursive bounds.

$$
\textsc{CoerTLamU} \quad \Lambda(\alpha \triangleright c : \tau) \Rightarrow \Gamma \vdash (\alpha, (c : \alpha \triangleright \tau) \vdash \rho) \triangleright \forall(\alpha \triangleright \tau)\, \rho
\qquad
\textsc{CoerTAppU} \quad \frac{\Gamma \vdash \sigma \;\mathsf{type} \qquad G \Rightarrow \Gamma \vdash \sigma \triangleright \tau[\alpha/\sigma]}{\cdot[\sigma \triangleright G] \Rightarrow \Gamma \vdash \forall(\alpha \triangleright \tau)\, \rho \triangleright \rho[\alpha/\sigma]}
$$

Finally, we extend the well-formedness rules. By rule TypeForU, the upper bounded polymorphic type $\forall(\alpha \triangleright \tau)\, \rho$ is well-formed if its body ρ is well-formed under the environment extended with the type binding α and coercion binding $(c : \alpha \triangleright \tau)$. So α is bound in ρ and we know that it is smaller than τ. By rule EnvTypeU, the extended environment $\Gamma, \alpha, (c : \alpha \triangleright \tau)$ is well-formed if the bound variables are not already bound in environment Γ and τ is well-formed under Γ, α which allows τ to mention α and thus permits recursive bounds.

$$
\textsc{TypeForU} \quad \frac{\Gamma, \alpha, (c : \alpha \triangleright \tau) \vdash \rho \;\mathsf{type}}{\Gamma \vdash \forall(\alpha \triangleright \tau)\, \rho \;\mathsf{type}}
\qquad
\textsc{EnvTypeU} \quad \frac{c, \alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \alpha \vdash \tau \;\mathsf{type}}{\Gamma, \alpha, (c : \alpha \triangleright \tau) \;\mathsf{env}}
$$

4.3 System F_ι^p

Now that the base system and the features are presented, we may compose them altogether into a type system that we call System F_ι^p. This language is described with a slightly different presentation in a paper [11]. In this framework, we made it clear that coercions act on typings and not only on types.

We sum up the syntax of the base system and all extensions in Figure 4.7. We write type variables α or β and coercion variables c. Coercion variables are only necessary in the explicit version of the type system. Terms are written M or N and only necessary in the explicit version. They contain variables x, abstractions $\lambda(x : \tau)\, M$, applications $M\, M$, pairs $\langle M, M \rangle$, projections $\mathsf{fst}\, M$ and $\mathsf{snd}\, M$, and coercion constructs $G\langle M \rangle$.

Types are written τ, σ, or ρ. They contain variables α, arrow types $\tau \to \tau$, product types $\tau \times \tau$, polymorphic types $\forall \alpha\, \tau$, the bottom type $\perp$, the top type $\top$, lower bounded polymorphic types $\forall(\alpha \triangleright \tau)\, \tau$, and upper bounded polymorphic types $\forall(\alpha \triangleleft \tau)\, \tau$. In bounded polymorphic

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
c & & & \text{Coercion variables} \\
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid \langle M, M\rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid G\langle M\rangle & \text{Explicit terms} \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall\alpha\, \tau \mid \bot \mid \top \mid \forall(\alpha \triangleleft \tau)\, \tau \mid \forall(\alpha \triangleright \tau)\, \tau & \text{Types} \\
G & ::= & c \mid \Diamond \mid G \circ G \mid *G \mid \Lambda\alpha \mid \bullet\, \tau \mid G \xrightarrow{\tau} G \mid G \times G \mid \bot\tau \mid \top & \text{Coercions} \\
& \mid & \Lambda(\alpha \triangleleft c : \tau) \mid \bullet[\tau \triangleleft G] \mid \Lambda(\alpha \triangleright c : \tau) \mid \bullet[\tau \triangleright G] & \\
\Gamma & ::= & \varnothing \mid \Gamma, (x:\tau) \mid \Gamma, \alpha \mid \Gamma, (c : \tau \triangleright \tau) & \text{Environments}
\end{array}
$$

Figure 4.7: System F_ι^p syntax

types $\forall(\alpha \triangleright \tau)\, \rho$ and $\forall(\alpha \triangleleft \tau)\, \rho$, the type variable α is bound in both the bound τ and the body ρ. It is bound in ρ because it is a polymorphic type, and it is bound in τ because we allow recursive bounds. The upper (resp. lower) bound polymorphic type $\forall(\alpha \triangleright \tau)\, \rho$ (resp. $\forall(\alpha \triangleleft \tau)\, \rho$) can be read as type ρ for all abstract type α such that a coercion from α to τ (resp. from τ to α) exists.

Coercions are written G and only necessary in the explicit version since they only occur in the coercion constructs of terms which are by definition only necessary in the explicit version. Coercions contain variables c, the reflexivity coercion $\Diamond$, transitivity coercions $G \circ G$, weakenings $*G$, type abstractions $\Lambda\alpha$, type applications $\bullet\, \tau$, arrow η-expansions $G \xrightarrow{\tau} G$, product η-expansions $G \times G$, bottom coercions $\bot\tau$, the top coercion $\top$, lower bounded type abstractions $\Lambda(\alpha \triangleleft c : \tau)$, lower bounded type applications $\bullet[\tau \triangleleft G]$, upper bounded type abstractions $\Lambda(\alpha \triangleright c : \tau)$, and upper bounded type applications $\bullet[\tau \triangleright G]$.

The transitivity coercions have to be read from right to left. The coercion $G_2 \circ G_1$ means that G_2 occurs after G_1, as we can see in rule RedTrans. The type annotation on the arrow η-expansion coercion is needed for rule RedArr. Upper and lower bounded abstractions bind the type variable α and the coercion variable c, which has type $\alpha \triangleright \tau$ and $\tau \triangleright \alpha$ respectively. Notice that for lower bounded abstraction, we write $\Lambda(\alpha \triangleleft c : \tau)$ with a reverse $\triangleright$. This is because we want to enhance that both c and α are bound and that c has coercion type $\tau \triangleright \alpha$. A similar reason holds for the lower bounded polymorphic type $\forall(\alpha \triangleleft \tau)\, \rho$. Upper and lower bounded applications are made of three parts: a type instantiation τ, a coercion instantiation G, and a direction $\triangleright$ or $\triangleleft$ which corresponds to upper or lower bounded application, respectively. The orientation is the same as its associated abstraction.

Finally, we define environments Γ as lists of bindings. The empty environment is written $\varnothing$ and extended environments are written with a comma. Environments extended with a term binding are written $\Gamma, (x : \tau)$, environments extended with a type binding are written Γ, α, and environments extended with a coercion binding are written $\Gamma, (c : \tau \triangleright \tau)$ in the explicit version and $\Gamma, (\tau \triangleright \tau)$ in the implicit version. Term bindings $(x : \tau)$ bind the term variable x to its type τ. A type binding α binds the type variable α. And coercion bindings $(c : \tau \triangleright \sigma)$ bind the coercion variable c to the coercion hypothesis $\tau \triangleright \sigma$. All variables bound in an environment have to be distinct. We ensure this restriction in the well-formedness judgment of environments.

We define some notations in Figure 4.8. Erasable environments are environments containing only erasable bindings. Erasable bindings are type or coercion bindings since they have to do with typing. Non-erasable bindings are term bindings since they have to do with computation. We write Σ for erasable environments.

$$
\begin{array}{rcll}
\Sigma & ::= & \varnothing \mid \Sigma, \alpha \mid \Sigma, (c : \tau \triangleright \tau) & \text{Erasable environments} \\
E & ::= & \lambda(x : \tau)\,[] \mid []\,M \mid M\,[] \mid \langle [], M \rangle \mid \langle M, [] \rangle \mid \mathsf{fst}\,[] \mid \mathsf{snd}\,[] \mid G\langle [] \rangle & \text{Evaluation contexts} \\
p & ::= & x \mid p\,v \mid \mathsf{fst}\,p \mid \mathsf{snd}\,p \mid \bullet\,\tau\langle p \rangle \mid (G \xrightarrow{\tau} G)\langle p \rangle \mid (G \times G)\langle p \rangle & \text{Prevalues} \\
& \mid & (\bot\tau)\langle p \rangle \mid (\bullet[\tau \triangleleft G])\langle p \rangle \mid (\bullet[\tau \triangleright G])\langle p \rangle \mid c\langle v \rangle & \\
v & ::= & p \mid \lambda(x : \tau)\,v \mid \langle v, v \rangle \mid \Lambda\alpha\langle v \rangle \mid \top\langle v \rangle \mid \Lambda(\alpha \triangleleft c : \tau)\langle v \rangle \mid \Lambda(\alpha \triangleright c : \tau)\langle v \rangle & \text{Values}
\end{array}
$$

Figure 4.8: System F_ι^p notations

RedCtx
$$\frac{M \rightsquigarrow_{\beta\iota} N}{E[M] \rightsquigarrow_{\beta\iota} E[N]}$$

RedApp
$$(\lambda(x : \tau)\,M)\,N \rightsquigarrow_\beta M[x/N]$$

RedFst
$$\mathsf{fst}\,\langle M, N \rangle \rightsquigarrow_\beta M$$

RedSnd
$$\mathsf{snd}\,\langle M, N \rangle \rightsquigarrow_\beta N$$

RedRefl
$$\Diamond\langle M \rangle \rightsquigarrow_\iota M$$

RedTrans
$$(G_2 \circ G_1)\langle M \rangle \rightsquigarrow_\iota G_2\langle G_1\langle M \rangle\rangle$$

RedWeak
$$(*G)\langle M \rangle \rightsquigarrow_\iota G\langle M \rangle$$

RedPoly
$$\bullet\,\tau\langle \Lambda\alpha\langle M \rangle\rangle \rightsquigarrow_\iota M[\alpha/\tau]$$

RedEtaArr
$$(G_1 \xrightarrow{\tau'} G_2)\langle \lambda(x : \tau)\,M \rangle \rightsquigarrow_\iota \lambda(x : \tau')\,G_2\langle M[x/G_1\langle x \rangle]\rangle$$

RedEtaProd
$$(G_1 \times G_2)\langle\langle M_1, M_2 \rangle\rangle \rightsquigarrow_\iota \langle G_1\langle M_1 \rangle, G_2\langle M_2 \rangle\rangle$$

RedPolyL
$$\bullet[\sigma \triangleleft G]\langle \Lambda(\alpha \triangleleft c : \tau)\langle M \rangle\rangle \rightsquigarrow_\iota M[\alpha/\sigma][c/G]$$

RedPolyU
$$\bullet[\sigma \triangleright G]\langle \Lambda(\alpha \triangleright c : \tau)\langle M \rangle\rangle \rightsquigarrow_\iota M[\alpha/\sigma][c/G]$$

Figure 4.9: System F_ι^p reduction rules

We define evaluation contexts, prevalues, and values. All of these are defined according to strong reduction. In strong reduction, evaluation contexts are all one-hole contexts of depth one. Prevalues are destructors applied to prevalues where a constructor is expected and values elsewhere. Finally, values are constructors applied to values. For System F_ι^p, prevalues contain in particular prevalues applied to a type $\bullet\,\tau\langle p \rangle$, prevalues applied to an η-expansion $(G \xrightarrow{\tau} G)\langle p \rangle$ and $(G \times G)\langle p \rangle$, instantiations of absurd prevalues $(\bot\tau)\langle p \rangle$, bounded type applications of prevalues $(\bullet[\tau \triangleright G])\langle p \rangle$ and $(\bullet[\tau \triangleleft G])\langle p \rangle$, and coerced values with an abstract coercion $c\langle v \rangle$. And values contain in particular type abstractions of values $\Lambda\alpha\langle v \rangle$, top values $\top\langle v \rangle$, and bounded type abstractions of values $\Lambda(\alpha \triangleright c : \tau)\langle v \rangle$ and $\Lambda(\alpha \triangleleft c : \tau)\langle v \rangle$.

We define the reduction rules in Figure 4.9. They are labeled with a β annotation for computational steps and ι annotation for typing steps or erasable steps. We use the meta-variable $\beta\iota$ to designate either of these annotations.

The first four rules mimic those of the λ-calculus. Rule RedCtx is the context rule and simply transfers the annotation from its subreduction to its whole reduction. If M reduces to N with annotation β (resp. ι), then $E[M]$ reduces to $E[N]$ with annotation β (resp. ι). Rules RedApp, RedFst, and RedSnd are β-reduction rules since they actually do a computational step. All following reduction rules are ι-reduction rules and only have to do with typings.

Rules RedRefl and RedTrans have to do with the closure properties of the coercion relation. The reflexivity coercion closes the coercion relation by reflexivity: typing $\Gamma \vdash \tau$ is smaller than typing $\Gamma \vdash \tau$ by coercion proof $\Diamond$. While the transitivity coercion closes the coercion relation by transitivity: if typing $\Gamma_1 \vdash \tau_1$ is smaller than typing $\Gamma_2 \vdash \tau_2$ by coercion proof G_1 and typing $\Gamma_2 \vdash \tau_2$ is smaller than typing $\Gamma_3 \vdash \tau_3$ by coercion proof G_2, then typing $\Gamma_1 \vdash \tau_1$ is smaller than typing $\Gamma_3 \vdash \tau_3$ by coercion proof $G_2 \circ G_1$. This is why $\Diamond\langle M\rangle$ ι-reduces to M since the typing does not change, and why $(G_2 \circ G_1)\langle M\rangle$ ι-reduces to $G_2\langle G_1\langle M\rangle\rangle$. Rule RedWeak simply applies the subcoercion.

Rule RedPoly corresponds to the usual rule of System F for polymorphism. A type abstraction followed by a type application results in a type substitution. In terms of typings and coercions, if we change the typing of a term M from typing $\Gamma, \alpha \vdash \tau$ to typing $\Gamma \vdash \forall\alpha\, \tau$ by coercion proof $\Lambda\alpha$ and then from typing $\Gamma \vdash \forall\alpha\, \tau$ to typing $\Gamma \vdash \tau[\alpha/\sigma]$ by coercion proof $\bullet\sigma$ where σ is a well-formed type under environment Γ, then the substitution of the free occurrences of the type variable α by the type σ, written $[\alpha/\sigma]$, changes the typing $\Gamma, \alpha \vdash \tau$ of the term M to the typing $\Gamma \vdash \tau[\alpha/\sigma]$, which is the same as what the original coercions where doing.

Rules RedEtaArr and RedEtaProd deal with η-expansion. They can be read using the η-expansion intuition. Coercions are erasable contexts and the η-expansion coercion $G_1 \overset{\tau'}{\to} G_2$ can be seen as the erasable context $\lambda(x : \tau')\, G_2\langle [\,]\, (G_1\langle x\rangle)\rangle$. When we fill this context with $\lambda(x : \tau)\, M$ as it is the case in rule RedEtaArr we get $\lambda(x : \tau')\, G_2\langle(\lambda(x : \tau)\, M)\, (G_1\langle x\rangle)\rangle$ which reduces to $\lambda(x : \tau')\, G_2\langle M[x/G_1\langle x\rangle]\rangle$ which is exactly the right-hand side of the reduction rule. The same mechanism works for rule RedEtaProd when taking $\langle G_1\langle \mathsf{fst}\, [\,]\rangle, G_2\langle \mathsf{snd}\, [\,]\rangle\rangle$ as the erasable context for coercion $G_1 \times G_2$.

Some additional or alternate rules could have been used instead of rules RedEtaArr and RedEtaProd. We focus on the arrow type only, since the product type is very similar. The two additional or alternate rules we consider are RedEtaArr1 and RedEtaArr2.

RedEtaArr1
$$(G_1 \overset{\tau'}{\to} G_2)\langle M\rangle\, N \rightsquigarrow_\iota G_2\langle M\, (G_1\langle N\rangle)\rangle$$

Rule RedEtaArr1 is the analog of rule RedEtaArr in the sense that an η-expansion has two potential redexes: one with the destructor above its hole, one with the constructor at its root. Rule RedEtaArr considers the redex under its hole while rule RedEtaArr1 considers the redex at its root. If we write down the η-expansion view of rule RedEtaArr1 as erasable contexts we get $(\lambda(x : \tau')\, G_2\langle M\, (G_1\langle x\rangle)\rangle)\, N$ which reduces at its root to $G_2\langle M\, (G_1\langle N\rangle)\rangle$. Notice that one advantage of this rule on rule RedEtaArr is that the type annotation τ' is not needed and the only reason for this annotation to be present in the syntax is to formulate rule RedEtaArr. However, rule RedEtaArr1 modifies the set of prevalues and values and in particular the classification of irreducible terms according to their type. For instance $(G_1 \overset{\tau'}{\to} G_2)\langle \lambda(x : \tau)\, M\rangle$ is an irreducible well-typed term if we replace rule RedEtaArr with rule RedEtaArr1. Both rules, RedEtaArr and RedEtaArr1, can be used independently or together.

RedEtaArr2
$$(G_1' \overset{\tau'}{\to} G_2')\langle (G_1 \overset{\tau}{\to} G_2)\langle M\rangle\rangle \rightsquigarrow_\iota ((G_1 \circ G_1') \overset{\tau'}{\to} (G_2' \circ G_2))\langle M\rangle$$

Rule RedEtaArr2 is special and actually problematic. Rule RedEtaArr1 was the analog of rule RedEtaArr because it was using the other redex of the hidden η-expansion, whereas rule RedEtaArr2 mixes the root half-redex of one η-expansion with the hole half-redex of another η-expansion. If we write down the η-expansion view of this rule as erasable contexts we get:

$$\lambda(x : \tau')\, G_2'\langle(\lambda(x : \tau)\, G_2\langle M\, (G_1\langle x\rangle)\rangle)\, (G_1'\langle x\rangle)\rangle$$

$$\frac{\Gamma\ \mathsf{env} \qquad (x:\tau)\in\Gamma}{x \Rightarrow x : \Gamma \vdash \tau}\ \textsc{TermVar} \qquad \frac{\Gamma\vdash\tau\ \mathsf{type} \qquad M \Rightarrow a : \Gamma,(x:\tau)\vdash\sigma}{\lambda(x:\tau)\,M \Rightarrow \lambda x\,a : \Gamma\vdash\tau\to\sigma}\ \textsc{TermLam}$$

$$\frac{M \Rightarrow a : \Gamma\vdash\tau\to\sigma \qquad N \Rightarrow b : \Gamma\vdash\tau}{M\,N \Rightarrow a\,b : \Gamma\vdash\sigma}\ \textsc{TermApp} \qquad \frac{M \Rightarrow a : \Gamma\vdash\tau \qquad N \Rightarrow b : \Gamma\vdash\sigma}{\langle M,N\rangle \Rightarrow \langle a,b\rangle : \Gamma\vdash\tau\times\sigma}\ \textsc{TermPair}$$

$$\frac{M \Rightarrow a : \Gamma\vdash\tau\times\sigma}{\mathsf{fst}\,M \Rightarrow \mathsf{fst}\,a : \Gamma\vdash\tau}\ \textsc{TermFst} \qquad \frac{M \Rightarrow a : \Gamma\vdash\tau\times\sigma}{\mathsf{snd}\,M \Rightarrow \mathsf{snd}\,a : \Gamma\vdash\sigma}\ \textsc{TermSnd}$$

$$\frac{M \Rightarrow a : \Gamma,\Sigma\vdash\tau \qquad G \Rightarrow \Gamma\vdash(\Sigma\vdash\tau)\triangleright\sigma}{G\langle M\rangle \Rightarrow a : \Gamma\vdash\sigma}\ \textsc{TermCoer}$$

Figure 4.10: System F_ι^p term judgment relation

It reduces to $\lambda(x:\tau')\,G_2'\langle G_2\langle M\,(G_1\langle G_1'\langle x\rangle\rangle)\rangle\rangle$ which is a reduct of the right-hand side of the reduction rule. This is where there is a problem since we break our intuition: coercions are erasable contexts. One way to recover it is to ask the closure reductions to be actually closure equivalences. In particular, we would have the following equivalence rules.

$$\Diamond\langle M\rangle \equiv M \qquad (G_2\circ G_1)\langle M\rangle \equiv G_2\langle G_1\langle M\rangle\rangle \qquad G_3\circ(G_2\circ G_1)\equiv(G_3\circ G_2)\circ G_1$$

Rule RedEtaArr2 would then be compatible with both RedEtaArr and RedEtaArr1. However, this rule alone is not enough to ensure progress since it does not act on computational terms as an η-expansion rule should do. So the six possible configurations for the η-expansion rules of the arrow type are: rule RedEtaArr alone, rule RedEtaArr1 alone, rules RedEtaArr and RedEtaArr1 together, rule RedEtaArr2 with rule RedEtaArr, rule RedEtaArr2 with rule RedEtaArr1, and rule RedEtaArr2 with rules RedEtaArr and RedEtaArr1. And each time we add rule RedEtaArr2 we have to modify the framework to allow term equivalence and add enough structural equivalence rules. Also, for each situation, the set of prevalues and values differ.

Rules RedPolyU and RedPolyL are similar to rule RedPoly. A bounded type abstraction followed by a similarly bounded type application results in a type substitution followed by a coercion substitution.

Term typing rules are given on Figure 4.10. The term typing judgment is of the form $M \Rightarrow a : \Gamma\vdash\tau$ where M is a term which is a partial proof that the term a has type τ under environment Γ. It is a partial proof because it only contains enough information to rebuild the term a, and enough information to rebuild the type τ given an environment Γ. The term is necessary only in the explicit version of the type system.

Rule TermVar gives type τ to the term variable x if x is bound to τ in environment Γ. The term witnessing this rule is the term variable x itself. Since we want derivations of the judgment $M \Rightarrow a : \Gamma\vdash\tau$ to hold the proof that its type τ is well-formed under its environment Γ, we have to ask the environment Γ to be well-formed. Rule TermLam gives type $\tau\to\sigma$ to the term $\lambda x\,a$ under environment Γ if the term a has type σ under the extended environment

$$\frac{}{\Diamond \Rightarrow \Gamma \vdash \tau \rhd \tau}\ \textsc{CoerRefl} \qquad \frac{G_1 \Rightarrow \Gamma, \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \rhd \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \rhd \tau_3}{G_2 \circ G_1 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \rhd \tau_3}\ \textsc{CoerTrans}$$

$$\frac{(c : \tau \rhd \sigma) \in \Gamma}{c \Rightarrow \Gamma \vdash \tau \rhd \sigma}\ \textsc{CoerVar} \qquad \frac{\Gamma, \Sigma\ \mathsf{env} \qquad G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma}{*G \Rightarrow \Gamma \vdash \tau \rhd \sigma}\ \textsc{CoerWeak} \qquad \frac{}{\Lambda\alpha \Rightarrow \Gamma \vdash (\alpha \vdash \tau) \rhd \forall\alpha\, \tau}\ \textsc{CoerTLam}$$

$$\frac{\Gamma \vdash \sigma\ \mathsf{type}}{\bullet\sigma \Rightarrow \Gamma \vdash \forall\alpha\, \tau \rhd \tau[\alpha/\sigma]}\ \textsc{CoerTApp} \qquad \frac{\Gamma \vdash \tau'\ \mathsf{type} \qquad G_1 \Rightarrow \Gamma, \Sigma \vdash \tau' \rhd \tau \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \sigma) \rhd \sigma'}{G_1 \xrightarrow{\tau'} G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau \to \sigma) \rhd \tau' \to \sigma'}\ \textsc{CoerEtaArr}$$

$$\frac{G_1 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \tau' \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \sigma) \rhd \sigma'}{G_1 \times G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau \times \sigma) \rhd \tau' \times \sigma'}\ \textsc{CoerEtaProd} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type}}{\bot\tau \Rightarrow \Gamma \vdash \bot \rhd \tau}\ \textsc{CoerBot} \qquad \frac{}{\top \Rightarrow \Gamma \vdash \tau \rhd \top}\ \textsc{CoerTop}$$

$$\frac{\Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Lambda(\alpha \lhd c : \tau) \Rightarrow \Gamma \vdash (\alpha, (c : \tau \rhd \alpha) \vdash \rho) \rhd \forall(\alpha \lhd \tau)\, \rho}\ \textsc{CoerTLamL} \qquad \frac{\Gamma \vdash \sigma\ \mathsf{type} \qquad G \Rightarrow \Gamma \vdash \tau[\alpha/\sigma] \rhd \sigma}{\bullet[\sigma \lhd G] \Rightarrow \Gamma \vdash \forall(\alpha \lhd \tau)\, \rho \rhd \rho[\alpha/\sigma]}\ \textsc{CoerTAppL}$$

$$\frac{\Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Lambda(\alpha \rhd c : \tau) \Rightarrow \Gamma \vdash (\alpha, (c : \alpha \rhd \tau) \vdash \rho) \rhd \forall(\alpha \rhd \tau)\, \rho}\ \textsc{CoerTLamU} \qquad \frac{\Gamma \vdash \sigma\ \mathsf{type} \qquad G \Rightarrow \Gamma \vdash \sigma \rhd \tau[\alpha/\sigma]}{\bullet[\sigma \rhd G] \Rightarrow \Gamma \vdash \forall(\alpha \rhd \tau)\, \rho \rhd \rho[\alpha/\sigma]}\ \textsc{CoerTAppU}$$

Figure 4.11: System F_ι^p coercion judgment relation

$\Gamma, (x : \tau)$ where the term variable x is now bound to the type τ. If we have M the term for the premise, we write $\lambda(x : \tau)\, M$ the term witnessing the conclusion. We need to add the type annotation τ since we cannot rebuild it from M. Rule TermApp gives type σ to the application of a to b under environment Γ if a has type $\tau \to \sigma$ under environment Γ and b has type σ under environment Γ. If M and N are the terms for a and b respectively, then we use $M\, N$ for the term of the conclusion.

Rule TermPair gives type $\tau \times \sigma$ to the pair $\langle a, b\rangle$ under environment Γ if a has type τ under environment Γ and b has type σ under environment Γ. If M and N are the terms for a and b respectively, then we use $\langle M, N\rangle$ for the term of the conclusion. Rule TermFst gives type τ to the first projection $\mathsf{fst}\, a$ under environment Γ if a has type $\tau \times \sigma$ under environment Γ. If M is the term for the premise, then we write $\mathsf{fst}\, M$ for the term of the conclusion. Rule TermSnd gives type σ to the second projection $\mathsf{snd}\, a$ under environment Γ if a has type $\tau \times \sigma$ under environment Γ. If M is the term for the premise, then we write $\mathsf{snd}\, M$ for the term of the conclusion.

Finally, rule TermCoer gives typing $\Gamma \vdash \sigma$ to the term a if it also has typing $\Gamma, \Sigma \vdash \tau$ and there is a coercion from typing $\Sigma \vdash \tau$ to type σ under Γ, written $\Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma$. If we name M the term witnessing $a : \Gamma, \Sigma \vdash \tau$ and G the coercion proof, then we write $G\langle M\rangle$ the term for the conclusion that a has typing $\Gamma \vdash \sigma$.

We can now define the coercion typing rules which are given on Figure 4.11. The coercion judgment is written $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma$. The coercion proof G witnesses the derivation of

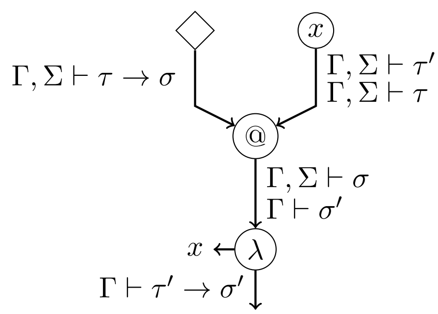
$\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$. This judgment means that the typing $\Sigma \vdash \tau$ is smaller than the type σ under environment Γ, which can also be seen as the polymorphic type $\forall \Sigma\, \tau$ being smaller than σ under environment Γ. When the erasable environment Σ is empty, we may write $\Gamma \vdash \tau \triangleright \sigma$.

Rules CoerRefl and CoerTrans are about the closure of the coercion relation by reflexivity and transitivity. The coercion proof $\Diamond$ is a witness that type τ is smaller than itself under environment Γ by rule CoerRefl. Rule CoerTrans tells that if coercion G_2 proves that $\Sigma_2 \vdash \tau_2$ is smaller than τ_3 under Γ and coercion G_1 proves that $\Sigma_1 \vdash \tau_1$ is smaller than τ_2 under Γ, Σ_2 then coercion $G_2 \circ G_1$ proves that $\Sigma_2, \Sigma_1 \vdash \tau_1$ is smaller than τ_3 under Γ.

Rule CoerVar looks in the environment Γ a coercion hypothesis. In the explicit version this lookup is done using the name given by the coercion variable c. If c is bound to $\tau \triangleright \sigma$ in Γ, then τ is smaller than σ. Rule CoerWeak forgets that a coercion G used to extend the environment if the extension is not used by the inner type τ. Concretely, if G witnesses a coercion from τ to σ extending Σ, then $*G$ witnesses a coercion from τ to σ with no environment extension. This is sound since τ is well-formed under Γ by hypothesis.

Rules CoerTLam and CoerTApp have to do with polymorphism. The first is type generalization and the second is type instantiation. We define coercion $\Lambda\alpha$ as a proof that typing $\alpha \vdash \tau$ is included in type $\forall \alpha\, \tau$ under Γ. And we define coercion $\bullet\sigma$ as a proof that we can go from type $\forall \alpha\, \tau$ to type $\tau[\alpha/\sigma]$ under Γ, given type σ is well-formed under Γ.

Rules CoerEtaArr and CoerEtaProd are about η-expansions so to understand the rules we need to look at their η-expansion views as erasable contexts. The coercion $G_1 \overset{\tau'}{\to} G_2$ can be viewed as the erasable context $\lambda(x : \tau')\, G_2\langle [] \, (G_1\langle x \rangle)\rangle$. We can display the following typing derivation as a graph:



or as a typing derivation where $\Gamma' = \Gamma, (x : \tau')$:

$$\dfrac{\dfrac{[] \Rightarrow [] : \Gamma', \Sigma \vdash \tau \to \sigma \qquad \dfrac{x \Rightarrow x : \Gamma', \Sigma \vdash \tau' \quad G_1 \Rightarrow \Gamma', \Sigma \vdash \tau' \triangleright \tau}{G_1\langle x \rangle \Rightarrow x : \Gamma', \Sigma \vdash \tau}}{[] \, (G_1\langle x \rangle) \Rightarrow [] \, x : \Gamma', \Sigma \vdash \sigma} \qquad G_2 \Rightarrow \Gamma' \vdash (\Sigma \vdash \sigma) \triangleright \sigma'}{\dfrac{G_2\langle [] \, (G_1\langle x \rangle)\rangle \Rightarrow [] \, x : \Gamma' \vdash \sigma'}{\lambda(x : \tau')\, G_2\langle [] \, (G_1\langle x \rangle)\rangle \Rightarrow \lambda x \, [] \, x : \Gamma \vdash \tau' \to \sigma'}}$$

Rule CoerTop tells that any type is smaller than the top type. And rule CoerBot tells that any well-formed type is bigger that the bottom type.

Finally, rules CoerTLamU, CoerTAppU, CoerTLamL, and CoerTAppL have to do with upper and lower bounded polymorphism and are thus similar to rules CoerTLam and CoerTApp. Rule CoerTLamU defines coercion $\Lambda(\alpha \triangleright c : \tau)$ as a proof that typing $\alpha, (c : \alpha \triangleright \tau) \vdash \rho$ is smaller than type $\forall(\alpha \triangleright \tau)\, \rho$ under Γ. Notice that this coercion binds two variables: the

$$\frac{\Gamma\ \mathsf{env} \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha\ \mathsf{type}}\ \textsc{TypeVar} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \to \sigma\ \mathsf{type}}\ \textsc{TypeArr} \qquad \frac{\Gamma \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash \sigma\ \mathsf{type}}{\Gamma \vdash \tau \times \sigma\ \mathsf{type}}\ \textsc{TypeProd}$$

$$\frac{\Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma \vdash \forall \alpha\, \tau\ \mathsf{type}}\ \textsc{TypeFor} \qquad \frac{\Gamma\ \mathsf{env}}{\Gamma \vdash \bot\ \mathsf{type}}\ \textsc{TypeBot} \qquad \frac{\Gamma\ \mathsf{env}}{\Gamma \vdash \top\ \mathsf{type}}\ \textsc{TypeTop} \qquad \frac{\Gamma, \alpha \vdash \tau\ \mathsf{type} \qquad \Gamma, \alpha, (c : \tau \rhd \alpha) \vdash \rho\ \mathsf{type}}{\Gamma \vdash \forall (\alpha \lhd \tau)\, \rho\ \mathsf{type}}\ \textsc{TypeForL}$$

$$\frac{\Gamma, \alpha \vdash \tau\ \mathsf{type} \qquad \Gamma, \alpha, (c : \alpha \rhd \tau) \vdash \rho\ \mathsf{type}}{\Gamma \vdash \forall (\alpha \rhd \tau)\, \rho\ \mathsf{type}}\ \textsc{TypeForU}$$

$$\textsc{EnvEmpty}\ \ \varnothing\ \mathsf{env} \qquad \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau\ \mathsf{type}}{\Gamma, (x : \tau)\ \mathsf{env}}\ \textsc{EnvTerm} \qquad \frac{\Gamma\ \mathsf{env} \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\Gamma, \alpha\ \mathsf{env}}\ \textsc{EnvType}$$

$$\frac{c, \alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma, \alpha, (c : \tau \rhd \alpha)\ \mathsf{env}}\ \textsc{EnvTypeL} \qquad \frac{c, \alpha \notin \mathsf{dom}(\Gamma) \qquad \Gamma, \alpha \vdash \tau\ \mathsf{type}}{\Gamma, \alpha, (c : \alpha \rhd \tau)\ \mathsf{env}}\ \textsc{EnvTypeU}$$

Figure 4.12: System F_ι^p well-formedness relations

type variable α and the coercion variable c with coercion type $\alpha \rhd \tau$. Rule CoerTAppU defines the upper bounded instantiation coercion $\cdot[\sigma \rhd G]$ as a proof that type $\forall(\alpha \rhd \tau)\,\rho$ is smaller than $\rho[\alpha/\sigma]$ under environment Γ given σ is well-formed and coercion G is a proof that σ is smaller than $\tau[\alpha/\sigma]$.

Similarly, rule CoerTLamL defines coercion $\Lambda(\alpha \lhd c : \tau)$ as a proof that typing $\alpha, (c : \tau \rhd \alpha) \vdash \rho$ is smaller than type $\forall(\alpha \lhd \tau)\,\rho$ under Γ. Notice that this coercion binds two variables: the type variable α and the coercion variable c with coercion type $\tau \rhd \alpha$. Rule CoerTAppL defines the lower bounded instantiation coercion $\cdot[\sigma \lhd G]$ as a proof that type $\forall(\alpha \lhd \tau)\,\rho$ is smaller than $\rho[\alpha/\sigma]$ under environment Γ given σ is well-formed and coercion G is a proof that $\tau[\alpha/\sigma]$ is smaller than σ.

It remains to define the well-formedness rules. A type τ is well-formed under environment Γ, written $\Gamma \vdash \tau\ \mathsf{type}$, if it has a derivation using the rules given in Figure 4.12. Similarly, an environment Γ is well-formed, written $\Gamma\ \mathsf{env}$, if it has a derivation using the rules given in the same figure.

Rule TypeVar tells that a type variable α is well-formed under environment Γ if it is bound in the environment. Since we want to extract from a type derivation that the environment is well-formed, we need to ask the environment Γ to be well-formed. Rule TypeArr tells that type $\tau \to \sigma$ is well-formed under environment Γ, if both τ and σ are well-formed under Γ. Similarly, type $\tau \times \sigma$ is well-formed under environment Γ by rule TypeProd, if both τ and σ are well-formed under Γ.

Rule TypeFor tells that the polymorphic type $\forall \alpha\, \tau$ is well-formed under environment Γ if type τ is well-formed under the extended environment Γ, α. The top and bottom types are well-formed, according to rules TypeTop and TypeBot respectively, if their environment is well-formed. Rules TypeForU and TypeForL are similar. Type $\forall(\alpha \rhd \tau)\,\rho$ is well-formed under environment Γ if type ρ is well-formed under the extended environment $\Gamma, \alpha, (c : \alpha \triangleright \tau)$. Similarly, type $\forall(\alpha \triangleleft \tau)\, \rho$ is well-formed under environment Γ if type ρ is well-formed under the extended environment $\Gamma, \alpha, (c : \tau \triangleright \alpha)$.

Since bindings are related to abstractions and not variables, and since System F_ι^p has abstractions binding more than one variable, the well-formedness derivation of an environment has to split the environment as a list of abstractions. System F_ι^p has four kind of abstractions: term abstractions with binding $(x : \tau)$, type abstraction with binding α, upper bounded type abstraction with binding $\alpha, (c : \alpha \triangleright \tau)$, and lower bounded type abstraction with binding $\alpha, (c : \tau \triangleright \alpha)$.

Once the environment is split by abstractions we proceed by recurrence over this list of abstractions. Either the list is empty and we have the empty environment $\varnothing$ which is always well-formed by rule EnvEmpty. Or we have a non-empty list of abstractions and we look at the last one. The remaining list has to be well-formed and the last abstraction has to be well-formed under the remaining environment. A term abstraction associating the term variable x to the type τ is well-formed by rule EnvTerm if the type τ is well-formed under the remaining environment Γ and the term variable x is not already bound in Γ. A type abstraction α is well-formed by rule EnvType if the type variable α is not already bound in the remaining environment. Upper and lower type abstractions, $\alpha, (c : \alpha \triangleright \tau)$ and $\alpha, (c : \tau \triangleright \alpha)$ respectively, are well-formed by rules EnvTypeU and EnvTypeL respectively if the type variable α and coercion variable c are not already bound in the remaining environment Γ and if the type τ is well-formed under the environment Γ, α since we allow recursive bounds.

4.4 Properties

We first describe the properties linking the implicit version and the explicit version of the type system as we did in Chapter 3. We then prove the strong normalization of the explicit reduction of well-typed terms by translating our judgments in System F. Using termination we prove the confluence of the explicit reduction. We then show how the explicit reduction corresponds to the implicit reduction in the bisimulation lemma. And we finally prove that System F_ι^p is sound and strongly normalizing in both its explicit and implicit versions. We can conclude from these properties that well-typed terms in System F_ι^p strongly normalize to a unique value without encountering any error.

4.4.1 Implicit vs. Explicit version

We first give the expected properties of an explicit type system. A judgment has to be unique according to its explicit entity. For instance, when $M \Rightarrow a : \Gamma \vdash \tau$ holds, then a is determined by M, and τ is a function of the term M and the environment Γ. For coercions, when we have $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$, then the erasable environment Σ is a function of coercion G and environment Γ, and the type σ is a function of coercion G, environment Γ, and type τ.

