On the Power of Coercion Abstraction

Julien Cretin, Didier Rémy

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On the Power of Coercion Abstraction

Julien Cretin — Didier Rémy

N° 7587

December 2011

Domaine 2

ISSN 0249-6399 ISRN INRIA/RR--7587--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

On the Power of Coercion Abstraction

Julien Cretin , Didier Rémy

Domaine : Algorithmique, programmation, logiciels et architectures
Équipes-Projets Gallium

Rapport de recherche n° 7587 — December 2011 — 59 pages

Abstract: Erasable coercions in System F_η, also known as retyping functions, are well-typed η-expansions of the identity. They may change the type of terms without changing their behavior and can thus be erased before reduction. Coercions in F_η can model subtyping of known types and some displacement of quantifiers, but not subtyping assumptions nor certain forms of delayed type instantiation. We generalize F_η by allowing abstraction over retyping functions. We follow a general approach where computing with coercions can be seen as computing in the λ-calculus but keeping track of which parts of terms are coercions. We obtain a language where coercions do not contribute to the reduction but may block it and are thus not erasable. We recover erasable coercions by choosing a weak reduction strategy and restricting coercion abstraction to value-forms or by restricting abstraction to coercions that are polymorphic in their domain or codomain. The latter variant subsumes F_η, $\mathsf{F}_{<:}$, and MLF in a unified framework.

Key-words: Type, System F, F-eta, Polymorphism, Coercion, Conversion, Retyping functions, Type containment, Subtyping, Bounded Polymorphism.

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

De l'Expressivité de l'Abstraction de Coercions

Résumé : Les coercions effaçables dans le Système F_η, aussi connues sous le nom de fonctions de retypage, sont des η-expansions de l'identité. Elles peuvent changer le type des termes sans en changer leur comportement et peuvent donc être effacées avant la réduction. Les coercions de F_η peuvent modéliser le sous-typage entre types connus ou le déplacement de quantificateurs, mais elles ne permettent pas certaines formes d'instanciation retardée ni de raisonner sous des hypothèses de sous-typage. Nous généralisons F_η en introduisant l'abstraction des fonctions de retypage. Nous suivons une approche générale où le calcul avec des coercions peut être vu comme une réduction dans le λ-calcul gardant trace de la partie des termes qui sont des coercions. Nous obtenons un langage où les coercions ne contribuent pas au calcul, mais peuvent le bloquer et ne sont donc pas effaçables. Nous retrouvons des coercions effaçables en choisissant une stratégie de réduction faible et en restreignant l'abstraction de coercions aux valeurs ou bien en restreignant l'abstraction aux coercions qui sont polymorphes en leur domaine ou en leur codomaine. Cette seconde variante généralise F_η, MLF et $\mathsf{F}_{<:}$ dans un cadre unifié.

Mots-clés : Types, Système F, Polymorphisme, Coercion, Conversion, Fonction de retypage, Type containment, Sous-typage, Bounded Polymorphism

Contents

1 Introduction **4**

2 The language F_ι **7**
2.1 Syntax of F_ι . . . 7
2.2 Typing rules . . . 9
2.3 Dynamic semantics . . . 11
2.4 Examples . . . 14

3 Properties of F_ι **15**
3.1 Soundness . . . 15
3.2 Termination of reduction . . . 17
3.3 Reification of F_ι in System F . . . 17
3.4 Confluence . . . 18
3.5 Forward simulation . . . 19

4 Coercions as retyping functions: F_ι^λ **19**
4.1 Definition of F_ι^λ . . . 20
4.2 Soundness . . . 23
4.3 Confluence . . . 24
4.4 Reification into System F . . . 25
4.5 Completeness . . . 25
4.6 Soundness . . . 26
4.7 Bisimulation between F_ι and F_ι^λ . . . 30

5 Parametric F_ι **30**
5.1 Syntax changes . . . 32
5.2 Adjustments to the semantics . . . 33
5.3 Properties . . . 34

6 Expressiveness of Parametric F_ι **36**

7 Weak F_ι **42**

8 Related work **45**

9 Discussion and future work **47**

A Delayed Proofs **50**

1 Introduction

When designing programming languages, types help choosing a small number of well-understood orthogonal language constructs; they also help programmers by ruling out all unsafe programs as ill-typed. However, type-safety is only an approximation of good-behavior by design: there will always remain useful well-behaved programs rejected as ill-typed as well as well-typed programs that don't behave as intended. These two gaps can be reduced simultaneously by increasing the expressiveness and accuracy of type systems and so capturing finer program invariants. Although this is an endless process, considerable progress has been made over the last couple of decades.

Parametric polymorphism and subtyping polymorphism are the two most popular means of increasing expressiveness of type systems: although first studied independently, they can be advantageously combined together. Each mechanism alone is relatively simple to understand and has a more or less canonical presentation. However, their combination is more complex. The most popular combination is the language $\mathsf{F}_{<:}$ [Cardelli, 1993]. However, this is just one (relatively easy) spot in the design space. In fact, much work in the 90's has been devoted to improving the combination of parametric and subtyping polymorphism, motivated by its application to the typechecking of object-oriented features.

Contravariance, the key ingredient of subtyping polymorphism, is already modeled in the language F_η proposed by Mitchell [1988]. One way to define F_η is as the closure of Curry-style System F by η-conversion. We write $\mathcal{C}[\mathcal{M}]$ for filling a context $\mathcal{C}$ with a term $\mathcal{M}$. A *retyping context* from τ to σ is a closed one-hole context $\mathcal{C}$ such that $\lambda x.\mathcal{C}\langle x\rangle$ is an η-expansion of the identity, also called a retyping function, and has type $\tau \to \sigma$ in System F. If $\mathcal{C}$ is a retyping context from τ to σ and $\mathcal{M}$ is a term of type τ, then $\mathcal{C}[\mathcal{M}]$ is a term of type σ in System F. In System F_η, the type-containment rule allows $\mathcal{M}$ itself to be claimed of type σ. Moving to Church-style System F, we may keep type-containment, *i.e.* filling of retyping contexts explicit. We write $G\langle M\rangle$ for the application of retyping context (*i.e.* a coercion) G to the term M. We write $\Diamond^\tau$ for the empty (retyping) context of type τ. Contravariance is induced by η-expansion as follows: if G_1 and G_2 are retyping contexts from τ_1 to τ_1' and from τ_2 to τ_2', then $\lambda(x : \tau_1)\ G_2\langle\Diamond^{\tau_1'\to\tau_2}\ (G_1\langle x\rangle)\rangle$ is a retyping context from type $\tau_1' \to \tau_2$ to $\tau_1 \to \tau_2'$.

Besides contravariance, η-expansion also introduces opportunities for inserting type abstractions and type applications, which may change polymorphism a posteriori. For instance, from the type $\forall\alpha.\,\tau \to \sigma$, we can find a retyping context to any type of the form $(\forall\alpha.\,\tau) \to (\forall\bar\beta.\,\sigma[\alpha \leftarrow \rho])$ provided $\bar\beta$ does not appear free in $\forall\alpha.\,\sigma$; this context is $\lambda(x : \forall\alpha.\,\tau)\ \lambda\bar\beta\ \Diamond^{\forall\alpha.\,\tau\to\sigma}\,\rho\ (x\,\rho)$. Such retypings are not supported in $\mathsf{F}_{<:}$ where polymorphism can only be introduced and eliminated explicitly at the topmost part of terms.

Conversely, $\mathsf{F}_{<:}$ allows reasoning under subtyping assumptions, which F_η does not support. Indeed, bounded quantification $\Lambda(\alpha <: \tau)\,M$ of $\mathsf{F}_{<:}$ introduces a type variable α that stands for any subtype of τ inside M. In particular, a *covariant occurrence* of α in M can be converted to type τ by subtyping.

Therefore F_η and $\mathsf{F}_{<:}$ are incomparable: is there a language that supersedes both? Before we tackle this question, let us first consider another form of retyping assumptions that have been introduced in MLF [Le Botlan and Rémy, 2009]: instance-bounded polymorphism $\Lambda(\alpha \geq \tau)\,M$ introduces a type variable α that stands for any instance of τ inside M. That is, an occurrence of type α within M in an *instantiable position* can be converted to any instance of τ. Instance-bounded quantification delays the choice of whether a polymorphic expression should be instantiated immediately or kept polymorphic. This mechanism enables expressions to have more general types and has been introduced in MLF to enable partial type inference in the presence of first-class second-order polymorphism and some type annotations.

Notice that bounded type instantiation allows for deep type instantiation of binders, as F_η does, but using a quite different mechanism. Bounded type instantiation has similarities with bounded quantification of $\mathsf{F}_{<:}$, but the two also differ significantly, since for instance, type conversion is not congruent on arrow types in MLF.