Lemma 42 (Uniqueness). *The following assertions hold.*

- *If $M \Rightarrow a_1 : \Gamma_1 \vdash \tau_1$ and $M \Rightarrow a_2 : \Gamma_2 \vdash \tau_2$ hold, then $a_1 = a_2$ holds.*
- *If $M \Rightarrow a : \Gamma \vdash \tau_1$ and $M \Rightarrow a : \Gamma \vdash \tau_2$ hold, then $\tau_1 = \tau_2$ hold.*
- *If $G \Rightarrow \Gamma \vdash (\Sigma_1 \vdash \tau_1) \triangleright \sigma_1$ and $G \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \triangleright \sigma_2$ hold, then $\Sigma_1 = \Sigma_2$ holds.*

$$
\begin{array}{rclcrclcrcl}
\lfloor x \rfloor &=& x & & \lfloor \langle M, N\rangle \rfloor &=& \langle \lfloor M \rfloor, \lfloor N \rfloor \rangle & & & & \\
\lfloor \lambda(x:\tau)\, M \rfloor &=& \lambda x\, \lfloor M \rfloor & & \lfloor \mathsf{fst}\, M \rfloor &=& \mathsf{fst}\, \lfloor M \rfloor & & \lfloor G\langle M\rangle \rfloor &=& \lfloor M \rfloor \\
\lfloor M\, N \rfloor &=& \lfloor M \rfloor\, \lfloor N \rfloor & & \lfloor \mathsf{snd}\, M \rfloor &=& \mathsf{snd}\, \lfloor M \rfloor & & & &
\end{array}
$$

Figure 4.13: System F_ι^p drop function

- *If $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma_1$ and $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma_2$ hold, then $\sigma_1 = \sigma_2$ holds.*

Proof. For each assertion, by induction on the first hypothesis and inversion of the second. The inversion leads to exactly one rule, which is actually the same as the one for the induction. Rule TermVar uses the fact that a well-formed environment binds each term variable at most once. All other cases simply use induction hypotheses. □

Actually a is a function of M even if M is not well-typed. This is not useful as a result, but the function in question is useful for the bisimulation properties. We call this function the *drop* function and we write it $\lfloor M \rfloor$. It is simply defined by dropping the annotations in M. The formal definition is given on Figure 4.13. This lemma explains why we usually omit a in explicit term judgments and write $M : \Gamma \vdash \tau$ instead of $M \Rightarrow a : \Gamma \vdash \tau$.

Lemma 43. *If $M \Rightarrow a : \Gamma \vdash \tau$ holds, then $a = \lfloor M \rfloor$ holds.*

Proof. By induction. □

The next lemma tells that M is actually contained in the implicit typing derivation. In other words, from a derivation of $a : \Gamma \vdash \tau$ we can extract the term M such that $M \Rightarrow a : \Gamma \vdash \tau$ holds. The environment Γ has to be filled with distinct coercion variables. Reciprocally, if $M \Rightarrow a : \Gamma \vdash \tau$ holds, we can extract a derivation of $a : \Gamma' \vdash \tau$ where Γ' is obtained from Γ by removing coercion variables. Similarly for coercions, we can add or remove the witness.

Lemma 44 (Equivalence)**.** *The following assertions hold.*

- *$a : \Gamma \vdash \tau$ holds if and once if $M \Rightarrow a : \Gamma \vdash \tau$ holds for some M.*
- *$\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ holds if and only if $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ holds for some G.*

Proof. By induction. The reciprocals are very easy since we only remove information. The implicit to explicit direction simply uses the induction hypotheses and the syntax of the side judgments. □

4.4.2 Termination

Termination for the explicit version is shown by reification into System F . We reify types, environments, terms, and coercion proofs into System F. We show that the reification of System F_ι^p derivations are also valid derivations in System F. So the reification of a well-typed explicit term of System F_ι^p remains well-typed in System F. We also show a simulation from the explicit reduction in System F_ι^p to the reduction of System F. Thus a non-terminating well-typed explicit term in System F_ι^p implies the existence of a non-terminating well-typed term in System F, which is known to be strongly normalizing. We use a hat to denote the reification of an object.

$$
\begin{array}{ccc}
\widehat{x} = x & \widehat{\langle M, N\rangle} = \langle \widehat{M}, \widehat{N}\rangle & \\
\widehat{\lambda(x : \tau)\, M} = \lambda(x : \widehat{\tau})\, \widehat{M} & \widehat{\mathsf{fst}\, M} = \mathsf{fst}\, \widehat{M} & \widehat{G\langle M\rangle} = \widehat{G}[\widehat{M}] \\
\widehat{M\, N} = \widehat{M}\, \widehat{N} & \widehat{\mathsf{snd}\, M} = \mathsf{snd}\, \widehat{M} &
\end{array}
$$

Figure 4.14: System F_ι^p term reification function

$$
\begin{array}{cc}
\widehat{\alpha} = \alpha & \widehat{\top} = \forall\alpha\, \alpha \to \alpha \\
\widehat{\tau \to \tau} = \widehat{\tau} \to \widehat{\sigma} & \widehat{\bot} = \forall\alpha\, \alpha \\
\widehat{\tau \times \tau} = \widehat{\tau} \times \widehat{\sigma} & \widehat{\forall(\alpha \triangleright \tau)\, \rho} = \forall\alpha\, (\alpha \to \widehat{\tau}) \to \widehat{\rho} \\
\widehat{\forall\alpha\, \tau} = \forall\alpha\, \widehat{\tau} & \widehat{\forall(\alpha \triangleleft \tau)\, \rho} = \forall\alpha\, (\widehat{\tau} \to \alpha) \to \widehat{\rho}
\end{array}
$$

Figure 4.15: System F_ι^p type reification function

We give the reification of explicit terms in Figure 4.14 and write $\widehat{M}$ for the reification of the term M. Terms of System F_ι^p are reified to terms of System F. Variables are reified to variables, abstractions to abstractions, applications to applications, pairs to pairs, and projections to projections, and recursively for subterms. The only interesting case is the coercion application $G\langle M\rangle$. Coercions of System F_ι^p are reified to System F multi-hole (but with at least one hole, see the reification of coercions) contexts. So the reification of the coercion application of G to the term M is the application of the context $\widehat{G}$ to the term $\widehat{M}$, leading to an possible duplication of $\widehat{M}$.

We give the reification of types in Figure 4.15. Type variables, arrow types, product types, and polymorphic types are simply translated to their equivalent in System F. The top type is reified to the polymorphic identity type. The bottom type is reified to the polymorphic bottom type $\forall\alpha\, \alpha$. Upper and lower bounded polymorphic types are reified to polymorphic arrow types: abstract coercions are reified to functions. Upper bounded polymorphism reification first abstracts over the type variable α, then over the function $\alpha \to \widehat{\tau}$, with body $\widehat{\rho}$. Lower bounded polymorphism is similar, but since τ is a lower bound, the reified abstract coercion is a function from $\widehat{\tau}$ to α.

The most interesting reification function is for coercions given in Figure 4.16. Coercions are reified to multi-hole contexts, with at least one hole. We write id for the identity function $\lambda(x : \tau)\, x$ of System F at the correct type τ depending of where it is used. There is always exactly one type, so it lighten the notations. We use id to add extra reduction steps for the simulation to hold. We also use top for $\lambda(y : \tau)\, \Lambda\alpha\, \lambda(x : \alpha)\, x$ where τ is the type where top is used, similarly to id. top is used to forget the type of an expression, but still keep it under hand to reduce inside it for the simulation.

Coercion variables are reified to term variables applied to the hole. We actually partition the term variables of System F into two parts: one part for the reification of System F_ι^p term variables, and one for coercion variables. The reflexivity coercion applies the identity function to the hole in order to have an extra reduction step. The transitivity coercion composes the two sub-contexts and applies the identity function to add one reduction step. Type abstraction and application are reified to their equivalent in System F. Arrow and product η-expansions are reified as computational η-expansions. Arrow η-expansion is $\lambda x\, []\, x$ where we add the two sub-contexts for the argument and body. Product η-expansion is $\langle \mathsf{fst}\, [], \mathsf{snd}\, []\rangle$. Similarly

$$
\begin{array}{rlrl}
\widehat{c} &= x_c\,[] & & \\
\widehat{\Diamond} &= \mathsf{id}\,[] & \widehat{\top} &= \mathsf{top}\,[] \\
\widehat{G_2 \circ G_1} &= \mathsf{id}\,(\widehat{G_2}[\widehat{G_1}]) & \widehat{\bot\tau} &= [][\widehat{\tau}] \\
\widehat{*G} &= \mathsf{id}\,\widehat{G} & \widehat{\Lambda(\alpha \triangleright c : \tau)} &= \Lambda\alpha\,\lambda(x_c : \alpha \to \widehat{\tau})\,[] \\
\widehat{\Lambda\alpha} &= \Lambda\alpha\,[] & \widehat{\cdot[\sigma \triangleright G]} &= [][\widehat{\sigma}]\,(\lambda(x : \widehat{\sigma})\,\widehat{G}[x]) \\
\widehat{\cdot\sigma} &= [][\widehat{\sigma}] & \widehat{\Lambda(\alpha \triangleleft c : \tau)} &= \Lambda\alpha\,\lambda(x_c : \widehat{\tau} \to \alpha)\,[] \\
\widehat{G_1 \xrightarrow{\tau'} G_2} &= \lambda(x : \widehat{\tau'})\,\widehat{G_2}[[]\,(\widehat{G_1}[x])] & \widehat{\cdot[\sigma \triangleleft G]} &= [][\widehat{\sigma}]\,(\lambda(x : \widehat{\tau}[\alpha/\widehat{\sigma}])\,\widehat{G}[x]) \\
\widehat{G_1 \times G_2} &= \langle \widehat{G_1}[\mathsf{fst}\,[]], \widehat{G_2}[\mathsf{snd}\,[]] \rangle & &
\end{array}
$$

Figure 4.16: System F_ι^p coercion reification function

$$
\begin{array}{rlrl}
\widehat{\varnothing} &= \varnothing & \widehat{\Gamma, \alpha} &= \widehat{\Gamma}, \alpha \\
\widehat{\Gamma, (x : \tau)} &= \widehat{\Gamma}, (x : \widehat{\tau}) & \widehat{\Gamma, (c : \tau \triangleright \sigma)} &= \widehat{\Gamma}, (x_c : \widehat{\tau} \to \widehat{\sigma})
\end{array}
$$

Figure 4.17: System F_ι^p environment reification function

we add the sub-contexts for the first and second components. Notice that this is the only coercion where we create more than one hole. This is not a problem since we only need a forward simulation from System F_ι^p to System F.

The top coercion is reified to the hole applied to top, which is a way to keep the hole under hand while the final type is the polymorphic identity type, which we use as the reification for top. The bottom instantiation coercion is reified to System F type instantiation. Upper bounded type abstraction $\Lambda(\alpha \triangleright c : \tau)$ is reified to a type abstraction $\Lambda\alpha$ followed by a function abstraction $\lambda(x_c : \alpha \to \widehat{\tau})$ for the abstract coercion $(c : \alpha \triangleright \tau)$. Conversely, upper bounded type application $\cdot[\sigma \triangleright G]$ is a type application $[\widehat{\sigma}]$ followed by the application of the function reifying the coercion $\lambda(x : \widehat{\sigma})\,\widehat{G}[x]$. To build this function, we close the context with a term abstraction and use the variable to fill the holes of the context. The type annotation of the term abstraction is the type of the hole of $\widehat{G}$ which is unique. Lower bounded abstraction and application are similar.

Finally, environments are reified in Figure 4.17. The empty environment is reified to the empty environment. Term bindings are reified to term bindings, type bindings to type bindings, and coercion bindings $(c : \tau \triangleright \sigma)$ to term bindings $(x_c : \widehat{\tau} \to \widehat{\sigma})$ because coercions are reified to functions.

We write $(J)_s$ for judgments in System F_ι^p, because it is the source language. And we write $(J)_t$ for the judgments of System F, because it is the target judgment. We show that derivations of System F_ι^p are reified to valid derivations of System F.

Lemma 45 (Typing preservation). *The following properties hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\widehat{\Gamma}\ \mathsf{env})_t$ *holds.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\widehat{\Gamma} \vdash \widehat{\tau}\ \mathsf{type})_t$ *holds.*
- *If* $(M \Rightarrow a : \Gamma \vdash \tau)_s$ *holds, then* $(\widehat{M} \Rightarrow \widehat{\Gamma} \vdash a : \widehat{\tau})_t$ *holds.*
- *If* $(G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma)_s$ *and* $(M \Rightarrow \widehat{\Gamma}, \widehat{\Sigma} \vdash a : \widehat{\tau})_t$ *hold, then* $(\widehat{G}[M] \Rightarrow \widehat{\Gamma} \vdash a : \widehat{\sigma})_t$ *holds.*

Proof. By induction. The only interesting assertion is the last one. Reflexivity, transitivity, and coercion variable are simple context transformations. The weakening coercion uses the weakening lemma of System F. Type abstraction and application use their homologous rule.

For rule CoerEtaArr, we first abstract over a fresh term variable x. We then use the induction hypothesis for the body context, which we feed with the application of the hypothesis weakened with the term binding of x, to x under the context of the argument coercion. Rule CoerEtaProd is similar without the weakening part because there is no binding.

Bottom instantiation uses the type instantiation rule. The top coercion uses the typing derivation of top. Upper and lower bounded polymorphic rules are simple uses of type and term abstractions and applications. □

Lemma 46 (Simulation). *If $M \rightsquigarrow_{\beta\iota} N$ holds, then $\widehat{M} \rightsquigarrow^{+} \widehat{N}$ holds.*

Proof. By induction. Rule RedCtx is obvious for all contexts but coercion application. We know that coercions are reified to multi-hole contexts with at least one hole. So we need to repeat the steps of the induction hypothesis as many times as there are holes. Rules RedApp, RedFst, and RedSnd are reified to their analog rules.

Rule RedRefl is reified to $\mathsf{id}\,\widehat{M}$ which reduces to $\widehat{M}$. Without id, it would not have been possible to do a reduction step in System F. Rule RedTrans and RedWeak use the same mechanism. Rule RedPoly reifies to the analog reduction rule.

For rules RedEtaArr and RedEtaProd we reduce the inner redexes, where the hole was. Because both rules reduce under coercion contexts, they may duplicate reduction steps. Rules RedPolyL and RedPolyU use type and term abstraction reduction rules. □

Lemma 47 (Termination). *If $M \Rightarrow a : \Gamma \vdash \tau$ holds, then M strongly normalizes.*

Proof. $\widehat{M}$ is well-typed in System F by Lemma 45. Using System F strong normalization result, we know that reduction of $\widehat{M}$ terminates. If we had an infinite reduction path for M, then by Lemma 46 we would have one for $\widehat{M}$ too, which is a contradiction. □

4.4.3 Confluence

The explicit reduction is locally confluent and thus confluent since reduction terminates.

Lemma 48 (Local confluence). *If $M \rightsquigarrow_{\beta\iota} M_1$ and $M \rightsquigarrow_{\beta\iota} M_2$ hold, then there is a N such that $M_1 \rightsquigarrow^{\star}_{\beta\iota} N$ and $M_2 \rightsquigarrow^{\star}_{\beta\iota} N$ hold.*

Proof. There are no critical pairs. □

Corollary 49 (Confluence). *If $M \rightsquigarrow^{\star}_{\beta\iota} M_1$ and $M \rightsquigarrow^{\star}_{\beta\iota} M_2$ hold, then there is a N such that $M_1 \rightsquigarrow^{\star}_{\beta\iota} N$ and $M_2 \rightsquigarrow^{\star}_{\beta\iota} N$ hold.*

Proof. By Newman's lemma and Lemma 48 and 47. □

4.4.4 Bisimulation

This section makes the link between the λ-calculus reduction and the explicit reduction. The main result is the bisimulation lemma. It helps to show the soundness of the implicit version but has also its own meaning. In a few words, the bisimulation lemma guarantees that typing annotations of the explicit term do not alter or block the reduction of its implicit underlying term. In other words, typing annotations are erasable (see Section 6.1.5 for non-erasability).

In order to show the bisimulation lemma, in particular the backward simulation, we need to show a classification lemma on ι-normal forms. This lemma also shows consistency of coercions. Only the first assertion of the iota classification lemma is used for bisimulation, and only the second assertion is used for consistency. The remaining assertions are used for the induction. We define retyping contexts Q as sequences of coercion applications: $Q ::= [] \mid G\langle Q \rangle$.

Lemma 50 (Iota classification). *If $Q[\lambda(x:\rho)\, M] \Rightarrow \lambda x\, a : \Gamma \vdash \tau$ (resp. $Q[\langle M, N\rangle] \Rightarrow \langle a, b\rangle : \Gamma \vdash \tau$) holds and the term is in ι-normal form, then the following assertions hold.*

- *If τ is $\tau_1 \to \tau_2$ (resp. $\tau_1 \times \tau_2$), then $Q = []$.*
- *τ is not $\tau_1 \times \tau_2$ (resp. $\tau_1 \to \tau_2$).*
- *If τ is $\forall \alpha\, \tau'$, then $Q = \Lambda\alpha\langle Q'\rangle$.*
- *If τ is $\forall(\alpha \triangleright \sigma)\, \tau'$, then $Q = \Lambda(\alpha \triangleright c : \sigma)\langle Q'\rangle$.*
- *If τ is $\forall(\alpha \triangleleft \sigma)\, \tau'$, then $Q = \Lambda(\alpha \triangleleft c : \sigma)\langle Q'\rangle$.*
- *τ is not $\bot$.*
- *For all $\alpha, (c : \alpha \triangleright \sigma)$ in Γ, τ is not α.*

Proof. We detail the arrow case. The product case is similar. We proceed by induction on Q. If Q is a hole, then all assertions hold. Let's proceed by cases on G when Q is of the form $G\langle Q \rangle$.

Reflexivity, transitivity, and weakening contradict the ι-normal form hypothesis. Type abstraction satisfies all assertions. If $Q = \bullet\,\sigma\langle Q'\rangle$, then $Q'[\lambda(x:\rho)\, M]$ is in ι-normal form and has type $\forall\alpha\,\tau'$ for some type τ'. By induction hypothesis $Q' = \Lambda\alpha\langle Q''\rangle$ which contradicts the ι-normal form hypothesis.

For the arrow η-expansion, we have $Q = (G_1 \overset{\tau'}{\to} G_2)\langle Q'\rangle$, so by typing and induction hypothesis we have $Q' = []$. This contradicts the ι-normal form hypothesis. Similarly for product η-expansion. The bottom instantiation is impossible by typing and induction hypothesis. The top coercion satisfies all assertions. Upper and lower bounded abstractions satisfy all assertions, while upper and lower bounded applications break the ι-normal form hypothesis. These cases are similar to polymorphism.

Finally, coercion variables rely on the fact that the environment Γ is well-formed. For upper bounded coercion variables $(c : \alpha \triangleright \sigma)$, there is a contradiction using the induction hypothesis, because $Q = c\langle Q'\rangle$ and Q' cannot end with type α. For lower bounded coercion variables $(c : \sigma \triangleright \alpha)$, all assertions are satisfied. The only interesting assertion is the last one. By the well-formedness of Γ, we know that α is bound just before $(c : \sigma \triangleright \alpha)$. Because α cannot be bound more than once, there is no $\alpha, (c' : \alpha \triangleright \sigma')$ bound in Γ. $\square$

Notice that the lemma does not require the whole term $Q[\lambda(x:\rho)\, M]$ to be ι-irreducible. The proof only needs the retyping context Q and its interaction with the term abstraction to be in ι-normal form. The term M may contain ι-redexes, the proof does not inspect M.

The forward simulation means that β-reduction corresponds to the λ-calculus reduction and that ι-reduction is only static: it does not change the computational behavior of the term. In other words β-steps are steps of the λ-calculus and ι-steps are erasable steps.

Lemma 51 (Forward simulation)**.** *The following assertions hold.*

- *If $M \rightsquigarrow_\beta N$ holds, then $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$ holds.*
- *If $M \rightsquigarrow_\iota N$ holds, then $\lfloor M \rfloor = \lfloor N \rfloor$ holds.*
- *If the well-typed term M is an explicit value, then $\lfloor M \rfloor$ is an implicit value.*

Proof. By induction for the first two assertions. For the last one, it suffices to see that prevalues drop on prevalues and values drop on values with one exception: values coerced with a coercion variable are prevalues. If $\lfloor c\langle v \rangle \rfloor$ is used as an implicit prevalue and $\lfloor v \rfloor$ is not a prevalue (it is either an abstraction or a pair), then by Lemma 50 we have a contradiction. The explicit value is actually of the form $E[Q_1[c\langle Q_2[M]\rangle]]$, where M is a computational constructor as $\lfloor v \rfloor$ is not a prevalue, and E ends with a computational destructor because $\lfloor c\langle v \rangle \rfloor$ is used as a prevalue. We use Lemma 50 with $Q_1[c\langle Q_2[M]\rangle]$. If the computational destructor and constructor correspond, then we are in the first assertion and $Q_1[c\langle Q_2[[]]\rangle]$ is empty. This is impossible because there is at least c. If the computational destructor and constructor do not correspond, then we are in the second assertion. Again, this is impossible. □

Reciprocally, the backward simulation tells that not only β-reduction steps are λ-calculus steps, but that they contain all possible λ-calculus steps modulo some erasable ι-reduction steps. This means that coercions cannot block a λ-calculus redex. For instance, the following term $((\bullet\,\mathsf{Int} \circ \Lambda\alpha)\langle \lambda(x : \alpha)\, x\rangle)\, 49$, which is the polymorphic identity instantiated with Int and applied to 49, drops on $(\lambda x\, x)\, 49$ which reduces to 49. However, in order to have access to this redex in the source term, we need to do two ι-steps: RedTrans and RedPoly.

Lemma 52 (Backward simulation)**.** *If $\lfloor M \rfloor \rightsquigarrow b$ holds and M is well-typed, then there is an N such that $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ and $b = \lfloor N \rfloor$ hold.*

Proof. By Lemma 47 we can assume M in ι-normal form. We proceed by induction. If the reduction rule is RedCtx, we know that M is of the form $G\langle E[M']\rangle$ with $\lfloor M' \rfloor \rightsquigarrow b'$. By induction hypothesis we find N' and use $G\langle E[N']\rangle$. We do rule RedApp in details. Rules RedFst and RedSnd are similar.

We know that $\lfloor M \rfloor = (\lambda x\, a_1)\, a_2$ which reduces on $a_1[x/a_2]$. By inversion of the drop function, we know that M is of the form $E[Q[\lambda(x : \tau_2)\, M_1]\, M_2]$ with $\lfloor M_1 \rfloor = a_1$ and $\lfloor M_2 \rfloor = a_2$. By typing we know that $Q[\lambda(x : \tau_2)\, M_1]$ has type $\tau_2 \to \tau_1$. By Lemma 50 we know that Q is $[]$. Hence M contains a redex the term abstraction is right under the application and reduces to $E[M_1[x/M_2]]$ which drops on $a_1[x/a_2]$ which concludes. □

4.4.5 Soundness

We prove soundness of the explicit and implicit type systems with the usual subject reduction and progress lemmas. However, the subject reduction and progress lemmas of the implicit version are proved using the subject reduction and progress lemmas of the explicit version and the bisimulation lemma. In order to do these syntactical proofs, we need to prove the usual syntactical lemma about weakening and substitution.

The weakening lemma tells that if an object is well-formed (resp. well-typed) under environment Γ, then it is also well-formed (resp. well-typed) under an extended well-formed environment Γ'.

Lemma 53 (Weakening)**.** *If $\Gamma \subseteq \Gamma'$ and Γ' env hold, then the following assertions hold:*

- *If $\Gamma \vdash \tau$ type holds, then $\Gamma' \vdash \tau$ type holds.*
- *If $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ holds and $\mathsf{dom}(\Sigma)$ is disjoint from $\mathsf{dom}(\Gamma')$, then Γ', Σ env and $G \Rightarrow \Gamma' \vdash (\Sigma \vdash \tau) \triangleright \sigma$ hold.*
- *If $M \Rightarrow a : \Gamma \vdash \tau$ holds, then $M \Rightarrow a : \Gamma' \vdash \tau$ holds.*

Proof. By mutual induction. We only give the non-trivial and non-similar cases. For rule TypeVar, if $\alpha \in \Gamma$ then $\alpha \in \Gamma'$ too. For rule TypeForL, because α could have been renamed we have that environment Γ', α is well-formed and by induction hypothesis that $\Gamma', \alpha \vdash \tau$ type holds. We deduce that $\Gamma', \alpha, (c : \tau \triangleright \alpha)$ holds and by induction hypothesis that $\Gamma', \alpha, (c : \tau \triangleright \alpha) \vdash \rho$ type holds. For rule TermCoer, we may rename the variables bound in Σ in order to avoid the domain of Γ'. We first use the induction hypothesis on the coercion since we need Γ', Σ env to use the induction hypothesis on the term judgment. □

If a type σ is well formed under environment Γ and τ is well-formed under $\Gamma, \alpha, C, \Gamma'$ (where C is a potential coercion binding), then $\tau[\alpha/\sigma]$ is well-formed under $\Gamma, \Gamma'[\alpha/\sigma]$. This lemma is called type substitution. The coercion binding C is maximally inserted, which means Γ' cannot start with a coercion binding. Notice that we don't ask for $\Gamma \vdash C[\alpha/\sigma]$ to hold because coercion variables are not used in type well-formedness judgment.

Lemma 54 (Type substitution)**.** *If $\Gamma \vdash \sigma$ type holds, then the following assertions hold.*

- *If $\Gamma, \alpha, C, \Gamma'$ env holds, then $\Gamma, \Gamma'[\alpha/\sigma]$ env holds.*
- *If $\Gamma, \alpha, C, \Gamma' \vdash \tau$ type holds, then $\Gamma, \Gamma'[\alpha/\sigma] \vdash \tau[\alpha/\sigma]$ type holds.*

Proof. By induction. The only interesting rule is TypeVar. There are two cases depending on whether the type variable to instantiate is the same as the one of the judgment. If we have $\Gamma, \alpha, \Gamma' \vdash \alpha$ type then we have to show that $\Gamma, \Gamma'[\alpha/\sigma] \vdash \sigma$ type holds which we do by Lemma 53 and induction hypothesis. If we have $\Gamma, \alpha, \Gamma' \vdash \beta$ type for $\beta \neq \alpha$, then by induction hypothesis we have $\Gamma, \Gamma'[\alpha/\sigma] \vdash \beta$ type. □

The extraction lemma tells when the sub-judgments of a well-formed judgment are also well-formed. From a type derivation we can extract that the environment is well-formed. From a coercion derivation, using the hypothesis that its left type is well-formed, we can extract the well-formedness of its right type. Finally, from a term derivation we can extract that its type is well-formed under its environment, and hence that the environment itself is well-formed.

Lemma 55 (Extraction)**.** *The following assertions hold.*

- *If $\Gamma \vdash \tau$ type holds, then Γ env holds.*
- *If $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ and $\Gamma, \Sigma \vdash \tau$ type hold, then $\Gamma \vdash \sigma$ type holds.*
- *If $a : \Gamma \vdash \tau$ holds, then $\Gamma \vdash \tau$ type holds.*

Proof. By mutual induction. We detail the cases where the conclusion is not in hypothesis (like rule TypeVar), nor obtained by inversion of the induction hypothesis (like rule TypeFor). For rule CoerVar, we have that Γ is well-formed by hypothesis from which we extract by Lemma 53 that σ is well-formed under it. For rule CoerWeak we use Lemma 53 to call the induction hypothesis. For rules CoerTApp, CoerTAppL, and CoerTAppU, we use Lemma 54. $\square$

Term substitution tells that the substitution $a[x/b]$ has type σ under environment Γ if the argument b has type τ under environment Γ and the body a has type σ under environment Γ extended with x associated to τ.

Lemma 56 (Term substitution)**.** *If $N \Rightarrow b : \Gamma \vdash \tau$ hold, then the following assertions hold.*

- *If $\Gamma, (x : \tau), \Gamma'$ env holds, then Γ, Γ' env holds.*
- *If $\Gamma, (x : \tau), \Gamma' \vdash \sigma$ type holds, then $\Gamma, \Gamma' \vdash \sigma$ type holds.*
- *If $G \Rightarrow \Gamma, (x : \tau), \Gamma' \vdash (\Sigma \vdash \sigma) \triangleright \sigma'$ holds, then $G \Rightarrow \Gamma, \Gamma' \vdash (\Sigma \vdash \sigma) \triangleright \sigma'$ holds.*
- *If $M \Rightarrow a : \Gamma, (x : \tau), \Gamma' \vdash \sigma$ holds, then $M[x/N] \Rightarrow a[x/b] : \Gamma, \Gamma' \vdash \sigma$ holds.*

Proof. By mutual induction. For rule TermVar, we use Lemma 53 when the term variables correspond. All other rules use induction hypotheses. $\square$

If a type σ and G' are well formed under environment Γ, then for all well-formed judgments under $\Gamma, \alpha, (c : \tau_1 \triangleright \tau_2), \Gamma'$ (where $(c : \tau_1 \triangleright \tau_2)$ is a potential coercion binding), the same judgment after the type substitution $[\alpha/\sigma]$ and coercion substitution $[c/G']$ is well-formed under $\Gamma, \Gamma'[\alpha/\sigma]$. This lemma is called bounded type substitution. The coercion binding $(c : \tau_1 \triangleright \tau_2)$ is maximally inserted, which means that Γ' cannot start with a coercion binding.

Lemma 57 (Bounded type substitution)**.** *If $\Gamma \vdash \sigma$ type and $G' \Rightarrow \Gamma \vdash \tau_1[\alpha/\sigma] \triangleright \tau_2[\alpha/\sigma]$ hold, then the following assertions hold.*

- *If $G \Rightarrow \Gamma, \alpha, (c : \tau_1 \triangleright \tau_2), \Gamma' \vdash (\Sigma \vdash \tau) \triangleright \tau'$ holds, then $G[\alpha/\sigma][c/G'] \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash (\Sigma[\alpha/\sigma] \vdash \tau[\alpha/\sigma]) \triangleright \tau'[\alpha/\sigma]$ holds.*
- *If $M \Rightarrow a : \Gamma, \alpha, (c : \tau_1 \triangleright \tau_2), \Gamma' \vdash \tau$ holds, then $M[\alpha/\sigma][c/G'] \Rightarrow a : \Gamma, \Gamma'[\alpha/\sigma] \vdash \tau[\alpha/\sigma]$ holds.*

Proof. By induction using Lemma 54 for types and environments. The only interesting rule is CoerVar. There are two cases depending whether the coercion variable to instantiate is the same as the one of the judgment. If we have $c \Rightarrow \Gamma, \alpha, (c : \tau_1 \triangleright \tau_2), \Gamma' \vdash \tau_1 \triangleright \tau_2$ then we have to show that $G' \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash \tau_1[\alpha/\sigma] \triangleright \tau_2[\alpha/\sigma]$ holds which we do by Lemma 53. If we have $c' \Rightarrow \Gamma, \alpha, (c : \tau_1 \triangleright \tau_2), \Gamma' \vdash \tau_1' \triangleright \tau_2'$ then we have to show $c' \Rightarrow \Gamma, \Gamma'[\alpha/\sigma] \vdash \tau_1' \triangleright \tau_2'$ which holds. $\square$

There are two soundness lemma: one for the explicit version and one for the implicit version. Both are done using a subject reduction and progress lemma. Subject reduction tells that if an term M is well-typed and reduces to N, then N is also well-typed with the same type and environment.

Lemma 58 (Explicit subject reduction)**.** *If $M \Rightarrow a : \Gamma \vdash \tau$ and $M \rightsquigarrow_{\beta\iota} N$ hold, then $N \Rightarrow b : \Gamma \vdash \tau$ holds where $b = \lfloor N \rfloor$.*

Proof. By induction on $M \rightsquigarrow_{\beta\iota} N$. Rule RedCtx is done by cases on the context E. Rule RedApp uses Lemma 56 and the fact that the drop function commutes with substitution. Rule RedWeak uses Lemma 53 to show that M also has type τ under the well-formed extended environment Γ, Σ. Rules RedPoly, RedPolyL, and RedPolyU use Lemma 57.

For rule RedEtaArr, we have $G_1 \Rightarrow \Gamma, \Sigma \vdash \tau' \triangleright \tau$ and $G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \sigma) \triangleright \sigma'$ for the coercions and $M \Rightarrow a : \Gamma, \Sigma, (x : \tau) \vdash \sigma$ for the term. We rename x into a fresh term variable y in the right-hand side of the reduction. By Lemma 53 we have $G_2 \Rightarrow \Gamma, (y : \tau') \vdash (\Sigma \vdash \sigma) \triangleright \sigma'$. It remains to show that $M[x/G_1\langle y \rangle] \Rightarrow a[x/y] : \Gamma, (y : \tau'), \Sigma \vdash \sigma$ holds. In order to use Lemma 56, we need to show two subgoals. We first use Lemma 53 to show $M \Rightarrow a : \Gamma, (y : \tau'), \Sigma, (x : \tau) \vdash \sigma$. We then show $G_1\langle y \rangle \Rightarrow y : \Gamma, (y : \tau'), \Sigma \vdash \tau$ by Lemma 53 and rules TermCoer and TermVar. Rules RedEtaProd is similar. ☐

We classify values according to their types in the following lemma. This lemma, called explicit classification, is used by the progress lemma, which follows. By looking at the type of a value, we get to know its form.

Lemma 59 (Explicit classification)**.** *If $v \Rightarrow a : \Gamma \vdash \tau$ holds, then either v is a prevalue or the following assertions hold.*

- *If τ is $\tau_1 \to \tau_2$, then v is $\lambda(x : \tau_1)\, v'$.*
- *If τ is $\tau_1 \times \tau_2$, then v is $\langle v_1, v_2 \rangle$.*
- *If τ is $\forall \alpha\, \tau'$, then v is $\Lambda \alpha \langle v' \rangle$.*
- *If τ is $\forall (\alpha \triangleright \sigma)\, \tau'$, then v is $\Lambda(\alpha \triangleright c : \sigma)\langle v' \rangle$.*
- *If τ is $\forall (\alpha \triangleleft \sigma)\, \tau'$, then v is $\Lambda(\alpha \triangleleft c : \sigma)\langle v' \rangle$.*
- *τ is not $\bot$.*

Proof. By induction on v and inversion of $v \Rightarrow a : \Gamma \vdash \tau$. ☐

The progress lemma tells that well-typed terms are either values or reducible. We make no distinction whether it is a β- or ι-reduction step.

Lemma 60 (Explicit progress)**.** *If $M \Rightarrow a : \Gamma \vdash \tau$ holds, then M is a value or there is N such that $M \rightsquigarrow_{\beta\iota} N$ holds.*

Proof. Let's assume that M is not a value. By induction on $M \Rightarrow a : \Gamma \vdash \tau$. Because of strong reduction, we can assume that all sub-terms of M are values, otherwise, M reduces using rule RedCtx. Rules TermVar, TermLam, and TermPair are values. Rules TermApp, TermFst, and TermSnd use Lemma 59 to reduce by rules RedApp, RedFst, and RedSnd, respectively.

For rule TermCoer, we proceed by cases on $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$. Rules CoerRefl, CoerTrans, and CoerWeak reduce by rules RedRefl, RedTrans, and RedWeak. Rules CoerVar, CoerTLam, CoerTop, CoerTLamL, and CoerTLamU are values. Rules CoerTApp, CoerEtaArr, CoerEtaProd, CoerTAppL, and CoerTAppU use Lemma 59 to reduce by rules RedPoly, RedEtaArr, RedEtaProd, RedPolyL, and RedPolyU, respectively.

The remaining rule is CoerBot which we refute with Lemma 59: there is no value of type $\bot$. ☐

Subject reduction and progress together proves the soundness lemma. An explicit term is sound if it reduces to a value.

Proposition 61 (Explicit soundness)**.** *If $M \Rightarrow a : \Gamma \vdash \tau$ holds, then M terminates to a value.*

Proof. By Lemma 47 and 49, M terminates to N. By Lemma 58 we have $N \Rightarrow \lfloor N \rfloor : \Gamma \vdash \tau$. And by Lemma 60 we have that N is a value. □

The same schema for soundness works for the implicit version. We prove subject reduction and progress for the implicit version from the explicit version using the bisimulation property.

Lemma 62 (Implicit subject reduction)**.** *If $a : \Gamma \vdash \tau$ and $a \rightsquigarrow b$ hold, then $b : \Gamma \vdash \tau$ holds.*

Proof. By Lemma 44 we have $M \Rightarrow a : \Gamma \vdash \tau$ for some M. By Lemma 52 we have $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ such that $\lfloor N \rfloor = b$. By Lemma 58 we have $N \Rightarrow b : \Gamma \vdash \tau$. We conclude with Lemma 44. □

The progress lemma for the implicit version of the type system resemble its explicit version. However, the proof may be short-cut using the bisimulation property and the explicit progress lemma.

Lemma 63 (Implicit Progress)**.** *If $a : \Gamma \vdash \tau$ holds, then a is a value or there is a term b such that $a \rightsquigarrow b$ holds.*

Proof. Let's assume a is not a value. By Lemma 44 we have $M \Rightarrow a : \Gamma \vdash \tau$ for some M. We proceed by induction on the strong normalization of M (Lemma 47). If M is a value, then a is a value by Lemma 51 and we are done. If M is not a value, then by Lemma 60 we have $M \rightsquigarrow_{\beta\iota} N$. If the reduction is a ι-step, we call the induction hypothesis since $\lfloor N \rfloor = \lfloor M \rfloor = a$. If the reduction is a β-step, then by Lemma 51, we have $a \rightsquigarrow \lfloor N \rfloor$. □

Now that we have shown explicit subject reduction, we can show the strong normalization of the implicit reduction. This result is useful to prove the soundness of the implicit language.

Proposition 64 (Implicit termination)**.** *If $a : \Gamma \vdash \tau$ holds, then a strongly normalizes.*

Proof. By Lemma 44 we have $M \Rightarrow a : \Gamma \vdash \tau$ with $\lfloor M \rfloor = a$. We proceed by induction on the strong normalization of M. If $a \rightsquigarrow b$, by Lemma 52 we have $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ with $\lfloor N \rfloor = b$. By Lemma 58, we have $N \Rightarrow x : \Gamma \vdash \tau$, which allows us to call the induction hypothesis to conclude. □

Similarly to the explicit soundness lemma, the implicit soundness lemma tells that well-typed lambda-terms strongly normalize to a value.

Proposition 65 (Implicit soundness)**.** *If $a : \Gamma \vdash \tau$ holds, then a terminates to a value.*

Proof. By Lemma 64, a terminates to b. By Lemma 62 we have $b : \Gamma \vdash \tau$. And by Lemma 63 we have that b is a value. □

4.5 Expressivity

To prove that a type system T_2 is more expressive than a type system T_1, we need to show that if a term a is well-typed in T_1 then it is also well-typed in T_2. In other words, there are more well-typed terms in the most expressive type-system than the less expressive one. Keeping in mind that well-typed terms are sound and that type systems are conservatives, an expressive type-system rejects fewer sound terms.

It is sufficient to show the inclusion between the implicit versions since explicit and implicit versions are equivalent. This simplifies the notations, but the proofs are mainly the same.

4.5.1 System F

For each judgment of System F, we show that the equivalent judgment in System F_ι^p also holds. Actually, we only need the polymorphism feature on top of the base system to include System F. And reciprocally, the base system extended with polymorphism is included into System F. As a consequence System F is equivalent to polymorphism, which is what is expected. We write $(J)_s$ for judgments in System F (as source language) and $(J)_t$ for those in the base system extended with the polymorphism extension (as target language).

Lemma 66. *The following assertions hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\Gamma\ \mathsf{env})_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\Gamma \vdash \tau\ \mathsf{type})_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash a : \tau)_s$ *holds, then* $(a : \Gamma \vdash \tau)_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma)_t$ *and* $(\Gamma, \Sigma \vdash a : \tau)_s$ *hold, then* $(\Gamma \vdash a : \sigma)_s$ *holds.*

Proof. First, we prove the implication from source to target and then we prove the reciprocal from target to source. Notice that the last assertion only occurs for the reciprocal. For both directions we proceed by mutual induction on all judgments.

For the implication, environment and type well-formedness rules are trivial since they are exactly the same. The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. The two interesting cases are rules TermGen and TermInst.

For rule TermGen, we have $(\Gamma, \alpha \vdash a : \tau)_s$ and we need to prove $(a : \Gamma \vdash \forall\alpha\,\tau)_t$. By induction hypothesis we have $(a : \Gamma, \alpha \vdash \tau)_t$. We use rule TermCoer on the preceding derivation. It remains to prove $(\Gamma \vdash (\alpha \vdash \tau) \triangleright \forall\alpha\,\tau)_t$ which we do with rule CoerTLam.