Surprisingly, among the three languages F_η, $\mathsf{F}_{<:}$, and MLF, any combination of two have features in common that the other one lacks! Hence, the challenge becomes whether all their features can be

combined together. This question has in fact already been raised in previous work on MLF [Rémy and Yakobowski, 2010].

Our contributions We answer positively by introducing a language F_ι^p that extends F_η with abstraction over retyping functions, combining all features simultaneously in a unified framework (§5). The language F_ι^p subsumes F_η, $\mathsf{F}_{<:}$, and MLF (§6); it also fixes and extends a previous language of coercions designed for modeling MLF alone [Manzonetto and Tranquilli, 2010]. Our subset of F_ι^p that coincides with MLF is *well-behaved*: it satisfies the subject reduction and progress lemmas and strongly normalizes. It also has an untyped semantics.

Actually, the extension of F_η with abstraction over coercion functions leads to a larger language F_ι of which F_ι^p is a restriction (§2). The language F_ι is well-behaved. We show that F_ι can be simulated into System F. Hence, reduction rules in F_ι are just particular instances of β-reduction (§4). F_ι can also be simulated into the untyped λ-calculus, by dropping coercions, which shows that coercions do not contribute to the computation. Unfortunately, they may block it, and are thus not erasable (§3). Erasability can be recovered by choosing a weak reduction strategy (§7), but this is not entirely satisfactory. So, other restrictions or extensions of F_ι with erasable coercions are still to be found. Nevertheless, we believe that F_ι is a solid ground for understanding erasable coercions (§9).

System F All languages we consider are second order calculi whose origin is System F. System F comes in two flavors: in Curry-style, terms do not carry type information and are thus a subset of the untyped λ-calculus, while in Church-style, terms carry explicit type information, namely type abstractions, type applications, and annotations of function parameters.

Of course, both presentations are closely related, since there is a bisimulation between the reduction of terms in Church-style and terms in Curry-style via type erasure, where the reduction of type application between terms in Church-style is reflected as an equality on terms in Curry-style. That is, calling ι the reduction of type applications and β the reduction of term applications, the type erasures of two explicitly-typed terms related by β-reduction (*resp.* ι-reduction) are related by β-reduction (*resp.* equality); conversely, if the erasure of a term M β-reduces to a term $\mathcal{M}'$, then M also reduces by a sequence of ι-reductions followed by a single β-reduction to a term whose erasure is $\mathcal{M}'$.

Both views are equally useful: we prefer source expressions to be explicitly typed, so that type checking is a trivial process and types can be easily maintained during program transformations; we also wish types to be erasable after compilation for efficiency of program execution. Moreover, a source language with an untyped semantics is generally simpler to understand, reason about, and use. We may argue that even if the source language has intentional polymorphism, it should first be compiled in a type-dependent way to an intermediate language with an untyped semantics [Crary et al., 2002].

From types to type conversions Our approach to coercions is similar to polymorphism in System F because we focus here on retyping functions that are *erasable*. In some circumstances, one may use *other* forms of coercions that may have some computational content, *e.g.* change the representation of values, and thus not be erasable. Then, we should compile source expressions into an intermediate language where remaining coercions, if any, are erasable; this is then the language we wish to study here.

Erasability also means that the dynamic semantics of our language is ultimately that of the underlying λ-calculus—possibly enriched with more constructs. Therefore the semantics only depends on the reduction strategy we choose and not on the typechecking details nor on the coercions we may use. Types are useful for programmers to understand their programs. It is also useful for programmers that types do not determine the semantics of their programs. At least, we should provide an intermediate representation in which this is true.

Coercions may also be introduced a posteriori to make type conversions explicit inside source terms. Coercions usually simplify the meta-theoretical study of the language by providing a

concrete syntax to manipulate typing derivations. Proofs such as subject reduction become *computation* on concrete terms instead of *reasoning* on derivations.

While in practice programming languages use weak evaluation strategies, strong evaluation strategies provide more insight into the calculus by also modeling reduction of open terms. Since our focus is on *understanding* the essence of coercions, and the meta-theoretical properties, we prefer strong reduction strategies. Imposing a weak reduction strategy on a well-behaved strong calculus afterward is usually easy—even if all properties do not automatically transfer. Conversely, properties for weak reduction strategies do not say much about strong reduction strategies.

The two faces of F_η Let us first return to the definition of F_η, which in Mitchell's original presentation is given in Curry-style. It is defined by adding to System F a *type containment* rule that allows to convert a term $\mathcal{M}$ of type τ to one of type σ whenever there exists a retyping context from type τ to σ, which we write $\vdash \tau \triangleright \sigma$. This judgment, called type containment, is equivalent to the existence of a (closed) retyping function $\mathcal{M}'$ of System F such that $\vdash \mathcal{M}' : \tau \to \sigma$, *i.e.* a function that is an η-expansion of the identity.

Interestingly, Mitchell gave another characterization of type containment, exhibiting a proof system for the judgment $\vdash \tau \triangleright \sigma$, which can be read back as the introduction of a language of coercions whose expressions G witness type containment derivations. Then, we write $\vdash G : \tau \triangleright \sigma$ where G fully determines the typing derivation—much as a Church-style System-F term M fully determines its typing derivation. For example, $G_1 \to G_2$ is a coercion that, given a function M, returns a function that coerces its argument with G_1, passes it to M, and coerces the result with G_2—hence the contravariance of type containment. (A full presentation of coercions appears in §2 where F_η is described as a subset of F_ι.)

The interpretation of coercions as λ-terms is more intuitive than coercions as proof witnesses. Unfortunately, its formal presentation F_ι^λ, which is equivalent to F_ι, is technically more involved for reasons explained in §4. Hence, we prefer to present F_ι first in §2 and only introduce F_ι^λ informally in §4. Interestingly, the reification of F_ι into System F given in §3.3 already reveals this intuitive interpretation of coercions—without the technicalities—and we refer to it when describing the typing rules and reduction rules of F_ι.

In Church-style System F, the use of a coercion G around a term M is witnessed explicitly as $G\langle M\rangle$. (We may continue seeing a coercion G as a retyping context and reading this as filling the hole of G or, equivalently, see G as a retyping function and read this as an application of a coercion to a term.) Reduction rules are added to reduce such applications when both G and M have been sufficiently evaluated—in a way depending on the form of both—so that a coercion G is never stuck in the middle of a (well-typed) redex as in $(G\langle\lambda(x:\tau)\ M\rangle)\ N$. The type system ensures that G is of a certain shape for which a reduction exists. In the above example, G may be $G_1 \xrightarrow{\sigma} G_2$ and then $G\langle\lambda(x:\tau)\ M\rangle$ can be reduced to $\lambda(x:\sigma)\ G_2\langle M[x \leftarrow G_1\langle x\rangle]\rangle$.

The genesis of F_ι To abstract over coercion functions, we introduce a new form $\lambda(c:\tau\triangleright\sigma)\ M$ in F_ι, where the parameter c stands for a coercion function that can be used inside M to convert an expression of type τ to one of type σ. This abstraction can be typed as $(\tau\triangleright\sigma)\Rightarrow\rho$ where ρ is the type of M. Correspondingly, we need a new application form $M\{G\}$ to pass a coercion G to a coercion abstraction, *i.e.* a term M of type $(\tau\triangleright\sigma)\Rightarrow\rho$.

By typing constraints, coercion abstractions can only be instantiated with coercions, which by construction are erasable. Thus, intuitively, coercions do not really contribute to the computation. Is this enough to erase them? Formally, we can exhibit a forward simulation between reduction of terms in F_ι and of their erasure in the untyped λ-calculus. Moreover, F_ι has the subject reduction property and is strongly normalizing. Still, coercions cannot be erased in F_ι, since although they do not create new evaluation paths, they may block existing evaluation paths: a subterm may be stuck while its erasure could proceed. Since coercions are erasable in F_η, this can only be due to the use of a coercion variable. Indeed, a coercion variable c may appear in the middle of a β-redex as in $(c\langle\lambda(x:\tau)\ M\rangle)\ N$. This is irreducible because reduction of coercion applications $G\langle M\rangle$ depends simultaneously on the shapes of G and M so that no rule fires when G is unknown.

$$
\begin{array}{rcll}
x, y & & & \text{variables} \\
\mathcal{M} & ::= & x \mid \lambda x.\mathcal{M} \mid \mathcal{M}\,\mathcal{M} \mid (\mathcal{M},\mathcal{M}) \mid \mathcal{M}.1 \mid \mathcal{M}.2 & \text{terms} \\
\mathcal{C} & ::= & \lambda x.[] \mid []\,\mathcal{M} \mid \mathcal{M}\,[] \mid ([],\mathcal{M}) \mid (\mathcal{M},[]) \mid [].1 \mid [].2 & \text{reduction contexts}
\end{array}
$$

$$
\frac{\mathcal{M} \rightsquigarrow \mathcal{M}'}{\mathcal{C}[\mathcal{M}] \rightsquigarrow \mathcal{C}[\mathcal{M}']}\ \textsc{RedContext}
\qquad
(\lambda x.\mathcal{M})\,\mathcal{M}' \rightsquigarrow \mathcal{M}[x \leftarrow \mathcal{M}']\ \textsc{RedBeta}
\qquad
(\mathcal{M}_1,\mathcal{M}_2).1 \rightsquigarrow \mathcal{M}_1\ \textsc{RedProjFirst}
\qquad
(\mathcal{M}_1,\mathcal{M}_2).2 \rightsquigarrow \mathcal{M}_2\ \textsc{RedProjSecond}
$$

Figure 1: λ-calculus: syntax and semantics

More generally, we call a wedge an irreducible term of the form $(G\langle\lambda(x:\tau)\ M\rangle)\ N$. Notice that the erasure of a wedge $(\lambda(x:\tau)\ \lfloor M\rfloor)\ \lfloor N\rfloor$ can be reduced, immediately. Hence, the existence of wedges in reduction contexts prevents erasability.

Taming coercions in F_ι^p An obvious solution to recover erasability is to make wedge configurations ill-typed—so that they never appear during the reduction of well-typed programs. One interesting restriction, called F_ι^p (read Parametric F_ι), is to request that coercion parameters be polymorphic in either their domain or their codomain. This allows coercion variables to appear either applied to a function or inside an application, but not both simultaneously.

Another solution is to change the semantics: choosing a weak reduction strategy for coercion abstractions and restricting them to appear only in front of value forms, coercion variables, hence wedges, cannot occur in a reduction context any more. This variant is called F_ι^w (read Weak F_ι).

Although our main goal—combining F_η, $\mathsf{F}_{<:}$, and MLF in a same language—is reached, both F_ι^p and F_ι^w are restrictions of F_ι. We may thus wonder whether other yet more interesting solutions exist. We further discuss some of the issues in §9, argue about some of the difficulties in the general case, and suggest other restrictions worth exploring. We defer a discussion of related works to §8.

2 The language F_ι

The language F_ι generalizes F_η with abstraction over coercions, but does not ensure erasability. Still coercions do not contribute to the evaluation. That is, reduction in F_ι can be simulated into the untyped λ-calculus, after erasure. Since coercions allow more terms to be typed, the coercion erasure will not in general be in the implicitly typed System F.

We recall the definition of the (untyped) λ-calculus on Figure 1. We include pairs and projections both to have non trivial errors (otherwise, even untyped terms cannot be stuck) and to have more interesting forms of subtyping. We assume an enumerable collection of term variables, ranged over by letters x and y. Untyped terms, written $\mathcal{M}$, include variables, abstractions $\lambda x.\mathcal{M}$, applications $\mathcal{M}\,\mathcal{M}'$, pairs $(\mathcal{M},\mathcal{M}')$, and projections $\mathcal{M}.1$ and $\mathcal{M}.2$. The semantics of untyped λ-terms is given by a small-step strong reduction relation. Reduction contexts of the λ-calculus are all one-hole contexts, written $\mathcal{C}$. We now write $\mathcal{C}[\mathcal{M}]$ for the term obtained by filling the hole of $\mathcal{C}$ with $\mathcal{M}$ and $\mathcal{M}[x \leftarrow \mathcal{M}']$ for the capture avoiding substitution of $\mathcal{M}'$ for x in $\mathcal{M}$. Expressions are considered equal up to the renaming of bound variables, which are defined in the usual way. This convention applies to the λ-calculus, as well as to all typed languages presented below.

2.1 Syntax of F_ι

The language F_ι is explicitly typed. Types are described on Figure 2. We assume given an enumerable set of type variables, ranged over by letters α and β. Types are type variables, arrow types $\tau \to \tau$, product types $(\tau * \tau)$, polymorphic types $\forall\alpha.\tau$, the top type $\top$, or coercion abstractions $\varphi \Rightarrow \tau$ where the coercion type φ is of the form $\tau \triangleright \tau$. Coercions are not first class, hence a coercion type φ is not itself a type.

$$
\begin{array}{rcll}
\tau, \sigma & ::= & \alpha \mid \tau \to \tau \mid (\tau * \tau) \mid \forall \alpha.\tau \mid \top & \text{Types} \\
& \mid & \varphi \Rrightarrow \tau & \textit{coercion abstraction} \\
\varphi & ::= & \tau \rhd \tau & \text{coercion type} \\
\\
M, N & ::= & x \mid \lambda(x:\tau)\ M \mid M\ M \mid (M, M) \mid M.\mathtt{1} \mid M.\mathtt{2} & \text{Terms} \\
& \mid & \lambda\alpha\ M \mid M\,\tau & \textit{type abs \& app} \\
& \mid & G\langle M \rangle & \textit{term coercion} \\
& \mid & \lambda(c:\varphi)\ M \mid M\{G\} & \textit{coercion abs \& app} \\
\\
G & ::= & c \mid \mathsf{Top}^{\tau} \mid \Diamond^{\tau} \mid G \xrightarrow{\tau} G \mid (G * G) & \text{Coercions} \\
& \mid & \mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau} \mid \mathsf{Dist}^{\varphi\Rrightarrow}_{\tau\to\tau} \mid \mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)} \mid \mathsf{Dist}^{\varphi\Rrightarrow}_{(\tau*\tau)} & \textit{distributivity} \\
& \mid & \lambda\alpha\ G \mid G\,\tau & \textit{type abs \& app} \\
& \mid & G\langle G \rangle & \textit{coercion coercion} \\
& \mid & \lambda(c:\varphi)\ G \mid G\{G\} & \textit{coercion abs \& app} \\
\\
\Gamma & ::= & \emptyset \mid \Gamma, \alpha \mid \Gamma, x:\tau \mid \Gamma, c:\varphi & \text{Typing environments}
\end{array}
$$

Figure 2: Syntax of F_ι.

The language of expressions is split into *terms* and *coercions*. We reuse the term variables of the λ-calculus. In addition, we assume an enumerable set of coercion variables written c. Terms are an extension of Church-style System F. Hence, they include type variables x, abstractions $\lambda(x:\tau)\ M$, applications MM, pairs (M, M), projections $M.\mathtt{1}$ and $M.\mathtt{2}$, type abstractions $\lambda\alpha\ M$, and type applications $M\,\tau$. A construct already present in F_η is the use of the application $G\langle M \rangle$ of a coercion G to a term M. There are two new constructs specific to F_ι and not present in F_η: coercion abstraction $\lambda(c:\varphi)\ M$ which is annotated with the coercion type φ; and coercion application $M\{G\}$ that passes a coercion G to a term M—and should not be confused with the earlier construct $G\langle M \rangle$ of F_η that places a coercion G around a term M.

Since the main purpose of coercions is to change types, we could postpone the description of coercion constructs together with their typing rules—and their associated reduction rules that justify the typing rules. Still, each coercion expression can be understood as a one-hole retyping context witnessing some type-containment rule. So we introduce each construct with the retyping context it stands for, also preparing for the reification of coercions as System-F terms given in §3.3.

A coercion variable c stands for the coercion it will be bound to. The opaque coercion Top^{τ} is a downgraded version of existential types (we currently do not handle existential types for reasons explained in §9): it turns a term of any type into an opaque term of type $\top$ that can only be used abstractly. The empty coercion $\Diamond^{\tau}$ stands for the empty retyping context and witnesses reflexivity of type containment. The arrow coercion $G_1 \xrightarrow{\tau} G_2$ stands for $\lambda(x:\tau)\ G_2\langle [\,]\ (G_1\langle x \rangle) \rangle$ and witnesses contravariance of the arrow type constructor. The distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ stands for $\lambda(x:\tau)\ \lambda\alpha\ [\,]\,\alpha\ x$ and permutes a type abstraction with a term abstraction: assuming the hole has type $\forall\alpha.\,\tau \to \sigma$ where α does not appear free in τ, it returns a term of type $\tau \to \forall\alpha.\,\sigma$. For instance, the coercion of a polymorphic function $\lambda\alpha\ \lambda(y:\tau)\ N$ makes it appear as if it had been defined as $\lambda(y:\tau)\ \lambda\alpha\ N$—which is actually what it will reduce to once coerced. The other distributivity coercion $\mathsf{Dist}^{\varphi\Rrightarrow}_{\tau\to\sigma}$, which stands for $\lambda(x:\tau)\ \lambda(c:\varphi)\ ([]\{c\}\ x)$, is similar but permutes a coercion abstraction with a term abstraction.