For rule TermInst, we have $(\Gamma \vdash a : \forall\alpha\,\tau)_s$ and $(\Gamma \vdash \sigma\ \mathsf{type})_s$ and we need to prove $(a : \Gamma \vdash \tau[\alpha/\sigma])_t$. By induction hypothesis we have $(a : \Gamma \vdash \forall\alpha\,\tau)_t$ and $(\Gamma \vdash \sigma\ \mathsf{type})_t$. We use rule TermCoer on the preceding term derivation. It remains to prove $(\Gamma \vdash \forall\alpha\,\tau \triangleright \tau[\alpha/\sigma])_t$ which we do with rule CoerTApp.

Let's now prove the reciprocal. Environment and type well-formedness rules are trivial again since they are exactly the same (notice that we only consider the base system extended with the polymorphism extension). The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. For rule TermCoer, we use the last assertion on the induction hypothesis.

It remains to prove the last assertion. Rule CoerRefl is done by hypothesis. Rule CoerTrans is done by composition. Rule CoerVar cannot occur since by extraction we have

$(\Gamma\ \mathsf{env})_s$ which implies that the environment Γ do not contain any coercion variables. Rule CoerWeak uses the weakening lemma. Finally the two interesting cases are rules CoerTLam and CoerTApp.

For rule CoerTLam we use rule TermGen and for rule CoerTApp we use rule TermInst. □

4.5.2 System F_η

For each judgment of System F_η, we show that the equivalent judgment in System F_ι^p also holds. Actually, we only need the polymorphism and η-expansion features on top of the base system to include F_η. And reciprocally, the base system extended with polymorphism and η-expansion is included into F_η. As a consequence F_η is equivalent to polymorphism and η-expansion, or equivalently extends System F with η-expansion as its name suggests. We write $(J)_s$ for judgments in System F_η and $(J)_t$ for those in the base system extended with the polymorphism and the η-expansion extensions. We define $\forall\Sigma\,\tau$ inductively on Σ.

$$\forall\varnothing\,\tau \stackrel{\mathsf{def}}{=} \tau$$
$$\forall(\Sigma,\alpha)\,\tau \stackrel{\mathsf{def}}{=} \forall\Sigma\,(\forall\alpha\,\tau)$$

Lemma 67. *The following assertions hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\Gamma\ \mathsf{env})_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\Gamma \vdash \tau\ \mathsf{type})_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash a : \tau)_s$ *holds, then* $(a : \Gamma \vdash \tau)_t$ *holds, and reciprocally.*
- *If* $(\Gamma \vdash \tau \rhd \sigma)_s$ *holds, then* $(\Gamma \vdash \tau \rhd \sigma)_t$ *holds.*
- *If* $(\Gamma \vdash (\Sigma \vdash \tau) \rhd \sigma)_t$ *holds, then* $(\Gamma \vdash \forall\Sigma\,\tau \rhd \sigma)_s$ *holds.*

Proof. First we prove the implication from source to target and then we prove the reciprocal from target to source. Notice that the last assertion only occurs for the reciprocal. For both directions we proceed by mutual induction on all judgments.

For the implication, environment and type well-formedness rules are trivial since they are exactly the same. The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. The next two cases, namely rules TermGen and TermInst, are similar to the proof for System F inclusion. Finally, for rule TermCont, we use rule TermCoer.

It remains to show the implication for the containment judgment. Rule ContRefl uses rule CoerRefl and rule ContTrans uses rule CoerTrans and weakening. Rule ContTLam uses rule CoerTLam. Rule ContTApp uses rule CoerTApp. Rule ContArr uses rule CoerEtaArr and weakening. Rule ContProd uses rule CoerEtaProd.

For rule ContCongr, we have $(\Gamma,\alpha \vdash \tau \rhd \sigma)_t$ by induction hypothesis and we show $(\Gamma \vdash \forall\alpha\,\tau \rhd \forall\alpha\,\sigma)_t$. At first we apply rule CoerWeak which leaves us to show $(\Gamma \vdash (\alpha \vdash \forall\alpha\,\tau) \rhd \forall\alpha\,\sigma)_t$. We apply a first time rule CoerTrans with rule CoerTLam which leaves us with $(\Gamma,\alpha \vdash \forall\alpha\,\tau \rhd \sigma)_t$ to prove. We apply a second time rule CoerTrans with the induction hypothesis this time. This leaves us $(\Gamma,\alpha \vdash \forall\alpha\,\tau \rhd \tau)_t$ to prove, for which we use rule CoerTApp with the type argument α.

For rule ContDistArr, we take α fresh. After the use of rule CoerWeak, it remains to show that $(\Gamma \vdash (\alpha \vdash \forall\alpha\,\tau \to \sigma) \triangleright \tau \to \forall\alpha\,\sigma)_t$ holds. We apply rule CoerTrans on $(\Gamma \vdash (\alpha \vdash \tau \to \sigma) \triangleright \tau \to \forall\alpha\,\sigma)_t$ and $(\Gamma, \alpha \vdash \forall\alpha\,\tau \to \sigma \triangleright \tau \to \sigma)_t$. We prove the last one with rule CoerTApp with type argument α. We prove the first one with rule CoerEtaArr, which gives one goal $(\Gamma, \alpha \vdash \tau \triangleright \tau)_t$ for the argument type and $(\Gamma \vdash (\alpha \vdash \sigma) \triangleright \forall\alpha\,\sigma)_t$ for the return type. We prove the first one with reflexivity and the second one with rule CoerTLam. The coercion witness for dist in System F_ι^p is thus $*((\Diamond \xrightarrow{\tau} \Lambda\alpha) \circ \bullet\,\alpha)$.

Rule ContDistProd is similar. After rule CoerWeak, we show $(\Gamma \vdash (\alpha \vdash \forall\alpha\,\tau \times \sigma) \triangleright \forall\alpha\,\tau \times \forall\alpha\,\sigma)_t$ with α a fresh type variable. We first instantiate with α then use the product η-expansion and type abstraction twice.

Let's now prove the reciprocal. Environment and type well-formedness rules are trivial again since they are exactly the same. The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. For rule TermCoer, we use the last assertion on the induction hypothesis.

It remains to prove the last assertion. Rule CoerRefl is done with rule ContRefl. For rule CoerTrans, we use rule ContTrans on $(\Gamma \vdash \forall\Sigma_2, \Sigma_1\,\tau_1 \triangleright \forall\Sigma_2\,\tau_2)_s$ and $(\Gamma \vdash \forall\Sigma_2\,\tau_2 \triangleright \tau_3)_s$. The last one holds by induction hypothesis. For the first one we use rule ContCongr repeatedly to get $(\Gamma, \Sigma_2 \vdash \forall\Sigma_1\,\tau_1 \triangleright \tau_2)_s$ which holds by induction hypothesis. Rule CoerVar cannot occur since by extraction we have $(\Gamma\ \mathsf{env})_s$ which implies that the environment Γ does not contain any coercion variables. For rule CoerWeak we use repeatedly a combination of rules ContTrans, ContCongr, and ContTLam to show $(\Gamma \vdash \tau \triangleright \forall\Sigma\,\tau)_s$. Rule CoerTLam uses rule ContRefl. CoerTApp uses rule ContTApp. The last remaining rules are CoerEtaArr and CoerEtaProd.

For rule CoerEtaArr, we use rule ContTrans on $(\Gamma \vdash \forall\Sigma\,(\tau \to \sigma) \triangleright \forall\Sigma\,(\tau' \to \sigma))_s$ and $(\Gamma \vdash \forall\Sigma\,(\tau' \to \sigma) \triangleright \tau' \to \sigma')_s$. For the first containment we use rule ContCongr repeatedly to get $(\Gamma, \Sigma \vdash \tau \to \sigma \triangleright \tau' \to \sigma)_s$ which we show with rule ContArr and the induction hypothesis. For the second containment we use rule ContTrans on $(\Gamma \vdash \forall\Sigma\,(\tau' \to \sigma) \triangleright \tau' \to \forall\Sigma\,\sigma)_s$ and $(\Gamma \vdash \tau' \to \forall\Sigma\,\sigma \triangleright \tau' \to \sigma')_s$. For the first containment we use rules ContTrans, ContCongr, and ContDistArr. For the second containment we use rules ContArr with the induction hypothesis. Rule CoerEtaProd is similar. □

4.5.3 MLF

For each judgment of MLF, we show that the equivalent judgment in System F_ι^p also holds. Actually, we only need the lower bounded polymorphism and bottom type feature on top of the base system to include MLF. And the reciprocal is almost true: MLF would include the base system extended with lower bounded polymorphism and bottom type, if recursive bounds were present in MLF. In other words, MLF is equivalent to the base system extended with non-recursive lower bounded polymorphism and the bottom type. We write $(J)_s$ for judgments in MLF and $(J)_t$ for those in the base system extended with lower bounded polymorphism and bottom type.

Lemma 68. *The following assertions hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\Gamma\ \mathsf{env})_t$ *holds.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\Gamma \vdash \tau\ \mathsf{type})_t$ *holds.*
- *If* $(\Gamma \vdash \tau \triangleright \sigma)_s$ *holds, then* $(\Gamma \vdash \tau \triangleright \sigma)_t$ *holds.*

- *If* $(\Gamma \vdash a : \tau)_s$ *holds, then* $(a : \Gamma \vdash \tau)_t$ *holds.*

Proof. We proceed by mutual induction on all judgments. Environment and type well-formedness rules are trivial since they are exactly the same. The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. Rule TermTAbs is a conjunction of rules CoerTLamL and TermCoer. Rule TermTApp uses rule TermCoer.

For the instantiation judgment, rule InstBot uses rule CoerBot. Rule InstId is rule CoerRefl. And rule InstComp is rule CoerTrans. Rule InstAbstr is rule CoerVar. For rule InstIntro, we use rule CoerWeak after rule CoerTLamL since the abstract type variable is not used. Rule InstElim uses rule CoerTAppL with the bound for the argument type and the reflexivity coercion (rule CoerRefl) for the argument coercion.

For rule InstUnder, we use rule CoerWeak on $\Gamma \vdash (\alpha, (\tau \rhd \alpha) \vdash \forall(\alpha \lhd \tau)\, \sigma_1) \rhd \forall(\alpha \lhd \tau)\, \sigma_2$, which we prove by using rule CoerTrans twice. First, we prove $\Gamma, \alpha, (\tau \rhd \alpha) \vdash \forall(\alpha \lhd \tau)\, \sigma_1 \rhd \sigma_1$ by rule CoerTAppL with α as the type argument and c as the coercion argument. Then, we prove $\Gamma, \alpha, (\tau \rhd \alpha) \vdash \sigma_1 \rhd \sigma_2$ by induction hypothesis. And we finally prove $\Gamma \vdash (\alpha, (\tau \rhd \alpha) \vdash \sigma_2) \rhd \forall(\alpha \lhd \tau)\, \sigma_2$ by rule CoerTLamL.

For rule InstInside, we use rule CoerWeak on $\Gamma \vdash (\alpha, (\tau_2 \rhd \alpha) \vdash \forall(\alpha \lhd \tau_1)\, \sigma) \rhd \forall(\alpha \lhd \tau_2)\, \sigma$, which we prove by using rule CoerTrans. First we prove $\Gamma, \alpha, (\tau_2 \rhd \alpha) \vdash \forall(\alpha \lhd \tau_1)\, \sigma \rhd \sigma$ by rule CoerTAppL with α as the type argument. For the coercion argument $\Gamma, \alpha, (\tau_2 \rhd \alpha) \vdash \tau_1 \rhd \alpha$, we use rule CoerTrans with the induction hypothesis and c. Then we prove $\Gamma \vdash (\alpha, (\tau_2 \rhd \alpha) \vdash \sigma) \rhd \forall(\alpha \lhd \tau_2)\, \sigma$ by rule CoerTLamL. □

We conclude from this inclusion, that MLF is sound and strongly normalizing. The proof of strong normalization of MLF is actually one of the contribution of this thesis.

4.5.4 System $\mathsf{F}_{<:}$

We remind that the version of System $\mathsf{F}_{<:}$ we describe in Section 3.6 corresponds to the most expressive version of System $\mathsf{F}_{<:}$, namely System F-bounded [4]. For each judgment of System $\mathsf{F}_{<:}$, we show that the equivalent judgment in System F_ι^p also holds. Actually, we only need the upper bounded polymorphism, the top type, and η-expansion features on top of the base system to include $\mathsf{F}_{<:}$. However, unlike for System F, System F_η, and MLF, the reciprocal is not true. The base system extended with upper bounded polymorphism, top, and η-expansion is strictly more expressive than System $\mathsf{F}_{<:}$. We explain why after the proof. We write $(J)_s$ for judgments in System $\mathsf{F}_{<:}$ and $(J)_t$ for those in the base system extended with upper bounded polymorphism, the top type, and η-expansion.

Lemma 69. *The following assertions hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\Gamma\ \mathsf{env})_t$ *holds.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\Gamma \vdash \tau\ \mathsf{type})_t$ *holds.*
- *If* $(\Gamma \vdash \tau \rhd \sigma)_s$ *holds, then* $(\Gamma \vdash \tau \rhd \sigma)_t$ *holds.*
- *If* $(\Gamma \vdash a : \tau)_s$ *holds, then* $(a : \Gamma \vdash \tau)_t$ *holds.*

Proof. We proceed by mutual induction on all judgments. Environment and type well-formedness rules are trivial since they are exactly the same. The same argument holds for the first six term judgment rules (those of the STLC): TermVar, TermLam, TermApp, TermPair, TermFst, and TermSnd. Rule TermGen is a conjunction of rules CoerTLamU and TermCoer. Rule TermInst is a conjunction of rules CoerTAppU and TermCoer. Rule TermSub uses rule TermCoer.

For the subtyping judgment, rules SubRefl, SubTrans, SubVar, SubTop, SubArr, and SubProd use rules CoerRefl, CoerTrans, CoerVar, CoerTop, CoerEtaArr, and CoerEtaProd respectively. For rule SubCongr, we use rule CoerWeak on $\Gamma \vdash (\alpha, (\alpha \triangleright \tau') \vdash \forall(\alpha \triangleright \tau)\, \sigma) \triangleright \forall(\alpha \triangleright \tau')\, \sigma'$ which we prove with two consecutive uses of rule CoerTrans. We first prove $\Gamma, \alpha, (\alpha \triangleright \tau') \vdash \forall(\alpha \triangleright \tau)\, \sigma \triangleright \sigma$ with rule CoerTAppU with α for the type argument and the induction hypothesis $\Gamma, \alpha, (\alpha \triangleright \tau') \vdash \alpha \triangleright \tau$ for the coercion argument. Then we prove $\Gamma, \alpha, (\alpha \triangleright \tau') \vdash \sigma \triangleright \sigma'$ with the second induction hypothesis. And we finally prove $\Gamma \vdash (\alpha, (\alpha \triangleright \tau') \vdash \sigma') \triangleright \forall(\alpha \triangleright \tau')\, \sigma'$ with rule CoerTLamU. □

The reciprocal does not hold because System $\mathsf{F}_{<:}$ misses a distributivity rule like we can find in System F_η. Distributivity naturally comes from polymorphism and η-expansion when expressed as coercions. System F_η and System $\mathsf{F}_{<:}$ resemble each other. They are composed of two type system features: η-expansion and some sort of polymorphism. However, they do not handle polymorphism in the same manner. In System F_η, polymorphism is in the coercion judgment. While in System $\mathsf{F}_{<:}$, (upper bounded) polymorphism is only available in the term judgment and thus not composable with the other type system features, in our case η-expansion (called subtyping in System $\mathsf{F}_{<:}$). More concretely, upper bounded instantiation is not a subtyping rule, but only a typing rule. While in System F_η, type instantiation is a containment rule.

The distributivity rule is not the only rule derivable in System $\mathsf{F}_{<:}$ if upper bounded polymorphism was in the coercion judgment. The congruence rule for upper bounded polymorphism would also be derivable. A general statement about erasable type system features (like polymorphism, recursive types, coercion abstraction) is that they come with an introduction and elimination coercion rule, from which we derive their congruence rule and distributivity rule for those about abstraction.

4.5.5 Summary

We can summarize all these inclusions and equivalences into the following table.

Type system features	F	F_η	MLF	$\mathsf{F}_{<:}$	F_ι^p
Polymorphism	$\checkmark$	$\checkmark$	-	-	$\checkmark$
Eta-expansion	-	$\checkmark$	-	$\checkmark$	$\checkmark$
Bottom	-	-	$\checkmark$	-	$\checkmark$
Top	-	-	-	$\checkmark$	$\checkmark$
Lower bounded polymorphism	-	-	$\checkmark$	-	$\checkmark$
Upper bounded polymorphism	-	-	-	$\simeq$	$\checkmark$

System F is equivalent to the base system extended with polymorphism. System F_η is equivalent to the base system extended with polymorphism and η-expansion. MLF is equivalent to the base system extended with the bottom type and lower bounded polymorphism.

Actually, the equivalence is almost true, but for recursive bounds. System $\mathsf{F}_{<:}$ is included in the base system extended with η-expansion, the top type, and upper bounded polymorphism. System $\mathsf{F}_{<:}$, which corresponds to the most expressive version of System $\mathsf{F}_{<:}$ [4], is in fact strictly included into this extension because the distributivity rule is derivable in the extended system but not in System $\mathsf{F}_{<:}$. In this sense, System $\mathsf{F}_{<:}$ is not complete according to the features it presents, whereas System F and System F_η are complete with respect to their features. The reason why the distributivity rule is not derivable is that the upper bounded polymorphism feature is not a subtyping (or coercion) rule in $\mathsf{F}_{<:}$. It is only a typing rule at the term level. The subtyping judgment only contains the congruence rule for upper bounded polymorphism. Finally, System F_ι^p contains all features and thus includes all presented type systems.

4.6 Beyond parametric coercion abstraction

In this chapter, we have defined a coercion type system where all features are expressed as coercions and thus fully composable and erasable. We called this coercion type system System F_ι^p. We defined it as a base system corresponding to the STLC and a series of orthogonal and composable features. We showed that the STLC, System F, System F_η, MLF, and System $\mathsf{F}_{<:}$ can be seen as the base system with some additional features. Notice however that System $\mathsf{F}_{<:}$ is not as expressive as it naturally gets in this approach. As a corollary, all these type systems are included in System F_ι^p which contains all features.

However, System F_ι^p is not as expressive as we might want. It has some restrictions and as a consequence it does not contain $\mathsf{Constraint\ ML}$, even though polymorphism is not even first class in $\mathsf{Constraint\ ML}$. The restriction in cause is coercion abstraction. As a matter of fact, upper and lower bounded polymorphism are forms of coercion abstraction, but they have the particularity that they only abstract over parametric coercions: coercions which are either from an abstract type or to an abstract type. In order to describe the feature of $\mathsf{Constraint\ ML}$ we would need an unrestricted coercion abstraction of the form $\Lambda(c : \tau \triangleright \sigma)$. We discuss this point in Section 4.6.1. Conclusions of this section leads us to a discussion about push in Section 4.6.2.

4.6.1 Unrestricted coercion abstraction

We could define a type for coercion polymorphism written $(\tau \triangleright \sigma) \Rightarrow \rho$ abstracting over the coercion $\tau \triangleright \sigma$ with body ρ. We would then naturally expect the following associated typing rules:

$$\textsc{CoerCLam}\qquad \Lambda(c : \tau \triangleright \sigma) \Rightarrow \Gamma \vdash ((c : \tau \triangleright \sigma) \vdash \rho) \triangleright (\tau \triangleright \sigma) \Rightarrow \rho$$

$$\textsc{CoerCApp}\qquad \frac{G \Rightarrow \Gamma \vdash \tau \triangleright \sigma}{\cdot G \Rightarrow \Gamma \vdash (\tau \triangleright \sigma) \Rightarrow \rho \triangleright \rho}$$

The problem is that in doing so (without any additional restriction) we break backward simulation. To see why, let's consider the following explicit term where c is a coercion variable: $(c\langle\lambda(x : \tau)\, M\rangle)\, N$. It drops to $(\lambda x\, \lfloor M \rfloor)\, \lfloor N \rfloor$ which is a redex, and thus can reduce. However, without allowing decomposition of abstract coercions (called push and described in Section 4.6.2), the explicit term is stuck and cannot do the same step as its dropped lambda term.

With push, backward simulation seems to be restored, but this breaks implicit progress and termination. Let's consider this non-terminating term $(\Lambda(c_1 : \alpha \triangleright \alpha \to \alpha) \circ \Lambda(c_2 : \alpha \to \alpha \triangleright \alpha))\langle a\,(c_2\langle a\rangle)\rangle$ where $a = \lambda(x : \alpha)\, c_1\langle x\rangle\, x$. This is the usual omega term. Its erasure loops, while this term builds always growing coercions by projection.

Independently of a push reduction rule, when abstracting over coercions, we need to make sure that we only abstract over consistent coercions. We shouldn't allow to abstract over c of type $\tau \times \sigma \triangleright \tau \to \sigma$. Let's consider the term $c\langle\langle M, N\rangle\rangle\, M$ which is well-typed. It's erasure is $\langle\lfloor M\rfloor, \lfloor N\rfloor\rangle\, \lfloor M\rfloor$ which is stuck therefore breaking the progress lemma.

A solution to avoid considering consistency is to keep coercions during reduction. The reduction does not go wrong, but coercions are no more erasable. GHC is doing something related for their first-class coercions. They do not fully erase them. They keep the termination information of coercions after compilation, and they don't reduce under coercion abstraction. This way, coercions can be represented with 0-bit since the termination information is only reduction steps. If a coercion type is inconsistent, then all coercions of this type would infinitely loop and avoid the machine to break. We extend our approach with 0-bit coercions in Section 6.1.5.

GHC adopts another solution for their top-level coercions. They have a consistency lemma which is needed to prove progress for the implicit language. In our case this lemma is ι-classification (Lemma 50). In GHC, they have a more restrictive lemma that asks all coercions not to touch their head type constructor. Our lemma is more subtle since it only ask this condition for non-erasable type constructors.

The solution used in System F_ι^p is inspired from $\mathsf{F}_{<:}$ and MLF. It allows coercion abstraction only for parametric coercion types. In $\mathsf{F}_{<:}$ the parametricity is on the argument type and in MLF it is on the return type. In System F_ι^p we allow both. This restriction prevents to build a coercion between two arrow types using a coercion variable. This is proved in the classification lemma (Lemma 50) which is needed to prove the bisimulation property.

The solution we develop in Chapter 5 with System L^c is to ask all abstract coercions to be syntactically inhabited, which implies consistency. The minimum requirement for soundness would be to ask a semantic witness that the abstract coercion is inhabited. This extension is discussed in Section 6.1.10.

4.6.2 Push

We call *abstract*, coercions containing coercion variables in places such that reduction cannot progress with the reduction rules of System F_ι^p.

In our current setting, only concrete coercions are decomposed. For example, an arrow η-expansion coercion applied to a function can be reduced (with Rule RedEtaArr) into a new function that coerces its argument and pass it to the previous function and then coerces the result. Decomposing abstract coercions is not needed, because reduction can always bring concrete coercions in places where decomposition of coercions would be needed. However, if we relax our restriction on coercion abstraction, we may have abstract coercions in-between redexes blocking the reduction that could be performed if the coercion were erased—thus breaking backward simulation. In this case, backward simulation may be recovered by adding a decomposition rule for abstract coercions—and associated coercion constructs.

Interestingly, decomposing coercions makes the backward simulation much easier, since wedges can now be decomposed instead of blocking the evaluation. However, the decomposition introduces new operations on abstract coercions that must eventually be reduced

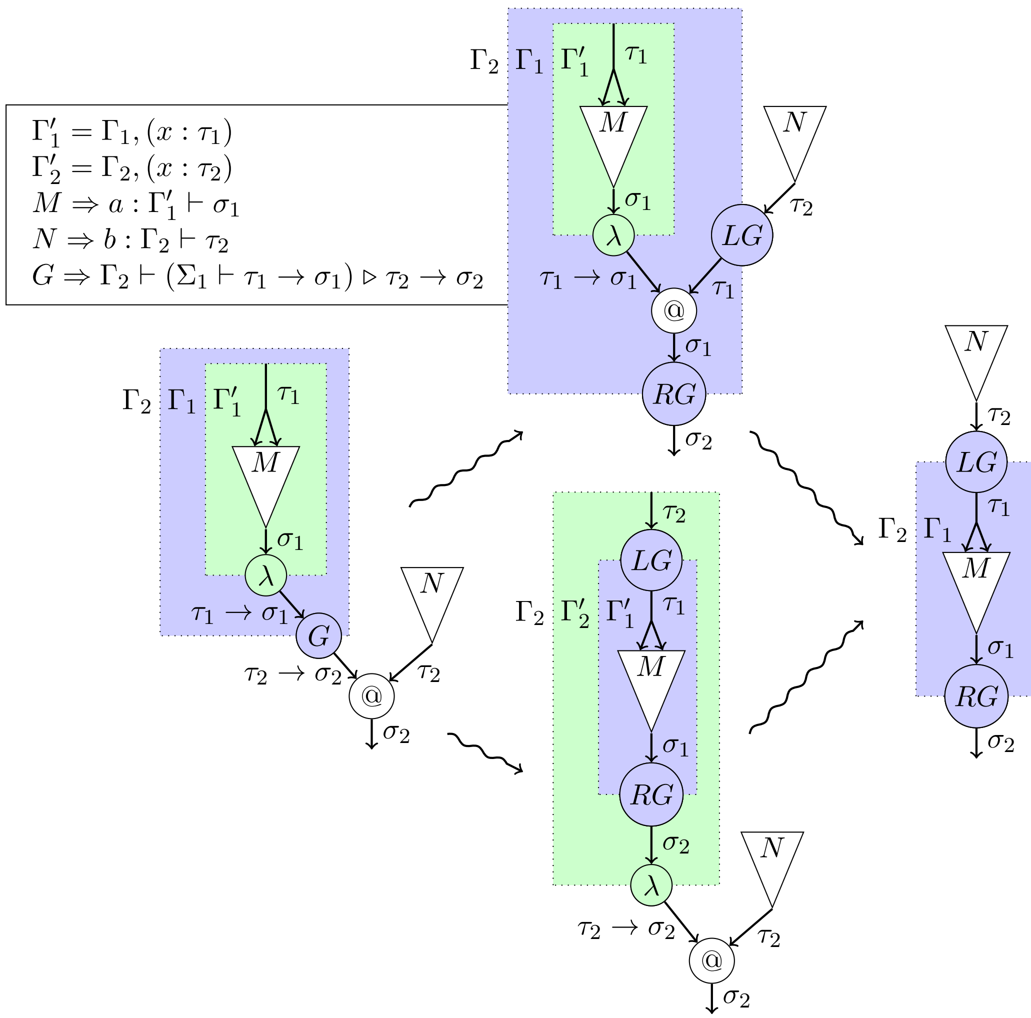


Figure 4.18: Push

when coercions become concrete: proof obligations are just moved elsewhere. Currently, explicit progress, consistency, and backward simulation are proved using coercion values in ι-classification. In order to have coercion values with push, we need to add reduction rules for coercions.

Figure 4.18 shows how a coercion in-between a redex can be decomposed. The initial term $G\langle\lambda(x : \tau_1)\, M\rangle\, N$, on the left, is the application of the function $\lambda(x : \tau_1)\, M$ coerced by G to the argument N. Its erasure is the application of the erasure of $\lambda(x : \tau_1)\, M$, namely $\lambda x\, a$, to the erasure of N, namely b, that is, the redex $(\lambda x\, a)\, b$. Can we make the redex apparent on the source term, without inspecting the coercion G, *i.e.* in a way that may also work when G is an abstract coercion (a coercion variable for instance)?

The terms are drawn upside-down to mimic their typing derivations (terms are explicitly typed and thus isomorphic to their typing derivations). Boxes represent scopes. The coercion G can extend the environment from Γ_2 to Γ_1 with the erasable environment Σ_1 ($\Gamma_1 = \Gamma_2, \Sigma_1$). The blue (dark gray for black and white prints) box delimits the separation between the Γ_2 environment (which is outside with a white background color) and the Γ_1 environment (which is inside with a blue background color). This blue part is more precisely the application of the environment action Σ_1 to Γ_2 (see Section 6.1.11). The lambda constructor opens a scope and thus also changes the environment. Its effect is to add a term binding. We use a green (or light gray for black and white versions) background color to show when x is added to the environment. We add a tick to the environment meta-variable when it is extended with the binding for x (see the first two definitions on Figure 4.18).

The coercion G lies on the left-hand side of the application and outside of the abstraction. It can be moved on other branches of the application node (upward arrow) or inside the abstraction (downward arrow). The argument is coerced directly in the former case, but indirectly on each use of the function parameter in the latter case.

In both cases we need to decompose G of type $\Gamma_2 \vdash (\Sigma_1 \vdash \tau_1 \to \sigma_1) \triangleright \tau_2 \to \sigma_2$ into RG of type $\Gamma_2 \vdash (\Sigma_1 \vdash \sigma_1) \triangleright \sigma_2$ and LG of type $\Gamma_1 \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_1$ (modulo the presence of x when we push under the lambda) where Σ_2 reverses the action of Σ_1 (see Section 6.1.11). If the only actions are bindings, like in System F_ι^p and usual type systems, LG would have type $\Gamma_1 \vdash \tau_2 \triangleright \tau_1$, binding nothing, and the blue boxes would spread upward the LG node and not only downward.

Besides, for both reduction rules, the scope of G moves. It starts at RG and ends at LG (or it ends at the leaves of the typing derivation if we don't use environment actions). We can actually see this in the condition that $\Gamma_2, \Sigma_1, \Sigma_2$ (the environment once we leave LG) should behave as Γ_2 (the environment before we enter RG).

Fortunately, the two possible reductions are locally confluent: both terms reduce with a β-reduction on the rightmost term where N got substituted in M and both N and M are coerced with the left and right projections of G—and the whole term remains well-typed. Therefore, only one of either rule may be retained, indifferently, or both rules may be offered, simultaneously. It is also possible to add a unique rule doing both steps at the same time. This rule would then be labeled as a β-reduction rule since its erasure does one step of computation.

At this stage, we may consider the following typing rules and reduction rule:

$$\textsc{CoerPushRight}\quad \frac{G \Rightarrow \Gamma_2 \vdash (\Sigma_1 \vdash \tau_1 \to \sigma_1) \triangleright \tau_2 \to \sigma_2}{RG \Rightarrow \Gamma_2 \vdash (\Sigma_1 \vdash \sigma_1) \triangleright \sigma_2} \qquad \textsc{CoerPushLeft}\quad \frac{G \Rightarrow \Gamma_2 \vdash (\Sigma_1 \vdash \tau_1 \to \sigma_1) \triangleright \tau_2 \to \sigma_2}{LG \Rightarrow \Gamma_2 \vdash \tau_2 \triangleright \tau_1}$$

$$\textsc{RedPushArr}\quad G\langle \lambda(x : \tau)\, M\rangle\, N \leadsto_\beta RG\langle M[x/LG\langle N\rangle]\rangle$$

Still, we need further reduction rules to recover confluence of the language and in particular have a progress lemma for coercions, because we need coercion values for the ι-classification lemma and thus implicit progress. Indeed, the decomposition introduced new left and right destructors for coercions, which must eventually be reduced in order to show that the implicit language is sound.

We define a γ-reduction written $G \leadsto_\gamma G$ for coercions (see Section 6.1.12). The goal of this reduction relation is to give closed coercions a value which has to be a concrete coercion of System F_ι^p. Some examples for this reduction relation are:

$$R(G_1 \xrightarrow{\tau} G_2) \leadsto_\gamma G_2 \qquad R\Diamond \leadsto_\gamma \Diamond$$
$$L(G_1 \xrightarrow{\tau} G_2) \leadsto_\gamma G_1 \qquad L\Diamond \leadsto_\gamma \Diamond$$

It may be actually quite hard to find a complete set of rules, because we need to define the left and right destructors for all previous coercions. Moreover, this push mechanism complicates the framework for future coercion extensions.

One example of a difficult γ-reduction rule is when we want to reduce the left or right projections of $\bullet\rho \circ \Lambda\alpha$, which happens when we instantiate and apply an argument to a polymorphic function. This coercion has type $\Gamma \vdash (\alpha \vdash \tau \to \sigma) \triangleright \tau[\alpha/\rho] \to \sigma[\alpha/\rho]$. We want to extract two coercions of type $\Gamma \vdash (\alpha \vdash \sigma) \triangleright \sigma[\alpha/\rho]$ for the right projection and $\Gamma, \alpha \vdash \tau[\alpha/\rho] \triangleright \tau$ for the left projection. We recognize type instantiation for the right projection and the same coercion $\bullet\rho \circ \Lambda\alpha$ works. For the left projection however the situation is more complicated. We need the reverse operation of type instantiation.

One solution to this problem is to preserve the link between the right and left projections and factor the initial coercion. One way to do so is to have the right projection bind the left projection. The right projection would then be $\pi(c)G$ and the left would be c. We would then have the following coercion and reduction rules:

$$\textsc{CoerPush}\quad \frac{G \Rightarrow \Gamma_2 \vdash (\Sigma_1 \vdash \tau_1 \to \sigma_1) \triangleright \tau_2 \to \sigma_2}{\pi(c)G \Rightarrow \Gamma_2 \vdash (\Sigma_1, (c : \tau_2 \triangleright \tau_1) \vdash \sigma_1) \triangleright \sigma_2}$$

$$\textsc{RedPushArr}\quad \frac{c \text{ is fresh}}{G\langle\lambda(x : \tau)\, M\rangle\, N \rightsquigarrow_\beta (\pi(c)G)\langle M[x/c\langle N\rangle]\rangle}$$

However, pursuing this path while preserving confluence, typing, subject reduction, and progress is one of the main difficulties of coercion decomposition. The other main difficulty is strong normalization. We cannot reify coercion projections as lambda terms, so our current proof of termination does not work. We may argue that strong normalization is not necessary, but it is only true for β-reduction, because the ι-normalization is used in the proof of soundness. Said otherwise, we need ι-normalization, but we do not need to prove the strong normalization of the implicit reduction.

In the next chapter, we extend System F_ι^p with unrestricted coercion abstraction. We only define the implicit version of this extension, called System L^c. This avoids defining the explicit reduction and as a consequence proving that the ι-reduction normalizes. More precisely, we do not need to exhibit a ι-reduction strategy to clean a redex in order to prove the backward simulation. We also avoid defining typing rules for coercion decomposition. There is however an underlying explicit version similar to System F_ι^p, but missing backward simulation. We do not know if it is possible to define an explicit version with subject reduction (see Section 5.7).

Chapter 5

An implicit calculus of coercions: System L^{c}

In this chapter we define a type system, called System L^{c}, with unrestricted but safe erasable coercion abstractions and unrestricted but safe recursive types. Coercion abstraction is featured with coherent polymorphism which implies the consistency of abstract coercions. Like for Constraint ML but unlike for other type systems, we only give an implicit version for System L^{c}, since the explicit version is hard to define and prove (see Section 4.6.2). The implicit version is proved with a step-index semantics. We developed a novel approach to step-index proofs, because we use a strong reduction setting while usual step-index proofs are done in a weak reduction setting. In particular, we put indices inside terms instead of aside them (see Section 5.2.1).

Since our approach to type systems is semantic in this chapter, we simply define three notions: kinds which are interpreted as sets of mathematical objects, types which are interpreted as mathematical objects (the unit object, pairs of objects, or sets of terms for instance), and propositions interpreted as mathematical propositions.

We show that System L^{c} satisfies the soundness proposition (Section 5.3), which we actually proved in Coq. We show that System F_ι^p can be seen as a sub-language (Section 5.4.2). And we describe how Constraint ML is included in System L^{c} (Section 5.4.3). The Coq formalization uses de-Bruijn indices while the current presentation uses names. Another difference is that the current presentation relies on the extraction property, while the Coq formalization inlines the extraction lemma when needed. This is necessary for the induction to work. These are the two differences between both presentations. We give more details about the Coq formalization in Section 5.6.

5.1 Definition

The definition of syntactic classes are given on Figure 5.1. We define a syntactical class, called kinds and written κ, to classify types. Kinds contain the star kind $\star$ to classify sets of terms, the unit kind 1 to classify the unit mathematical object, product kinds $\kappa \times \kappa$ to classify mathematical pairs, and constrained kinds $\{\alpha : \kappa \mid \mathrm{P}\}$ to constrain the set κ to its elements α satisfying the proposition P. For instance, the constrained kind $\{\alpha : \star \mid (\varnothing \vdash \alpha) \triangleright \tau\}$ corresponds to the set of types σ smaller than $\tau[\alpha/\sigma]$, which also corresponds to the domain of the upper bounded polymorphic type of System $\mathsf{F}_{<:}$ or F_ι^p.

$$
\begin{array}{rcll}
\kappa & ::= & \star \mid 1 \mid \kappa \times \kappa \mid \{\alpha : \kappa \mid \mathrm{P}\} & \text{Kinds} \\
\tau, \sigma & ::= & \alpha \mid \langle\rangle \mid \langle \tau, \tau \rangle \mid \mathsf{fst}\, \tau \mid \mathsf{snd}\, \tau & \text{Types} \\
 & \mid & \tau \to \tau \mid \tau \times \tau \mid \forall(\alpha : \kappa)\, \tau \mid \mu\alpha\, \tau \mid \bot \mid \top & \\
\mathrm{P} & ::= & \top \mid \mathrm{P} \wedge \mathrm{P} \mid (\Sigma \vdash \tau) \triangleright \tau \mid \exists\, \kappa \mid \forall(\alpha : \kappa)\, \mathrm{P} & \text{Propositions} \\
\Sigma & ::= & \varnothing \mid \Sigma, (\alpha : \kappa) & \text{Type envs} \\
\Theta & ::= & \varnothing \mid \Theta, \mathrm{P} & \text{Proof envs} \\
\Gamma & ::= & \varnothing \mid \Gamma, (x : \tau) & \text{Term envs} \\
\mathsf{rec} & ::= & \mathsf{NE} \mid \mathsf{WF} & \text{Well-foundness signs}
\end{array}
$$

Figure 5.1: System $\mathsf{L^c}$ syntax

As usual we define the syntactical class of types, written τ or σ. Types contain variables α, the unit object $\langle\rangle$, pairs $\langle \tau, \tau \rangle$, and projections $\mathsf{fst}\, \tau$ and $\mathsf{snd}\, \tau$. Types additionally contain the usual arrow types $\tau \to \tau$, product types $\tau \times \tau$, polymorphic types $\forall(\alpha : \kappa)\, \tau$, recursive types $\mu\alpha\, \tau$, the bottom type $\bot$, and the top type $\top$. These last types are interpreted as sets of terms. We define the projections of pairs equal to the associated components: $\mathsf{fst}\, \langle \tau, \sigma \rangle$ is equal to τ and $\mathsf{snd}\, \langle \tau, \sigma \rangle$ is equal to σ. This notion of equality is reflexive, symmetrical, transitive, and congruent for all syntactical constructs (see the Coq inductive `typesystem.jeq` for more details). We could also have added $\mu\alpha\, \tau = \tau[\alpha/\mu\alpha\, \tau]$ in the equality rules, but this rule needs typing conditions to ensure that τ is actually of kind $\star$ and we do not have typing conditions in our syntactical equality rules. We could have defined recursive types as $\mu(\alpha : \kappa)\tau$ and then have the given equality, but the semantics of recursive types for arbitrary kinds is more involved. As a consequence, we have the recursive type equality at the coercion level, where type kinding is available. See Section 6.1.4 for discussion about arbitrary kind recursive types.