The product coercion $(G_1 * G_2)$ stands for $(G_1\langle [].\mathtt{1} \rangle, G_2\langle [].\mathtt{2} \rangle)$ and allows congruence on the product type constructor. The distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}$ stands for $(\lambda\alpha\ ([]\,\alpha).\mathtt{1}, \lambda\alpha\ ([]\,\alpha).\mathtt{2})$ and permutes a type abstraction with a pair constructor: assuming the hole has type $\forall\alpha.\,(\tau*\sigma)$, it returns a term of type $((\forall\alpha.\,\tau)*(\forall\alpha.\,\sigma))$. The other distributivity coercion $\mathsf{Dist}^{\varphi\Rrightarrow}_{(\tau*\sigma)}$, which stands for $(\lambda(c:\varphi)\ ([]\{c\}).\mathtt{1}, \lambda(c:\varphi)\ ([]\{c\}).\mathtt{2})$ is similar but permutes a coercion abstraction with a pair construct.

$$
\begin{array}{rlrl}
\lfloor x \rfloor &= x & & \\
\lfloor \lambda(x:\tau)\ M \rfloor &= \lambda x.\lfloor M \rfloor & \lfloor \lambda\alpha\ M \rfloor &= \lfloor M \rfloor \\
\lfloor M\ N \rfloor &= \lfloor M \rfloor\ \lfloor N \rfloor & \lfloor M\,\tau \rfloor &= \lfloor M \rfloor \\
\lfloor (M,N) \rfloor &= (\lfloor M \rfloor, \lfloor N \rfloor) & \lfloor G\langle M \rangle \rfloor &= \lfloor M \rfloor \\
\lfloor M.1 \rfloor &= \lfloor M \rfloor.1 & \lfloor \lambda(c:\varphi)\ M \rfloor &= \lfloor M \rfloor \\
\lfloor M.2 \rfloor &= \lfloor M \rfloor.2 & \lfloor M\{G\} \rfloor &= \lfloor M \rfloor
\end{array}
$$

Figure 3: Coercion erasure

We may need more distributivity coercions when extending the language of terms. Hence, the notation Dist_b^a uses the following mnemonic: the superscript a and the subscript b indicate the kind of the first and second type constructs, respectively. The first type construct a should be an erasable quantifying type construct, *i.e.* a binding coercion type construct, like type or coercion abstraction. The second type construct b should be a *stlc* (simply-typed lambda-calculus) type construct, like arrow or product. The type of the hole is ab and the coerced type is b where all positive holes c become ac and negative holes stay the same. When b is $[] \to []$, the positive (covariant) hole is the right one, while the negative (contravariant) one is the left one. When b is $([] * [])$, both holes are positive. This is why we get $(a[] * a[])$. This heavy-weighted distributivity mechanism might in the end overcome the difficulties about binding coercions in F_ι^λ (§4), which would then become preferable to work with.

The remaining coercions are the lifting of all term constructs without computational content to coercions: type abstraction $\lambda\alpha\ G$ and type application $G\,\tau$; coercion of a coercion $G'\langle G \rangle$ which intuitively stands for $G'\langle G\langle [] \rangle\rangle$ and witnesses transitivity of coercions: it has type $\rho \triangleright \sigma$ if G' and G have coercion types $\tau \triangleright \sigma$ and $\rho \triangleright \tau$, respectively; finally, coercion abstraction $\lambda(c:\varphi)\ G$ and coercion application $G'\{G\}$. All these coercions are of the form $P[G]$ where P is one of the contexts $\lambda\alpha\ []$, $[]\,\tau$, $G'\langle [] \rangle$, $\lambda(c:\varphi)\ []$, or $[]\{G'\}$, where the hole is filled with G. It is convenient to overload the notation P when the hole holds a term instead of a coercion, although this is formally another syntactic node.

We recover the syntax of System F_η by removing coercion types from types and coercion variables, coercion abstractions and applications from both terms and coercions. We recover the syntax of System F by further removing the top type, term coercions, and all coercion forms, which become vacuous.

The coercion erasure, written $\lfloor \cdot \rfloor$, defined on Figure 3, is as expected: type annotations on function parameters and coercions are erased, while other constructs are projected on their equivalent constructs in the untyped λ-calculus.

2.2 Typing rules

Typing environments, written Γ, are lists of bindings where bindings are either type variables α, coercion variables along with their coercion type $c : \varphi$, or term variables along with their type $x : \tau$ (Figure 2). We write $\Gamma \vdash M : \tau$ if term M has type τ under Γ and $\Gamma \vdash G : \varphi$ if coercion G has coercion type φ under Γ.

The two typing judgments are recursively defined on figures 4 and 5. They use auxiliary well-formedness judgments for types and typing contexts: we write $\Gamma \vdash ok$ to mean that typing environment Γ is well-formed and $\Gamma \vdash \tau$ or $\Gamma \vdash \varphi$ to mean that type τ or coercion type φ is well-formed in Γ.

As usual, we require that typing contexts do not bind twice the same variable, which is not restrictive as all expressions are considered equal up to renaming of bound variables. This is enforced by well-formedness judgments defined on Figure 6. This restriction allows us to see Γ as a partial function from term, coercion, or type variables to their types if they have ones.

Typing rules for terms are described in Figure 4. Rules TermVar, TermTermLam, TermTermApp, TermPair, TermFirst, TermSecond, TermTypeLam, and TermTypeApp are exactly

$$\textsc{TermVar}\ \frac{\Gamma \vdash ok \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{TermTermLam}\ \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau)\ M : \tau \to \sigma} \qquad \textsc{TermTermApp}\ \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \sigma}$$

$$\textsc{TermPair}\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : (\tau_1 * \tau_2)} \qquad \textsc{TermFirst}\ \frac{\Gamma \vdash M : (\tau_1 * \tau_2)}{\Gamma \vdash M.1 : \tau_1} \qquad \textsc{TermSecond}\ \frac{\Gamma \vdash M : (\tau_1 * \tau_2)}{\Gamma \vdash M.2 : \tau_2} \qquad \textsc{TermTypeLam}\ \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \lambda\alpha\ M : \forall\alpha.\, \tau}$$

$$\textsc{TermTypeApp}\ \frac{\Gamma \vdash M : \forall\alpha.\, \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash M\,\sigma : \tau[\alpha \leftarrow \sigma]} \qquad \textsc{TermCoer}\ \frac{\Gamma \vdash G : \tau \triangleright \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash G\langle M \rangle : \sigma} \qquad \textsc{TermCoerLam}\ \frac{\Gamma, c : \varphi \vdash M : \tau}{\Gamma \vdash \lambda(c : \varphi)\ M : \varphi \Rightarrow \tau}$$

$$\textsc{TermCoerApp}\ \frac{\Gamma \vdash M : \varphi \Rightarrow \tau \qquad \Gamma \vdash G : \varphi}{\Gamma \vdash M\{G\} : \tau}$$

Figure 4: System F_ι: term typings

$$\textsc{CoerDot}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \Diamond^\tau : \tau \triangleright \tau} \qquad \textsc{CoerForget}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{Top}^\tau : \tau \triangleright \top} \qquad \textsc{CoerArrow}\ \frac{\Gamma \vdash G_1 : \tau_1 \triangleright \tau_1' \qquad \Gamma \vdash G_2 : \tau_2 \triangleright \tau_2'}{\Gamma \vdash G_1 \overset{\tau_1}{\to} G_2 : (\tau_1' \to \tau_2) \triangleright (\tau_1 \to \tau_2')}$$

$$\textsc{CoerDistTypeArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\forall\alpha.}_{\tau \to \sigma} : \forall\alpha.\, (\tau \to \sigma) \triangleright \tau \to \forall\alpha.\, \sigma} \qquad \textsc{CoerDistCoerArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \varphi \qquad \Gamma \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\varphi \Rightarrow}_{\tau \to \sigma} : (\varphi \Rightarrow (\tau \to \sigma)) \triangleright (\tau \to (\varphi \Rightarrow \sigma))}$$

$$\textsc{CoerPair}\ \frac{\Gamma \vdash G_1 : \tau_1 \triangleright \tau_1' \qquad \Gamma \vdash G_2 : \tau_2 \triangleright \tau_2'}{\Gamma \vdash (G_1 * G_2) : (\tau_1 * \tau_2) \triangleright (\tau_1' * \tau_2')} \qquad \textsc{CoerDistTypeProd}\ \frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\forall\alpha.}_{(\tau * \sigma)} : \forall\alpha.\, (\tau * \sigma) \triangleright (\forall\alpha.\, \tau * \forall\alpha.\, \sigma)}$$