Instead of having only proofs of inclusion between sets of terms, that we call coercions, we define a general notion of propositions and define a syntactical class, written P, for them. Propositions contain the true proposition $\top$, conjunctions $\mathrm{P} \wedge \mathrm{P}$, coercions $(\Sigma \vdash \tau) \triangleright \tau$, coherence propositions $\exists\, \kappa$, and polymorphic propositions $\forall(\alpha : \kappa)\, \mathrm{P}$. These propositions are what we can usually find in type systems based on constraints, which gives an intuition of how System $\mathsf{L^c}$ compares to $\mathsf{Constraint\ ML}$. Notice that the coherence proposition of the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, namely $\exists\, \{\alpha : \kappa \mid \mathrm{P}\}$, gives the usual existential proposition. Because coherence corresponds to habitation and a type τ is in the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ if it is in κ and satisfies P by definition.

We define environments for types, proofs, and terms. Type environments, written Σ, are lists of bindings of the form $(\alpha : \kappa)$ which binds the type variable α to its kind κ. Notice that since kinds depend upon types, they may refer to type variables, and thus type environments are dependent and the order really matters. Proof environments, written Θ, collect which propositions hypotheses are accessible and hence they are lists of propositions. We don't need a variable to designate the proof since we only define the implicit version of our type system. Finally, term environments, written Γ, are lists of bindings of the form $(x : \tau)$ binding the term variable x to its type τ. The order in proof and term environments does not matter, but we still keep them as lists since this is how they are represented in the Coq proof.

Finally, we define the well-foundness signs NE and WF, which we designate with the meta-variable rec. These signs are used to know which type functors are non-expansive, for sign

$$\textsc{RecVar}\quad \alpha \mapsto \alpha : \mathsf{NE} \qquad \textsc{RecArr}\ \frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \mapsto \sigma : \mathsf{NE}}{\alpha \mapsto \tau \to \sigma : \mathsf{WF}} \qquad \textsc{RecProd}\ \frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \mapsto \sigma : \mathsf{NE}}{\alpha \mapsto \tau \times \sigma : \mathsf{WF}}$$

$$\textsc{RecFor}\ \frac{\alpha \mapsto \tau : \mathsf{rec} \qquad \alpha \notin \mathsf{fv}(\kappa)}{\alpha \mapsto \forall(\beta : \kappa)\, \tau : \mathsf{rec}} \qquad \textsc{RecMu}\ \frac{\alpha \mapsto \tau : \mathsf{rec} \qquad \beta \mapsto \tau : \mathsf{WF}}{\alpha \mapsto \mu\beta\, \tau : \mathsf{rec}}$$

$$\textsc{RecWF}\ \frac{\alpha \notin \mathsf{fv}(\tau)}{\alpha \mapsto \tau : \mathsf{WF}} \qquad \textsc{RecNE}\ \frac{\alpha \mapsto \tau : \mathsf{WF}}{\alpha \mapsto \tau : \mathsf{NE}}$$

Figure 5.2: System L^{c} well-foundness judgment relation

NE, or well-founded, for sign WF.

Similarly to System $\mathsf{F}_{\mathsf{rec}}$, we define a well-foundness judgment, written $\alpha \mapsto \tau : \mathsf{rec}$, which is interpreted as the functor associating the type variable α to type τ is non-expansive (resp. well-founded) if rec is NE (resp. WF). The rules of this judgment are given on Figure 5.2. The identity functor $\alpha \mapsto \alpha$ is non-expansive according to rule RecVar. The arrow and product types are well-founded with respect to the type variable α, if their components are non-expansive with respect to α, according to rules RecArr and RecProd. Rule RecFor tells that the polymorphic type $\forall(\beta : \kappa)\, \tau$ has the same well-foundness sign than τ with respect to the type variable α which has to be different from β. Besides, the kind κ must not mention the type variable α. Similarly, rule RecMu tells that the recursive type $\mu\beta\, \tau$ preserves the well-foundness sign of its body τ with respect to type variable α. The additional condition is that $\mu\beta\, \tau$ has to be a well-formed recursive type, in other words, the type τ has to be well-founded with respect to the type variable β. The last two rules are generic ways to prove well-foundness or non-expansiveness. Constant functors are well-founded by rule RecWF and well-founded functors are non-expansive by rule RecNE.

Because most of the judgments in System L^{c} are mutually recursive, we give a snapshot of all possible judgments with their intuition and mathematical interpretation. The only particularities are the well-foundness judgment, which we just defined, because it has no dependencies, and the term judgment, because no judgment depends on it. We write $\Sigma \Vdash \kappa$ when the kind κ is well-formed under the type environment Σ. Similarly, we write $\Sigma \Vdash \mathrm{P}$, $\Sigma \Vdash \Sigma$, and $\Sigma \Vdash \Theta$, for proposition, type environment, and proposition environment well-formedness, respectively. Well-formedness judgments have no mathematical interpretation. They are only used for extraction (Lemma 94), which we describe when presenting each judgment.

The kind judgment $\Sigma \vdash \kappa$ means that the kind κ is coherent under Σ. It is interpreted as the mathematical proposition that the interpretation of κ under interpretations of Σ is inhabited. When Σ is well-formed (under the empty environment), we can extract from $\Sigma \vdash \kappa$ that κ is well-formed under Σ. The type judgment $\Sigma \vdash \tau : \kappa$ means that the type τ has kind κ under environment Σ. Its interpretation is that the interpretation of τ is an element of the interpretation of κ under interpretations of Σ. When Σ is well-formed, we can extract from $\Sigma \vdash \tau : \kappa$ that κ is well-formed under Σ. The proposition judgment $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$ means that the proposition P holds under Σ with accessible coinduction hypotheses Θ_0 and protected

$$\frac{\Sigma; \varnothing; \varnothing \vdash \exists \kappa}{\Sigma \vdash \kappa}\ \textsc{Kind}$$

Figure 5.3: System $\mathsf{L^c}$ kind judgment relation

$$\frac{\Sigma \vdash \tau : \kappa \qquad \kappa = \kappa' \qquad \Sigma \Vdash \kappa'}{\Sigma \vdash \tau : \kappa'}\ \textsc{TypeConv} \qquad \frac{\Sigma \vdash \kappa' \qquad \Sigma, (\alpha' : \kappa') \vdash \tau : \kappa \qquad \alpha' \notin \mathsf{fv}(\tau, \kappa)}{\Sigma \vdash \tau : \kappa}\ \textsc{TypeStr}$$

$$\frac{(\alpha : \kappa) \in \Sigma}{\Sigma \vdash \alpha : \kappa}\ \textsc{TypeVar} \qquad \frac{\Sigma \vdash \tau : \star \qquad \Sigma \vdash \sigma : \star}{\Sigma \vdash \tau \to \sigma : \star}\ \textsc{TypeArr} \qquad \frac{\Sigma \vdash \tau : \star \qquad \Sigma \vdash \sigma : \star}{\Sigma \vdash \tau \times \sigma : \star}\ \textsc{TypeProd} \qquad \frac{\Sigma, (\alpha : \kappa) \vdash \tau : \star}{\Sigma \vdash \forall (\alpha : \kappa)\, \tau : \star}\ \textsc{TypeFor}$$

$$\frac{\alpha \mapsto \tau : \mathsf{WF} \qquad \Sigma, (\alpha : \star) \vdash \tau : \star}{\Sigma \vdash \mu\alpha\, \tau : \star}\ \textsc{TypeMu} \qquad \frac{}{\Sigma \vdash \bot : \star}\ \textsc{TypeBot} \qquad \frac{}{\Sigma \vdash \top : \star}\ \textsc{TypeTop} \qquad \frac{}{\Sigma \vdash \langle\rangle : 1}\ \textsc{TypeUnit}$$

$$\frac{\Sigma \vdash \tau_1 : \kappa_1 \qquad \Sigma \vdash \tau_2 : \kappa_2}{\Sigma \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \times \kappa_2}\ \textsc{TypePair} \qquad \frac{\Sigma \vdash \tau : \kappa_1 \times \kappa_2}{\Sigma \vdash \mathsf{fst}\, \tau : \kappa_1}\ \textsc{TypeFst} \qquad \frac{\Sigma \vdash \tau : \kappa_1 \times \kappa_2}{\Sigma \vdash \mathsf{snd}\, \tau : \kappa_2}\ \textsc{TypeSnd}$$

$$\frac{\Sigma, (\alpha : \kappa) \Vdash \mathrm{P} \qquad \Sigma \vdash \tau : \kappa \qquad \Sigma; \varnothing; \varnothing \vdash \mathrm{P}[\alpha/\tau]}{\Sigma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}\ \textsc{TypePack} \qquad \frac{\Sigma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Sigma \vdash \tau : \kappa}\ \textsc{TypeUnpack}$$

Figure 5.4: System $\mathsf{L^c}$ type judgment relation

coinduction hypotheses Θ_1. We use coinduction for recursive types (see Section 5.4.4). Our notion of productivity comes from computational types (see rule CoerArr for example). The mathematical interpretation of the proposition judgment is involved and uses step-indices (see Lemma 99). When Σ is well-formed and both Θ_0 and Θ_1 are well-formed under Σ, we can extract from $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$ that P is well-formed under Σ.

We define in Figure 5.3 when a kind is coherent. A kind κ is coherent with respect to a type environment Σ, written $\Sigma \vdash \kappa$, when the kind coherence proposition $\exists \kappa$ is satisfied under Σ. Rule Kind is the only rule for the kind coherence judgment. This means that the kind judgment is just a notation for the kind coherence proposition. Notice that we give no coinduction hypotheses to the proposition, which explains the two empty proposition environments in the premise.

A type τ has kind κ under type environment Σ, written $\Sigma \vdash \tau : \kappa$, if it satisfies the rules given in Figure 5.4. Rule TypeConv permits to convert from κ to κ' the kind of a type τ, given both kinds are equal and the final kind κ' is well-formed under the given environment. Rule TypeStr is the elimination rule of the coherence proposition (see rule PropExi). Coherence proposition actually behaves like an existential: $\exists \kappa$ means that there is a type τ of kind κ. Since the kind judgment is actually a notation for the coherence proposition, the premise

$\Sigma \vdash \kappa'$ is actually $\Sigma; \varnothing; \varnothing \vdash \exists\, \kappa'$ and we are eliminating this existential proposition. To do so, we use the second premise assuming a type α' of kind κ' to prove that τ has kind κ. Since neither τ or κ mention the witness type α', we can conclude that τ has kind κ without the hypothesis about the coherence of κ'.

Rule TypeVar tells that the type variable α has kind κ under the type environment Σ, if α is bound to κ in Σ. Arrow and product types have the star kind if their components do too, by rules TypeArr and TypeProd respectively. The type environment remains the same since none of these two type constructs binds type variables. One example for type binding is rule TypeFor for polymorphic types. The polymorphic type $\forall(\alpha : \kappa)\, \tau$ has the star kind under the type environment Σ if the body type τ has the star kind under the environment Σ extended with the type binding $(\alpha : \kappa)$ associating the type variable α to its kind κ. We do not ask κ to be coherent either, because this condition is only necessary for the type generalization coercion rule CoerGen.

Rule TypeMu tells that the recursive type $\mu\alpha\, \tau$ has the star kind if its body has the star kind under the same environment extended with α bound to the star kind. First, notice that recursive types are only of the star kind (relaxing this restriction is discussed in Section 6.1.4). Then, a recursive type has also to be well-founded as a functor to be well-kinded, which is ensured by hypothesis $\alpha \mapsto \tau : \mathsf{WF}$. The bottom and top type have kind star according to rules TypeBot and TypeTop.

By rule TypeUnit, the unit type has the unit kind. The pair of types τ_1 and τ_2 has the product kind $\kappa_1 \times \kappa_2$ under type environment Σ, given type τ_1 has kind κ_1 and type τ_2 has kind κ_2 under Σ. This rule is similar to the rule for pairs in terms. And like for terms, type projections extract their kind from their premise. The first projection $\mathsf{fst}\, \tau$ and second projection $\mathsf{snd}\, \tau$ have kinds κ_1 and κ_2 respectively, if the type τ has the product kind $\kappa_1 \times \kappa_2$ under the same environment.

Finally the last two rules are particular since they just change the kind of a type leaving the type unchanged. In this sense they are similar to coercions but at the kind level (see Section 6.1.8). This is actually called subkinding since in our case the environment are not modified. Rule TypePack tells that if a type τ has kind κ and satisfies the well-formed property P, then it also has the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ of types of kind κ satisfying P. Rule TypeUnpack acts the opposite way: if type τ has constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, then it also has kind κ by forgetting the fact that it satisfies also the property P. The constrained kind can be seen as a dependent sum of a kind and a proposition. It is built with a type and a proof that this type satisfies a proposition by rule TypePack. It can be eliminated on its first component by rule TypeUnpack. And it can also be eliminated on its second component by rule PropRes in the proposition judgment.

The proposition judgment is the most complicated one and is actually split in two figures: one for the logic and one for coercions. The logical rules are given in Figure 5.5, while coercion rules are given in Figure 5.6. The proposition judgment is written $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$. It tells that the proposition P is true under the type environment Σ, given additional coinduction hypotheses in Θ_0 and Θ_1. The hypotheses in Θ_0 are accessible while those in Θ_1 will be accessible only after some productive proof step has been done. Productive proof steps are exclusively done in the η-expansion coercion rules of computational types. Coinduction hypotheses are extended in the coinduction proof step. These are the only rule altering the coinduction hypotheses.

Similar to rule TypeConv for types, we can convert the proof of a proposition from P to P' under the same environment, given that P is equal to P' and P' is well-formed under the

$$\textsc{PropConv}\quad \frac{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P} \qquad \mathrm{P} = \mathrm{P}' \qquad \Sigma \Vdash \mathrm{P}'}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}'}$$

$$\textsc{PropStr}\quad \frac{\Sigma \vdash \kappa \qquad \Sigma, (\alpha : \kappa); \Theta_0; \Theta_1 \vdash \mathrm{P} \qquad \alpha \notin \mathsf{fv}(\Theta_0, \Theta_1, \mathrm{P})}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}} \qquad \textsc{PropVar}\quad \frac{\mathrm{P} \in \Theta_0}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}} \qquad \textsc{PropTrue}\quad \Sigma; \Theta_0; \Theta_1 \vdash \top$$

$$\textsc{PropPair}\quad \frac{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_1 \qquad \Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_2}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_1 \wedge \mathrm{P}_2} \qquad \textsc{PropFst}\quad \frac{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_1 \wedge \mathrm{P}_2}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_1} \qquad \textsc{PropSnd}\quad \frac{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_1 \wedge \mathrm{P}_2}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}_2}$$

$$\textsc{PropExi}\quad \frac{\Sigma \vdash \tau : \kappa}{\Sigma; \Theta_0; \Theta_1 \vdash \exists\, \kappa} \qquad \textsc{PropGen}\quad \frac{\Sigma \Vdash \kappa \qquad \Sigma, (\alpha : \kappa); \Theta_0; \Theta_1 \vdash \mathrm{P} \qquad \alpha \notin \mathsf{fv}(\Theta_0, \Theta_1)}{\Sigma; \Theta_0; \Theta_1 \vdash \forall(\alpha : \kappa)\, \mathrm{P}}$$

$$\textsc{PropInst}\quad \frac{\Sigma; \Theta_0; \Theta_1 \vdash \forall(\alpha : \kappa)\, \mathrm{P} \qquad \Sigma \vdash \tau : \kappa}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}[\alpha/\tau]} \qquad \textsc{PropRes}\quad \frac{\Sigma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}[\alpha/\tau]} \qquad \textsc{PropFix}\quad \frac{\Sigma \Vdash \mathrm{P} \qquad \Sigma; \Theta_0; \Theta_1, \mathrm{P} \vdash \mathrm{P}}{\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}}$$

Figure 5.5: System $\mathsf{L^c}$ proposition judgment relation

same environment. This rule is named PropConv and given in Figure 5.5. Again, similarly to the type judgment rule TypeStr, we can eliminate the existential proposition when proving a proposition. By rule PropStr, if the kind κ is coherent under Σ and the proposition P holds under the extended environment $\Sigma, (\alpha : \kappa)$ with coinduction hypotheses Θ_0 and Θ_1 given that the type variable α is not free in Θ_0, Θ_1, and P.

Rule PropVar tells that a proposition P is true if it is present in the accessible coinduction hypotheses Θ_0. The true proposition is always true, according to rule PropTrue. Rule PropPair tells that the conjunction proposition $\mathrm{P}_1 \wedge \mathrm{P}_2$ is true if its components P_1 and P_2 are true under the same environments. If a conjunction proposition $\mathrm{P}_1 \wedge \mathrm{P}_2$ is true, then its components P_1 and P_2 are also true under the same environments by rules PropFst and PropSnd respectively. These conjunction rules resemble those of the product type and the product kind, except we are not interested in their proof terms. Rule PropExi is the associated introduction rule of rules TypeStr and PropStr. A coherence proposition $\exists\, \kappa$ is true if its kind κ is inhabited, which is our notion of coherence.

Rule PropGen and PropInst deal with proposition polymorphism. The polymorphic proposition $\forall(\alpha : \kappa)\, \mathrm{P}$ holds under the type environment Σ and the coinduction hypotheses Θ_0 and Θ_1, if its inner proposition P holds under the extended type environment $\Sigma, (\alpha : \kappa)$, given that κ is well-formed under Σ and α is not free in the coinduction hypotheses. To eliminate such polymorphic proposition, we supply it a type of the given kind. If $\forall(\alpha : \kappa)\, \mathrm{P}$ holds and τ has kind κ, then the substituted proposition $\mathrm{P}[\alpha/\tau]$ holds under the same environments.

Rule PropRes allows to extract from a type of a constrained kind the fact that the property holds for this type. This is the second projection rule of the constrained kind—the first being rule TypeUnpack. If τ has kind $\{\alpha : \kappa \mid \mathrm{P}\}$, then $\mathrm{P}[\alpha/\tau]$ is true under the same type environment and any coinduction environments. A rule we do not usually find in type systems

$$\frac{\Sigma \vdash \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \tau) \triangleright \tau}\ \textsc{CoerRefl}$$

$$\frac{\mathsf{dom}(\Sigma_2) \cap \mathsf{fv}(\Theta_0, \Theta_1) = \varnothing \qquad \Sigma, \Sigma_2; \Theta_0; \Theta_1 \vdash (\Sigma_1 \vdash \tau_1) \triangleright \tau_2 \qquad \Sigma; \Theta_0; \Theta_1 \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_3}{\Sigma; \Theta_0; \Theta_1 \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \triangleright \tau_3}\ \textsc{CoerTrans}$$

$$\frac{\mathsf{dom}(\Sigma') \cap \mathsf{fv}(\tau) = \varnothing \qquad \Sigma; \Theta_0; \Theta_1 \vdash (\Sigma' \vdash \tau) \triangleright \sigma}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \tau) \triangleright \sigma}\ \textsc{CoerWeak}$$

$$\frac{\mathsf{dom}(\Sigma') \cap \mathsf{fv}(\Theta_0, \Theta_1, \tau) = \varnothing \qquad \Sigma, \Sigma'; \Theta_0, \Theta_1; \varnothing \vdash (\varnothing \vdash \tau) \triangleright \tau' \qquad \Sigma; \Theta_0, \Theta_1; \varnothing \vdash (\Sigma' \vdash \sigma') \triangleright \sigma}{\Sigma; \Theta_0; \Theta_1 \vdash (\Sigma' \vdash \tau' \to \sigma') \triangleright \tau \to \sigma}\ \textsc{CoerArr}$$

$$\frac{\Sigma; \Theta_0, \Theta_1; \varnothing \vdash (\Sigma' \vdash \tau') \triangleright \tau \qquad \Sigma; \Theta_0, \Theta_1; \varnothing \vdash (\Sigma' \vdash \sigma') \triangleright \sigma}{\Sigma; \Theta_0; \Theta_1 \vdash (\Sigma' \vdash \tau' \times \sigma') \triangleright \tau \times \sigma}\ \textsc{CoerProd}$$

$$\frac{\Sigma \vdash \kappa \qquad \Sigma, (\alpha : \kappa) \vdash \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing, (\alpha : \kappa) \vdash \tau) \triangleright \forall (\alpha : \kappa)\, \tau}\ \textsc{CoerGen} \qquad \frac{\Sigma, (\alpha : \kappa) \vdash \tau : \star \qquad \Sigma \vdash \sigma : \kappa}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \forall (\alpha : \kappa)\, \tau) \triangleright \tau[\alpha/\sigma]}\ \textsc{CoerInst}$$

$$\frac{\Sigma \vdash \mu\alpha\, \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \mu\alpha\, \tau) \triangleright \tau[\alpha/\mu\alpha\, \tau]}\ \textsc{CoerUnfold} \qquad \frac{\Sigma \vdash \mu\alpha\, \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \tau[\alpha/\mu\alpha\, \tau]) \triangleright \mu\alpha\, \tau}\ \textsc{CoerFold}$$

$$\frac{\Sigma \vdash \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \bot) \triangleright \tau}\ \textsc{CoerBot} \qquad \frac{\Sigma \vdash \tau : \star}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \tau) \triangleright \top}\ \textsc{CoerTop}$$

Figure 5.6: System $\mathsf{L^c}$ coercion judgment relation

is rule PropFix: it allows to prove a proposition by coinduction. If P is true assuming P in the protected coinduction environment, then P is true without this additional hypothesis. We can derive from this general rule the usual rules about recursive types we can find in other type systems (see Section 5.4.4).

Coercion rules resemble very closely to what we already have seen in System F_ι^p. A coercion proposition $(\Sigma' \vdash \tau') \triangleright \tau$ is well-formed under Σ, if its binding environment Σ' is coherent under Σ (all its kind are coherent), its argument type τ' has kind star under the extended environment Σ, Σ', and its return type τ has kind star under Σ. See rule WfPCoer in Figure 5.8 for the formal statement.

Coercions are closed by reflexivity and transitivity since they are interpreted as inclusions. Rule CoerRefl in Figure 5.6 tells that a type τ of kind $\star$ is smaller than itself. Rule CoerTrans tells that τ_1 extended with Σ_2, Σ_1 is smaller than τ_3 under Σ, if τ_2 extended with Σ_2 is smaller than τ_3 under Σ and τ_1 extended with Σ_1 is smaller than τ_2 under Σ, Σ_2. The coinduction environments remain the same, but the type variables bound in Σ_2 should not be free in the coinduction environments. If τ extended with Σ' is smaller than σ and τ does not mention the type variables in Σ', then τ is smaller than σ by rule CoerWeak.

Rules CoerArr and CoerProd are about η-expansions of computational types. Computational types are the only types that need η-expansion coercion rules, because η-expansion

$$\frac{}{\varnothing \vdash \Sigma}\ \textsc{TEnvNil} \qquad \frac{\Sigma \vdash \Sigma' \qquad \Sigma, \Sigma' \vdash \kappa \qquad \alpha \notin \mathsf{dom}(\Sigma, \Sigma')}{\Sigma \vdash \Sigma', (\alpha : \kappa)}\ \textsc{TEnvCons}$$

Figure 5.7: System $\mathsf{L^c}$ type environment judgment relation

coercion rules for erasable types are derivable from their introduction and elimination rules. Notice however that the η-expansion of recursive types uses coinduction (see Section 5.4.4). As η-expansion rules of computational types, rules CoerArr and CoerProd are productive proofs for the coinduction. This is why the protected coinduction environment, namely Θ_1, of the conclusion is accessible in the premises. If σ' extended with Σ' is smaller than σ under Σ and τ is smaller than τ' under Σ, Σ', then $\tau' \to \sigma'$ extended with Σ' is smaller than $\tau \to \sigma$ by rule CoerArr. Similarly if τ' extended with Σ' is smaller than τ and σ' extended with Σ' is smaller than σ, then $\tau' \times \sigma'$ extended with Σ' is smaller than $\tau \times \sigma$.

Rules CoerGen and CoerInst are about coherent polymorphism. The type τ extended with the type binding $(\alpha : \kappa)$ is smaller than the polymorphic type $\forall(\alpha : \kappa)\,\tau$, given that κ is coherent and τ has kind star under the extended environment. This coherence condition is an important feature of System $\mathsf{L^c}$: polymorphism is erasable (and hence a coercion proposition) only if it is coherent. Notice that the usual polymorphism for kind star is always coherent, because the star kind is trivially inhabited (with the top or bottom type for instance). Notice also that bounded polymorphism is also trivially coherent (see Section 5.4.2). Without this coherence condition we could give the term $27 + \mathsf{true}$ (the addition of the integer constant 27 and the boolean constant true) the type $\forall(\alpha : \{\alpha : 1 \mid (\varnothing \vdash \mathsf{Bool}) \rhd \mathsf{Int}\})\,\mathsf{Int}$. We simply use rule PropRes to extract from the abstract type variable α the fact that booleans can be coerced to integers. This kind of feature is actually useful to deal with GADTs and we present it as a Coq-proved extension in Section 5.5. Similarly to type generalization, type instantiation is a coercion and the polymorphic type $\forall(\alpha : \kappa)\,\tau$ is smaller than its instantiation $\tau[\alpha/\sigma]$ with type σ of kind κ.

Rules CoerUnfold and CoerFold are about recursive types. The well-kinded recursive type $\mu\alpha\,\tau$ is equivalent (*i.e.* smaller and bigger than) to its unfolding $\tau[\alpha/\mu\alpha\,\tau]$. Finally, rules CoerBot and CoerTop are about extrema. The bottom type $\perp$ is smaller than all types of kind star and the top type $\top$ is bigger than all types of kind star.

The type environment Σ' is coherent under Σ, written $\Sigma \vdash \Sigma'$, if its type variables are bound at most once and the associated kinds are coherent in their preceding type environment. The rules for type environment coherence are given in Figure 5.7. The empty type environment $\varnothing$ is always coherent according to rule TEnvNil. The extended environment $\Sigma', (\alpha : \kappa)$ is coherent under Σ if the type variable α is not already bound in Σ, Σ' and the kind κ is coherent under the extended environment Σ, Σ'.

We use the notation $\Sigma \Vdash \cdot$ for the well-formedness judgment of all syntactical classes subject to well-formedness rules: kinds, propositions, type environment, and coinduction environments. The well-formedness judgment has no mathematical interpretation. It is only use for the extraction lemma (Lemma 94). The well-formedness rules are given in Figure 5.8. An object is well-formed if its components are well-formed in an extended type environment when the object behaves as a binder. The only particular rule is for the coercion proposition, namely rule WfPCoer, because it asks for coherence and well-kindness instead of well-formedness. A

$$\frac{}{\Sigma \Vdash \star}\textsc{WfKStar} \qquad \frac{}{\Sigma \Vdash 1}\textsc{WfKOne} \qquad \frac{\Sigma \Vdash \kappa_1 \qquad \Sigma \Vdash \kappa_2}{\Sigma \Vdash \kappa_1 \times \kappa_2}\textsc{WfKProd} \qquad \frac{\Sigma \Vdash \kappa \qquad \Sigma, (\alpha : \kappa) \Vdash \mathrm{P}}{\Sigma \Vdash \{\alpha : \kappa \mid \mathrm{P}\}}\textsc{WfKRes} \qquad \frac{}{\Sigma \Vdash \top}\textsc{WfPTrue}$$

$$\frac{\Sigma \Vdash \mathrm{P}_1 \qquad \Sigma \Vdash \mathrm{P}_2}{\Sigma \Vdash \mathrm{P}_1 \wedge \mathrm{P}_2}\textsc{WfPAnd} \qquad \frac{\Sigma \vdash \Sigma' \qquad \Sigma, \Sigma' \vdash \tau' : \star \qquad \Sigma \vdash \tau : \star}{\Sigma \Vdash (\Sigma' \vdash \tau') \triangleright \tau}\textsc{WfPCoer} \qquad \frac{\Sigma \Vdash \kappa}{\Sigma \Vdash \exists \kappa}\textsc{WfPExi} \qquad \frac{\Sigma, (\alpha : \kappa) \Vdash \mathrm{P}}{\Sigma \Vdash \forall (\alpha : \kappa)\, \mathrm{P}}\textsc{WfPFor}$$

$$\frac{}{\Sigma \Vdash \varnothing}\textsc{WfHNil} \qquad \frac{\Sigma \Vdash \Sigma' \qquad \Sigma, \Sigma' \Vdash \kappa \qquad \alpha \notin \mathsf{dom}(\Sigma, \Sigma')}{\Sigma \Vdash \Sigma', (\alpha : \kappa)}\textsc{WfHCons} \qquad \frac{}{\Sigma \Vdash \varnothing}\textsc{WfYNil} \qquad \frac{\Sigma \Vdash \Theta \qquad \Sigma \Vdash \mathrm{P}}{\Sigma \Vdash \Theta, \mathrm{P}}\textsc{WfYCons}$$

Figure 5.8: System $\mathsf{L^c}$ well-formedness relation

$$\frac{}{\Sigma \vdash \varnothing}\textsc{EnvNil} \qquad \frac{\Sigma \vdash \Gamma \qquad \Sigma \vdash \tau : \star \qquad x \notin \mathsf{dom}(\Gamma)}{\Sigma \vdash \Gamma, (x : \tau)}\textsc{EnvCons}$$

Figure 5.9: System $\mathsf{L^c}$ term environment judgment relation

coercion proposition $(\Sigma' \vdash \tau') \triangleright \tau$ is well-formed if Σ' is coherent and both τ and τ' have kind star under their respective environments.

A term environment Γ is valid under Σ, written $\Sigma \vdash \Gamma$, if all its term variables are bound at most once and their associated types have kind star under Σ. The rules are given in Figure 5.9. The empty term environment is valid under any type environment by rule EnvNil. The extended term environment $\Gamma, (x : \tau)$ is valid if τ has kind star and the term variable x is not already bound in Γ, which has to be valid as well, according to rule EnvCons.

Since we separated type environments and term environments, the term judgment changes a little. We write $a : \Sigma; \Gamma \vdash \tau$ when the term a has type τ under the type environment Σ and the term environment Γ. The rules for the term judgment are given on Figure 5.10. These are the usual STLC rules plus the additional coercion typing rule. The mathematical interpretation of this judgment is that for all Σ-instantiation and all Γ-substitution in this instantiation, the term a after substitution is in the type τ after instantiation. Syntactically, we can extract from the term judgment that the type has kind star, if the type environment is well-formed and the term environment is valid.

Rule TermVar tells that the term variable x has type τ under type environment Σ and term environment Γ if x is bound to τ in Γ. The term abstraction $\lambda x\, a$ has type $\tau \to \sigma$ under Σ and Γ if its body a has type σ under the extended term environment $\Gamma, (x : \tau)$ and the same type environment Σ by Rule TermLam. If, under type environment Σ and term environment Γ, the term a has type $\tau \to \sigma$ and b has type τ, then their application $a\, b$ has type σ.

Rule TermPair tells that if a has type τ and b has type σ under type environment Σ and term environment Γ, then the pair $\langle a, b \rangle$ has type $\tau \times \sigma$. The other way around, if a has type $\tau \times \sigma$, then its first projection $\mathsf{fst}\, a$ has type τ and its second projection $\mathsf{snd}\, a$ has type σ under the same environments.

Finally, the coercion rule TermCoer changes the typing of term a from $\Sigma, \Sigma'; \Gamma \vdash \tau'$ to

$$\frac{(x : \tau) \in \Gamma}{x : \Sigma; \Gamma \vdash \tau}\ \textsc{TermVar} \qquad \frac{\Sigma \vdash \tau : \star \qquad a : \Sigma; \Gamma, (x : \tau) \vdash \sigma}{\lambda x\, a : \Sigma; \Gamma \vdash \tau \to \sigma}\ \textsc{TermLam} \qquad \frac{a : \Sigma; \Gamma \vdash \tau \to \sigma \qquad b : \Sigma; \Gamma \vdash \tau}{a\, b : \Sigma; \Gamma \vdash \sigma}\ \textsc{TermApp}$$

$$\frac{a : \Sigma; \Gamma \vdash \tau \qquad b : \Sigma; \Gamma \vdash \sigma}{\langle a, b\rangle : \Sigma; \Gamma \vdash \tau \times \sigma}\ \textsc{TermPair} \qquad \frac{a : \Sigma; \Gamma \vdash \tau \times \sigma}{\mathsf{fst}\, a : \Sigma; \Gamma \vdash \tau}\ \textsc{TermFst} \qquad \frac{a : \Sigma; \Gamma \vdash \tau \times \sigma}{\mathsf{snd}\, a : \Sigma; \Gamma \vdash \sigma}\ \textsc{TermSnd}$$

$$\frac{\begin{array}{c}\mathsf{dom}(\Sigma') \cap \mathsf{fv}(\Gamma) = \varnothing \\ a : \Sigma, \Sigma'; \Gamma \vdash \tau' \qquad \Sigma; \varnothing; \varnothing \vdash (\Sigma' \vdash \tau') \triangleright \tau\end{array}}{a : \Sigma; \Gamma \vdash \tau}\ \textsc{TermCoer}$$

Figure 5.10: System $\mathsf{L^c}$ term judgment relation

$\Sigma; \Gamma \vdash \tau$, given there is a coercion proposition $(\Sigma' \vdash \tau') \triangleright \tau$ under the type environment Σ. To respect scoping, the type environment Σ' should not bind type variables that are free in Γ. If this is the case, it suffices to rename these variables in Σ' and τ' with fresh ones.

5.2 Semantics

A term is sound if none of its reductions lead to an error (see Section 2.5.3). To avoid the negation, it is easier to reason with valid terms defined as the complement of Ω, *i.e.* terms that are not errors, which we write $\mho$. To remember these notations, Ω is the omega greek letter usually used for errors, while $\mho$ looks like the letter v for valid terms. Notice also that they are 180 degrees rotation of one another because they are complement of one another. Hence, a term is sound if all its reduction paths lead to valid terms. Since this construction appears repeatedly, we define the *expansion* of a set of terms R, which we write $(\leadsto^\star \mathsf{R})$, the set of terms a such that any reduction path starting with a leads to a term in R. The set $\mathcal{S}$ of sound terms is the expansion $(\leadsto^\star \mho)$ of valid terms.

Head normal forms Δ are terms whose root node is a constructor, *i.e.* abstractions $\lambda x\, a$ and pairs $\langle a, a\rangle$, while neutral terms ∇ are variables x, applications $a\, a$, and projections $\mathsf{fst}\, a$ and $\mathsf{snd}\, a$. Notice that Δ and ∇ are complement of one another, *i.e.* terms are the disjoint union of Δ and ∇. To remember which notation is associated to which notion, we can see Δ as stable (it has a large basis) and ∇ as unstable (it has a pointy basis). Stability, in our analogy, means that we have a head constructor which cannot change by reduction, while instability means that reduction can modify our root node, to finally become a head constructor. Notice also that, like for errors and valid terms, head normal forms and neutrals are 180 degrees of one another because they are complement of one another.

Progress is a way to double-check the definition of the semantics, by defining values syntactically and checking that semantic values (irreducible valid terms) are syntactic values. Prevalues are simply neutral values.

Lemma 70 (Progress)**.** *If $a \in \mho$ and $a \not\leadsto$, then a is of the form v.*

Proof. By induction on a. For each case, either it is an error, or it can reduce, or it is a value. $\square$

$$
\begin{array}{lll}
x, y & & \text{Term variables} \\
e, f & ::= x^k \mid \lambda^k x\, e \mid (e\, e)^k \mid \langle e, e\rangle^k \mid \mathsf{fst}^k\, e \mid \mathsf{snd}^k\, e & \text{Terms}
\end{array}
$$

Figure 5.11: Indexed Calculus syntax

$$
\begin{array}{rcll}
E^k & ::= & \lambda^k x\, [] \mid ([]\, e)^k \mid (e\, [])^k \mid \langle [], e\rangle^k \mid \langle e, []\rangle^k \mid \mathsf{fst}^k\, [] \mid \mathsf{snd}^k\, [] & \text{Evaluation contexts} \\
s & ::= & E^k[s] \mid (\langle s, s\rangle^k\, s)^k \mid \mathsf{fst}^k\, (\lambda^k x\, s) \mid \mathsf{snd}^k\, (\lambda^k x\, s) & \text{Errors}
\end{array}
$$

Figure 5.12: Indexed Calculus notations

The converse is also true, *i.e.* values do not contain errors. However, this won't remain true when we restrict the strategy, *e.g.* to call-by-value weak reduction. In this case, redefining the grammar of values, progress still holds, but some grammatically well-formed values may contain "inaccessible" errors, such as errors occurring under an abstraction.

Type soundness states that well-typed terms are sound. We prove this by interpreting syntactic types as semantic types which are themselves sets of terms. However, since we allow general recursive types the evaluation of terms may not terminate. This is not a problem, since type soundness is not about termination, but ruling out unsound terms, which if they reach an error do so in a finite number of steps.

The idea of step-indexed techniques is to stop the reduction after a certain number of steps, as if some initially available fuel (the number of allowed reduction steps) had all been consumed. Since errors are necessarily reached after a finite number of steps, we may always detect errors with some finite but arbitrary large number of reduction steps.

However, there is a difficulty applying this technique with strong reduction strategies, which we solve by including the fuel inside terms, called indexed terms, and block the reduction internally when terms do not have enough fuel, rather than control the number of reduction steps externally. The difficulty of using step indices in a strong reduction setting is that the usual semantic of the arrow type is not stable by reduction.

5.2.1 The Indexed Calculus

Terms of the Indexed Calculus are terms of the λ-calculus where each node is annotated with a natural number called the index (or fuel) of this node. They are written with letter f or e and formally defined on Figure 5.11. Indexed terms are variables x^k, abstractions $\lambda^k x\, e$, applications $(e\, e)^k$, pairs $\langle e, e\rangle^k$, and projections $\mathsf{fst}^k\, e$ and $\mathsf{snd}^k\, e$.

As for the λ-calculus, we define the indexed errors. We write them s and define them on Figure 5.12. They contain terms which contain an error under a context and terms which are the: application of a pair or the projection of an abstraction, similarly to lambda errors. The indices are ignored in this definition. We write E^k for one-hole context with index k. We have all contexts since we are in strong reduction. Finally we define neutrals and head normal forms which are complement of one another according to the set of indexed terms. Neutrals are variables x^k, applications $(e\, e)^k$, and projections $\mathsf{fst}^k\, e$ and $\mathsf{snd}^k\, e$, while head normal forms are abstractions $\lambda^k x\, e$ and pairs $\langle e, e\rangle^k$.

We write ∇ for neutrals, Δ for head normal forms, Ω for indexed errors, and $\mho$ for valid indexed terms, by overloading the notation of the λ-calculus. The overloading is not ambiguous because we use distinct meta-variables for lambda terms and indexed terms. Moreover, these

$$
\begin{array}{rclcrcl}
\lfloor x^k \rfloor_j &=& x^{kj} & \quad & \lfloor \langle e, f\rangle^k \rfloor_j &=& \langle \lfloor e \rfloor_j, \lfloor f \rfloor_j \rangle^{kj} \\
\lfloor \lambdabar^k x\, e \rfloor_j &=& \lambdabar^{kj} x\, \lfloor e \rfloor_j & & \lfloor \mathsf{fst}^k\, e \rfloor_j &=& \mathsf{fst}^{kj}\, \lfloor e \rfloor_j \\
\lfloor (e\, f)^k \rfloor_j &=& (\lfloor e \rfloor_j\, \lfloor f \rfloor_j)^{kj} & & \lfloor \mathsf{snd}^k\, e \rfloor_j &=& \mathsf{snd}^{kj}\, \lfloor e \rfloor_j
\end{array}
$$

Figure 5.13: Lower function

RedCtx
$$\frac{e \rightsquigarrow f}{E^{k+1}[e] \rightsquigarrow E^k[f]}$$

RedApp
$$((\lambdabar^{j+1} x\, e)\, f)^{k+1} \rightsquigarrow \lfloor e[x/f] \rfloor_{kj}$$

RedFst
$$\mathsf{fst}^{k+1}\, \langle e, f\rangle^{j+1} \rightsquigarrow \lfloor e \rfloor_{kj}$$

RedSnd
$$\mathsf{snd}^{k+1}\, \langle e, f\rangle^{j+1} \rightsquigarrow \lfloor f \rfloor_{kj}$$

Figure 5.14: Indexed Calculus reduction relation

sets are closely related. The set of neutral lambda terms and the set of neutral indexed terms are the same up to the erasure of indices, and similarly for head normal forms, errors, and valid terms.

Intuitively, indices indicate the maximum number of reduction steps allowed under the given node. We want this invariant to hold during reduction, so reduction has to update indices. Since redexes usually delete several nodes of the left-hand part and reorganize subterms for the right-hand side, we must remember the indices of the removed nodes one way or another on their subterms. The easiest solution is to take the minimum of indices of the redex nodes and lower their subterms with this minimum. To do so we use an auxiliary lowering function on indexed terms. It is written $\lfloor e \rfloor_j$ and defined on Figure 5.13 (Coq definition `Flanguage.lower`). We use concatenation of indices to denote the minimum of their values. This is not ambiguous since we never use multiplication of indices. Lowering simply changes all indices in the term e with their minimum with j. With this definition, the β-reduction rule now becomes: $((\lambdabar^{j+1} x\, e)\, f)^{k+1} \rightsquigarrow \lfloor e[x/f] \rfloor_{kj}$.

The capture avoiding substitution $e[x/f]$ of term f for variable x in the term e replaces in e all free occurrences x^j of x by $\lfloor f \rfloor_j$. The definition is generalized in the obvious way to simultaneous substitutions. We use letter γ to range over substitutions. The lowering of substituted occurrences is necessary to make substitution commute with the lowering function. In particular, renaming commutes with the lowering function.

Lemma 71. $\lfloor e[x/f] \rfloor_k = \lfloor e \rfloor_k [x/f] = \lfloor e \rfloor_k [x/\lfloor f \rfloor_k]$

Proof. The first equality is proved with the Coq lemma `lower_subst` and the last equality is proved with `subst_lower` in file `Flanguage.v`. □

The reduction rules of the Indexed Calculus mimic those of the λ-calculus, but with some index manipulation, as described in Figure 5.14 (Coq inductive `Flanguage.red`). Reduction can only proceed when the indices on the nodes involved in the reduction are strictly positive; the indices are lowered after reduction by the minimum of the involved indices decremented by one. As a corollary, reduction cannot occur at or under a node with a null index. This applies both to head reduction rules (RedApp, RedFst, and RedSnd) and to reductions in an evaluation context (Rule RedCtx). That is, a head reduction can only be applied along a path of the form $E_1^{k_1}[\dots E_p^{k_p}[e]]$ when indices k_i's are all strictly positive; they are all decremented after the reduction.

For example, here is a decorated reduction of apply (the lambda term $\lambda x\, \lambda y\, x\, y$) applied to two terms e and f:

$$\begin{array}{ll} & (((\lambda^{k_3+1} x\, \lambda^{j_1} y\, (x^{j_3}\, y^{j_4})^{j_2})\, e)^{k_2+1}\, f)^{k_1+1} \\ \rightsquigarrow & ((\lambda^{j_1 k_2 k_3} y\, (\lfloor e \rfloor_{j_3 k_2 k_3}\, y^{j_4 k_2 k_3})^{j_2 k_2 k_3})\, f)^{k_1} \end{array}$$

Since the reduction happens under the external application, it must have some fuel $k_1 + 1$, which is decreased by one in the result. Then, for the redex to fire, the application must have some fuel $k_2 + 1$ as well as the abstraction $k_3 + 1$, which are both decreased by one and combined as $k_2 k_3$ to lower the result of the reduction. Before that, the term e, which has been substituted for x^{j_3} has been lowered to j_3 in the result. The important feature is that f has not been lowered, which is an important difference with what would happen with the traditional step-indexed approach when indices are outside terms.

Strong normalization By design, the Indexed Calculus is strongly normalizing, *i.e.* all reduction paths of all terms are finite. In particular, they are bounded by the index of their root node.

Lemma 72 (Strong normalization). *The indexed calculus is strongly normalizing.*

Proof. The proof is done in Coq lemma `Fnormalization.wf_der` and uses the measure that associates for each indexed term its root index. □

5.2.2 Bisimulation

To show that reduction between undecorated terms and decorated terms coincides, we define $\lfloor e \rfloor$ (without subindex) the erasure of an indexed term e obtained by dropping all indices. We lift this function to sets of terms: $\lfloor \mathsf{R} \rfloor$ is the set $\{\lfloor e \rfloor \mid e \in \mathsf{R}\}$. By construction, dropping is stronger than lowering, *i.e.* dropping after lowering is the same as dropping, $\lfloor \lfloor e \rfloor_j \rfloor = \lfloor e \rfloor$. As for lowering, dropping commutes with substitution: $\lfloor e[x/f] \rfloor = \lfloor e \rfloor [x/\lfloor f \rfloor]$. We overload the notation $\mathcal{S}$ for the set of sound indexed terms. Although it is defined as for lambda terms as $(\rightsquigarrow^{\star} \mho)$, the meaning is different since the reduction is now bridled by indices.

The calculus on indexed terms is just an instrumentation of the λ-calculus that behaves the same up to the consumption of all the fuel. Formally, we show that they can simulate one another, up to some condition on the indices.

Indexed terms can be simulated by lambda terms. That is, if an indexed term can reduce, then the same reduction step can be performed after dropping indices.

Lemma 73 (Forward simulation). *If $e \rightsquigarrow f$, then $\lfloor e \rfloor \rightsquigarrow \lfloor f \rfloor$.*

Proof. The proof is in the Coq lemma `Llanguage.red_drop`. □

In order to make the other direction concise, we lift predicates on integers to predicates on indexed terms by requiring the predicate to hold for all indices occurring in the term. For instance, $e > 0$ means that the index of all the nodes of e are greater than zero while $e \leq k$ means that the indices in e are smaller or equal to k. This is formally defined on Figure 5.15 (Coq definition `Flanguage.unary_fuel`).

$$
\begin{array}{rlrl}
\mathsf{P}\,x^k &= \mathsf{P}\,k & \mathsf{P}\,\langle e, f\rangle^k &= \mathsf{P}\,k \wedge \mathsf{P}\,e \wedge \mathsf{P}\,f \\
\mathsf{P}\,(\lambda^k x\, e) &= \mathsf{P}\,k \wedge \mathsf{P}\,e & \mathsf{P}\,(\mathsf{fst}^k\, e) &= \mathsf{P}\,k \wedge \mathsf{P}\,e \\
\mathsf{P}\,(e\, f)^k &= \mathsf{P}\,k \wedge \mathsf{P}\,e \wedge \mathsf{P}\,f & \mathsf{P}\,(\mathsf{snd}^k\, e) &= \mathsf{P}\,k \wedge \mathsf{P}\,e
\end{array}
$$

Figure 5.15: Lifting integer predicates to indexed terms

$$
\begin{aligned}
x^{k_1} \star x^{k_2} &= k_1 \star k_2 \\
(\lambda^{k_1} x\, e_1) \star (\lambda^{k_2} x\, e_2) &= k_1 \star k_2 \wedge e_1 \star e_2 \\
(e_1\, f_1)^{k_1} \star (e_2\, f_2)^{k_2} &= k_1 \star k_2 \wedge e_1 \star e_2 \wedge f_1 \star f_2 \\
\langle e_1, f_1\rangle^{k_1} \star \langle e_2, f_2\rangle^{k_2} &= k_1 \star k_2 \wedge e_1 \star e_2 \wedge f_1 \star f_2 \\
\mathsf{fst}^{k_1}\, e_1 \star \mathsf{fst}^{k_2}\, e_2 &= k_1 \star k_2 \wedge e_1 \star e_2 \\
\mathsf{snd}^{k_1}\, e_1 \star \mathsf{snd}^{k_2}\, e_2 &= k_1 \star k_2 \wedge e_1 \star e_2
\end{aligned}
$$

Figure 5.16: Lifting of a binary predicate $\star$ on indices to terms

Indexed terms can simulate lambda terms, provided they have enough fuel. This means that if an indexed term has strictly positive indices and can be reduced after dropping its indices, then the same reduction step can be performed on the indexed term.

Lemma 74 (Backward simulation)**.** *If $e > 0$ and $\lfloor e \rfloor \rightsquigarrow a'$, then there exists e' such that $e \rightsquigarrow e'$ and $\lfloor e' \rfloor = a'$.*

Proof. The proof is in the Coq lemma `Llanguage.drop_red_exists`. □

Using this last lemma, we can show that when a sound indexed term e has indices greater or equal to k, then its erasure can safely do at least k steps. This lemma is crucial to transfer the soundness result in the indexed calculus to the λ-calculus. A lambda term is sound if it is sound for all number of steps.

Lemma 75. *If $e \geq k$ and $e \in \mathcal{S}$ hold, then $\lfloor e \rfloor$ is sound for all paths of size smaller than k steps.*

Proof. The proof is in the Coq lemma `Lsoundness.term_ge_OK`. □

5.2.3 Semantic types

In order to define semantic types concisely, it is convenient to have a few helper operations on sets of indexed terms. We first lift binary properties on indices to indexed terms. This is done by asking the two terms to share the same skeleton (they drop on the same lambda term) and the indices of corresponding nodes to be related by the property on indices. A formal definition is given on Figure 5.16 (Coq definition `Flanguage.binary_fuel`).

The *interior* of a set R is the set $\mathsf{R}{\downarrow}$ containing all terms smaller than a term in R, *i.e.* $\{f \mid \exists e \in \mathsf{R}, f \leq e\}$ (Coq definition `Fsemantics.Dec`). The *contraction* of a set R is the set $(\mathsf{R}{\rightsquigarrow})$ of all terms obtained by one-step reduction of a term in R, *i.e.* $\{f \mid \exists e \in \mathsf{R}, e \rightsquigarrow f\}$ (Coq definition `Fsemantics.Red`).

A *pretype* is a set of sound terms that contains both its interior and its contraction (Coq definition `Fsemantics.C`). We write $\mathcal{C}$ the set of pretypes.

Definition 76 (Pretypes)**.** $\mathcal{C} \stackrel{\text{def}}{=} \{\mathsf{R} \subseteq \mathcal{S} \mid \mathsf{R}{\downarrow} \cup (\mathsf{R}{\leadsto}) \subseteq \mathsf{R}\}$

Notice that the empty set and $\mathcal{S}$ are pretypes. Pretypes only contain sound terms since types are pretypes and types are sets of sound terms. The closure of pretypes by interior is just technical: if a property holds for some index, we want it to also hold for smaller indices. The main property of pretypes is to be closed by reduction. Types are pretypes that are also closed by a form of expansion. As a first approximation, sound terms that reduce to a term in a type R should also be in R. However, a type R should still not contain unsound terms even if these reduce to some term in R. Moreover, the meaning of a set of terms R is in essence determined by its set of head normal forms, which we call the kernel of R. We use concatenation for intersection of sets of terms. Hence, the kernel of R is $\Delta\mathsf{R}$. A type R needs not to contain every head normal form that reduces to some term in R. Consider for example the term e_0 equal to $\lambda x\, x$ and one of its expansion is the term e_1 equal to $\lambda x\, (\lambda y\, x)\, (x\, x)$. The sets $\{e_0\}$ and $\{e_0, e_1\}$ have quite different meanings The second set has additional requirements for the argument: it should be sound to apply it to itself. Notice that by definition, the kernel is an idempotent operation: $\Delta(\Delta\mathsf{R}) = \Delta\mathsf{R}$.

The expansion-closure of a set of terms R, written $\diamondsuit\,\mathsf{R}$, is the set $({\leadsto}^\star(\nabla\mho \uplus \Delta\mathsf{R}))$, which contains terms of which every reduction path leads to either a valid neutral term or a head normal form of R. By definition, the expansion closure is monotonic: if $\mathsf{R} \subseteq \mathsf{S}$, then $\diamondsuit\,\mathsf{R} \subseteq \diamondsuit\,\mathsf{S}$; it is also idempotent: $\diamondsuit\,(\diamondsuit\,\mathsf{R}) = \diamondsuit\,\mathsf{R}$.

Finally, semantic types are pretypes that are stable by expansion closure (Coq definition `Fsemantics.CE`):

Definition 77 (Semantic types)**.** $\mathbb{T} \stackrel{\text{def}}{=} \{\mathsf{R} \in \mathcal{C} \mid \diamondsuit\,\mathsf{R} \subseteq \mathsf{R}\}$.

The kernel of a type is a pretype—but not a type. Conversely, the expansion-closure of a pretype is a type. Actually expansion-closure and kernel are almost inverse of one another: if R is a type, then $\diamondsuit\,(\Delta\mathsf{R}) = \mathsf{R}$.

The smallest type, called the bottom type and written $\bot$, is equal to $\diamondsuit\,\{\}$, that is $({\leadsto}^\star(\nabla\mho))$. The largest type, $\top$, called the top type is the set $\mathcal{S}$ of sound terms.

5.2.4 Simple types

We can now define the semantics of arrows and products as semantic type operators. We overload the arrow and product notations, but there is no ambiguity with the syntactic type operators since the semantic operators take semantic types R or S, while syntactic type operators take syntactic types τ or σ as arguments.

Definition 78 (Arrow and product operators)**.**
$\mathsf{R} \to \mathsf{S} \stackrel{\text{def}}{=} \diamondsuit\,\{\lambda^k x\, e \in \mathcal{S} \mid k > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x/f] \rfloor_{k-1} \in \mathsf{S}\}$
$\mathsf{R} \times \mathsf{S} \stackrel{\text{def}}{=} \diamondsuit\,\left\{\langle e, f\rangle^k \in \mathcal{S} \mid k > 0 \Rightarrow \lfloor e \rfloor_{k-1} \in \mathsf{R} \wedge \lfloor f \rfloor_{k-1} \in \mathsf{S}\right\}$

The arrow semantic operator is the expansion closure of sound term abstractions $\lambda^k x\, e$ satisfying the following property when the index k is non-zero. For all argument f such that its lowering by $k-1$ is in the domain R, the lowering by $k-1$ of the substitution of x by f in e has to be in the range S. The product semantic operator is similar. It is the expansion closure of sound pairs satisfying the following property for non-zero indices k. Each component of the pair has to be in its associated type after lowering by $k-1$.

In order to prove that arrow and product semantic type operators preserves type, we need to define and prove the following easy properties on indices:

- $\lfloor\lfloor e\rfloor_j\rfloor_k = \lfloor e\rfloor_{kj}$ (Coq lemma `Flanguage.lower_lower`)
- If $k' \leq k$ and $e' \leq e$, then $\lfloor e'\rfloor_{k'} \leq \lfloor e\rfloor_k$. (Coq lemma `Flanguage.le_term_lower`)
- If $e' \leq e$ and $f' \leq f$, then $e'[x/f'] \leq e[x/f]$. (Coq lemma `Flanguage.le_term_subst`)

And this less easy one:

Lemma 79. *If $e \rightsquigarrow f$ holds, then for all k, $\lfloor e\rfloor_{k+1} \rightsquigarrow f'$ and $\lfloor f\rfloor_k \leq f'$ hold for some f'.*

Proof. Coq lemma `Flanguage.red_lower` □

The arrow and product operators preserve types.

Lemma 80. *If R and S are types, then so are $\mathsf{R} \to \mathsf{S}$ and $\mathsf{R} \times \mathsf{S}$.*

Proof. Coq lemma `CE_EArr` and `CE_EProd` in `Fsemantics.v`.

We only detail the proof for the arrow operator, which uses indexed terms in a crucial way. The proof for the product operator is similar, but easier. Since the arrow operator is defined by expansion-closure, it is a type if its kernel is a pretype. Its kernel contains only sound terms by definition. So it remains to show that the definition contains its interior and contraction.

Let $\lambda^j x\, e' \leq \lambda^k x\, e$ (**1**), $\lambda^k x\, e \in \mathcal{S}$ (**2**), and $k > 0 \Rightarrow \forall f, \lfloor f\rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x/f]\rfloor_{k-1} \in \mathsf{S}$ (**3**), and show that $\lambda^j x\, e' \in \mathcal{S}$ (**4**) and $j > 0 \Rightarrow \forall f, \lfloor f\rfloor_{j-1} \in \mathsf{R} \Rightarrow \lfloor e'[x/f]\rfloor_{j-1} \in \mathsf{S}$ (**5**). The first assertion (4) comes easily with (1) and (2) since $\mathcal{S}$ contains its interior. To show (5), let $j > 0$ and $\lfloor f\rfloor_{j-1} \in \mathsf{R}$ (**6**) and show $\lfloor e'[x/f]\rfloor_{j-1} \in \mathsf{S}$ (**7**). By (1) we have $j \leq k$, so $k > 0$. We also have $\lfloor\lfloor f\rfloor_{j-1}\rfloor_{k-1} = \lfloor f\rfloor_{j-1}$ which is in R by (6). So from (3) we have $\lfloor e[x/\lfloor f\rfloor_{j-1}]\rfloor_{k-1} \in \mathsf{S}$. Since S is a type, it contains its interior so $\lfloor e'[x/\lfloor f\rfloor_{j-1}]\rfloor_{j-1} \in \mathsf{S}$. Since the substitution and the lowering function commute, we conclude (7).

Let $\lambda^k x\, e \rightsquigarrow e_1$ (**8**), $\lambda^k x\, e \in \mathcal{S}$ (**9**), and $k > 0 \Rightarrow \forall f, \lfloor f\rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x/f]\rfloor_{k-1} \in \mathsf{S}$ (**10**). By inversion of the reduction relation we have $k = k' + 1$ and $e_1 = \lambda^{k'} x\, e'$ for some k' and e' such that $e \rightsquigarrow e'$ (**11**). We now have to show that $\lambda^{k'} x\, e' \in \mathcal{S}$ (**12**) and $k' > 0 \Rightarrow \forall f, \lfloor f\rfloor_{k'-1} \in \mathsf{R} \Rightarrow \lfloor e'[x/f]\rfloor_{k'-1} \in \mathsf{S}$ (**13**). We show (12) with (8) and (9) since $\mathcal{S}$ contains its contraction. To show (13), let $k' > 0$ and $\lfloor f\rfloor_{k'-1} \in \mathsf{R}$ (**14**) and show $\lfloor e'[x/f]\rfloor_{k'-1} \in \mathsf{S}$ (**15**). We have $\lfloor\lfloor f\rfloor_{k'-1}\rfloor_{k-1} = \lfloor f\rfloor_{k'-1}$ which is in R by (14). So from (10) we have $\lfloor e[x/\lfloor f\rfloor_{k'-1}]\rfloor_{k-1} \in \mathsf{S}$. Since S is a type, it contains its contraction and interior so $\lfloor e'[x/\lfloor f\rfloor_{k'-1}]\rfloor_{k'-1} \in \mathsf{S}$ by Lemma 79. Since the substitution and the lowering function commute, we conclude (15). □

5.2.5 Intersection types

The intersection $\bigcap_{i\in\mathsf{I}} \mathsf{R}_i$ of a nonempty family of types $(\mathsf{R}_i)^{i\in I}$ is a type. As a particular case, the bottom type $\bot$ is also the intersection of all types. We extend this definition to the empty family with the top type, which is the set of sound terms. Because all types are sets of sound terms, we can define the intersection operator of a family of types as the intersection with the set of sound terms (Coq definition `Fsemantics.EFor`).

Definition 81 (Intersection operator)**.** *We define $\forall\mathsf{I}\,\mathsf{F}$ as the set of terms $\mathcal{S} \cap \bigcap_{i\in\mathsf{I}} \mathsf{F}\,i$.*

The intersection operator preserves semantic types:

Lemma 82. *If for all $i \in \mathsf{I}$, $\mathsf{F}\,i$ is a semantic type, then $\forall\mathsf{I}\,\mathsf{F}$ is a semantic type.*

Proof. Coq lemma `Fsemantics.CE_EFor` □

5.2.6 Recursive types

This section follows the usual description of recursive types using the notion of approximations as done in [3]. Adding recursive types is the main reason for using a step-indexed technique, assuming we have a semantics approach due to our notion of coherent abstraction. While recursive types motivate the use of step-indexed semantics, they do not raise any additional difficulty once the semantics has been correctly set up.

The k-approximation of a set R, written $\langle \mathsf{R} \rangle_k$ is the subset $\{e \in \mathsf{R} \mid e < k\}$ of element of R that are smaller than k (Coq definition `Fsemantics.approx`). The following properties of approximations immediately follow from the definition: $\langle \mathsf{R} \rangle_0$ is the empty set; a sequence of approximations is the approximation by the minimum of the sequence: $\langle \langle \mathsf{R} \rangle_j \rangle_k = \langle \mathsf{R} \rangle_{jk}$; two sets of terms that are equal at all approximations are equal: if $\langle \mathsf{R} \rangle_k = \langle \mathsf{S} \rangle_k$ holds for all k, then $\mathsf{R} = \mathsf{S}$.

Definition 83 (Well-foundness)**.** *A function F on sets of terms is well-founded (resp. non-expansive) if for any set of terms R, the approximations of $\mathsf{F}\,\mathsf{R}$ and $\mathsf{F}\,\langle \mathsf{R} \rangle_k$ are equal at rank $k+1$ (resp. k),* i.e. $\langle \mathsf{F}\,\mathsf{R} \rangle_{k+1} = \langle \mathsf{F}\,\langle \mathsf{R} \rangle_k \rangle_{k+1}$ *(resp. $\langle \mathsf{F}\,\mathsf{R} \rangle_k = \langle \mathsf{F}\,\langle \mathsf{R} \rangle_k \rangle_k$)*

Intuitively, well-foundness (resp. non-expansiveness) ensures that given terms sound for k steps, F returns terms sound for $k+1$ (resp. k) steps. The Coq definition of well-foundness (resp. non-expansiveness) is given in `WF` (resp. `NE`) in `Fsemantics.v`.

The iteration of a well-founded function F does not look at its argument for terms of small indices: $\langle \mathsf{F}^k\,\mathsf{R} \rangle_k$ is independent of R; in particular, it is equal to $\langle \mathsf{F}^k\,\bot \rangle_k$. This lemma helps us to prove the more useful lemmas that allows us to prove that a term is in the kth iteration of a well-founded functor, given it is already in the jth iteration and smaller than both j and k: $\langle \mathsf{F}^j\,\mathsf{R} \rangle_{kj}$ and $\langle \mathsf{F}^k\,\mathsf{R} \rangle_{kj}$ are equal.

Definition 84 (Recursive operator)**.** *Given a well-founded function F on sets of terms, we define $\mu\,\mathsf{F}$ as the set of terms $\bigcup_{k \geq 0} \langle \mathsf{F}^k \bot \rangle_k$.*

The recursive operator preserves semantic types:

Lemma 85. *If F is well-founded and maps semantic types to semantic types, then $\mu\,\mathsf{F}$ is a semantic type.*

Proof. Coq lemma `Fsemantics.CE_EMu` □

Moreover, recursive types can be unfolded or folded as expected: if F is well-founded, then $\mu\,\mathsf{F} = \mathsf{F}\,(\mu\,\mathsf{F})$. This is proved by showing that $\langle \mu\,\mathsf{F} \rangle_k$ is equal to both $\langle \mathsf{F}^k\,\bot \rangle_k$ and $\langle \mathsf{F}\,(\mu\,\mathsf{F}) \rangle_k$ for every k. The proofs are done in Coq lemma `Mu_fold` and `Mu_unfold` in file `Fsemantics.v`.

The following Lemma, although in a different settings, is stated exactly as with traditional step-indexed semantics [3]:

Lemma 86. *We have the following properties:*

- *Every well-founded function is non-expansive.*
- $\mathsf{X} \mapsto \mathsf{X}$ *is non-expansive.*
- $\mathsf{X} \mapsto \mathsf{R}$ *where* X *is unused in* R *(*R *is constant) is well-founded.*
- *The composition of non-expansive functors is non-expansive.*

- *The composition of a non-expansive functor with a well-founded functor (in either order) is well-founded.*
- *If* F *and* G *are non-expansive, then* $\mathsf{X} \mapsto (\mathsf{F}\,\mathsf{X} \to \mathsf{G}\,\mathsf{X})$ *and* $\mathsf{X} \mapsto (\mathsf{F}\,\mathsf{X} \times \mathsf{G}\,\mathsf{X})$ *are well-founded.*
- *If* $(\mathsf{F}_i)^{i \in I}$ *is a family of non-expansive (resp. well-founded) functors, then the functor* $\mathsf{X} \mapsto \bigcap_{i \in I} (\mathsf{F}_i\,\mathsf{X})$ *is non-expansive (resp. well-founded).*
- *If* $\mathsf{X} \mapsto \mathsf{F}\,\mathsf{X}\,\mathsf{Y}$ *is non-expansive (resp. well-founded) for every* Y *and* $\mathsf{F}\,\mathsf{X}$ *is well founded for every* X*, then* $\mathsf{X} \mapsto \mu\,(\mathsf{F}\,\mathsf{X})$ *is non-expansive (resp. well-founded).*

Proof. Lemma `NE_id`, `WF_CST`, `WF_PArr`, `WF_PProd`, `WFj_For`, and `WFj_Mu` in the `Fsemantics.v` Coq file. □

Just for illustration $\mathsf{X} \mapsto \mathsf{X} \to \mathcal{S}$ is well-founded since $\mathsf{X} \mapsto \mathsf{X}$ is non-expansive and $\mathsf{X} \mapsto \mathcal{S}$ is constant, thus well-founded, and therefore non-expansive.

5.2.7 Semantic judgment

A binding is a pair $(x : \mathsf{R})$ of a variable and a semantic type. A context G is a set of bindings, defining a finite mapping from term variables to semantic types. We say that a substitution γ is *compatible* with a context G, written $\gamma : \mathsf{G}$, if $\mathsf{dom}(\gamma)$ and $\mathsf{dom}(\mathsf{G})$ coincide and for all $(x : \mathsf{R})$ in G, the term $\gamma\,x$ is in R.

We define the semantic judgment $\mathsf{G} \models \mathsf{S}$ as the set of terms e such that $\gamma\,e$ is in S for any substitution γ "compatible" with G (Coq definition `Fsemantics.EJudg`).

Definition 87 (Semantic judgment)**.**

$$\begin{array}{rcl} \gamma : \mathsf{G} & \overset{\mathit{def}}{=} & \forall (x : \mathsf{R}) \in \mathsf{G}, \gamma\,x \in \mathsf{R} \\ \mathsf{G} \models \mathsf{S} & \overset{\mathit{def}}{=} & \{e \mid \forall \gamma : \mathsf{G}, \gamma\,e \in \mathsf{S}\} \end{array}$$

We may now present the semantic typing rules for the STLC.

Lemma 88 (Variable)**.** *If* R *is a type and* $(x : \mathsf{R})$ *is in* G*, then* x^k *is in* $\mathsf{G} \models \mathsf{R}$*.*

Proof. Coq lemma `Fsemantics.EVar_sem`

Let γ be compatible with G (**1**). We show that $\gamma\,x^k$ is in R. Since $(x : \mathsf{R})$ is in G, we have $\gamma\,x$ in R by (1), Being a type, R is closed by lowering. Hence, $\lfloor \gamma\,x \rfloor_k$ is also in R. By definition of substitution, this is equal to $\gamma\,x^k$, which is thus also in R. □

Lemma 89 (Abstraction)**.** *If* R *and* S *are types and* e *is in* $\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S}$*, then* $\lambda^k x\,e$ *is in* $\mathsf{G} \models \mathsf{R} \to \mathsf{S}$*.*

Proof. Coq lemma `Fsemantics.ELam_sem`

Let γ be compatible with G (**1**). We show that $\gamma\,(\lambda^k x\,e)$ is in $\mathsf{R} \to \mathsf{S}$ (**2**). Assume $\gamma\,(\lambda^k x\,e) \leadsto^\star e_1$. Then e_1 is necessarily of the form $\lambda^j x\,e'$ where $\gamma\,e \leadsto^\star e'$.

We first show that $\lambda^j x\,e' \in \mathcal{S}$ (**3**). Since γ is compatible with G, $\gamma, x \mapsto x$ is compatible with $\mathsf{G}, (x : \mathsf{R})$ as variables are in all types. Since e is in $(\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S})$, we have $(\gamma, x \mapsto x)\,e$, *i.e.* $\gamma\,e$ in S. Since S is closed by reduction, we have e' in S and a fortiori in $\mathcal{S}$. This implies (3).

Assume $j > 0$ and $\lfloor f \rfloor_{j-1} \in \mathsf{R}$. Let γ' be $\gamma, x \mapsto \lfloor f \rfloor_{j-1}$. By construction $\gamma' : \mathsf{G}, (x : \mathsf{R})$. Since e is in $(\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S})$, we have $\gamma' e$ in S and, since S is closed by reduction, $e'[x/\lfloor f \rfloor_{j-1}]$ is also in S. By decreasing index we have $\lfloor e'[x/\lfloor f \rfloor_{j-1}] \rfloor_{j-1} \in \mathsf{S}$, from which by Lemma 71 becomes $\lfloor e'[x/f] \rfloor_{j-1} \in \mathsf{S}$. This ends the proof of (2). □

Lemma 90 (Application). *If* R *and* S *are types,* e *is in* $\mathsf{G} \models \mathsf{R} \to \mathsf{S}$*, and* f *is in* $\mathsf{G} \models \mathsf{R}$*, then* $(e\, f)^k$ *is in* $\mathsf{G} \models \mathsf{S}$ *for any* k*.*

Proof. Coq lemma `Fsemantics.EApp_sem`

Let γ be compatible with G. We show that $\gamma\,(e\, f)^k \in \mathsf{S}$. By hypotheses we have $\gamma\, e \in \mathsf{R} \to \mathsf{S}$ and $\gamma\, f \in \mathsf{R}$. We prove the more general result that for all k, e, and f, if $e \in \mathsf{R} \to \mathsf{S}$ and $f \in \mathsf{R}$ hold, then $(e\, f)^k \in \mathsf{S}$ also holds. This is proved is by induction over the strong normalization of e and f using the closure expansion of S.

The term $(e\, f)^k$ is neutral. It is also valid since e and f are sound and, by construction of $\mathsf{R} \to \mathsf{S}$, e is an abstraction when in normal form. If $(e\, f)^k$ reduces by a context rule, we use our induction hypothesis. Otherwise, e must be of the form $\lambda^{j+1} x\, e'$ for some j and e' and k be of the form $k' + 1$ and the reduction is $(e\, f)^k \rightsquigarrow \lfloor e'[x/f] \rfloor_{jk'}$. It remains to show $\lfloor e'[x/f] \rfloor_{jk'} \in \mathsf{S}$ (**1**). We have $\lfloor f \rfloor_j \in \mathsf{R}$ by stability under decreasing index. So, we have $\lfloor e'[x/f] \rfloor_j \in \mathsf{S}$ by definition of the arrow operator. Then (1) follows by stability under decreasing index. □

Lemma 91 (Pair). *If* R_i *is a type and* e_i *is in* $\mathsf{G} \models \mathsf{R}_i$*, then* $\langle e_1, e_2 \rangle^k$ *is in* $\mathsf{G} \models \mathsf{R}_1 \times \mathsf{R}_2$*.*

Proof. Coq lemma `Fsemantics.EPair_sem` □

Lemma 92 (Projections). *If* R *and* S *are types and* e *in* $\mathsf{G} \models \mathsf{R} \times \mathsf{S}$*, then* $\mathsf{fst}^k\, e$ *is in* $\mathsf{G} \models \mathsf{R}$ *and* $\mathsf{snd}^k\, e$ *is in* $\mathsf{G} \models \mathsf{S}$*.*

Proof. Coq lemma `EFst_sem` and `ESnd_sem` in file `Fsemantics.v`. □

Note that when R is a type for all $(x : \mathsf{R}) \in \mathsf{G}$ and S is a type, then $\mathsf{G} \models \mathsf{S}$ is a pretype.

5.3 Soundness

The soundness proof is not direct. We translate L^{c} type system from the λ-calculus to a temporary type system on the indexed calculus. We prove soundness for the indexed calculus type system and migrate the result to the λ-calculus type system. The relation between both type systems is that if a lambda term is well-typed then all indexed terms that drop on this lambda term are well-typed too. And reciprocally, if an indexed term is well-typed, then its dropped lambda term is well-typed too. Both directions preserve the typing (the pair of the environment and type). Notice that only the term judgment needs to be changed since it is the only one talking about terms.

Syntactically, the indexed term judgment $e : \Sigma; \Gamma \vdash \tau$ contains the exact same rules as those of the lambda term judgment. However index annotations now appear on the term node we are typing. This annotation has no constraint, which gives us that if a term is typed with annotations it can be typed without and reciprocally if a term is typed without annotations it can be typed with any annotations.

Lemma 93. *The following assertions hold:*

$$
\begin{array}{rcl}
|\star|_\eta & = & \mathbb{T} \\
|1|_\eta & = & \{\langle\rangle\} \\
|\kappa_1 \times \kappa_2|_\eta & = & \left\{\langle x_1, x_2\rangle \mid x_1 \in |\kappa_1|_\eta \wedge x_2 \in |\kappa_2|_\eta\right\} \\
|\{\alpha : \kappa \mid \mathrm{P}\}|_\eta & = & \left\{x \mid x \in |\kappa|_\eta \wedge \forall k\ |\mathrm{P}|^k_{\eta,\alpha\mapsto x}\right\}
\end{array}
$$

Figure 5.17: Kind interpretation

- *If* $e : \Sigma; \Gamma \vdash \tau$ *holds, then* $\lfloor e \rfloor : \Sigma; \Gamma \vdash \tau$ *holds.*
- *If* $a : \Sigma; \Gamma \vdash \tau$ *holds, then* $e : \Sigma; \Gamma \vdash \tau$ *holds for all* e *such that* $\lfloor e \rfloor = a$.

Proof. Coq lemma `jterm_aux` and `jterm_aux_rev` in file `Lsoundness.v`. □

We assume the extraction lemma which is used to prove soundness. It permits to add premises to some typing rules. These premises have to be present for the induction to work, but are simple consequences of all rules taken at once. The premises in question are specifically marked with a comment in the definition of `typesystem.jobj` and `Ftypesystem.jterm`. This lemma has been proved in Coq for a prior but similar version of the System. It is not yet proved for the current version, however the proof is done on paper.

Lemma 94 (Extraction)**.** *The following assertions hold.*

- *If* $\Sigma \vdash \kappa$ *and* $\varnothing \Vdash \Sigma$ *hold, then* $\Sigma \Vdash \kappa$ *holds.*
- *If* $\Sigma \vdash \tau : \kappa$ *and* $\varnothing \Vdash \Sigma$ *hold, then* $\Sigma \Vdash \kappa$ *holds.*
- *If* $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$, $\varnothing \Vdash \Sigma$, $\Sigma \Vdash \Theta_0$, *and* $\Sigma \Vdash \Theta_1$ *hold, then* $\Sigma \Vdash \mathrm{P}$ *holds.*
- *If* $\Sigma \vdash \Sigma'$ *and* $\varnothing \Vdash \Sigma$ *hold, then* $\Sigma \Vdash \Sigma'$ *holds.*
- *If* $e : \Sigma; \Gamma \vdash \tau$, $\varnothing \Vdash \Sigma$, *and* $\Sigma \vdash \Gamma$ *hold, then* $\Sigma \vdash \tau : \star$ *holds.*

Proof. The proof is done by induction and uses the usual weakening and substitution lemma. □

To state and prove the soundness of the indexed type system we interpret (syntactic) kinds, types, propositions, and typing environments as sets of semantic objects, semantic objects, semantic propositions, and mappings from type variables to semantic objects respectively. For each judgment, we also give its semantic interpretation as a mathematical assertion. The Coq interpretation function is `Fsoundness.semobj`. It is defined as a binary relation, but we show in `semobj_eq` that it behaves as a function.

We define the interpretation of kinds on Figure 5.17. Kinds are interpreted as sets of semantic objects under a mapping η from type variables α to semantic objects. The star kind is interpreted as the set of semantic types $\mathbb{T}$. The unit kind is interpreted as the singleton set containing the unit object $\langle\rangle$. The product kind $\kappa_1 \times \kappa_2$ is interpreted as the set of pairs of which the first component is in the interpretation of κ_1 and the second component is in the interpretation of κ_2. Finally the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ is interpreted as the subset of the interpretation of κ for which its elements x satisfy the interpretation of the proposition P under the extended mapping where α maps to x and for all k.

$$
\begin{aligned}
|\alpha|_\eta &= \eta(\alpha) \\
|\tau \to \sigma|_\eta &= |\tau|_\eta \to |\sigma|_\eta \\
|\tau \times \sigma|_\eta &= |\tau|_\eta \times |\sigma|_\eta \\
|\forall(\alpha : \kappa)\,\tau|_\eta &= \forall\,|\kappa|_\eta\ (x \mapsto |\tau|_{\eta,\alpha\mapsto x}) \\
|\mu\alpha\,\tau|_\eta &= \mu\,(x \mapsto |\tau|_{\eta,\alpha\mapsto x}) \\
|\bot|_\eta &= \bot \\
|\top|_\eta &= \top
\end{aligned}
\qquad
\begin{aligned}
|\langle\rangle|_\eta &= \langle\rangle \\
|\langle\tau,\sigma\rangle|_\eta &= \langle|\tau|_\eta\,, |\sigma|_\eta\rangle \\
|\mathsf{fst}\,\tau|_\eta &= \mathsf{fst}\,|\tau|_\eta \\
|\mathsf{snd}\,\tau|_\eta &= \mathsf{snd}\,|\tau|_\eta
\end{aligned}
$$

Figure 5.18: Type interpretation

$$
\begin{aligned}
|\top|^k_\eta &= \top \\
|\mathrm{P}_1 \wedge \mathrm{P}_2|^k_\eta &= |\mathrm{P}_1|^k_\eta \wedge |\mathrm{P}_2|^k_\eta \\
|(\Sigma' \vdash \tau') \rhd \tau|^k_\eta &= \forall e < k\ (\forall \eta' \in |\Sigma'|_\eta\ e \in |\tau'|_{\eta,\eta'}) \Rightarrow e \in |\tau|_\eta \\
|\exists\,\kappa|^k_\eta &= \exists x \in |\kappa|_\eta \\
|\forall(\alpha : \kappa)\,\mathrm{P}|^k_\eta &= \forall x \in |\kappa|_\eta\ \ |\mathrm{P}|^k_{\eta,\alpha\mapsto x}
\end{aligned}
$$

Figure 5.19: Proposition interpretation

The interpretation of syntactic types, given in Figure 5.18, are semantic objects, and it is parametrized over a mapping from type variables to semantic objects written η. Semantic objects may be semantic types, the unit object $\langle\rangle$, or pairs of semantic objects $\langle x_1, x_2\rangle$. The interpretation of a type variable is its value in the mapping. If it is not present in the mapping, the unit semantic object is returned. The interpretation of arrow and product types simply use the arrow and product operators defined in 5.2.4. The interpretation of the polymorphic type $\forall(\alpha : \kappa)\,\tau$ under η is the intersection of all interpretations of τ under η extended with α mapping to R in the interpretation of κ under η. This is defined using the intersection operator of Section 5.2.5. The interpretation of the recursive type $\mu\alpha\,\tau$ under η is the infinite iteration of the functor mapping X under the extension of η mapping α to X—which corresponds to the infinite unfolding of the recursive type (see Section 5.2.6). Finally, top and bottom are mapped to their semantic equivalent.

The unit type is interpreted as the unit semantic object. The syntactic pair of types τ and σ is interpreted as the semantic pair of the interpretations of τ and σ under the same mapping. Similarly for the projections: $\mathsf{fst}\,\tau$ and $\mathsf{snd}\,\tau$ are interpreted to the mathematical projections of respectively the first and second component of semantic pairs.