$$\textsc{CoerDistCoerProd}\ \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\varphi \Rightarrow}_{(\tau * \sigma)} : (\varphi \Rightarrow (\tau * \sigma)) \triangleright ((\varphi \Rightarrow \tau) * (\varphi \Rightarrow \sigma))} \qquad \textsc{CoerTypeLam}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda\alpha\ G : \tau \triangleright \forall\alpha.\, \sigma}$$

$$\textsc{CoerTypeApp}\ \frac{\Gamma \vdash G : \tau' \triangleright \forall\alpha.\, \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash G\,\tau : \tau' \triangleright \sigma[\alpha \leftarrow \tau]} \qquad \textsc{CoerCoer}\ \frac{\Gamma \vdash G : \tau \triangleright \sigma \qquad \Gamma \vdash G' : \rho \triangleright \tau}{\Gamma \vdash G\langle G' \rangle : \rho \triangleright \sigma} \qquad \textsc{CoerCoerLam}\ \frac{\Gamma, c : \varphi \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda(c : \varphi)\ G : \tau \triangleright (\varphi \Rightarrow \sigma)}$$

$$\textsc{CoerCoerApp}\ \frac{\Gamma \vdash G' : \tau \triangleright (\varphi \Rightarrow \sigma) \qquad \Gamma \vdash G : \varphi}{\Gamma \vdash G'\{G\} : \tau \triangleright \sigma} \qquad \textsc{CoerVar}\ \frac{\Gamma \vdash ok \qquad c : \varphi \in \Gamma}{\Gamma \vdash c : \varphi}$$

Figure 5: System F_ι: coercion typings

$$\textsc{TypeVar}\ \frac{\Gamma \vdash ok \qquad \alpha \in \mathsf{dom}(\Gamma)}{\Gamma \vdash \alpha} \qquad \textsc{TypeArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau \to \sigma} \qquad \textsc{CoerType}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau \triangleright \sigma} \qquad \textsc{TypeProd}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash (\tau * \sigma)}$$

$$\textsc{TypeForall}\ \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall\alpha.\, \tau} \qquad \textsc{TypeCoerArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \varphi}{\Gamma \vdash \varphi \Rightarrow \tau} \qquad \textsc{EnvCoer}\ \frac{c \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \varphi}{\Gamma, (c : \varphi) \vdash ok}$$

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

Figure 6: System F: well-formedness rules

the typing rules of System F. Rule TermCoer is similar to rule TermTermApp, except that a coercion G of coercion type $\tau \triangleright \sigma$ is used instead of a function M of type $\tau \to \sigma$. Rule TermCoerLam is similar to TermTermLam, except that the parameter c stands for a coercion of coercion type φ instead of a term of type σ: the result is a coercion abstraction of type $\varphi \Rightarrow \tau$. Consistently, TermCoerApp applies a term that is a coercion abstraction of type $\varphi \Rightarrow \tau$ to a coercion G of coercion type φ.

Typing rules for coercions are described in Figure 5. They are all straightforward when read with the retyping context that the coercion stands for in mind. Rule CoerVar reads the coercion type of a coercion variable from its typing context. The empty coercion has type $\tau \triangleright \tau$ provided τ is well-formed in the current context. As all basic coercions, it contains just enough type information so that its typing rule is syntax-directed. The top coercion Top^τ converts an expression of type τ to the top type, provided τ is well-formed. The arrow coercion $G_1 \xrightarrow{\tau_1} G_2$ turns an arrow type $\tau_1' \to \tau_2$ into an arrow type $\tau_1 \to \tau_2'$, provided G_i coerces type τ_i into τ_i' for i in $\{1,2\}$. The distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ turns an expression of type $\forall\alpha.\,\tau \to \sigma$ into one of type $\tau \to \forall\alpha.\,\sigma$ provided τ is well-formed in the current environment, which prevents α from appearing free in τ, and σ is well-formed in the current environment extended with α. Finally, Rule CoerDistCoerArrow is similar to CoerDistTypeArrow, but swaps a coercion abstraction and a term abstraction.

The product coercion $(G_1 * G_2)$ turns a product type $(\tau_1 * \tau_2)$ into a product type $(\tau_1' * \tau_2')$, provided G_i coerces type τ_i into τ_i' for i in $\{1,2\}$. The distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}$ turns an expression of type $\forall\alpha.(\tau * \sigma)$ into one of type $(\forall\alpha.\,\tau * \forall\alpha.\,\sigma)$ provided τ and σ are well-formed in the current environment extended with α. Rule CoerDistCoerProd is similar to CoerDistTypeProd, but swaps a coercion abstraction and a pair constructor.

The remaining rules CoerTypeLam, CoerTypeApp, CoerCoer, CoerCoerLam, and CoerCoerApp are similar to their counterpart for terms, but where the term M of type τ has been replaced by a coercion (*i.e.* a one-hole context) G of coercion type $\tau_1 \triangleright \tau_2$, where τ_1 is the type of the hole and τ_2 the type of the body. Rule CoerTypeLam for typing $\lambda\alpha\, G$ introduces a variable α that is bound in G and can be used in the type of the body of G but not in the type of its hole, which is enforced by the first premise. In particular, $\lambda\alpha\, G$ builds a coercion to a polymorphic type $\tau \triangleright \forall\alpha.\,\sigma$ and not a polymorphic coercion $\forall\alpha.\,\tau \triangleright \sigma$. Accordingly, only the codomain of the type of the conclusion is polymorphic. Rule CoerCoerLam is typed in a similar way: $\lambda(c:\varphi)\, G$ has type $\tau_1 \triangleright (\varphi \Rightarrow \tau_2)$ and not $\varphi \Rightarrow (\tau_1 \triangleright \tau_2)$ as one could naively expect—which would be ill-formed. Type and coercion applications are typed accordingly (rules CoerTypeApp and CoerCoerApp).

The typing rules for F_η are obtained by removing TermCoerLam and TermCoerApp for terms and their counter parts CoerCoerLam and CoerCoerApp for coercions as well as Rule CoerVar for coercion variables and Rules CoerDistCoerArrow and CoerDistCoerProd for distributivity of coercion abstraction.

The type superscripts that appear in reflexivity, distributivity, and top coercions make type checking syntax directed. The type superscript in arrow coercions is not needed for typechecking but to keep reduction a local rewriting rule. (We may leave superscripts implicit when they are unimportant or can be unambiguously reconstructed from the context.)

Our presentation of F_ι is in Church-style. Curry-style F_ι is the image of F_ι by coercion erasure. That is, it is the subset of terms of the untyped λ-calculus that are the erasure of a term of Church-style System F_ι. We write $\Gamma \vdash \mathcal{M} : \tau$ to mean that there exists M such that $\Gamma \vdash M : \tau$ and $\lfloor M \rfloor$ is $\mathcal{M}$.

2.3 Dynamic semantics

The dynamic semantics of System F_ι is given by a standard small-step strong reduction relation. The syntax of values and reduction contexts is recalled on Figure 7.

A value is an abstraction of a value, a pair of values, an opaque value $\mathsf{Top}^\tau\langle v\rangle$, or a prevalue. A prevalue is a variable, a prevalue applied to a value, type, or coercion, a projection of a prevalue, a value coerced by a coercion variable, or a partial application of a distributivity coercion. Reduction contexts C are all one-hole term contexts. For convenience, we have distinguished a subset of

$$
\begin{array}{rcll}
p & ::= & x \mid p\ v \mid p.\mathtt{1} \mid p.\mathtt{2} \mid p\,\tau \mid p\{G\} \mid c\langle v\rangle & \text{Prevalues} \\
& \mid & \mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}\langle p\rangle \mid \mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}\langle\lambda\alpha\ p\rangle \mid (G \xrightarrow{\tau} G)\langle p\rangle & \\
& \mid & \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}\langle p\rangle \mid \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}\langle\lambda(c:\varphi)\ p\rangle & \\
& \mid & \mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}\langle p\rangle \mid \mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}\langle\lambda\alpha\ p\rangle \mid (G * G)\langle p\rangle & \\
& \mid & \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}\langle p\rangle \mid \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}\langle\lambda(c:\varphi)\ p\rangle & \\
v & ::= & p \mid \lambda(x:\tau)\ v \mid (v,v) \mid \lambda\alpha\ v \mid \lambda(c:\varphi)\ v \mid \mathtt{Top}^{\tau}\langle v\rangle & \text{Values} \\
C & ::= & \lambda(x:\tau)\ [] \mid []\ M \mid M\ [] \mid ([],M) \mid (M,[]) \mid []\mathtt{.1} \mid []\mathtt{.2} \mid P & \text{Reduction contexts} \\
P & ::= & \lambda\alpha\ [] \mid []\,\tau \mid G\langle[]\rangle \mid \lambda(c:\varphi)\ [] \mid []\{G\} & \text{Retyping contexts}
\end{array}
$$