The interpretation of propositions is given on Figure 5.19. A proposition is interpreted to a mathematical assertion and depends on a mapping η from type variables to semantic objects and on an index k used to control coinduction. The true and conjunction propositions are interpreted as the true and conjunction mathematical assertions. The interpretation of the coercion proposition $(\Sigma' \vdash \tau') \rhd \tau$ under η and k asks $\forall\,|\Sigma'|_\eta\ (\eta' \mapsto |\tau'|_{\eta,\eta'})$ to be included in $|\tau|_\eta$ for indexed terms e with indices smaller than k. Because all indexed terms are controlled by an index k (Coq lemma `Flanguage.term_lt_exists`), if the inclusion holds for all indices, then it holds for the specific index k, and thus it holds for the term e. The interpretation of the coherence proposition $\exists\,\kappa$ under η (the coinduction index k is not used) is the assertion

$$
\begin{aligned}
|\varnothing|_\eta &= \{\varnothing\} \\
|\Sigma, (\alpha : \kappa)|_\eta &= \left\{\eta', \alpha \mapsto x \mid \eta' \in |\Sigma|_\eta \wedge x \in |\kappa|_{\eta,\eta'}\right\} \\
|\varnothing|_\eta^k &= \top \\
|\Theta, \mathrm{P}|_\eta^k &= |\Theta|_\eta^k \wedge |\mathrm{P}|_\eta^k \\
|\varnothing|_\eta &= \varnothing \\
|\Gamma, (x : \tau)|_\eta &= |\Gamma|_\eta , (x : |\tau|_\eta)
\end{aligned}
$$

Figure 5.20: Environments interpretation

that there exists a semantic object x in the interpretation of the kind κ under η. Finally, the polymorphic proposition $\forall(\alpha : \kappa)\,\mathrm{P}$ is interpreted under η and k by a quantified assertion that P has to hold for k and the mapping η extended with α mapped to x for all objects x in the interpretation of κ under η.

We define the interpretation of type environments as a set of semantic object mappings (from type variables to semantic objects) in Figure 5.20. The interpretation is parametrized by a surrounding mapping η. The empty environment is interpreted by the singleton set containing the empty mapping $\varnothing$. The interpretation of an environment Σ extended with a type binding $(\alpha : \kappa)$ extends the mapping η' in the interpretation of Σ under η with α bound to a semantic object in the interpretation of κ under the extended mapping η, η'. We write $|\Sigma|$ to stand for $|\Sigma|_\varnothing$. The following lemma and corollary demonstrate how mapping extension relates to type environment concatenation. If η is a mapping of the interpretation of the type environment Σ and η' is a mapping of the interpretation of Σ' under the surrounding mapping η, then η' is actually an extension of η for the interpretation of the concatenation Σ, Σ'.

Lemma 95. *If $\eta_2 \in |\Sigma_2|_{\eta_1}$ and $\eta_3 \in |\Sigma_3|_{\eta_1,\eta_2}$, then $\eta_2, \eta_3 \in |\Sigma_2, \Sigma_3|_{\eta_1}$.*

Corollary 96. *If $\eta \in |\Sigma|$ and $\eta' \in |\Sigma'|_\eta$, then $\eta, \eta' \in |\Sigma, \Sigma'|$.*

Proof. Coq lemma `Fsoundness.Happ_fH` □

The interpretation of proof environments is given in Figure 5.20. The interpretation $|\Theta|_\eta^k$ of the proof environment Θ under the mapping η and with index k is a mathematical proposition. The interpretation of the empty environment is the tautology proposition, which is always true. The extension of an environment Θ with proposition P under η with k is the conjunction of the interpretations of Θ and P under the same arguments η and k. Said otherwise $|\Theta|_\eta^k$ holds if and only if all its propositions hold under η and k.

Finally, the interpretation of term environments is given in Figure 5.20. The interpretation $|\Gamma|_\eta$ of the term environment Γ under the mapping η is the semantic context G where all syntactic types have been interpreted to a semantic type R under η.

The semantic weakening lemma tells that the interpretation of a syntactical object (a kind, a type, a proposition, a type environment, a proof environment, or a term environment) under the mapping η, only looks at the values of η for its free type variables. As a consequence, we may freely change the mapping η as long as we do not touch the semantic objects associated to the free type variables of the syntactical object we consider, it will not change its interpretation.

Lemma 97 (Semantic weakening)**.** *If η' and η agree on the free variables of κ (resp. τ, P, Σ, and Θ), then $|\kappa|_{\eta'} = |\kappa|_\eta$ (resp. $|\tau|_{\eta'} = |\tau|_\eta$, $|\mathrm{P}|^k_{\eta'} = |\mathrm{P}|^k_\eta$ for all k, $|\Sigma|_{\eta'} = |\Sigma|_\eta$, $|\Theta|^k_{\eta'} = |\Theta|^k_\eta$ for all k, and $|\Gamma|_{\eta'} = |\Gamma|_\eta$) holds.*

Proof. Coq lemma `Fsoundness.semobj_lift` □

The semantic substitution lemma relates type substitution with mapping extensions. The interpretation under η of a semantic object in which we substituted the type variable α with type σ is equal to the interpretation of the same object without substitution under the extension of η where α maps to the interpretation of σ under η.

Lemma 98 (Semantic substitution)**.** *Let $\eta' = \eta, \alpha \mapsto |\sigma|_\eta$. The following assertions hold.*

- $|\kappa[\alpha/\sigma]|_\eta = |\kappa|_{\eta'}$ *holds.*
- $|\tau[\alpha/\sigma]|_\eta = |\tau|_{\eta'}$ *holds.*
- $|\mathrm{P}[\alpha/\sigma]|^k_\eta = |\mathrm{P}|^k_{\eta'}$ *holds for all k.*
- $|\Sigma[\alpha/\sigma]|_\eta = |\Sigma|_{\eta'}$ *holds.*
- $|\Theta[\alpha/\sigma]|^k_\eta = |\Theta|^k_{\eta'}$ *holds for all k.*
- $|\Gamma[\alpha/\sigma]|_\eta = |\Gamma|_{\eta'}$ *holds.*

Proof. Coq lemma `Fsoundness.semobj_subst` □

We can now define and prove the soundness of each syntactic judgment according to its semantic. If τ is non-expansive (resp. well-founded) with respect to α, then its interpretation as a functor under any mapping is non-expansive (resp. well-founded). If a kind κ is coherent under environment Σ, then its interpretation under any mapping η of the interpretation of Σ is inhabited. If a type τ has kind κ under the environment Σ, then its interpretation is in the interpretation of its kind, for all mapping η of the interpretation of Σ.

If the proposition P holds under Σ, Θ_0, and Θ_1, then for all mapping η of the interpretation of Σ and for all index k, if the interpretation of Θ_0 holds under η for all indices j smaller or equal than k and the interpretation of Θ_1 holds under η for all indices j smaller than k, then the interpretation of P holds under η with k. We see in the interpretation of the proposition judgment that Θ_0 can be used at level k while Θ_1 is only accessible at level $k-1$. So in order for P to use an hypothesis in Θ_1, it has to do something constructive in order to decrease the level and use the hypothesis.

If a type environment Σ' is coherent under Σ, then its interpretation is inhabited under any mapping η of the interpretation of Σ. If the term environment Γ is valid under the type environment Σ, then for all mappings η of the interpretation of Σ the interpretation under η of the types of Γ are semantic types. If the indexed term e has type τ under Σ and Γ, then for all type mapping η of the interpretation of Σ the term e is in the semantic judgment $|\Gamma|_\eta \models |\tau|_\eta$.

Lemma 99 (Judgment soundness)**.** *The following assertions hold.*

- *If $\alpha \mapsto \tau : \mathsf{NE}$ holds, then $\forall \eta\ \mathsf{X} \mapsto |\tau|_{\eta,\alpha\mapsto\mathsf{X}}$ is non-expansive.*
- *If $\alpha \mapsto \tau : \mathsf{WF}$ holds, then $\forall \eta\ \mathsf{X} \mapsto |\tau|_{\eta,\alpha\mapsto\mathsf{X}}$ is well-founded.*

$$
\begin{aligned}
\lceil x \rceil^k &= x^k & \lceil \langle a, b\rangle \rceil^k &= \langle \lceil a\rceil^k, \lceil b\rceil^k \rangle^k \\
\lceil \lambda x\, a \rceil^k &= \lambda^k x\, \lceil a \rceil^k & \lceil \mathsf{fst}\, a \rceil^k &= \mathsf{fst}^k\, \lceil a \rceil^k \\
\lceil a\, b \rceil^k &= (\lceil a \rceil^k\, \lceil b \rceil^k)^k & \lceil \mathsf{snd}\, a \rceil^k &= \mathsf{snd}^k\, \lceil a \rceil^k
\end{aligned}
$$

Figure 5.21: Fill function

- *If* $\Sigma \vdash \kappa$ *holds, then* $\forall \eta \in |\Sigma| \;\; |\kappa|_\eta \neq \varnothing$.
- *If* $\Sigma \vdash \tau : \kappa$ *holds, then* $\forall \eta \in |\Sigma| \;\; |\tau|_\eta \in |\kappa|_\eta$.
- *If* $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$ *holds, then* $\forall \eta \in |\Sigma| \;\; \forall k \; (\forall j \leq k \;\; |\Theta_0|_\eta^j) \wedge (\forall j < k \;\; |\Theta_1|_\eta^j) \Rightarrow |\mathrm{P}|_\eta^k$.
- *If* $\Sigma \vdash \Sigma'$ *holds, then* $\forall \eta \in |\Sigma| \;\; |\Sigma'|_\eta \neq \varnothing$.
- *If* $\Sigma \vdash \Gamma$ *holds, then* $\forall \eta \in |\Sigma| \;\; \forall (x : \mathsf{R}) \in |\Gamma|_\eta \;\; \mathsf{R} \in \mathbb{T}$.
- *If* $e : \Sigma; \Gamma \vdash \tau$ *holds, then* $\forall \eta \in |\Sigma| \;\; e \in |\Gamma|_\eta \models |\tau|_\eta$.

Proof. Lemma `jrec_sound`, `jobj_sound`, and `jterm_sound` in the `Fsoundness.v` Coq file. □

To prove the soundness of System L^c, we need to define the filling $\lceil a \rceil^k$ of lambda term a at rank k as the indexed term obtained by annotating each node of a with index k (Figure 5.21 and Coq definition `Llanguage.kfill`). By construction, we have $\lfloor \lceil a \rceil^k \rfloor = a$. The drop and fill functions are used to go back and forth between both type systems.

We can now define and prove the soundness lemma. If a is well-typed under coherent and valid environments then a is sound.

Theorem 100 (Soundness of System L^c)**.** *If* $\varnothing \vdash \Sigma$, $\Sigma \vdash \Gamma$, *and* $a : \Sigma; \Gamma \vdash \tau$ *hold, then* $a \in \mathcal{S}$.

Proof. Coq lemma `Lsoundness.jterm_sound`

The lambda term a is sound if it is sound for all number of steps k. Let's show that it is sound for at least k steps. Let e be $\lceil a \rceil^k$. We have $e \geq k$, and by Lemma 75 it suffices to show that e is sound. We have $e : \Sigma; \Gamma \vdash \tau$ by Lemma 93. By Lemma 94, we have $\varnothing \Vdash \Sigma$ in a first step and $\Sigma \vdash \tau : \star$ in a second step. By Lemma 99, $|\Sigma|$ is inhabited and there is a mapping η such that $\eta \in |\Sigma|$. We also have that all sets in $|\Gamma|_\eta$ are semantic types. And we also have that $|\tau|_\eta$ is a semantic type. And finally we have e is in the semantic judgment $|\Gamma|_\eta \models |\tau|_\eta$, which is a pretype. We deduce $e \in \mathcal{S}$. □

Termination in the absence of recursive types Although evaluation may not terminate because of the presence of recursive types, it remains interesting to show that recursive types are the only source of non-termination. We already know this in System F. We show that coercions do not themselves introduce non-termination, as long as all types remain non-recursive. The proof was done in Coq for a prior but similar version of the System. It is based on reducibility candidates as for System F and does not raise any difficulties. We thus omit the details.

Theorem 101 (Termination)**.** *If* $\varnothing \vdash \Sigma$, $\Sigma \vdash \Gamma$, *and* $a : \Sigma; \Gamma \vdash \tau$ *hold in the sublanguage without recursive types, then* a *strongly normalizes.*

5.4 Expressivity

System $\mathsf{L^c}$ goal was to extend System F_ι^p to gain the expressivity of Constraint ML. In order to do so we added recursive types, coherent coercion abstraction. Recursive types are necessary because consistency in Constraint ML is only defined with recursive types. There is apparently no simple criteria for consistency implying strong normalization in addition to soundness. In System $\mathsf{L^c}$ however, it is simple to remove recursive types and thus allow us to write programs with mechanisms similar to Constraint ML which are sound and terminating.

Coherent coercion abstraction was actually added using the coherent polymorphism type system feature. Polymorphism permits to abstract over an erasable content as long as it is coherent. We classify erasable contents, *i.e.* types, with kinds. We added a particular kind which permits to restrain an existing kind for types satisfying a proposition, and propositions contain coercions. In a first section, we define some surface notations. In the next two sections we show how System F_ι^p and Constraint ML are included in System $\mathsf{L^c}$. We then discuss the differences between equi-recursive types which we provide in System $\mathsf{L^c}$ and iso-recursive types which we defined in System $\mathsf{F_{rec}}$ in Section 3.3. The final section shows how our notion of coinduction subsumes existing notions about recursive type equality and recursive type congruence.

5.4.1 Surface notations

To ease readability, we define a few surface notations that easily desugar to System $\mathsf{L^c}$.

Type bindings with constrained kinds The first notation is about type bindings $(\alpha : \kappa)$ when the kind κ is a constrained kind $\{\alpha' : \kappa' \mid \mathrm{P}'\}$. The binding is thus written $(\alpha : \{\alpha' : \kappa' \mid \mathrm{P}'\})$. On the one hand, the type variable α do not bind in the constrained kind and is thus not free in κ' and P'. But it binds for what comes next (which is the role of a binder); if the binder is used for a polymorphic type $\forall(\alpha : \{\alpha' : \kappa' \mid \mathrm{P}'\})\,\sigma$, it binds in σ; if it is used in a type environment $\Sigma_1, (\alpha : \{\alpha' : \kappa' \mid \mathrm{P}'\}), \Sigma_2$, it binds in Σ_2 and what comes after (coinductive hypotheses or what is after the turnstile). On the other hand, the type variable α' binds in P', but not in what comes after. It also does bind in its kind κ'. Since these scopes are not ambiguous we may reuse the same type variable for α and α', and thus write $(\alpha : \{\alpha : \kappa' \mid \mathrm{P}'[\alpha'/\alpha]\})$ (which is what happens when using de Bruijn indices).

However these two type variables are essentially the same: they represent the same type. We may thus simply write $(\alpha : \kappa \mid \mathrm{P})$ instead of $(\alpha : \{\alpha : \kappa \mid \mathrm{P}\})$. In other words the constrained kind, with parentheses instead of accolades, becomes a binder for the constrained kind itself, reusing its type variable for the binding type variable. This notation applies only when the constrained kind is used where a binder is expect. Here follow two examples: one for the polymorphic type and one for type environment extension.

$$\begin{aligned} \forall(\alpha : \kappa \mid \mathrm{P})\,\sigma &\stackrel{\mathsf{def}}{=} \forall(\alpha : \{\alpha : \kappa \mid \mathrm{P}\})\,\sigma \\ \Sigma, (\alpha : \kappa \mid \mathrm{P}) &\stackrel{\mathsf{def}}{=} \Sigma, (\alpha : \{\alpha : \kappa \mid \mathrm{P}\}) \end{aligned}$$

Existential propositions In type systems with constraints, like Constraint ML, there is usually an existential proposition. As we saw in rule PropExi, our coherence proposition $\exists\,\kappa$ behaves an existential. It is introduced with a rule forgetting the witness type, and eliminated with rules TypeStr and PropStr which are similar to existential elimination. This is thus no surprise that we can easily encode existential propositions with the coherence proposition of a constrained kind.

$$\exists(\alpha : \kappa)\,\mathrm{P} \stackrel{\mathsf{def}}{=} \exists\,\{\alpha : \kappa \mid \mathrm{P}\}$$

The existential proposition $\exists(\alpha : \kappa)\,\mathrm{P}$ is thus true, if there is a witness type τ in the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, which by unfolding rule TypePack means that τ has kind κ and satisfies the proposition P.

Lists of type variables and lists of kinds The reason we added the unit kind and product kinds was to simplify the Coq development when dealing with lists of types. Our prior version had a polymorphic type of the form $\forall(\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})\sigma$, which meant that we abstract over all type variables in $\overline{\alpha}$ satisfying all the coercions in $\overline{\tau_1 \triangleright \tau_2}$. We eliminated these two uses of lists using the unit kind and product kind for the type variables, and the true proposition and conjunction proposition for the coercions. However, a notation of the form $\{\overline{\alpha : \kappa} \mid \mathrm{P}\}$, and thus $\forall(\overline{\alpha : \kappa} \mid \mathrm{P})\,\sigma$ using the type binding with constrained kind notation, may be useful.

The definition of such notation is not local to the binder, but needs access to what is in the scope of the binder. In the case of the constrained kind $\{\overline{\alpha : \kappa} \mid \mathrm{P}\}$ we need to modify the proposition P. Similarly, for the polymorphic type $\forall(\overline{\alpha : \kappa})\,\sigma$, we need access to the type σ. And finally, for the combination of both $\forall(\overline{\alpha : \kappa} \mid \mathrm{P})\,\sigma$, we need to modify both P and σ.

We will study the situation with the constrained kind polymorphic type, but it applies for other binding structures. We define the notation $\forall(\overline{\alpha : \kappa} \mid \mathrm{P})\,\sigma$ as $\forall(\alpha' : \kappa' \mid \mathrm{P}')\,\sigma'$, where κ', P', and σ' are defined as follows. The type variable α' is taken fresh. We define κ' as the list $\overline{\kappa}$ encoded using the product kind for cons and the unit kind for nil. For example, the list κ_1, κ_2 is encoded as $\kappa_1 \times (\kappa_2 \times 1)$. The proposition P' and the type σ' are substitutions of P and σ respectively. We substitute each use of α_i (the ith element of the list $\overline{\alpha}$) as the first type projection of $(i-1)$ iterations of the second type projection of the type variable α'. For example, if the list of type variables was α_1, α_2, the substitution will be $[\alpha_1/\mathsf{fst}\,\alpha'][\alpha_2/\mathsf{fst}\,(\mathsf{snd}\,\alpha')]$. When instantiating such kind of notations, we need to define the encoding of a list of types. The idea is similar to the encoding of the list of kinds. The list of types $\overline{\tau}$ is encoded as a list where the pair type is used for cons and the unit type is used for nil. For example, the list τ_1, τ_2 is encoded as $\langle \tau_1, \langle \tau_2, \langle\rangle\rangle\rangle$.

Here is a list of other places where this notation may be used: polymorphic types $\forall(\overline{\alpha : \kappa})\,\sigma$, constrained kinds $\{\overline{\alpha : \kappa} \mid \mathrm{P}\}$, constrained kind polymorphic types $\forall(\overline{\alpha : \kappa} \mid \mathrm{P})\,\sigma$ as we saw, existential propositions $\exists(\overline{\alpha : \kappa})\,\mathrm{P}$ because they use the constrained kind, type extensions $\Sigma, (\overline{\alpha : \kappa})$, and constrained kind type extensions $\Sigma, (\overline{\alpha : \kappa} \mid \mathrm{P})$.

The star kind A very simple notation, but easing readability, is to omit the star kinds in type bindings. We may for example write: $\forall\alpha\,\tau$ for polymorphic types of the kind star, $\forall\overline{\alpha}\,\tau$ for polymorphic types over a list of types of the kind star, $\forall(\overline{\alpha} \mid \mathrm{P})\,\sigma$ for polymorphic types over a list of types of the kind star satisfying the proposition P, or $\exists\overline{\alpha}\,\mathrm{P}$ for existential propositions over a list of type variables of the kind star satisfying the proposition P.

Erasable isomorphisms It is sometime useful to have a notion of isomorphisms, like the equality coercions in FC. We can encode this notion using two inverse coercions. We define the isomorphism proposition notation $\tau \equiv \sigma$ as $(\varnothing \vdash \tau) \triangleright \sigma \wedge (\varnothing \vdash \sigma) \triangleright \tau$. However, this definition can only be used with types of kind star, because coercions are only defined at kind

$$
\begin{aligned}
\widehat{\alpha} &= \alpha \\
\widehat{\tau \to \sigma} &= \widehat{\tau} \to \widehat{\sigma} \\
\widehat{\tau \times \sigma} &= \widehat{\tau} \times \widehat{\sigma} \\
\widehat{\forall \alpha\, \tau} &= \forall \alpha\, \widehat{\tau} \\
\widehat{\top} &= \top \\
\widehat{\bot} &= \bot
\end{aligned}
\qquad
\begin{aligned}
\forall \widehat{(\alpha \triangleright \tau)}\, \sigma &= \forall(\alpha \mid (\varnothing \vdash \alpha) \triangleright \widehat{\tau})\, \widehat{\sigma} \\
\forall \widehat{(\alpha \triangleleft \tau)}\, \sigma &= \forall(\alpha \mid (\varnothing \vdash \widehat{\tau}) \triangleright \alpha)\, \widehat{\sigma}
\end{aligned}
$$

Figure 5.22: System F_ι^p type translation function

$$
\begin{aligned}
\widehat{\varnothing}^{\Sigma} &= \varnothing \\
\widehat{\Gamma, (x : \tau)}^{\Sigma} &= \widehat{\Gamma}^{\Sigma} \\
\widehat{\Gamma, \alpha}^{\Sigma} &= \widehat{\Gamma}^{\Sigma}, \alpha \\
\Gamma, \widehat{\alpha, (\tau \triangleright \sigma)}^{\Sigma} &= \widehat{\Gamma}^{\Sigma}, (\alpha \mid (\varnothing \vdash \widehat{\tau}) \triangleright \widehat{\sigma})
\end{aligned}
\qquad
\begin{aligned}
\widehat{\varnothing}^{\Gamma} &= \varnothing \\
\Gamma, \widehat{(x : \tau)}^{\Gamma} &= \widehat{\Gamma}^{\Gamma}, (x : \widehat{\tau}) \\
\widehat{\Gamma, \alpha}^{\Gamma} &= \widehat{\Gamma}^{\Gamma} \\
\Gamma, \widehat{(\tau \triangleright \sigma)}^{\Gamma} &= \widehat{\Gamma}^{\Gamma}
\end{aligned}
$$

Figure 5.23: System F_ι^p environment translation function

star. In order to define a notion of isomorphism at any kind, we lift our notion of erasable isomorphism at kind star to other kinds by extensionality. We index the notation $\tau \equiv_\kappa \sigma$ with the kind κ of types τ and σ. We may consider heterogeneous isomorphism, but we restrict to homogeneous isomorphism for simplicity. The isomorphism at kind κ is defined inductively on the kind as follows (the last line concerns type-level functions defined in Section 6.1.3):

$$
\begin{aligned}
\tau \equiv_\star \sigma &\overset{\mathsf{def}}{=} \tau \equiv \sigma \\
\tau \equiv_1 \sigma &\overset{\mathsf{def}}{=} \top \\
\tau \equiv_{\kappa_1 \times \kappa_2} \sigma &\overset{\mathsf{def}}{=} (\mathsf{fst}\,\tau \equiv_{\kappa_1} \mathsf{fst}\,\sigma) \wedge (\mathsf{snd}\,\tau \equiv_{\kappa_2} \mathsf{snd}\,\sigma) \\
\tau \equiv_{\{\alpha:\kappa \mid \mathrm{P}\}} \sigma &\overset{\mathsf{def}}{=} \tau \equiv_\kappa \sigma \\
\tau \equiv_{\kappa_1 \to \kappa_2} \sigma &\overset{\mathsf{def}}{=} \forall(\alpha\beta : \kappa_1 \times \kappa_1 \mid \alpha \equiv_{\kappa_1} \beta)\, \tau\, \alpha \equiv_{\kappa_2} \sigma\, \beta
\end{aligned}
$$

5.4.2 System F_ι^p

We define two translations from System F_ι^p to System L^c: one for types and one for environments. We write $\widehat{\tau}$ the translation of the type τ. We give its definition in Figure 5.22. Type variables, arrows, products, polymorphic types, top and bottom are translated to their equivalent in System L^c. Notice that the polymorphic type uses the notation omitting to mention the kind star (see Section 5.4.1). The bounded polymorphic types are translated to polymorphic types over a constrained kind (we can use the notations we defined in Section 5.4.1). The upper bounded polymorphic type $\forall(\alpha \triangleright \tau)\,\sigma$ is translated to a polymorphic type with body $\widehat{\sigma}$ over the type variable α ranging over the constrained kind $\{\alpha \mid (\varnothing \vdash \alpha) \triangleright \tau\}$. This constrained kind contains types (of the star kind) satisfying the proposition that α can be coerced to τ, which is exactly what upper bounded polymorphism requires. Lower bounded polymorphism is similar.

We define the translation for environments in Figure 5.23. Environments can be translated as type environments, written $\widehat{\Gamma}^{\Sigma}$, or as term environments, written $\widehat{\Gamma}^{\Gamma}$. The exponent symbol is not a metavariable but an annotation to differentiate the translation. The translation as term environments simply filters the term bindings and translates their types. The translation for type environments translates sole type bindings to type bindings of the star kind, and bounded type bindings to type bindings of a constrained kind with the coercion as the constraining proposition. Again, we use the notations defined in Section 5.4.1.

To prove the inclusion of System F_ι^p in System L^c, we need to prove that if a judgment holds in System F_ι^p, then its translation holds in System L^c. We write $(J)_s$ for System F_ι^p judgments and $(J)_t$ for System L^c judgments. From the environment judgment we actually extract two translated judgments: one for the type environment and one for the term environment which has to hold under the translated type environment. All types in System F_ι^p have the star kind as we can see in the second assertion. And all coercions are typed without coinduction hypotheses as we can see in the third assertion.

Lemma 102. *The following assertions hold.*

- *If* $(\Gamma\ \mathsf{env})_s$ *holds, then* $(\varnothing \vdash \widehat{\Gamma}^\Sigma)_t$ *and* $(\widehat{\Gamma}^\Sigma \vdash \widehat{\Gamma}^\Gamma)_t$ *hold.*
- *If* $(\Gamma \vdash \tau\ \mathsf{type})_s$ *holds, then* $(\widehat{\Gamma}^\Sigma \vdash \widehat{\tau} : \star)_t$ *holds.*
- *If* $(\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma)_s$ *and* $(\Gamma, \Sigma \vdash \tau\ \mathsf{type})_s$ *hold, then* $(\widehat{\Gamma}^\Sigma; \varnothing; \varnothing \vdash (\widehat{\Sigma}^\Sigma \vdash \widehat{\tau}) \triangleright \widehat{\sigma})_t$ *holds.*
- *If* $(a : \Gamma \vdash \tau)_s$ *holds, then* $(a : \widehat{\Gamma}^\Sigma; \widehat{\Gamma}^\Gamma \vdash \widehat{\tau})_t$ *holds.*

Proof. We proceed by mutual induction. For rule EnvEmpty, the assertion holds because the empty type environment is well formed, and the empty term environment is well-formed under well-formed type environment. Rules EnvTerm and EnvType hold by induction hypotheses without difficulties.

Rule EnvTypeL is more involved. This is actually the most interesting part of the proof because this is where we forge (very simple) witnesses. If we write Σ the translated type environment by induction hypothesis, then we have to show that $\Sigma \vdash \{\alpha \mid (\varnothing \vdash \widehat{\tau}) \triangleright \alpha\}$ holds. By rule Kind we have to show $\Sigma; \varnothing; \varnothing \vdash \exists\alpha\, (\varnothing \vdash \widehat{\tau}) \triangleright \alpha$. We use rule TypePack with the top type. The well-formedness premise for the proposition $\Sigma, \alpha \Vdash (\varnothing \vdash \widehat{\tau}) \triangleright \alpha$ uses the induction hypothesis to show $\Sigma, \alpha \vdash \widehat{\tau} : \star$. The top type has kind star by rule TypeTop. We prove the last premise $\Sigma; \varnothing; \varnothing \vdash (\varnothing \vdash \widehat{\tau}[\alpha/\top]) \triangleright \top$ by rule CoerTop and the substitution lemma to show $\Sigma \vdash \widehat{\tau}[\alpha/\top] : \star$. Rule EnvTypeU is similar with the bottom type as witness.

The second assertion is not difficult and uses only induction hypotheses. For the third assertion, proof environments are always empty. Rules CoerRefl, CoerTrans, CoerWeak, CoerEtaArr, CoerEtaProd, CoerTop, and CoerBot have analog rules.

Rule CoerVar is interesting because it does not use a coercion variable rule, as there are no such rule. It uses rule PropRes with the instance type equal to the type variable associated to the coercion, which does exists because the environment is well-formed. For instance, if the coercion judgment we have to translate is $\Gamma, \alpha, (\alpha \triangleright \tau), \Gamma' \vdash \alpha \triangleright \tau$ then we have $(\alpha : \{\alpha \mid (\varnothing \vdash \alpha) \triangleright \widehat{\tau}\})$ in our translated type environment. With rule TypeVar we derive $\Sigma \vdash \alpha : \{\alpha \mid (\varnothing \vdash \alpha) \triangleright \widehat{\tau}\}$. From which using rule PropRes we get $\Sigma; \varnothing; \varnothing \vdash (\varnothing \vdash \alpha) \triangleright \widehat{\tau}$.

Rules CoerTLam and CoerTApp use rules CoerGen and CoerInst with the star kind. Rules CoerTLamL, CoerTLamU, CoerTAppL, and CoerTAppU are similar with more elaborated kinds. We detail rule CoerTAppU. We have to show that $\Sigma \vdash \widehat{\sigma} : \{\alpha \mid (\varnothing \vdash \alpha) \triangleright \widehat{\tau}\}$ holds, if we write Σ the translated type environment. We use rule TypePack. The first premise $\Sigma \vdash \widehat{\sigma} : \star$ is proved by the first induction hypothesis. The second premise $\Sigma; \varnothing; \varnothing \vdash (\varnothing \vdash \widehat{\sigma}) \triangleright \widehat{\tau}[\alpha/\widehat{\sigma}]$ is proved by the second induction hypothesis.

The last assertion only contains analog rules. ∎

5.4.3 Constraint ML

In $\mathsf{Constraint\ ML}$, the main feature is constraint generalization. We write $\widehat{\tau}$ the type translation of τ. We can translate constraints as propositions. The constraint $\tau \triangleright \sigma$ becomes the coercion $(\varnothing \vdash \widehat{\tau}) \triangleright \widehat{\sigma}$. Constraint conjunctions $C_1 \wedge C_2$ are translated to proposition conjunctions $\widehat{C_1} \wedge \widehat{C_2}$. The existential constraints $\exists\overline{\alpha}.C$ become $\exists\overline{\alpha}\,\widehat{C}$, which are existential propositions over a series of type variables (see Section 5.4.1). We translate types in the same manner. In particular, type schemes $\forall\overline{\alpha}.C \Rightarrow \sigma$ are translated to $\forall(\overline{\alpha} \mid \widehat{C})\,\widehat{\sigma}$. Environments are simply translated by translating the type schemes.

We translate the term judgment $C;\Gamma \vdash a : \tau$ into $a : \varnothing, (\overline{\alpha} \mid \widehat{C}); \widehat{\Gamma} \vdash \widehat{\tau}$. The type environment contains a unique type binding for all the type variables (of the star kind using notations of Section 5.4.1) in C, Γ, and τ. The type binding has a constrained kind resulting from the constraint of the $\mathsf{Constraint\ ML}$ judgment. Most term typing rules are simple translations. We focus on the most interesting ones.

Rule TermSub uses rule TermCoer as long as $C \vdash \tau \triangleright \tau'$ can be translated to a derivation of $\varnothing, (\overline{\alpha} \mid \widehat{C}); \varnothing; \varnothing \vdash (\varnothing \vdash \widehat{\tau}) \triangleright \widehat{\tau'}$ which depends on the power of the $\mathsf{Constraint\ ML}$ constraint judgment.

We remind rule TermIntro:

$$\frac{\begin{array}{l}\text{TermIntro}\\ C \wedge D; \Gamma \vdash a : \tau \qquad \overline{\alpha} \notin \mathsf{fv}(C, \Gamma)\end{array}}{C \wedge \exists\overline{\alpha}.D; \Gamma \vdash a : \forall\overline{\alpha}.D \Rightarrow \tau}$$

We use $\overline{\beta}$ to designate the type variables of the judgment that are not in $\overline{\alpha}$. By induction hypothesis, we have $a : \varnothing, (\overline{\beta}\overline{\alpha} : \{\overline{\beta}\overline{\alpha} : \overline{\star} \times \overline{\star} \mid \widehat{C}\theta \wedge \widehat{D}\theta\}); \widehat{\Gamma} \vdash \widehat{\tau}\theta$, where θ is the substitution $[\overline{\alpha}/\mathsf{snd}\,\overline{\beta}\overline{\alpha}][\overline{\beta}/\mathsf{fst}\,\overline{\beta}\overline{\alpha}]$. We now weaken the type environment to $\varnothing, (\overline{\beta} \mid \widehat{C} \wedge \exists\overline{\alpha}\,\widehat{D}), (\overline{\alpha} \mid \widehat{D})$. We clearly have the same type variables with the same kinds (the star kind) because we only partitioned the existing binding. It remains to show that this partition is still coherent. The first binding is coherent by hypothesis. And the second binding $(\overline{\alpha} \mid \widehat{D})$ is coherent under $\varnothing, (\overline{\beta} \mid \widehat{C} \wedge \exists\overline{\alpha}\,\widehat{D})$ by rules Kind, PropSnd, PropRes, and TypeVar.

Rule TermElim relies on rule TermCoer with CoerInst. We have to show that $\varnothing, (\overline{\beta} \mid \widehat{C}) \vdash \widehat{\overline{\tau}} : \{\overline{\alpha} \mid \widehat{D}\}$ holds. Rule TypePack asks $\varnothing, (\overline{\beta} \mid \widehat{C}); \varnothing; \varnothing \vdash \widehat{D}[\overline{\alpha}/\widehat{\overline{\tau}}]$. Depending on the power of $\mathsf{Constraint\ ML}$ constraint judgment, we may have a translation from $C \vdash D\overline{[\alpha \leftarrow \tau]}$ to our judgment.

To conclude, System $\mathsf{L^c}$ features $\mathsf{Constraint\ ML}$ mechanisms for the term judgment. Since $\mathsf{Constraint\ ML}$ is parametrized over its constraint mechanism, the inclusion is complete if the constraint judgment implies its translation as a System $\mathsf{L^c}$ proposition. In particular, constraint implication $C \vdash D$ should imply its translation, which is the proposition $\varnothing, (\overline{\alpha} \mid \widehat{C}); \varnothing; \varnothing \vdash \widehat{D}$ where $\overline{\alpha}$ are the free variables of the constraints C and D. A simple situation when this implication holds, is when the constraint mechanism exhibits type witnesses. These type witnesses can then be used to build a derivation of the translated proposition.

5.4.4 Recursive coercions

We added recursive types to System $\mathsf{L^c}$, because the solvability condition of $\mathsf{Constraint\ ML}$ uses them ([25] and [28]). There are two sorts of recursive types: equi-recursive types and iso-recursive types. Equi-recursive types leave no traces in terms. They define the unfolding of recursive types as a type equality. The types $\mu\alpha\,\tau$ and $\tau[\alpha/\mu\alpha\,\tau]$ are really the same type.

By contrast, these are different types in the iso-recursive view and fold and unfold coercions allow to change one into the other. The version of System $\mathsf{F_{rec}}$ we presented in Section 3.3 uses iso-recursive types.

Equi-recursive types are also stronger than iso-recursive types for two reasons. First, because the folding and unfolding of equi-recursive types may be used at any depth. Notice however, that in type systems with subtyping or coercions, this is not a difference. Then, because they may equal two recursive types of different periods. For instance, the types $\mu\alpha\,\mathsf{Int} \to \alpha$ and $\mu\alpha\,\mathsf{Int} \to \mathsf{Int} \to \alpha$ are equal with equi-recursive types, but there is no possible conversion from one to the other with iso-recursive types.

In System $\mathsf{L^c}$, although recursive types are folded and unfolded in our coercion judgment (as it is the case with iso-recursive types) and not in the type equality (as it is the case with equi-recursive types), they have the expressivity of equi-recursive types. This is possible for two reasons. First, our recursive types do not leave any trace at the term level: they are erasable. Erasability is a requirement for equi-recursive types. Then, we have the possibility to prove coercions by coinduction, which is crucial in order to equal periods.

We present how to derive the usual rules for reasoning on recursive types [2] by using rule PropFix. For the usual rule EquivPeriod, we use our notion of erasable isomorphism, which we also call equivalence, instead of type equality. This is not a problem because equivalence is interpreted as equality. This rule says that two types τ_1 and τ_2 are equivalent if each of them is equivalent to some unfolding by a common well-founded functor $\alpha \mapsto \sigma$.

$$\textsc{EquivPeriod}\quad \frac{\alpha \mapsto \sigma : \mathsf{WF} \qquad \Sigma; \Theta_0; \Theta_1 \vdash \tau_1 \equiv \sigma[\alpha/\tau_1] \qquad \Sigma; \Theta_0; \Theta_1 \vdash \tau_2 \equiv \sigma[\alpha/\tau_2]}{\Sigma; \Theta_0; \Theta_1 \vdash \tau_1 \equiv \tau_2}$$

This rule can be used to show that one-step τ-lists $\mu\alpha\, 1 + \tau \times \alpha$ are equivalent to two-steps τ-lists $\mu\alpha\, 1 + \tau \times (1 + \tau \times \alpha)$. It suffices to take τ_1 to be one-step τ-lists, τ_2 to be two-steps τ-lists, and σ to be $1 + \tau \times (1 + \tau \times \alpha)$. To show the first equivalence, we have to unfold τ_1 twice, and to show the second equivalence, we have to unfold τ_2 once.

Rule EquivPeriod is derivable for each type σ because the derivation follows the structure of σ. We first use rule PropFix to memorize the $\tau_1 \equiv \tau_2$ proposition. We then split the conjunction and show only the side $\tau_1 \triangleright \tau_2$, because the other side is similar. We the use both hypotheses to get $\sigma[\alpha/\tau_1] \triangleright \sigma[\alpha/\tau_2]$. We now follow the structure of σ using η-expansions to go under constructors and reflexivity when subtypes are free from the type variable α. When we reach α, it means that we have to show either $\tau_1 \triangleright \tau_2$ or $\tau_2 \triangleright \tau_1$ depending on the variance. But in both cases we went under a computational type constructor, because σ is well-founded with respect to α, and the $\tau_1 \equiv \tau_2$ hypothesis is now accessible. This remark ends the proof.

The second usual rule about recursive types is CoerEtaMu. It tells when two recursive types are in the coercion relation. The recursive type $\mu\alpha\,\tau$ is smaller than $\mu\beta\,\sigma$ if τ is smaller than σ given that α is smaller than β. This rule can be used to prove that τ-lists $\mu\alpha\, 1 + \tau \times \alpha$ are smaller than σ-lists $\mu\beta\, 1 + \sigma \times \beta$ if τ is smaller than σ. We simply use η-expansions for the sum and product types, and the hypotheses for $\tau \triangleright \sigma$ and $\alpha \triangleright \beta$.

$$\textsc{CoerEtaMu}\quad \frac{\Sigma, (\alpha\beta : \star \times \star \mid (\varnothing \vdash \alpha) \triangleright \beta); \Theta_0; \Theta_1 \vdash (\varnothing \vdash \tau) \triangleright \sigma}{\Sigma; \Theta_0; \Theta_1 \vdash (\varnothing \vdash \mu\alpha\,\tau) \triangleright \mu\beta\,\sigma}$$

This rule is derivable in System $\mathsf{L^c}$. We use rule PropFix to memorize the $\mu\alpha\,\tau \triangleright \mu\beta\,\sigma$ coercion. We then use fold and unfold to get $\tau[\alpha/\mu\alpha\,\tau] \triangleright \sigma[\beta/\mu\beta\,\sigma]$. We now use our body hypothesis and modify its use of $\alpha \triangleright \beta$ by our memorized coercion, which is accessible because $\mu\alpha\,\tau$ and $\mu\beta\,\sigma$ are well-formed recursive types and thus $\alpha \mapsto \tau$ and $\beta \mapsto \sigma$ are well-founded.

5.5 Incoherent Polymorphism

Incoherent polymorphism is a necessity for some type system features, but it may also be a simplification. First, as we describe in Section 6.1.2, incoherent polymorphism is necessary to describe the kind of existentials used in GADTs. On the one hand, coherent polymorphism requires the kind to be coherent, and thus the existence of a witness type. On the other hand, incoherent polymorphism permits type abstraction for any well-formed kind: inhabited kinds, potentially inhabited kinds, and empty kinds. In the polymorphic type $\forall(\alpha : \kappa)\,\tau$, the coherence of kind κ may depend over type variables β of the type environment. Depending on how they are instantiated, the kind κ may or may not be inhabited. A simple example will follow when illustrating GADTs. Apart from these cases, we may simply omit to prove that the kind κ is coherent to ease inference. The result will be inaccessible code until the abstraction is instantiated. Since such abstraction is equivalent to a unit abstraction (see the last paragraph of this section about weak reduction), we may say it has a zero-bit cost as in FC. In the rest of this section, we describe how to extend System L^{c} with incoherent polymorphism, and illustrate its use with GADTs. This extension is in the Coq development and thus proved sound.

We extend the syntax of the λ-calculus with type abstraction Λa and type application $a\,\circ$. Type abstraction does not bind anything. Its only use is to block reduction. Hence, we only add the evaluation context for type application $[]\,\circ$. The type abstraction context $\Lambda\,[]$ is not an evaluation context. This is the most important part of this extension. In particular, Λr, where r is an error, is sound. Prevalues and values are extended accordingly: $p\,\circ$ is a prevalue and Λa is a value. We do not ask the body a of the type abstraction to be a value because we do not reduce under type abstraction. Finally, we extend the set of errors with destructors applied to the wrong constructor.

$$\begin{array}{rcll}
a, b & ::= & \dots \mid \Lambda a \mid a\,\circ & \text{Terms} \\
E & ::= & \dots \mid []\,\circ & \text{Evaluation contexts} \\
p & ::= & \dots \mid p\,\circ & \text{Prevalues} \\
v & ::= & \dots \mid \Lambda a & \text{Values} \\
r & ::= & \dots \mid (\Lambda a)\,a \mid \mathsf{fst}\,(\Lambda a) \mid \mathsf{snd}\,(\Lambda a) \mid (\lambda x\,a)\,\circ \mid \langle a, a\rangle\,\circ & \text{Errors}
\end{array}$$

We add the reduction rule for type abstraction and type application. When a type abstraction occurs right under a type application, both constructs erase themselves and leave the body of the type abstraction which can now be reduced.

$$\begin{array}{l}\textsc{RedTApp} \\ (\Lambda a)\,\circ \rightsquigarrow a\end{array}$$

We now extend the type system. We extend types with incoherent polymorphism $\Pi(\alpha : \kappa)\,\tau$. The notation differs from coherent polymorphism $\forall(\alpha : \kappa)\,\tau$ only by the $\forall$ quantifier which becomes a Π quantifier (that has nothing to do with the quantifier for dependent products).

$$\tau, \sigma \ ::= \ \ldots \mid \Pi(\alpha : \kappa)\,\tau \qquad \text{Types}$$

We extend typing rules as follows. We add a rule WFPi to allow recursive type variables to occur in the body of an incoherent polymorphic type. Because incoherent polymorphic types are computational, they are well-founded as long as their body is non-expansive. Notice that we ask the recursive type variable not to be free in the kind of the abstract type variable. Rule TypePi is similar to rule TypeFor. An incoherent polymorphic type has the star kind if its body has the star kind under an extended environment.

$$\frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \notin \mathsf{fv}(\kappa)}{\alpha \mapsto \Pi(\beta : \kappa)\,\tau : \mathsf{WF}}\ \textsc{RecPi} \qquad \frac{\Sigma, (\alpha : \kappa) \vdash \tau : \star}{\Sigma \vdash \Pi(\alpha : \kappa)\,\tau : \star}\ \textsc{TypePi}$$

Rule TermGen is the introduction rule for incoherent polymorphism. There are three main differences between TermGen and CoerGen, which is the introduction rule of coherent polymorphism. First, TermGen is a typing rule for terms, while CoerGen is a coercion rule, as their names suggest. This comes from the fact that incoherent polymorphic types are computational, while coherent polymorphic types are erasable. This implies the second difference: TermGen leaves a mark on the term, while CoerGen is erasable. Finally, TermGen only requires the kind to be well-formed, while CoerGen needs the kind to be coherent. These are very important distinctions, because relaxing the coherence condition for rule CoerGen would be unsound. Rule TermInst is the elimination rule of incoherent polymorphism. The only difference with rule CoerInst is erasability: TermInst leaves a mark while CoerInst is erasable.

$$\frac{\Sigma \Vdash \kappa \qquad a : \Sigma, (\alpha : \kappa); \Gamma \vdash \tau \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Lambda a : \Sigma; \Gamma \vdash \Pi(\alpha : \kappa)\,\tau}\ \textsc{TermGen} \qquad \frac{a : \Sigma; \Gamma \vdash \Pi(\alpha : \kappa)\,\tau \qquad \Sigma \vdash \sigma : \kappa}{a \circ : \Sigma; \Gamma \vdash \tau[\alpha/\sigma]}\ \textsc{TermInst}$$

Since computational types have non-erasable introduction and elimination rules, we cannot derive their η-expansion coercion as we can do with erasable types. Incoherent polymorphic types are computational and we thus need to add an η-expansion coercion rule CoerPi for them. Like other η-expansion rules, coinductive hypotheses are made accessible. This is due to the computational behavior of incoherent polymorphic types.

$$\frac{\begin{array}{c}\Sigma \Vdash \kappa \qquad \Sigma \vdash \Sigma' \qquad \Sigma, \Sigma', (\alpha' : \kappa') \vdash \tau' : \star \qquad \alpha \notin \mathsf{fv}(\Theta_0, \Theta_1, \Sigma', \kappa', \tau') \\ \Sigma, (\alpha : \kappa); \Theta_0, \Theta_1; \varnothing \vdash (\Sigma' \vdash \tau'[\alpha'/\sigma']) \rhd \tau \qquad \Sigma, (\alpha : \kappa), \Sigma' \vdash \sigma' : \kappa'\end{array}}{\Sigma; \Theta_0; \Theta_1 \vdash (\Sigma' \vdash \Pi(\alpha' : \kappa')\,\tau') \rhd \Pi(\alpha : \kappa)\,\tau}\ \textsc{CoerPi}$$

To understand the main ideas of this somewhat complicated rule, we have to draw its explicit coercion as an η-expansion context. We use the line between nodes for the turnstile: type environments are on the left and types are on the right. We start to read this typing derivation from the bottom, which is the resulting typing of the η-expansion $\Sigma \vdash \Pi(\alpha : \kappa)\,\tau$. We first use TermGen which moves the type binding from the type to the environment to get the typing $\Sigma, (\alpha : \kappa) \vdash \tau$. We then use the subcoercion and assume it binds Σ'. We do not do any assumption about the argument type of the subcoercion to be principal. We find it by using TermInst which gives a type substitution of α' in τ' for some σ'. By typing, we deduce that we need $\Sigma, (\alpha : \kappa), \Sigma' \vdash \sigma' : \kappa'$. Finally, we realize that we need to use a weakening rule to remove the extra binding $(\alpha : \kappa)$ to get the argument typing of the η-expansion $\Sigma, \Sigma' \vdash \Pi(\alpha' : \kappa')\,\tau'$. The weakening rule requires α not to be free in Σ', κ', and τ' (plus the coinduction hypotheses Θ_0 and Θ_1 which are not drawn).

$$
\begin{array}{rcl}
\Sigma, \Sigma' & \downarrow & \Pi(\alpha' : \kappa')\,\tau' \\
 & \text{weak} & \\
\Sigma, (\alpha : \kappa), \Sigma' & \downarrow & \Pi(\alpha' : \kappa')\,\tau' \\
 & \odot \longleftarrow & \Sigma, (\alpha : \kappa), \Sigma' \vdash \sigma' : \kappa' \\
\Sigma, (\alpha : \kappa), \Sigma' & \downarrow & \tau'[\alpha'/\sigma'] \\
 & \text{coer} & \\
\Sigma, (\alpha : \kappa) & \downarrow & \tau \\
 & \Lambda & \\
\Sigma & \downarrow & \Pi(\alpha : \kappa)\,\tau
\end{array}
$$

The well-formedness of kind κ and the well-kindness of type τ' are for extraction purposes. Notice that we do not ask coherence for the kind κ because this is the η-expansion of the incoherent polymorphic type. However, we ask for the coherence of the type environment extension Σ' under Σ. This is a very important premise because we do not want the incoherence of κ to leak in Σ' and thus under the coercion, because coercion are erasable.

A simple counter-example would be to use rule CoerGen to prove $\Sigma, (\alpha : \kappa); \varnothing; \varnothing \vdash (\varnothing, (\beta : \kappa) \vdash \tau) \triangleright \forall(\beta : \kappa)\,\tau$, which holds using α as a witness. We could then give this sub-coercion to rule CoerPi with instantiation type α to get $\Sigma; \varnothing; \varnothing \vdash (\varnothing, (\beta : \kappa) \vdash \Pi(\alpha : \kappa)\,\tau) \triangleright \Pi(\alpha : \kappa)\,\forall(\beta : \kappa)\,\tau$. We could thus build an erasable coercion that extends the environment with a potentially incoherent binding like $(\beta : 1 \mid (\varnothing \vdash \mathsf{Bool}) \triangleright \mathsf{Int})$.

The indexed calculus is extended accordingly to the extensions of the λ-calculus. We define the interpretation of $\Pi(\alpha : \kappa)\,\tau$ under η as a variant of the arrow type and the coherent polymorphic type, because it is a computational type and it behaves as an intersection in terms of typing. The main difference is that we do not ask the inner term to be sound. All these extensions are formalized in Coq and thus proved sound.

$$|\Pi(\alpha : \kappa)\,\tau|_\eta \stackrel{\mathsf{def}}{=} \Diamond \left\{ \Lambda^k\, e \mid k > 0 \Rightarrow \forall x \in |\kappa|_\eta \;\; \lfloor e \rfloor_{k-1} \in |\tau|_{\eta, \alpha \mapsto x} \right\}$$

Let's illustrate a practical case when incoherent polymorphism is useful. Let's first define existentials by CPS encoding (see Section 6.1.2). Because we have two notions of polymorphism, coherent and incoherent, we also have two notions of existentials: coherent and incoherent. We write $\exists(\alpha : \kappa)\,\tau$ for coherent existential types and $\Sigma(\alpha : \kappa)\,\tau$ for incoherent existential types with the following definitions:

$$
\begin{array}{rl}
\text{coherent:} & \exists(\alpha : \kappa)\,\tau \stackrel{\mathsf{def}}{=} \forall\beta\,(\forall(\alpha : \kappa)\,(\tau \to \beta)) \to \beta \\
\text{incoherent:} & \Sigma(\alpha : \kappa)\,\tau \stackrel{\mathsf{def}}{=} \forall\beta\,(\Pi(\alpha : \kappa)\,(\tau \to \beta)) \to \beta
\end{array}
$$

We define the pack and unpack term syntactic sugar for the coherent existential, and ipack and $\mathsf{iunpack}$ for their incoherent version. Notice that the body of the $\mathsf{iunpack}$ sugar is hidden under an incoherent type abstraction, and as such is allowed to be unsound because it cannot be reduced.

$$\begin{array}{lll} \text{coherent:} & \mathsf{pack}\,a \stackrel{\mathsf{def}}{=} \lambda x\,x\,a & \mathsf{unpack}\,a\,\mathsf{as}\,x\,\mathsf{in}\,b \stackrel{\mathsf{def}}{=} a\,(\lambda x\,b) \\ \text{incoherent:} & \mathsf{ipack}\,a \stackrel{\mathsf{def}}{=} \lambda x\,x \circ a & \mathsf{iunpack}\,a\,\mathsf{as}\,x\,\mathsf{in}\,b \stackrel{\mathsf{def}}{=} a\,(\Lambda\,\lambda x\,b) \end{array}$$

Let's assume we have type-level functions (see Section 6.1.3) and sum types as in Section 2.4.3. We also use the erasable isomorphism notation of Section 5.4.1. We can now define the following GADT, named Term, and with kind $\star \to \star$:

$$\begin{array}{rl} \mathsf{Term}\,\alpha & \stackrel{\mathsf{def}}{=} \Sigma(\beta : \star \times \star \mid \alpha \equiv (\mathsf{fst}\,\beta \to \mathsf{snd}\,\beta))\,\alpha \\ & + \;\exists\beta\,\mathsf{Term}\,(\beta \to \alpha) \times \mathsf{Term}\,\beta \end{array}$$

This GADT is the sum of an incoherent existential type and a coherent existential type. The incoherent existential type asks α to be an arrow type and stores a term of such type. It also names $\mathsf{fst}\,\beta$ the argument type and $\mathsf{snd}\,\beta$ its return type. The coherent existential type adds no constraint on α but stores a pair such that its first component applied to its second component is of type α. It names β the intermediate type. The Term GADT contains two constructors: one for the left-hand side of the sum injecting functions and one for the right-hand side of the sum freezing function applications. We can define its two constructors in the following manner:

$$\begin{array}{rcll} \mathsf{Lam}\,x & \stackrel{\mathsf{def}}{=} & \mathsf{inl}\,(\mathsf{ipack}\,x) & : \forall\alpha\,\forall\beta\,(\alpha \to \beta) \to \mathsf{Term}\,(\alpha \to \beta) \\ \mathsf{App}\,y\,x & \stackrel{\mathsf{def}}{=} & \mathsf{inr}\,(\mathsf{pack}\,\langle y, x\rangle) & : \forall\alpha\,\forall\beta\,\mathsf{Term}\,(\alpha \to \beta) \to \mathsf{Term}\,\alpha \to \mathsf{Term}\,\beta \end{array}$$

We can now define a recursive eval function taking a term of type $\mathsf{Term}\,\alpha$ and returning a term of type α for all type variable α. Said otherwise, eval has type $\forall\alpha\,\mathsf{Term}\,\alpha \to \alpha$. When the argument is on the left-hand side of the sum, eval simply unpacks the inside argument and returns it. When the argument is on the right-hand side of the sum, eval unpacks the inside argument, which is a pair. It applies the evaluation of the first component to the evaluation of the second component. On the left-hand side, we use the incoherent version of unpack, while we use the coherent version on the right-hand side.

$$\begin{array}{l} \mathsf{eval}\,x = \mathsf{case}\,x\,\mathsf{of}\,\{\mathsf{inl}\,x_1 \mapsto \mathsf{iunpack}\,x_1\,\mathsf{as}\,y\,\mathsf{in}\,y \\ \qquad\qquad\qquad\qquad \mid \mathsf{inr}\,x_2 \mapsto \mathsf{unpack}\,x_2\,\mathsf{as}\,y\,\mathsf{in}\,(\mathsf{eval}\,(\mathsf{fst}\,y))\,(\mathsf{eval}\,(\mathsf{snd}\,y))\} \end{array}$$

Let's now suppose that we call eval with a term of type $\mathsf{Term}\,(\tau\times\sigma)$. This term is necessarily from the right-hand side of the sum because $\tau \times \sigma$ cannot be equivalent to an arrow type by consistency. However, in the first branch, in the body of the inconsistent unpack, we have access to the proposition $\tau\times\sigma \equiv \mathsf{fst}\,\beta \to \mathsf{snd}\,\beta$ which is inconsistent. This sort of inconsistency in some branches of case expression on GADTs is frequent. Notice however, that in the second branch we can reduce for any instantiation of α because we used a coherent existential type: there is a witness for β for any instance of α.

Weak reduction If the language is equipped (in addition to our current strong term abstraction) with a weak term abstraction, *i.e.* a term abstraction under which reduction is not allowed, then it is possible to reuse this existing abstraction to implement incoherent type abstraction. Let's write $\Lambda x\,a$ this weak term abstraction. It can use the existing application construct, because weak reduction is a property of the abstraction, not the application. We do not extend evaluation contexts, and in particular we do not add $\Lambda x\,[]$. We add the value $\Lambda x\,a$ for weak term abstractions. And we extend errors accordingly. We extend the type system with the weak arrow type $\Pi(\alpha : \kappa)\tau \Rightarrow \tau$, which does a type abstraction first, and then a weak term abstraction.

$$\begin{array}{rcll} a, b & ::= & \ldots \mid \Lambda x\, a & \text{Terms} \\ v & ::= & \ldots \mid \Lambda x\, a & \text{Values} \\ r & ::= & \ldots \mid \mathsf{fst}\,(\Lambda x\, a) \mid \mathsf{snd}\,(\Lambda x\, a) & \text{Errors} \\ \tau, \sigma & ::= & \ldots \mid \Pi(\alpha : \kappa)\tau \Rightarrow \tau & \text{Types} \end{array}$$

We add a reduction rule similar to RedApp. The main difference is in the typing rule of this term abstraction. We also need to add a typing rule for application of such abstraction. Rule TermLamWeak tells that $\Lambda x\, a$ has type $\Pi(\alpha : \kappa)\tau \Rightarrow \sigma$ under Σ and Γ, if α is not free in Γ and if the body a has type σ after type abstraction $(\alpha : \kappa)$ and term abstraction $(x : \tau)$. Rule TermAppWeak tells that $a\, b$ has type $\tau_2[\alpha/\sigma]$, if a has type $\Pi(\alpha : \kappa)\tau_1 \Rightarrow \tau_2$, σ has kind κ, and b has type $\tau_1[\alpha/\sigma]$.

RedAppWeak
$$(\Lambda x\, a)\, b \rightsquigarrow a[x/b]$$

TermLamWeak
$$\frac{\Sigma \Vdash \kappa \qquad \Sigma, (\alpha : \kappa) \vdash \tau : \star \qquad a : \Sigma, (\alpha : \kappa); \Gamma, (x : \tau) \vdash \sigma \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Lambda x\, a : \Sigma; \Gamma \vdash \Pi(\alpha : \kappa)\tau \Rightarrow \sigma}$$

TermAppWeak
$$\frac{a : \Sigma; \Gamma \vdash \Pi(\alpha : \kappa)\tau_1 \Rightarrow \tau_2 \qquad \Sigma \vdash \sigma : \kappa \qquad b : \Sigma; \Gamma \vdash \tau_1[\alpha/\sigma]}{a\, b : \Sigma; \Gamma \vdash \tau_2[\alpha/\sigma]}$$

We can now simulate type abstraction $\Pi(\alpha : \kappa)\,\tau$ with $\Pi(\alpha : \kappa)1 \Rightarrow \tau$ and we can also simulate the usual weak term abstraction $\tau \Rightarrow \sigma$ with $\Pi(\alpha : 1)\tau \Rightarrow \sigma$. This enhances the fact that incoherent type abstractions are actually unit weak abstractions. Both views are actually equivalent for well-typed terms. We can encode the weak abstraction $\Lambda x\, a$ using type abstraction and strong abstraction to form $\Lambda\, \lambda x\, a$. And we can encode the application $a\, b$ for weak abstraction (which we know by typing) using type application and application to form $a \circ b$. Reciprocally, we can encode the type abstraction Λa as $\Lambda x\, a$ for some fresh term variable x. And we can encode the type application $a \circ$ as $a\, \langle\rangle$, where $\langle\rangle$ is the unit term.

5.6 Coq formalization

The Coq [9] formalization has been developed with Coq version 8.4pl2. It can be found online at `http://phd.ia0.fr/`. There is a Makefile, so it suffices to run `make` to compile the Coq files or `make html` to also build the html version. The formalization merges ideas from strong normalization proofs of System F and step-indexed techniques. The strong normalization proof techniques is initially inspired from [12] but adapted for soundness proofs and step-indices. The step-indexed techniques are inspired from [3] but modified for strong reduction.

Our formalization has evolved with each extension of the language: either in the type system features or in the presentation. The last major version differed from System $\mathsf{L^c}$ in features, presentation, but also in the formalization. The new features are the notion of kinds, the notion of propositions, incoherent polymorphism. The old presentation was using a list of type variables (of kind star) and a list of coercions for coherent polymorphism, and thus type and coercion abstraction. But the old version had additional formalized results (see later in this section), which are not yet migrated to System $\mathsf{L^c}$.

The main differences between the version of System $\mathsf{L^c}$ presented in this chapter and its formalization in Coq are the use of de Bruijn indices and the inlining of the extraction property. Using de Bruijn indices makes it a lot easier and cleaner to deal with binders. Inlining the extraction property is necessary for the induction to go well. However, it would be interesting to prove the extraction in Coq as we did for the last major version.

We will now give a small glimpse of the Coq code for the reader to find its way through the files, definitions, and lemma. Files prefixed with the capital letter `F` refer to the indexed calculus: this letter stands for fuel. Files prefixed with the capital letter `L` refer to the lambda calculus. Files without a prefix letter are more general, like `typesystem.v` which factorizes the type systems for both the indexed calculus and the lambda calculus. We describe the files in dependency order.

We first have a few independent files. The file `ext.v` defines the extensionality axioms we use. Propositional and functional extensionality are the only axioms used. The other extensionality rules are lemmas. The file `set.v` defines a type for potentially infinite sets as predicates in `Prop`. The file `minmax.v` defines the tactic `minmax` to deal with indices. Finally, the file `list.v` defines useful lemmas about lists, which we use for environments and mappings.

We can now define the indexed calculus in file `Flanguage.v`. We define the inductive `term` for terms. All constructors of this inductive are prefixed with a `nat` representing the index of the node. We then define a few functions to traverse terms, from which we define lifting of index predicates to term, and the `lift` and `subst` functions for de Bruijn lifting and substitution. We then define the reduction relation in the inductive `red`. We finally define errors in `Err` and valid terms `V`. What follows is a list of lemmas about lifting, substitutions, lowering, and other functions over indices.

We prove the strong normalization of the pure indexed calculus in file `Fnormalization.v`. This file is quite simple to follow: we define a `measure`, prove that it strictly decreases with reduction in `red_measure`, and finally prove that reduction is well founded in `wf_der`.

We can now define a semantics for this indexed calculus in file `Fsemantics.v`. We define the notion of interior in `Dec`, the notion of contraction in `Red`, and the notion of expansion in `Exp`. Using expansion, we can define the set of sound terms `OK`. We define pretypes in `C` and types in `CE`. We define the closure of a set in `Cl`, in order to define the arrow operator `EArr`, product operator `EProd`, and incoherent abstraction operator `EPi`. We show that these operators preserve types in `CE_EArr`, `CE_EProd`, and `CE_EPi`. We also define erasable types such as the coherent polymorphic type `EFor`, the top type `ETop`, the bottom type `EBot`, or recursive types `EMu`. We then define the notions we need to show that recursive types are equal to their unfolding. And we finally define the semantic judgment `EJudg` and the semantic typing rules of the STLC, such as `ELam_sem`. We also define a subtyping rule `ECoer_sem` and a distributivity rule `Edistrib` which will be used together to prove rule TermCoer.

Once that the indexed calculus is defined, we may define the lambda calculus and the functions to go back and forth between them in file `Llanguage.v`. The structure of this file is similar to `Flanguage.v` with the difference that it now contains a `drop` and `kfill` function to translate terms from one language to the other. It also contains the key lemma `drop_red_exists` for the bisimulation between the reduction relation of both languages.

Independently from the indexed calculus and the lambda calculus, we can define the type part of System $\mathsf{L^c}$: everything but the term judgment. This is done in `typesystem.v` and is actually shared by both `Ftypesystem.v` and `Ltypesystem.v`, which define the term judgment for the indexed type system and lambda type system respectively. The last two files are exactly the same up to indices. There are two things to explain in the treatment of the type system.

First, all the syntax is gathered is a single Coq inductive, namely `obj`. This simplifies a lot the treatment of operations on the syntax, such as lifting or substitution, which are defined only once. In order to keep track of syntactical classes, we define a judgment `cobj` to classify each object in its grammatical class. In the paper version, we naturally assume that everything is syntactically well-formed, while we have to state it explicitly in Coq.

Then, each premise of the type system is annotated with one of the three letter `C`, `E`, or `S`, or simply not annotated. When a premise is annotated with `C`, it is a syntactical well-formedness judgment, which is only present in the Coq version and assumed to be true in the paper version. When a premise is annotated with `E`, it is commented and only necessary for the extraction lemma to hold, and thus only present in the paper version. When a premise is annotated with `S`, it is used to strengthen the induction of the soundness proof but is a consequence of the extraction lemma. As such it is only present in Coq and not in the paper version. Ideally, we would parametrize the type system on whether we are in the `E` presentation where extraction holds or the `S` presentation used to prove soundness. And we would prove that each judgment in the `E` version can be translated to its `S` version. This parametrization and extraction proof was done for the last major version of the system. This is not a difficult proof, but it takes some time to prove it.

The proof that each judgment of the indexed type system is sound lies in `Fsoundness.v`. We start by defining a notion of semantic objects `sobj`. A semantic object is either a set of indexed terms, the unit object, or a pair of objects. We then define the signature of the interpretation of each syntactical class in `sem`. Kinds are interpreted as sets of semantic objects, types as semantic objects, propositions as indexed propositions, type environments as sets of semantic environments (lists of semantics objects because we use de Bruijn indices), coinduction environments as indexed propositions, and finally term environments as semantic term environments (lists of semantic types, when the syntactical environment is valid). All these interpretation are parametrized by an surrounding semantic environment. We define the interpretation function `Fsoundness.semobj` as a binary relation, but we show in `semobj_eq` that it behaves as a function. We then prove semantic lifting and substitution properties. And we finally prove the soundness of each judgment. The soundness of `jfoo` is proved in `jfoo_sound` (Lemma 99).

We finally lift this soundness proof from the indexed type system to the lambda type system in `Lsoundness.v`. We first define when a term a is sound for a least k steps in `OKstep` and when it is sound for all number of steps in `OK` by coinduction. We prove that these two notions coincide. We then show how to transpose soundness from the indexed calculus to the lambda calculus in `term_ge_OK` (Lemma 75). We prove that both type systems coincide and finally prove the soundness of System L^c in `jterm_sound` (Theorem 100).

We give a few examples of derivations in `examples.v`. We prove that the identity function $\mathsf{id} \stackrel{\mathsf{def}}{=} \lambda x\, x$ has type $\forall \alpha\, \alpha \to \alpha$, that the term $\mathsf{delta} \stackrel{\mathsf{def}}{=} \lambda x\, x\, x$ has type $\forall \beta\, \mu\alpha\, (\alpha \to \beta)$, and finally that the term $\mathsf{omega} \stackrel{\mathsf{def}}{=} \mathsf{delta}\, \mathsf{delta}$ has type $\forall \alpha\, \alpha$.

5.7 Discussion about the explicit version

What System L^c lacks to have an explicit version is subject reduction. A typical example where we loose subject reduction is inspired from Section 4.6.2 and involves a coercion variable in-between a redex, which requires decomposing coercions to remain well-typed after reduction.

Let's take the following definitions:

- $\Sigma \stackrel{\mathsf{def}}{=} \varnothing, (\alpha\beta \mid (\varnothing \vdash \mathsf{Int} \to \alpha) \triangleright \mathsf{Int} \to \beta)$
- $\Gamma \stackrel{\mathsf{def}}{=} \varnothing, (y : \alpha)$
- $a \stackrel{\mathsf{def}}{=} (\lambda x\, y)\, 3$

We first need to show that Σ is coherent. To do so, it suffices to take $\bot$ as a witness for α and $\top$ as a witness for β. We have $a : \Sigma; \Gamma \vdash \beta$ and $a \leadsto y$, but we do not have $y : \Sigma; \Gamma \vdash \beta$. We would have to prove $\alpha \triangleright \beta$ from our hypothesis $\mathsf{Int} \to \alpha \triangleright \mathsf{Int} \to \beta$. This decomposition on the right-hand side of the arrow type is actually semantically valid, and it is possible to add it and thus restore subject reduction for this example. However, it is unclear whether the decomposition on the left-hand side holds (see later in this section). We can adapt the counter-example to subject reduction to use the left-hand side decomposition. We take the following definitions:

- $\Sigma \stackrel{\mathsf{def}}{=} \varnothing, (\alpha\beta \mid (\varnothing \vdash \alpha \to \alpha) \triangleright \beta \to \alpha)$
- $\Gamma \stackrel{\mathsf{def}}{=} \varnothing, (y : \beta)$
- $a \stackrel{\mathsf{def}}{=} (\lambda x\, x)\, y$

The type environment Σ is coherent by taking $\top$ as the witness of α and $\bot$ as the witness for β. We have $a : \Sigma; \Gamma \vdash \alpha$ and $a \leadsto y$, but we do not have $y : \Sigma; \Gamma \vdash \alpha$. We would have to prove $\beta \triangleright \alpha$ from our hypothesis $\alpha \to \alpha \triangleright \beta \to \alpha$.

We can prove the decomposition for the right-hand side of the arrow type when the coercion is not binding anything. This means that the initial coercion is simply of the form $\tau' \to \sigma' \triangleright \tau \to \sigma$ and not of the form $(\Sigma' \vdash \tau' \to \sigma') \triangleright \tau \to \sigma$. From this coercion we can extract a coercion from σ' to σ. If the coercion was binding something, using a distributivity rule prior to this lemma should work.

Lemma 103 (Push right)**.** *If* S' *and* R *are types and* $\mathsf{R}' \to \mathsf{S}' \subseteq \mathsf{R} \to \mathsf{S}$ *holds, then* $\mathsf{S}' \subseteq \mathsf{S}$ *holds.*

Proof. Coq lemma `Fsemantics.Push_right_Arr`

Let $e \in \mathsf{S}'$ and show that $e \in \mathsf{S}$. Let k be the greatest index in e. We have $e \leq k$. Let x be a fresh variable, and in particular not free in e. We have $\lambda^{k+1} x\, e \in \mathsf{R}' \to \mathsf{S}'$ by definition of the arrow operator and because $\lfloor e \rfloor_k = e$. By hypothesis, we also have $\lambda^{k+1} x\, e \in \mathsf{R} \to \mathsf{S}$. And because all semantic types are inhabited, we use the arrow operator definition to get $\lfloor e \rfloor_k \in \mathsf{S}$ which concludes. ☐

It is less clear whether or not the decomposition on the left-hand side of the arrow operator holds. However, we can show that this decomposition does not hold in other semantics, like in weak reduction for example. Some semantics only consider closed terms and some others permit semantic types to be empty. One example of such semantics is when semantic types are sets of closed strong normalizing terms. In this setting, the bottom type is the empty set. And functions types from inhabited types to the bottom type are empty too. In particular $\mathsf{Int} \to \bot$ is empty and thus included in $\mathsf{Bool} \to \bot$ which is also empty. If we could decompose coercions on the left of arrow types, we would get $\mathsf{Bool} \triangleright \mathsf{Int}$ which is unsound.

So, for some settings, coercion decomposition is unsound. For our setting, this is an open question, although we have a positive result for the right-hand side of the arrow operator. And finally, for more restrictive settings, coercion decomposition is sound and thus part of the system. Such restrictive setting is FC that considers only equality coercions. Equality coercions enforce structural coercions which simplifies the framework, as the main difficulty comes from type generalization and instantiation being coercions. Another less restrictive simplification would be to remove polymorphism from the coercion judgment and use it only in the typing judgment. However, we would loose a lot of the current expressivity and generality.

Chapter 6

Discussions

This work should be applicable to other programming languages instead of the λ-calculus, because the idea is to say that once we defined approximations (or invariants) for programs, it is natural to study the order relation between these approximations, define a syntactical composable judgment to prove inclusions between invariants, and study how to abstract over this judgment. However what can be done in the λ-calculus has not yet been fully explored. Several extensions remain to be studied: from new erasable types to changes of the framework. These extensions are described in Section 6.1. We then discuss related works in Section 6.2, and applications in Section 6.3.

6.1 Extensions

In this section, we discuss a series of extensions. Some of them are only conjectures, while some have more technical materials. These materials may either be ideas to further study or strong candidates waiting to be proved. We detail for each extension in which state it lies and which parts may be not quite correct as stated.

6.1.1 Data types

The λ-calculus can be extended with sums $\tau + \sigma$, integers Int, booleans Bool, the unit type 1, and the void type 0. Extending System F^p_ι and System L^c to handle Int, Bool, and 1 should be anecdotal, but almost no work has been done. It is known that sums may raise difficulties, so extending our type systems with $\tau + \sigma$ and 0 may not be trivial a priori. We however assume these extensions.

6.1.2 Existentials

There are two ways to add existentials: one is to use their CPS encoding with polymorphic types and another one is to give them their own meaning as union. The first solution always work while the second one may raise difficulties.

Existentials as CPS encoding The well-known encoding from existentials with polymorphism also works in System L^c. We can define the type $\exists(\alpha : \kappa)\,\tau$ and add the following type equality (or notation): $\exists(\alpha : \kappa)\,\tau = \forall\beta\,(\forall(\alpha : \kappa)\,\tau \to \beta) \to \beta$. We can also define $\mathsf{pack}\,a$ as a sugar for $\lambda x\,x\,a$ and $\mathsf{unpack}\,a\,\mathsf{as}\,x\,\mathsf{in}\,b$ as a sugar for $a\,(\lambda x\,b)$.

This easy extension is actually is enough to express GADTs using incoherent polymorphism like in FC. See Section 5.5 for an example of GADTs using incoherent polymorphism, this encoding of existentials, and sum types.

Existentials as unions By contrast with intersection of types, union of types is not obviously a type in our semantics. Our semantics is the usual semantics for reducibility candidates modified with step-indices. Without the expansion-closure, it is not obvious that the union $\mathsf{R} \cup \mathsf{S}$ of two types is a type. The problem is that a term e in $\Diamond(\mathsf{R} \cup \mathsf{S})$ could a priori reduce to both e_1 in $\Delta\mathsf{R} \setminus \Delta\mathsf{S}$ and e_2 in $\Delta\mathsf{S} \setminus \Delta\mathsf{R}$ and then not be in $\mathsf{R} \cup \mathsf{S}$.

In the current setting, where the underlying programming language is the λ-calculus, it seems that e should always be in $\mathsf{R} \cup \mathsf{S}$ by a standardization argument [30]. However, this argument is already complex in the absence of indices and may not be applicable in the case of indices—or force us to have a more involved definition of indexing compatible with standardization. This is an interesting question to explore further.

6.1.3 Type-level functions

Type-level functions and recursive types at arbitrary kinds (Section 6.1.4) are the last element we need to feature user data-types. User defined data-types are usually a recursive type of a series of type abstractions followed by a sum of products. For instance, the type for lists is given by $\mathsf{list} \stackrel{\mathsf{def}}{=} \mu(\beta : \star \to \star)\, \lambda(\alpha : \star)\, 1 + (\alpha \times \beta\, \alpha)$ where 1 is the type containing exactly one element. It should be possible to extend System $\mathsf{L^c}$ with type functions, but nothing has been proved yet.

System F_ω (Chapter 30 of [26]) is the simplest type system featuring polymorphism and type-level functions. In order to extend System $\mathsf{L^c}$ with type-level functions, we need to extend syntactical types with the type abstraction $\lambda(\alpha : \kappa)\, \tau$ and type application $\tau\, \tau$. We also need to extend syntactical kinds with the arrow kind $\kappa \to \kappa$. We add the β-reduction rule at the type-level as an equality between types: $(\lambda(\alpha : \kappa)\, \tau)\, \sigma = \tau[\alpha/\sigma]$. We finally give the two kinding rules for type abstraction and application, and the well-formedness rule for the arrow kind.

$$\textsc{TypeLam}\ \frac{\Sigma \Vdash \kappa_1 \qquad \Sigma, (\alpha : \kappa_1) \vdash \tau : \kappa_2}{\Sigma \vdash \lambda(\alpha : \kappa_1)\, \tau : \kappa_1 \to \kappa_2} \qquad \textsc{TypeApp}\ \frac{\Sigma \vdash \tau : \kappa_1 \to \kappa_2 \qquad \Sigma \vdash \sigma : \kappa_1}{\Sigma \vdash \tau\, \sigma : \kappa_2} \qquad \textsc{WfKArr}\ \frac{\Sigma \Vdash \kappa_1 \qquad \Sigma \Vdash \kappa_2}{\Sigma \Vdash \kappa_1 \to \kappa_2}$$

This is all we need in order to add type-level functions. We can recover subtyping rules and the notion of variance with the notions we already have: polymorphic propositions and coercions. We do so by lifting our notion of coercions to higher kinds, as we did with isomorphisms in Section 5.4.1. However, the definition of subtyping at the arrow kind is not canonical and depends on the variance we want to consider. So we will index our subtyping relation with a vkind (variance kind), which is a kind where arrow kinds are annotated with a variance. Type systems with polarized higher-order subtyping [31] have a common notion for kinds and vkinds. We believe these are two distinct notions and keep them separated.

There are at least four possible variances, written d: $+$ for covariant, $-$ for contravariance, $=$ for invariance, and $\varnothing$ for bivariance. Vkinds, written K, are the star vkind $\star$, the unit vkind 1, the product vkind $K \times K$, the constrained vkind $\{\alpha : K \mid \mathrm{P}\}$, and the arrow vkind $K\,[\mathrm{d}]\!\!\to K$. We write $\lfloor K \rfloor$ the function from vkinds to kinds that drops the variance annotations. The only interesting case is $\lfloor K_1\,[\mathrm{d}]\!\!\to K_2 \rfloor$ which is defined by $\lfloor K_1 \rfloor \to \lfloor K_2 \rfloor$.

We write $\tau \triangleright \sigma : K$ to say that τ is smaller than σ at vkind K. Both types τ and σ have kind $\lfloor K \rfloor$. We may consider heterogeneous subtyping later, and focus on homogeneous subtyping for simplicity. Notice that it makes no sense to talk about binding subtyping (or coercions) at higher kinds, because the notion of binding is a notion of terms and thus only relevant at the kind star. The subtyping at vkind K is inductively defined. The first cases are straightforward:

$$\begin{array}{ll} \tau \triangleright \sigma : \star & \stackrel{\mathsf{def}}{=} (\varnothing \vdash \tau) \triangleright \sigma \\ \tau \triangleright \sigma : 1 & \stackrel{\mathsf{def}}{=} \top \\ \tau \triangleright \sigma : K_1 \times K_2 & \stackrel{\mathsf{def}}{=} (\mathsf{fst}\,\tau \triangleright \mathsf{fst}\,\sigma : K_1) \wedge (\mathsf{snd}\,\tau \triangleright \mathsf{snd}\,\sigma : K_2) \\ \tau \triangleright \sigma : \{\alpha : K \mid \mathrm{P}\} & \stackrel{\mathsf{def}}{=} \tau \triangleright \sigma : K \end{array}$$

For the arrow vkind, there is one definition for each possible variance:

$$\begin{array}{ll} \tau \triangleright \sigma : K_1 [+]\!\!\mapsto K_2 & \stackrel{\mathsf{def}}{=} \forall(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor \mid \alpha \triangleright \beta : K_1)\, \tau\,\alpha \triangleright \sigma\,\beta : K_2 \\ \tau \triangleright \sigma : K_1 [-]\!\!\mapsto K_2 & \stackrel{\mathsf{def}}{=} \forall(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor \mid \beta \triangleright \alpha : K_1)\, \tau\,\alpha \triangleright \sigma\,\beta : K_2 \\ \tau \triangleright \sigma : K_1 [=]\!\!\mapsto K_2 & \stackrel{\mathsf{def}}{=} \forall(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor \mid \alpha \triangleright \beta : K_1 \wedge \beta \triangleright \alpha : K_1)\, \tau\,\alpha \triangleright \sigma\,\beta : K_2 \\ \tau \triangleright \sigma : K_1 [\varnothing]\!\!\mapsto K_2 & \stackrel{\mathsf{def}}{=} \forall(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor)\, \tau\,\alpha \triangleright \sigma\,\beta : K_2 \end{array}$$

We recover the usual definition that a type τ of kind $\star \to \star$ has variance d if the subtyping proposition $\tau \triangleright \tau : \star [\mathrm{d}]\!\!\mapsto \star$ holds, by looking at the diagonal of the subtyping relation. We can actually see each vkind K as a constrained kind $\{\alpha : \lfloor K \rfloor \mid \alpha \triangleright \alpha : K\}$ as it is done in polarized higher-order subtyping systems. This definition permits to talk about the kind $\star [+]\!\!\mapsto \star$ of covariant type functions, for instance.