Figure 7: System F_ι: values and reduction contexts

$$\textsc{RedContextBeta}\ \frac{M \rightsquigarrow_\beta N}{C[M] \rightsquigarrow_\beta C[N]} \qquad \textsc{RedContextIota}\ \frac{M \rightsquigarrow_\iota N}{C[M] \rightsquigarrow_\iota C[N]}$$

$$\textsc{RedTerm}\ (\lambda(x:\tau)\ M)\ N \rightsquigarrow_\beta M[x \leftarrow N] \qquad \textsc{RedFirst}\ (M,N).\mathtt{1} \rightsquigarrow_\beta M$$

$$\textsc{RedSecond}\ (M,N).\mathtt{2} \rightsquigarrow_\beta N \qquad \textsc{RedType}\ (\lambda\alpha\ M)\,\tau \rightsquigarrow_\iota M[\alpha \leftarrow \tau] \qquad \textsc{RedCoer}\ (\lambda(c:\varphi)\ M)\{G\} \rightsquigarrow_\iota M[c \leftarrow G]$$

$$\textsc{RedCoerArrow}\ (G_1 \xrightarrow{\tau} G_2)\langle\lambda(x:\sigma)\ M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\ G_2\langle M[x \leftarrow G_1\langle x\rangle]\rangle$$

$$\textsc{RedCoerDistTypeArrow}\ \mathsf{Dist}^{\forall\alpha.}_{\tau'\to\sigma'}\langle\lambda\alpha\ \lambda(x:\tau)\ M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\ \lambda\alpha\ M$$

$$\textsc{RedCoerDistCoerArrow}\ \mathsf{Dist}^{\varphi'\Rightarrow}_{\tau'\to\sigma'}\langle\lambda(c:\varphi)\ \lambda(x:\tau)\ M\rangle \rightsquigarrow_\iota \lambda(x:\tau)\ \lambda(c:\varphi)\ M \qquad \textsc{RedCoerProd}\ (G_1 * G_2)\langle(M,N)\rangle \rightsquigarrow_\iota (G_1\langle M\rangle, G_2\langle N\rangle)$$

$$\textsc{RedCoerDistTypeProd}\ \mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}\langle\lambda\alpha\ (M,N)\rangle \rightsquigarrow_\iota (\lambda\alpha\ M, \lambda\alpha\ N)$$

$$\textsc{RedCoerDistCoerProd}\ \mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}\langle\lambda(c:\varphi)\ (M,N)\rangle \rightsquigarrow_\iota (\lambda(c:\varphi)\ M, \lambda(c:\varphi)\ N) \qquad \textsc{RedCoerDot}\ \Diamond^{\tau}\langle M\rangle \rightsquigarrow_\iota M$$

$$\textsc{RedCoerFill}\ (P[G])\langle M\rangle \rightsquigarrow_\iota P[G\langle M\rangle]$$

Figure 8: Reduction rules for F_ι

reduction contexts P, called *retyping reduction contexts*: a term M placed in a retyping reduction context is just a retyping of M, *i.e.* a term that behaves as M but possibly with another type.

Reduction rules are defined on Figure 8. We have indexed the reduction rules so as to distinguish between β-steps with computational content (RedTerm, RedProjFirst, and RedProjSecond), that are preserved after erasure, and ι-steps (RedType) that become equalities after erasure. We write $\rightsquigarrow_{\beta\iota}$ for the union of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\iota$.

Hence, Rule RedContext is split into two rules, so as to preserve the index of the premise. The only β-redexes are RedTerm, RedProjFirst, and RedProjSecond; all other reductions are ι-reductions. Rule RedType is type reduction (a ι-reduction). The first six rules cover System F. Notice that RedContext allows all possible contexts. Hence, there is no particular reduction strategy and a call-by-value evaluation would be a particular case of reduction.

Rule RedCoer is the counterpart of β-reduction for coercion application $M\{G\}$. It only reduces a term applied to a coercion; a coercion applied to a coercion is a coercion and is not reduced directly, but only when it is applied to a term so that rule RedCoerCoerApp eventually applies.

All other rules reduce the application $G\langle M\rangle$ of a coercion G to a term M, which plays the role of a destructor: both G and M must be sufficiently evaluated before it reduces—except when G is the opaque coercion or a variable since $\mathsf{Top}^\tau\langle v\rangle$ and $c\langle v\rangle$ are values.

Other coercion nodes are all constructors. We thus have one rule for each possible shape of G. The most interesting rules are for basic coercions:

- When G is an arrow coercion $G_1 \xrightarrow{\tau} G_2$ and M is a function $\lambda(x:\sigma)\ M$, Rule RedCoerArrow reduces the application by pushing G_1 on all occurrences of x in M and G_2 outside of M. This changes the type of the parameter x from σ to τ, hence the need for the annotation τ on arrow coercions.

- When G is a distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{\tau'\to\sigma'}$ and M is a polymorphic function $\lambda\alpha\ \lambda(x:\tau)\ M$, Rule RedCoerDistTypeArrow reduces the application to $\lambda(x:\tau)\ \lambda\alpha\ M$ by exchanging the type and value parameters; this is sound since α cannot be free in τ.

- When G is a distributivity coercion $\mathsf{Dist}^{\varphi'\Rightarrow}_{\tau'\to\sigma'}$ and M is a coercion abstraction followed by a value abstraction $\lambda(c:\varphi)\ \lambda(x:\tau)\ M$, Rule RedCoerDistCoerArrow reduces the application to $\lambda(x:\tau)\ \lambda(c:\varphi)\ M$ by exchanging the coercion and value parameters.

  Notice that as in the previous rule, the type annotation on Dist and the parameters need not be identical since reduction does not assume that terms are well-typed.

- When G is a product coercion $(G_1 * G_2)$ and M is a pair (M,N), Rule RedCoerProd reduces the application by pushing G_1 on M and G_2 on N.

- When G is a distributivity coercion from a forall on a product, $\mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}$, and M is a polymorphic product $\lambda\alpha\ (M,N)$, Rule RedCoerDistTypeProd reduces the application by exchanging the type and pair construct to $(\lambda\alpha\ M, \lambda\alpha\ N)$.

- When G is a distributivity coercion $\mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}$ and M is a coercion abstraction followed by a product $\lambda(c:\varphi)\ (M,N)$, Rule RedCoerDistCoerProd similarly exchanges the parameters to $(\lambda(c:\varphi)\ M, \lambda(c:\varphi)\ N)$.

The remaining cases for G can be factored as $P[G']$. Rule RedCoerFill fills G' with M, transforming $P[G']\langle M\rangle$ into $P[G'\langle M\rangle]$. Notice that the two occurrences of P are different abstract nodes on each side of the rule—a coercion on the left-hand side and a term on the right-hand side. Rule RedCoerFill is actually a meta-rule that could be expanded into, and should be understood as, the following five different rules:

$$
\begin{aligned}
(\lambda\alpha\ G)\langle M\rangle &\rightsquigarrow_\iota \lambda\alpha\ (G\langle M\rangle) && \textsc{RedCoerTypeLam}\\
(G\,\tau)\langle M\rangle &\rightsquigarrow_\iota (G\langle M\rangle)\,\tau && \textsc{RedCoerTypeApp}\\
(G_2\langle G_1\rangle)\langle M\rangle &\rightsquigarrow_\iota G_2\langle G_1\langle M\rangle\rangle && \textsc{RedCoerCoer}\\
(\lambda(c:\varphi)\ G)\langle M\rangle &\rightsquigarrow_\iota \lambda(c:\varphi)\ (G\langle M\rangle) && \textsc{RedCoerCoerLam}\\
(G_1\{G_2\})\langle M\rangle &\rightsquigarrow_\iota (G_1\langle M\rangle)\{G_2\} && \textsc{RedCoerCoerApp}
\end{aligned}
$$

The use of the meta-rule emphasizes the similarity between all five cases; it is also more concise.

For example, the application $G_1\{G_2\}$ of a coercion abstraction G_1 to a coercion G_2 is only reduced when it is further applied to a term M (as other complex coercions), by first wrapping elements of G around M (two first steps below) so that Rule RedCoer can finally fire (last step):

$$
\begin{aligned}
((\lambda(c:\tau\triangleright\sigma)\,G)\{G'\})\langle M\rangle &\rightsquigarrow_\iota ((\lambda(c:\tau\triangleright\sigma)\,G)\langle M\rangle)\{G'\}\\
&\rightsquigarrow_\iota (\lambda(c:\tau\triangleright\sigma)\,(G\langle M\rangle))\{G'\}\\
&\rightsquigarrow_\iota (G\langle M\rangle)[c\leftarrow G']
\end{aligned}
$$

The reduction rules for System F_η are obtained by removing rules RedCoer, RedCoerCoerLam, RedCoerCoerApp, RedCoerDistCoerArrow, and RedCoerDistCoerProd

Optional reduction rules Our presentation of F_ι could be extended with additional reduction rules for arrow, distributivity and product coercions such as:

$$(\mathsf{Dist}^{\forall\alpha.}_{\tau'\to\sigma'}\langle M\rangle)\ N\,\rho \rightsquigarrow_\iota M\,\rho\ N \qquad (\mathsf{Dist}^{\forall\alpha.}_{\tau'\to\sigma'}\langle\lambda\alpha\ M\rangle)\ N \rightsquigarrow_\iota \lambda\alpha\ M\ N$$

$$(\mathsf{Dist}^{\varphi'\Rightarrow}_{\tau'\to\sigma'}\langle M\rangle)\ N\,G \rightsquigarrow_\iota M\,G\ N \qquad (\mathsf{Dist}^{\varphi'\Rightarrow}_{\tau'\to\sigma'}\langle\lambda(c:\varphi)\ M\rangle)\ N \rightsquigarrow_\iota \lambda(c:\varphi)\ M\ N$$

$$\overset{\textsc{RedCoerArrowApp}}{((G_1 \overset{\tau}{\to} G_2)\langle M\rangle)\ N \rightsquigarrow_\iota G_2\langle M\ (G_1\langle N\rangle)\rangle}$$

$$((G_1 * G_2)\langle M\rangle).1 \rightsquigarrow_\iota G_1\langle M.1\rangle \qquad ((G_1 * G_2)\langle M\rangle).2 \rightsquigarrow_\iota G_2\langle M.2\rangle$$

$$(\mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}\langle M\rangle).1\,\rho \rightsquigarrow_\iota (M\,\rho).1 \qquad (\mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}\langle M\rangle).2\,\rho \rightsquigarrow_\iota (M\,\rho).2$$

$$(\mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}\langle\lambda\alpha\ M\rangle).1 \rightsquigarrow_\iota \lambda\alpha\ M.1 \qquad (\mathsf{Dist}^{\forall\alpha.}_{(\tau'*\sigma')}\langle\lambda\alpha\ M\rangle).2 \rightsquigarrow_\iota \lambda\alpha\ M.2$$

$$(\mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}\langle M\rangle).1\{G\} \rightsquigarrow_\iota (M\{G\}).1 \qquad (\mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}\langle M\rangle).2\{G\} \rightsquigarrow_\iota (M\{G\}).2$$

$$(\mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}\langle\lambda(c:\varphi)\ M\rangle).1 \rightsquigarrow_\iota \lambda(c:\varphi)\ M.1 \qquad (\mathsf{Dist}^{\varphi'\Rightarrow}_{(\tau'*\sigma')}\langle\lambda(c:\varphi)\ M\rangle).2 \rightsquigarrow_\iota \lambda(c:\varphi)\ M.2$$

However, this narrows the set of values and reestablishing progress would require *binding coercions*, as for F^λ_ι described in §4, which are technically more involved. For sake of simplicity, the current presentation has fewer, but sufficiently many, reduction paths. To better understand why we need binding coercions, let's forget about pairs and focus on arrows. The rule RedCoerArrowApp is telling us that $(G_1 \overset{\tau}{\to} G_2)\langle M\rangle$ behaves like a arrow constructor, since when it is under the arrow destructor (which is the application node) they reduce. This morally means that $(G_1 \overset{\tau}{\to} G_2)\langle M\rangle$ behaves like $\lambda(x:\tau)\ M'$—actually this can be easily understood after reification (Section 3.3). This correspondence implies that we should define an additional reduction rule with $(G_1 \overset{\tau}{\to} G_2)\langle M\rangle$ everywhere we previously had a $\lambda(x:\tau)\ M'$, and in particular in RedCoerDist. So we have to define the reduction of $\mathsf{Dist}^{\forall\alpha.}_{\tau''\to\sigma''}\langle\lambda\alpha\ (G_1 \overset{\tau}{\to} G_2)\langle M\rangle\rangle$ where $\Gamma,\alpha\vdash M:\tau'\to\sigma'$, $\Gamma,\alpha\vdash G_1:\tau\triangleright\tau'$, and $\Gamma,\alpha\vdash G_2:\sigma'\triangleright\sigma$. We would like to say it reduces to $(G_1 \overset{\tau}{\to} (\lambda\alpha\ G_2))\langle M\rangle$ but we need the $\lambda\alpha$ to bind additionally in both G_1 and M.

2.4 Examples

Let us first see examples in the F_η subset. Retyping functions in F_η allow for the commutation of quantifiers and removal of useless quantifiers. They also let terms have more principal types. For example, in System F, the S-combinator $\lambda x.\lambda y.\lambda z.x\,z\,(y\,z)$ can be given the two incomparable types:

$$\forall\alpha.\,\forall\beta.\,\forall\gamma.\,(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma$$
$$(\forall\alpha.\,\alpha\to\alpha)\to(\forall\alpha.\,\alpha\to\alpha)\to(\forall\alpha.\,\alpha\to\alpha)\to(\forall\alpha.\,\alpha\to\alpha)$$

However, the former type is more general as it can be coerced to the latter (already in F_η), using three η-expansions. This example does not use distributivity, but the following example, still in F_η, does. (In the examples, we use type constructors List and D, which we assume to be covariant.) The map function has type:

$$\forall\alpha.\,\forall\beta.\,(\alpha\to\beta)\to\mathsf{List}\ \alpha\to\mathsf{List}\ \beta \qquad (\mathbf{1})$$

It can also be given the type

$$(\forall\alpha.\,\alpha\to D\,\alpha)\to\forall\alpha.\,\mathsf{List}\ (D\,\alpha)\to\mathsf{List}\ (D(D\,\alpha)) \qquad (\mathbf{2})$$

for some type constructor D, using the following coercion, which is already typable in F_η:

$$\big((\Diamond \to \lambda\alpha\ \Diamond\,(D\,\alpha))\langle \mathsf{Dist}^\forall\rangle\big)\langle\lambda\alpha\ (\Diamond\,\alpha \to \Diamond)\langle\Diamond\,\alpha\,(D\,\alpha)\rangle\rangle$$

Indeed, applying the coercion $\lambda\alpha\ (\Diamond\,\alpha \to \Diamond)\langle\Diamond\,\alpha\,(D\,\alpha)\rangle$ turns a term of type (1) into one of type:

$$\forall\alpha.\,(\forall\alpha.\,\alpha \to D\,\alpha) \to \mathsf{List}\ (\alpha) \to \mathsf{List}\ (D\,\alpha)) \qquad (\mathbf{3})$$

which in turn $(\Diamond \to \lambda\alpha\ \Diamond\,(D\,\alpha))\langle\mathsf{Dist}^\forall\rangle$ coerces to type (2). This example also illustrates the low-level nature of the language of coercions, to which we will come back in §4.

The last two examples illustrate coercion abstraction. We define a function first, inspired from $\mathsf{F}_{<:}$, that implements the first projection for non-empty tuples of arbitrary length. Tuples are encoded as chained pairs ending with $\top$. The function first

$$\lambda\beta\ \lambda\alpha\ \lambda(c : \alpha \triangleright (\beta * \top))\ \lambda(x : \alpha)\ (c\langle x\rangle).\mathtt{1}$$

of type $\forall\beta.\forall\alpha.\,(\alpha \triangleright (\beta * \top)) \Rightarrow \alpha \to \beta$ abstracts over a coercion c from arbitrary tuples to the singleton tuple. It can be applied to any non-zero tuple by passing the appropriate coercion. (In this example, subtyping could be encoded with just polymorphism instead of coercion abstration, but this is not true in general.)