We also recover the well-known facts that a bivariant type is also covariant and contravariant, and that covariant or contravariant types are also invariant. We can actually show in System L^{c} the proposition that bivariance implies covariance (and also the other implications), which gives us access to variance demoting. The proposition is:

$$\forall(\alpha : 1 \mid \tau \triangleright \sigma : K_1 [\varnothing]\!\!\mapsto K_2)\, \tau \triangleright \sigma : K_1 [+]\!\!\mapsto K_2$$

This is a polymorphic proposition over the constrained kind that contains the unit type $\langle\rangle$ of kind 1 if τ is in the bivariance relation with σ at kind $K_1 \to K_2$. The proof uses rule PropGen to introduce both $(\alpha : 1 \mid \tau \triangleright \sigma : K_1 [\varnothing]\!\!\mapsto K_2)$ and $(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor \mid \alpha \triangleright \beta : K_1)$ in the type environment. We show $\tau\,\alpha \triangleright \sigma\,\beta : K_2$ by first fetching α with rule TypeVar, then applying rule PropRes to extract its proposition $\forall(\alpha\beta : \lfloor K_1 \rfloor \times \lfloor K_1 \rfloor)\, \tau\,\alpha \triangleright \sigma\,\beta : K_2$, and finally using PropInst with α and β.

6.1.4 Recursive types at arbitrary kinds

Our current version of recursive types is of the form $\mu\alpha\,\tau$ and only permits to build types of the star kind. Moreover, the folding and unfolding of recursive types are coercions and not type equalities. We showed in Section 5.4.4 that we do not loose the power of equi-recursive types by using coercions. But the kind star restriction for recursive types is a real limitation when the type system has other sorts of kinds. Recursive types at product kinds permit to build mutually recursive types, while arrow kinds permit to build data-structures (see Section 6.1.3). We do not know how to exactly extend System L^{c} with recursive types of arbitrary kinds, but we have a privileged path to follow, which we describe below.

Recursive types at arbitrary kind would be written $\mu(\alpha : \kappa)\,\tau$. Since their unfolding rule is at kind κ, it cannot be a coercion anymore and has to be defined as a type equality $\mu(\alpha : \kappa)\,\tau = \tau[\alpha/\mu(\alpha : \kappa)\,\tau]$. This implies that the type equality judgment needs to check the well-foundness of recursive types, for the folding and unfolding to be sound. The kinding rule for the recursive type has to be modified to take the kind into account:

$$\textsc{TypeMu}\quad \frac{\alpha \mapsto \tau : \mathsf{WF} \qquad \Sigma, (\alpha : \kappa) \vdash \tau : \kappa}{\Sigma \vdash \mu(\alpha : \kappa)\,\tau : \kappa}$$

We also need to extend the notion of well-founded and non-expansive functors to all type constructs. This extension will follow the intuition that all occurrences of a recursive type variable have to cross a computational type for the recursive type to be well-formed. To prove the soundness of such rules, we extend the notion of k-approximations to all mathematical objects: not only sets of terms in System L^{c}, but also to the unit object, pairs of objects, and functions on objects (see Section 6.1.3). We would also need to define the limit of a series of objects when k approaches infinity.

A notion of sort has to be added for mathematical objects: sets, unit, products, and arrows. This corresponds to a simply-typed object calculus. The k-approximation and limit operators would depend on the sort of objects we give them. We have four k-approximation:

- $\langle \mathsf{R} \rangle_k^{\text{set}} = \{e \in \mathsf{R} \mid e < k\}$
- $\langle \mathsf{R} \rangle_k^{1} = \langle\rangle$
- $\langle \mathsf{R} \rangle_k^{s_1 \times s_2} = \langle \langle \mathsf{fst}\,\mathsf{R} \rangle_k^{s_1}, \langle \mathsf{snd}\,\mathsf{R} \rangle_k^{s_2} \rangle$
- $\langle \mathsf{R} \rangle_k^{s_1 \to s_2} = \mathsf{X} \mapsto \langle \mathsf{R}\,\mathsf{X} \rangle_k^{s_2}$ or $\mathsf{X} \mapsto \langle \mathsf{R}\,\langle \mathsf{X} \rangle_k^{s_1} \rangle_k^{s_2}$

We η-expand the objects according to their sort and use the approximation for their subsorts for their subcomponents. We use the same mechanism for limits. We have not fully explored this idea and do not know whether or not the details can be worked out. It may also require that type functions be non-expansive.

6.1.5 Non-erasable coercions

In this thesis, we have only studied erasable coercions. What would be a more general notion of coercions? There are several ways to extend erasable coercions to non-erasable ones and generalize the notion of coercions. We did not study erasable coercions, although we feature the example of the next paragraph.

A first example of non-erasable coercions is given in FC. A coercion is represented by the unit term with no information. Abstracting over a coercion results in a simple abstraction without bindings because the unit element contains no information. The information of a coercion is only present at typing. This view is compatible with type-erasure in the sense that no type is needed at runtime. However, the unit element has to be present during runtime, because, although it does not contain information in itself, its sole existence is a proof that code depending on it behaves correctly. We developed this example of non-erasable coercions with incoherent polymorphism in Section 5.5.

A second definition of non-erasable coercions is to simulate a rich runtime over a poor runtime. Coercions would be erasable in the rich runtime while non-erasable in the poor one.

An illustration of non-erasable coercions in this setting are record subtyping. A rich runtime may freely forget additional fields, while the poor runtime would need to copy the record value without its additional fields. For example, in the rich runtime $\{\mathsf{name} = \text{"}\mathsf{bob}\text{"}; \mathsf{age} = 23\}$ accepts type $\{\mathsf{age} : \mathsf{Int}\}$, while in the poor runtime it only has type $\{\mathsf{name} : \mathsf{String}; \mathsf{age} : \mathsf{Int}\}$ and needs to be coerced with non-erasable code to have type $\{\mathsf{age} : \mathsf{Int}\}$. OCaml has actually both forms of coercions. The subtyping for variants and objects is using a rich runtime: coercions are erasable. And the subtyping for modules uses a poor runtime: coercions copy the module by η-expansion.

Finally, a loose definition is to allow any form of code as coercions. The typing implications of this definition is not obvious, but it may be a framework to define code inference by restraining the inferred code to be valid according to the coercion judgment. We might expect the main feature of such coercions to be η-expansion, because it follows the structure of types. The rule for coercion application would resemble the following typing rule:

$$\frac{\overset{\textsc{TermCoer}}{M \Rightarrow \Gamma, \Sigma \vdash a : \tau} \qquad G \Rightarrow \Gamma \vdash C : (\Sigma \vdash \tau) \triangleright \sigma}{G\langle M\rangle \Rightarrow \Gamma \vdash C[a] : \sigma}$$

This rule differs from its preceding version in Figure 4.10 by the presence of a computational context C in the coercion judgment. Similarly to the computational term a in the term judgment, we now say that the coercion G erases to the context C. And we can see that the erasure of the coercion application $G\langle M\rangle$ is now a context application $C[a]$. It is interesting to study restrictions of this general setting in the framework of implicit coercions (see Section 6.2.3): boring coercions could be inferred.

6.1.6 First-class coercions

First-class coercions is the possibility to pass coercions as arguments to functions and to return coercions as objects. A simple way to achieve this feature is to use incoherent existential types (see Section 6.1.2 and 5.5). This solution does not permit to reduce parts of terms relying on first-class coercions to preserve soundness.

We actually define first-class propositions, which imply first-class coercions, because coercions are propositions. We write [P] the first-class type for the proposition P. It is a notation for $\Sigma(\alpha : 1 \mid \mathrm{P})\, 1$, where 1 is the type containing only the unit term. When we receive a first-class proposition as argument we may unpack it to access the proposition in the constrained kind. Since we use the incoherent version of the unpack construct, its body cannot be reduced until we know that the proposition holds, which we know when the associated pack construct is made accessible. We may also return such first-class proposition by packing. The incoherent packing construct has no reduction limitations.

One example of first-class propositions is to write a function of type $[\tau_1 \equiv \tau_2] \to \sigma$ which relies of the fact that τ_1 and τ_2 are equal to return a term of type σ.

6.1.7 Dependent types

A much more difficult extension would be to redefine the framework starting from the dependently typed λ-calculus instead of the STLC. In a dependently typed setting the return type of a term abstraction may mention its term argument. As a consequence, terms may appear in types. Or said otherwise, types may depend on terms, which is why dependent types are called so. The arrow type changes from $\tau \to \sigma$ to $\Pi(x:\tau)\,\sigma$ (which should not be confused with the incoherent polymorphic type $\Pi(\alpha:\kappa)\,\tau$), where the term argument x of type τ is bound in the return type σ. Rule TermLam shows this new typing for term abstraction. In rule TermApp, we can see that term application remembers its argument in its type. It is the return type σ of its function where the term argument variable x has been substituted with the actual argument b.

$$\textsc{TermLam}\quad \frac{a : \Gamma, (x:\tau) \vdash \sigma}{\lambda x\, a : \Gamma \vdash \Pi(x:\tau)\,\sigma} \qquad\qquad \textsc{TermApp}\quad \frac{a : \Gamma \vdash \Pi(x:\tau)\,\sigma \qquad b : \Gamma \vdash \tau}{a\, b : \Gamma \vdash \sigma[x/b]}$$

The implications of such extension on the framework and the semantics are not negligible. The following works handle erasable contents in a dependent setting. However, these works do not take a coercion approach with a distinct composable judgment for erasable typing transformations. They reuse the term judgment by adding implicit rules as we usually see.

The Implicit Calculus of Constructions (ICC) defined in [21] can be seen as an extension of System F_η to dependent types. The presentation uses the η-expansion term judgment rule, where a has type $\tau \to \sigma$ if $\lambda x\, a\, x$ has type $\tau \to \sigma$ and x is not free in a. The subtyping judgment (between types) is derived from the term judgment. System L^{c} extended with dependent types would extends ICC with coercion abstraction. Some major difficulties are expected when dealing with strong normalization, types (which are terms) inhabitation, and semantics. A work by Bruno Barras and Bruno Bernardo [5] gives an explicit version to a restriction of ICC. Large elimination is handled in [1].

Instead of extending the framework to handle dependent types, another solution is to simulate dependent type features by lifting terms to types and types to kinds, and using singleton types to link term values to their associated types. This solution is used in System FC (see Section 6.1.8).

6.1.8 Kind coercions

System FC has kind equality coercions [36]. The goal of such feature is to give GHC a type system with dependent type features. Actually, they achieve this by merging types and kinds into the same syntactical class. As a consequence, type equality coercions naturally extend to kind equality coercions. In System L^{c}, studying kind coercions (a form of subkinding) seems natural, because kinds are sets of types and are thus naturally ordered. In particular the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ is by definition a subkind of κ. But we also have that $\{\alpha : \kappa \mid \mathrm{P}_1\}$ is a subkind of $\{\alpha : \kappa \mid \mathrm{P}_2\}$ if P_1 implies P_2.

Notice that we already have a notion of subkinding for coherence: a kind κ is a subkind of the kind κ' under the type environment Σ, if the coherence of κ under Σ implies the coherence of κ' under Σ. We can prove such propositions with rule PropStr where the conclusion proposition is $\Sigma; \varnothing; \varnothing \vdash \exists\, \kappa'$ and the kind coherence premise is $\Sigma \vdash \kappa$. This notion of subkinding for coherence is limited and cannot be used to rekind a type. However, we somehow use this idea of coherence propagation with the instance type σ' in the η-expansion rule of incoherent polymorphism CoerPi.

There are two alternatives to study kind coercions. The first path is to keep types and kinds distinct and develop a new theory about kind coercions. Another path is to mimic FC and merge types and kinds and reuse the type coercions mechanism for kinds too. The first solution may look safer, and maybe easier to prove, but it may also imply more duplication of rules.

6.1.9 Intersection types

We could perhaps add binary intersection types as coercions following the approach of Wells on *branching types* [37]. The same way we extended subtyping from types to typings, we could extend branching types to *branching typings*, which would be trees of typings where leaves are usual typings and nodes are chunks of typing environments. We would have the following grammar for invariants, instead of the usual single typing $\Gamma \vdash \tau$.

$$\Phi \ ::= \ \Gamma \vdash \tau \mid \Gamma\langle \Phi \cap \Phi \rangle \qquad \text{Branching typings}$$

For the intuition, we can define a function $\lfloor \cdot \rfloor$ from branching typings to sets of usual typings. We would have $\lfloor \Gamma \vdash \tau \rfloor$ be the singleton set containing the typing $\Gamma \vdash \tau$. And we would define $\lfloor \Gamma\langle \Phi_1 \cap \Phi_2 \rangle \rfloor$ as $\{\Gamma, \Gamma' \mid \Gamma' \in \lfloor \Phi_1 \rfloor \cup \lfloor \Phi_2 \rfloor\}$. Intuitively, if a term accepts the branching typing Φ, it means that it accepts all the typings in $\lfloor \Phi \rfloor$. In other words, a term is in the semantics $|\Phi|$ if it is in the intersection of the semantics of the elements of $\lfloor \Phi \rfloor$.

We would also need to modify the usual STLC typing rules to act simultaneously on all leaves. For instance, we would need to define $\Phi_1 \times \Phi_2$, where Φ_1 and Φ_2 are binary trees with the same structure where only the types at the leaves may change, as the same binary tree where leaves $\Gamma \vdash \tau_1$ and $\Gamma \vdash \tau_2$ become $\Gamma \vdash \tau_1 \times \tau_2$. Coercions are now between these branching typings. We also need to add coercions from $\tau \cap \sigma$ to τ and σ, and a coercion from the branching typing $\langle (\varnothing \vdash \tau) \cap (\varnothing \vdash \sigma) \rangle$ to the type $\tau \cap \sigma$.

6.1.10 Semantic consistency

In System L^{c}, the coherent polymorphism abstraction rule is well-typed when the kind is inhabited. When the kind is a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, the proposition P has to hold for at least one inhabitant of κ. Currently, this proof of coherence is done using syntactical and concrete witnesses. We could extend the logic to prove propositions with more semantic proofs, or we may delay proof of propositions to top-level and use some algorithm to check them. We can already model this last feature using polymorphic propositions and a top-level type environment Σ_0 with all the propositions we will ever use in the derivation. For example, when we need to prove $\Sigma; \Theta_0; \Theta_1 \vdash \mathrm{P}$, we may add a top-level type binding of the form $(\alpha : 1 \mid \forall \Sigma\, \mathrm{P})$ in Σ_0. Then, instead of proving the coherence of Σ_0 using the syntactical rules of Figure 5.7, we may prove its semantic interpretation using mathematics, or with an algorithm. This is actually the solution followed in Constraint ML.

6.1.11 Environment actions

The framework of coercions defined in Part II does not handle coercions that would act on the environment in other ways than bindings. For instance, the weakening coercion shrinks the current environment while usual constructs extend the environment.

If we were considering coercions in their generality and not only coercion extending environments, how should we present them? They should still be written with the judgment given in System F_ι^p, namely $G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$, however, the meaning of the erasable environments Σ would change to an environment action. An environment action is the modification a coercion may do on its surrounding environment. For instance, in System F_ι^p with erasable environments, the two environment actions we have are: type binding and coercion binding.

However we all know that we may come up with more environment actions like: weakening, binding swapping, or environment shuffling.

As a consequence of this modification from erasable environment to environment actions, we need to modify the notion of simple concatenation we had Γ, Σ, to a more subtle environment actions composition $\Gamma + \Sigma$. For example, if we define $\mathsf{W}\alpha$ the weakening binding for α, then the following environment actions application $\Gamma, \alpha, \Gamma' + \mathsf{W}\alpha$ evaluates to Γ by removing the type binding α and all the following bindings that may depend on it. The term typing rule for coercions now becomes:

$$\textsc{TermCoer}\quad \frac{M \Rightarrow a : \Gamma + \Sigma \vdash \tau \qquad G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma}{G\langle M\rangle \Rightarrow a : \Gamma \vdash \sigma}$$

and the coercion typing rules for transitivity becomes:

$$\textsc{CoerTrans}\quad \frac{G_1 \Rightarrow \Gamma + \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \triangleright \tau_2 \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_3}{G_1 \circ G_2 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \triangleright \tau_3}$$

where Σ_2, Σ_1 is the concatenation or sequencing of environment actions: we first do the actions of Σ_2, followed by the actions of Σ_1: $\Gamma + (\Sigma_2, \Sigma_1)$ evaluates like $(\Gamma + \Sigma_2) + \Sigma_1$.

Environment actions subsume the more intuitive notation $\Gamma \vdash (\Sigma_1 \vdash \tau_1) \triangleright (\Sigma_2 \vdash \tau_2)$. Indeed, this last coercion can be encoded as $\Gamma, \Sigma_2 \vdash (-\Sigma_2 + \Sigma_1 \vdash \tau_1) \triangleright \tau_2$, where $-\Sigma_2$ is the opposite action of Σ_2. This coercion can be used in a similar way, because the typing $\Gamma, \Sigma_2, -\Sigma_2, \Sigma_1 \vdash \tau_1$ is actually equivalent to $\Gamma, \Sigma_1 \vdash \tau_1$.

6.1.12 Coercion reduction

When the coercion language gets bigger or when an intensive use of them is done, it becomes interesting to reduce them to make the typing derivations smaller. Coercion reduction may also be necessary to prove soundness (see Section 4.6.2) using the fact that coercions strongly normalize to a coercion value, which we can decompose.

In System F_ι^p, our term reduction relation contains β and ι steps. We never reduce coercions directly, but only when applied to terms. The reason is that there is no need for doing so since our coercions can be erased. Moreover, it keeps the presentation simpler.

Still, introducing a reduction relation $G \rightsquigarrow_\gamma G$ between coercions themselves is possible, and could be interesting in other contexts. For example, coercions in System FC [35] are elaborated from the surface language, inferred from the constraint solver, and made bigger by term optimizations. In all these cases, coercions may grow very large. Hence, reducing coercions is a way to make them smaller.

We briefly describe how to add coercion reduction in our language. Since we use strong reduction, coercion evaluation contexts are all coercion contexts:

$$\textsc{RedIGamma}\quad \frac{G_1 \rightsquigarrow_\gamma G_2}{G_1\langle M\rangle \rightsquigarrow_\iota G_2\langle M\rangle} \qquad\qquad \textsc{RedGCtx}\quad \frac{G_1 \rightsquigarrow_\gamma G_2}{F[G_1] \rightsquigarrow_\gamma F[G_2]}$$

Rule RedIGamma is the ι-reduction linking γ-reduction to ι-reduction. If a coercion reduces, then its application to a term reduces too. Rule RedGCtx is the context rule for

γ-reduction. Then, some obvious reduction rules are pushing transitivity down, pulling reflexivity up, and reducing coercion redexes:

$$\textsc{RedGEtaArr}\quad (G_1 \stackrel{\tau}{\to} G_2) \circ (G_1' \stackrel{\tau'}{\to} G_2') \rightsquigarrow_\gamma (G_1' \circ G_1) \stackrel{\tau}{\to} (G_2 \circ G_2')$$

$$\textsc{RedGEtaProd}\quad (G_1 \times G_2) \circ (G_1' \times G_2') \rightsquigarrow_\gamma (G_1 \circ G_1') \times (G_2 \circ G_2') \qquad \textsc{RedGFold}\quad \mathsf{unfold} \circ \mathsf{fold}_{\mu\alpha.\tau} \rightsquigarrow_\gamma \Diamond$$

Rules RedGEtaArr, RedGEtaProd, and RedGFold are simple transposition of their ι counterparts (which could then be removed). These reduction rules are sound. They are also incomplete: other rules, which are not derivable could also be added—but they require other operations on coercions that have not been defined yet. For instance, all ι-reduction rules involving no untyped constructs on their left-hand side could be transformed into the combination of an equivalent γ-reduction rule and rule RedIGamma. For instance, one remaining case is rule RedType. If we had a coercion for explicit type substitution, we would be able to have the following γ-reduction rule:

$$\textsc{RedGType}\quad \bullet\tau \circ \Lambda\alpha \rightsquigarrow_\gamma [\alpha/\tau]$$

Notice that one must now prove the termination of γ-reduction rules since they are included in ι-reduction through rule RedIGamma.

6.1.13 Side effects

Extending the λ-calculus with side effects is a natural question since real-world programs do have side effects. The main modification is to add a notion of memory and how programs interact with it. Since this memory has a semantic meaning, invariants have to be stated according to the memory. Invariants are no longer a pair of an environment and a type, but a triplet of an environment, a memory invariant, and a type. It should be possible to adapt the step-indexed semantics according to [3].

6.1.14 Dead code

Although incoherent polymorphism is required to express GADTs (see Section 5.5), we might still want to discriminate between potentially inconsistent branches and always inconsistent branches. A branch is potentially inconsistent if its coherence depends on the instantiation of its environment, while a branch is always inconsistent if it is incoherent for all its environment instantiations. For example, the kind $\{\beta : 1 \mid \alpha \equiv \mathsf{Int}\}$ in the environment $\varnothing, \alpha$, is potentially incoherent. It is coherent if the type variable α is instantiated with Int, but it is incoherent if the type variable α is instantiated with Bool. Let's now consider the kind $\{\beta : 1 \mid \mathsf{Bool} \equiv \mathsf{Int}\}$ in the empty environment. It is always incoherent, and we might want to either reject terms generalizing on this kind or warn the user that he is writing dead code. Such type abstraction is actually not instantiable.

In pedagogical systems [8], an abstraction (such as type abstraction) is *useful*, if its domain (such as kinds in the case of type abstraction) is inhabited for some instantiation of the environment. This condition implies that the abstraction is not dead code and can be instantiated. We can use this idea to define an alternative version of incoherent polymorphism, by strengthening its condition from the kind to be well-formed to the kind to be coherent for some instantiation of the environment. This useful incoherent polymorphic type $\overline{\overline{\Pi}}(\alpha : \kappa)\, \tau$ would have the following introduction rule:

$$\frac{\exists \Sigma\, \kappa \qquad a : \Sigma, (\alpha : \kappa); \Gamma \vdash \tau \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Lambda a : \Sigma; \Gamma \vdash \overline{\overline{\Pi}}(\alpha : \kappa)\, \tau}\ \textsc{TermGenUseful}$$

The premise $\exists \Sigma, \kappa$ asks the kind κ to be coherent for some instantiation of Σ. This rule is in the term judgment, because the usefulness does not imply consistency and the abstraction cannot be erasable. We can reuse the existing type abstraction, which we already use for incoherent polymorphism, because we can always tell them apart by typing. The other rules for useful incoherent polymorphic types are identical to the incoherent polymorphic versions.

Such extension would tell us that we cannot reduce under the useful incoherent abstraction now, but that it is possible to reach this part of the code for some instantiation of the environment.

Unfortunately, this property is not stable by reduction, because reduction could instantiate the environment making the kind incoherent for this particular instantiation. If one want this dead code property to be stable by reduction, he would need to have some form of coherent polymorphism, which we already have.

6.2 Related work

To the best of our knowledge there is no previous work considering coercions as an inclusion of typings. However, the use of coercions to study features of type system is not at all new. This section presents how coercions have been used to study subtyping, GADTs, recursive types, etc.

Our work on step-indices is greatly inspired from existing works in weak reduction settings. We actually merged the typical semantic proofs of System F in a strong reduction setting, with the idea of fuel to limit the reduction of terms.

6.2.1 System $\mathsf{F}_{<:}$

Record subtyping in System $\mathsf{F}_{<:}$ may be compiled away into records without subtyping in plain System F by inserting coercions with computational content [7] that change the representation of records whenever subtyping is used. Since these coercions are not erasable and can be inserted in different ways, the soundness of the approach depends on a coherence result to show that the semantics of the translation does not actually depend on the places where coercions are inserted.

Another method for eliminating subtyping has been used by Crary [10]: bounded polymorphism $\forall(\alpha \triangleright \tau)\, \sigma$ is compiled away into an intersection type $\forall \alpha\, \sigma[\alpha/\alpha \cap \tau]$ while intersection types are themselves encoded with explicit erasable coercions. This directly relates to our work by their canonization, which is similar to our ι-reduction, and their use of bisimulation up to canonization to show erasability of coercions. Of course, the languages are different, as we do not consider intersection types while they do have neither coercion abstraction nor distributivity and only consider call-by-value reduction.

6.2.2 System FC

In System FC [32] (the core language of the Glasgow Haskell Compiler), coercions are bidirectional: they are proofs of equality instead of proofs of inclusion. As a consequence they are structural: when two types are equal according to an equality coercion, then their head type constructors are the same. This permits drastic simplifications and in particular to handle push much more easily. Notice that consistency already asks the computational head type constructors of coercions to be equal. In FC, this condition applies to all type constructors, even the erasable ones. As a consequence, the coercion $(\forall\alpha\,\tau \to \sigma) = (\tau \to \forall\alpha\,\sigma)$, when α is not free in τ, is ill-formed, even though these two types are semantically equal: $(\forall\alpha\,\tau \to \sigma) \equiv (\tau \to \forall\alpha\,\sigma)$ holds in System F_ι^p and System L^c.

Coercion abstractions are not fully erasable in System FC, but they are zero-bit, because they only have to do with typings and do not modify the computational content of the term they are applied to. They are not erased because they have to block the reduction, as coercion abstraction in FC is incoherent. See Section 5.5 to see why incoherent coercion are useful and how we extend System L^c with incoherent polymorphism. Only top-level coercion axioms are checked for consistency, because we have to reduce under them.

In System FC, coercions are not as powerful as other type systems, since they are only equality and structural, however they permit to define a number of derivable type system features such as GADTs, type families, type synonyms, etc.

System FC actually has an heterogeneous equality for coercions and provides an additional notion of kind equality. This feature permits to lift ([38] and [36]) all sorts of data types to the type level and use them to index singleton types in order to simulate dependent type mechanisms.

6.2.3 Implicit Coercions

The notion of implicit objects (implicit terms, implicit coercions, etc.) is a feature of surface languages and in particular inference. It is orthogonal to the notion of erasability. On the one hand, an object is erasable if it present during typing but absent at runtime. Typically, types and coercions are erasable is System L^c. On the other hand, an object is implicit if it is absent in the source code but may be present at runtime. Most of the time, inference elaborates implicit objects from the surface language to the kernel language. So when we study implicit objects, we study objects inference.

In Coq, coercions are not erasable but they are implicit. They are not added to extend the expressiveness of the language, but to lighten the source language. They permit to implicitly coerce a term of a particular type to another type, by giving particular functions which may be used in this manner.

Implicit coercions may also be used to express a wide variety of practical features, from dynamic software updating to provenance tracking [34]. However, the more powerful the inference of coercions, the greater the possibility of several semantically different elaborations. As a consequence, trade-offs between expressiveness and ambiguity have been studied. This notion of ambiguity can be solved by proving that all rewriting of the source term into a kernel term with coercions have the same semantics, no matter how they are introduced. This path is followed by [33] to ease programming with monads. The binds, units, and monad-to-monad morphisms necessary when programming in a monadic-style, are inferred based on types. This idea is called coherence in [6], which uses implicit coercions to control logical properties in pure type systems.

In coercive subtyping [19], a type τ is a subtype of σ whenever there is a unique coercion from τ to σ. Coercive subtyping extends an existing type theory T with a subtyping judgment C to form a new type system $T[C]$. The coherence of coercions comes from the equivalence of $T[C]$ with $T[C]^*$, which is a version of $T[C]$ where the position of coercions are marked. As a consequence, coercive subtyping lightens the source code without ambiguity. This work is done in a dependent type setting and solves the difficulty of Σ-types [18], which is that the congruence on the first component of a Σ-type and the first projection of a Σ-type form a set of incoherent coercions.

This approach of coercions as implicit objects is orthogonal to our work and may thus be used at the same time, to get the most of both worlds. Implicit coercions inference happens from the source language to the kernel language, while erasable coercions are a property of the erasure of the kernel language, which is independent.

6.2.4 Step-indices

Numerous works have been done on step-indices ([3], [13]), and all that we know of are in a weak-reduction setting. The reason is probably that step indices are usually used to give a semantics to programming languages with side effects, which usually come with a weak reduction strategy. In all these works there is a distinction between the values of type τ, written $\mathcal{V}[\tau]$, and expressions of type τ, written $\mathcal{E}[\tau]$ and defined by the terms having their normal form in $\mathcal{V}[\tau]$ with the remaining fuel. The set of values of arrow type $\tau \to \sigma$ is then the set of term abstractions $\lambda x\, a$ with index k such that for all indices j smaller than k and for all values v at j in $\mathcal{V}[\tau]$, the substitution $a[x/v]$ is $\mathcal{E}[\sigma]$ at j. This definition is not stable by reduction if we reduce under the abstraction. However, step-indices are not only used for side-effects, they are also used for recursive types, which makes perfect sense in a language with strong reduction, as we do. This is why we use them.

6.3 Applications

There are several applications to this work, although most of them would be more practical with some of the extensions discussed in Section 6.1. However, we can see System L^{c} as a good starting point for these applications.

Designing features that are easy to merge Our initial theoretical motivation was to express type systems features in a unique framework and in an orthogonal way. When type system features are written as coercions, they can easily be composed with pre-existing type system features. Besides the ease of merging, studying type system features as coercions naturally gives the most out of the fusion of two features. The typical example is for System $\mathsf{F}_{<:}$ which merges η-expansion (called subtyping) with upper bounded polymorphism. Merging these two features using coercions naturally gives the distributivity rule which is not present in System $\mathsf{F}_{<:}$.

Extending FC with subtyping A more practical motivation to study coercions in a general setting, was the use of type equality coercions in FC to express useful surface features such as GADTs, type families, and others. There are several main differences between System L^{c} and

FC. On the one hand, our type system only has an implicit version, although we discuss a possible explicit version in Section 5.7. On the other hand, our type system features subtyping. A very important and potentially difficult extension to consider are side-effects (Section 6.1.13). A less important but potentially difficult extension are kind coercions (Section 6.1.8).

The other differences between both systems are more anecdotal. Zero-bit coercion abstractions of FC may be added using incoherent polymorphism (Section 5.5). Type-level functions of FC may be added as explained in Section 6.1.3. Type equality coercions are simply erasable isomorphisms: τ is equal to σ if there is a coercion from τ to σ and reciprocally. Higher-kind recursive types are required to define user data-types (Section 6.1.4). Finally, coercion reduction (see Section 6.1.12) may be considered to optimize the compiler, because surface language elaboration, constraint solving, and code optimization may build large coercions and slow down the compilation phase.

Merging inference systems If two type systems with inference rely on the same programming language, and both type systems can be projected into a common kernel type system, then a program may use one or the other inference system for each of its compilation unit, as long as the interfaces of the compilation units are in the intersection of both type systems. A concrete example would be to use MLF and Constraint ML with ML types for interfaces.

MLF and Constraint ML are surface type systems with powerful inference mechanism. Merging the two inference systems may be a very hard task. However, if one only needs to write some parts of his program using the MLF inference system and other parts with the Constraint ML system, he can partition his compilation units according to the inference system he wishes to use for them. Each compilation unit should have an interface in ML, which is a common sublanguage of MLF and Constraint ML.

Suppose we have a module M written in MLF with an interface in ML and a module N written in Constraint ML with an interface in ML. As we showed in Section 5.4, MLF and Constraint ML are subsumed in System L^{c}. So, once M and N have been inferred in their respective language with the other module interface, we can translate their typing derivations to System L^{c} and then link them to get the final program. The linking works correctly since the interface types are the same.

Chapter 7

Conclusion

We presented two type systems, System F_ι^p and System L^c, based on a coercion framework. Coercions are an extension of subtyping from type ordering to typing ordering. The particularity of the coercion framework is that all type system features are expressed as coercions. The underlying programming language we use is the λ-calculus extended with pairs. The main object of study is coercion abstraction.

Both type systems are sound. They are also strongly normalizing after the removal of recursive types. A type system is sound (resp. strongly normalizing) when all well-typed terms are sound (resp. strongly normalizing). A term is sound when it cannot reach an error state. A term is strongly normalizing when it cannot indefinitely reduce. When these properties hold in a strong reduction setting, they also hold in all usual reduction strategies (weak reduction, call by value, call by need, etc.). We thus described our type systems and proved these properties in a strong reduction setting.

Coercions are erasable typing transformations in the implicit version and erasable contexts in the explicit version. System F_ι^p has an implicit and explicit version, while System L^c only has an implicit version. With an implicit version, coercions are naturally erasable since they are not present in the term. With an explicit version, coercions are erasable if they satisfy the bisimulation lemma, which tells that coercions should not introduce, remove, or block computational steps. System F_ι^p satisfies the bisimulation lemma.

Both type systems feature polymorphism, η-expansion, and coercion abstraction as coercions. The main difference is that coercion abstraction is restricted to parametric coercions in System F_ι^p, while it is unrestricted in System L^c. Parametric coercions are coercions where either the argument type or the return type is an abstract type. Parametric coercion abstraction can actually be seen as an extension from polymorphism to bounded polymorphism. In System L^c, we also extend polymorphism by extending the kinds on which abstract types may range, with a particular kind constraining the abstract type to satisfy a proposition, which may be a conjunction of coercions for instance. The η-expansion feature gives both type systems congruence rules for computational types. This permits to use coercions deeply in a type according to the variance of the path. This feature is at the root of subtyping and comes from System F_η.

Both type systems subsume System F_η, MLF, and System $\mathsf{F}_{<:}$. System L^c additionally subsumes $\mathsf{Constraint\ ML}$. As a consequence, MLF and $\mathsf{Constraint\ ML}$ which have particularly good inference mechanisms, can be seen as surface languages for System L^c. The main missing feature, to describe OCaml core type system and FC (GHC core type system), is side effects.

The remaining features are more anecdotal to study.

Another main difference with FC, is that FC has an explicit version with the subject reduction lemma, while only System F_ι^p does. System L^c is more expressive than System F_ι^p, since it features unrestricted coercion abstraction and unrestricted recursive types. But it only has an implicit version without subject reduction. Restoring subject reduction and defining an explicit version would require to study coercion decomposition and the push reduction rule. The main difficulty is the proof of consistency which is not completely clear to hold in our semantics. Moving polymorphism from the coercion language to the term language may be a solution to restore subject reduction and the explicit version at the cost of losing deep type and coercion abstraction and instantiation.

A last difference with FC, is that FC has top-level coherent coercion abstraction and local incoherent coercion abstraction. In System L^c, we permit also local coherent coercion abstraction by asking the kind of the type abstraction to be inhabited. We also permit incoherent coercion abstraction as an extension, which requires blocking abstraction at the computational level: such as unit abstraction or weak lambdas.

A final contribution of this thesis is an adaptation of step-indexed techniques to prove soundness in a strong reduction setting. Step-indexed techniques are usually studied in a weak reduction setting due to the presence of side effects in the considered languages. However, the definition they use for the arrow type does not work for strong reduction semantics. The reason comes from the fact that indices, representing the fuel of the terms, are external to the terms. As a consequence, substitution and reduction do not permute. We restore this commutation by internalizing the notion of indices inside terms.

Bibliography

[1] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1):1–36, 2012. TYPES'10 special issue.

[2] Roberto Amadio and Luca Cardelli. Subtyping recursive types. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 15(4):575–631, 1993.

[3] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems.*, 23(5), September 2001.

[4] Paolo Baldan, Giorgio Ghelli, and Alessandra Raffaetà. Basic theory of F-bounded quantification. *Inf. Comput.*, 153:173–237, September 1999.

[5] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *Proceedings of the Theory and practice of software, 11th international conference on Foundations of software science and computational structures*, FOSSACS'08/ETAPS'08, pages 365–379, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Gilles Barthe. Implicit coercions in type systems. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 1996.

[7] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[8] Loïc Colson and Vincent Demange. Investigations on a pedagogical calculus of constructions. *CoRR*, abs/1203.3568, 2012.

[9] The Coq Proof Assistant `http://coq.inria.fr/what-is-coq`.

[10] Karl Crary. Typed compilation of inclusive subtyping. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 68–81, New York, NY, USA, 2000. ACM.

[11] Julien Cretin and Didier Rémy. On the power of coercion abstraction. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 361–372, New York, NY, USA, 2012. ACM.

[12] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply-typed lambda-calculus and system f.

[13] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proceedings of the 2009 24th Annual IEEE Symposium on Logic In Computer Science*, LICS '09, pages 71–80, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Jean H. Gallier and Jean H. Gallier. On girard's candidats de réductibilité. In *Logic and Computer Science*. North, 1990.

[15] The Haskell Programming Language `http://www.haskell.org/haskellwiki/Haskell`.

[16] Didier Le Botlan and Didier Rémy. Mlf: raising ml to the power of system f. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP '03, pages 27–38, New York, NY, USA, 2003. ACM.

[17] Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6):726–785, 2009.

[18] Yong Luo and Zhaohui Luo. Combining incoherent coercions for sigma-types. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2003.

[19] Z. Luo, S. Soloviev, and T. Xue. Coercive subtyping: Theory and implementation. *Inf. Comput.*, 223:18–42, February 2013.

[20] Ralph Matthes. Monotone fixed-point types and strong normalization. In *Proceedings of the 12th International Workshop on Computer Science Logic*, pages 298–312, London, UK, UK, 1999. Springer-Verlag.

[21] Alexandre Miquel. The implicit calculus of constructions: extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th international conference on Typed lambda calculi and applications*, TLCA'01, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag.

[22] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76):211–249, 1988.

[23] The OCaml Programming Language `http://caml.inria.fr/ocaml/index.en.html`.

[24] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, January 1999.

[25] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *ACM Trans. Program. Lang. Syst.*, 17(4):576–599, July 1995.

[26] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[27] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2004.

[28] Francois Pottier. Simplifying subtyping constraints. In *In Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 122–133. ACM Press, 1996.

[29] Didier Rémy and Boris Yakobowski. A church-style intermediate language for mlf. In Matthias Blume, Naoki Kobayashi, and German Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin / Heidelberg, 2010.

[30] Colin Riba. On the stability by union of reducibility candidates. In *Proceedings of the 10th international conference on Foundations of software science and computational structures*, FOSSACS'07, pages 317–331, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] Martin Steffen. Polarized higher-order subtyping, 1997.

[32] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[33] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ml. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 15–27, New York, NY, USA, 2011. ACM.

[34] Nikhil Swamy, Michael Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 329–340, New York, NY, USA, 2009. ACM.

[35] Dimitrios Vytiniotis and Simon L. Peyton Jones. Evidence normalization in system fc (invited talk). In Femke van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 20–38. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[36] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System fc with explicit kind equality. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 275–286, New York, NY, USA, 2013. ACM.

[37] Joe B. Wells and Christian Haack. Branching types. In *Proc. of the European Symposium On Programming Languages and Systems*, 2002.

[38] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.

List of Figures

Index