The other example of coercion abstraction, inspired from MLF, delays the instantiation of a call to the polymorphic function choose of type $\forall\gamma.\,\gamma \to \gamma \to \gamma$, say σ_{ch}, when given itself as an argument. Let chch be $\lambda\gamma\ \lambda(c : \sigma_{ch} \triangleright \gamma)\ \mathsf{choose}\ \gamma\ (c\langle\mathsf{choose}\rangle)$ of type $\forall\gamma.\,(\sigma_{ch} \triangleright \gamma) \Rightarrow \gamma \to \gamma$. We may then pass chch the function plus of type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$, say σ_{plus}. This application is written $(\mathsf{chch}\ \sigma_{plus})\{\Diamond^{\sigma_{ch}}\ \mathsf{int}\}\ \mathsf{plus}$ and has type σ_{plus}.

3 Properties of F_ι

In this section, we show that F_ι is well-behaved: it has the subject reduction property and strongly normalizes; moreover, there is a forward simulation between terms of F_ι and their coercion erasure. Hence, coercions do not really contribute to the reduction. However, coercions are not erasable as they may sometimes appear in wedges and block the reduction.

The termination of F_ι is proved by reifying proof terms as plain System-F terms in §3.3, which shows that the dynamic semantics of proof-terms is in fact derivable. Unfortunately, in System F, one cannot distinguish between terms that are reification of proof terms and terms that compute. We then present the retyping function view of coercions, which is much closer to the reified approach in System F. We may regain this disjunction in F_ι^λ, which is the presentation of F_ι as retyping functions (§4). Terms of F_ι^λ may still be reified into System F, but the reification is a so simple transformation that F_ι^λ can be seen almost (but not quite) as an annotated version of System F.

3.1 Soundness

Type soundness of F_ι follows as usual from the subject reduction and progress lemmas. The proof of subject reduction uses substitution lemmas for terms, types, and coercions, which in turn use weakening. The proof is easy because coercions are explicit. So the reduction rules actually *are* the proof.

Definition 1 (Valid Extension)**.** *A* valid extension *of a well-formed context Γ is a well-formed context Γ' that contains Γ (*i.e. *as it extends Γ as a partial function).*

Lemma 2 (Extract Environment)**.** *If $\Gamma \vdash M : \tau$, $\Gamma \vdash G : \varphi$, or $\Gamma \vdash \tau$ holds, then $\Gamma \vdash ok$.*

Lemma 3 (Weakening)**.** *If Γ' is a valid extension of Γ, then:*

1. *If $\Gamma \vdash \tau$ holds, then $\Gamma' \vdash \tau$ holds.*

$$
\begin{array}{rlrl}
\lceil \alpha \rceil &= \alpha & \lceil x \rceil &= x \\
\lceil \tau \to \sigma \rceil &= \lceil \tau \rceil \to \lceil \sigma \rceil & \lceil \lambda(x:\tau)\ M \rceil &= \lambda(x:\lceil \tau \rceil)\ \lceil M \rceil \\
\lceil (\tau * \sigma) \rceil &= (\lceil \tau \rceil * \lceil \sigma \rceil) & \lceil M\ N \rceil &= \lceil M \rceil\ \lceil N \rceil \\
\lceil \forall \alpha.\, \tau \rceil &= \forall \alpha.\, \lceil \tau \rceil & \lceil (M, N) \rceil &= (\lceil M \rceil, \lceil N \rceil) \\
\lceil \varphi \Rightarrow \tau \rceil &= \lceil \varphi \rceil \to \lceil \tau \rceil & \lceil M.\mathtt{1} \rceil &= \lceil M \rceil.\mathtt{1} \\
\lceil \top \rceil &= \forall \alpha.\, (\forall \beta.\, \beta \to \alpha) \to \alpha & \lceil M.\mathtt{2} \rceil &= \lceil M \rceil.\mathtt{2} \\
\lceil \tau \triangleright \sigma \rceil &= \lceil \tau \rceil \to \lceil \sigma \rceil & &
\end{array}
$$

$$
\begin{array}{rlrl}
\lceil \lambda \alpha\ M \rceil &= \lambda \alpha\ \lceil M \rceil & \lceil \emptyset \rceil &= \emptyset \\
\lceil M\, \tau \rceil &= \lceil M \rceil\ \lceil \tau \rceil & \lceil \Gamma, B \rceil &= \lceil \Gamma \rceil, \lceil B \rceil \\
\lceil G\langle M \rangle \rceil &= \lceil G \rceil\ \lceil M \rceil & \lceil \alpha \rceil &= \alpha \\
\lceil \lambda(c:\varphi)\ M \rceil &= \lambda(x_c : \lceil \varphi \rceil)\ \lceil M \rceil & \lceil (x:\tau) \rceil &= (x : \lceil \tau \rceil) \\
\lceil M\{G\} \rceil &= \lceil M \rceil\ \lceil G \rceil & \lceil (c:\varphi) \rceil &= (x_c : \lceil \varphi \rceil)
\end{array}
$$

$$
\begin{array}{rl}
\lceil c \rceil &= \lceil x_c \rceil \\
\lceil \Diamond^{\tau} \rceil &= \lceil \lambda(x:\tau)\ x \rceil \\
\lceil \mathsf{Top}^{\tau} \rceil &= \lceil \lambda(y:\tau)\ \lambda\alpha\ \lambda(x : \forall\beta.\, \beta \to \alpha)\ x\, \tau\ y \rceil \\
\lceil G_1 \xrightarrow{\tau} G_2 \rceil &= \lceil \lambda(y : \mathsf{dom}(G_1 \xrightarrow{\tau} G_2))\ \lambda(x:\tau)\ G_2\langle y\ (G_1\langle x \rangle) \rangle \rceil \\
\lceil \mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}))\ \lambda(x:\tau)\ \lambda\alpha\ y\, \alpha\ x \rceil \\
\lceil \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}))\ \lambda(x:\tau)\ \lambda(c:\varphi)\ y\{c\}\ x \rceil \\
\lceil (G_1 * G_2) \rceil &= \lceil \lambda(y : \mathsf{dom}(G_1 * G_2))\ (G_1\langle y.\mathtt{1} \rangle, G_2\langle y.\mathtt{2} \rangle) \rceil \\
\lceil \mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}))\ (\lambda\alpha\ (y\, \alpha).\mathtt{1}, \lambda\alpha\ (y\, \alpha).\mathtt{2}) \rceil \\
\lceil \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\sigma)} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\sigma)}))\ (\lambda(c:\varphi)\ (y\{c\}).\mathtt{1}, \lambda(c:\varphi)\ (y\{c\}).\mathtt{2}) \rceil \\
\lceil P[G] \rceil &= \lceil \lambda(x : \mathsf{dom}(G))\ P[G\langle x \rangle] \rceil
\end{array}
$$

Figure 9: Reification of F_ι into System F

2. *If* $\Gamma \vdash M : \tau$ *holds, then* $\Gamma' \vdash M : \tau$ *holds.*
3. *If* $\Gamma \vdash G : \varphi$ *holds, then* $\Gamma' \vdash G : \varphi$ *holds.*

(Proof p. 50)

Lemma 4 (Type Substitution)**.** *If* $\Gamma \vdash \rho$ *holds and* θ *is* $[\alpha \leftarrow \rho]$*, we have:*

1. *If* $(x:\tau) \in \Gamma, \alpha, \Gamma'$ *holds, then* $(x : \tau\theta) \in \Gamma, \Gamma'\theta$ *holds.*
2. *If* $(c:\varphi) \in \Gamma, \alpha, \Gamma'$ *holds, then* $(c : \varphi\theta) \in \Gamma, \Gamma'\theta$ *holds.*
3. *If* $\beta \in \mathsf{dom}(\Gamma, \alpha, \Gamma')$ *holds and* $\alpha \neq \beta$*, then* $\beta \in \mathsf{dom}(\Gamma, \Gamma'\theta)$ *holds.*
4. *If* $\Gamma, \alpha, \Gamma' \vdash ok$ *holds, then* $\Gamma, \Gamma'\theta \vdash ok$ *holds.*
5. *If* $\Gamma, \alpha, \Gamma' \vdash \tau$ *holds, then* $\Gamma, \Gamma'\theta \vdash \tau\theta$ *holds.*
6. *If* $\Gamma, \alpha, \Gamma' \vdash M : \tau$ *holds, then* $\Gamma, \Gamma'\theta \vdash M\theta : \tau\theta$ *holds.*
7. *If* $\Gamma, \alpha, \Gamma' \vdash G : \varphi$ *holds, then* $\Gamma, \Gamma'\theta \vdash G\theta : \varphi\theta$ *holds.*

(Proof p. 50)

Lemma 5 (Extract Type)**.** *The following assertions hold:*

1. *If* $\Gamma \vdash M : \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*
2. *If* $\Gamma \vdash G : \varphi$ *holds, then* $\Gamma \vdash \varphi$ *holds.*

(Proof p. 50)

Lemma 6 (Term Substitution)**.** *If* $\Gamma \vdash N : \rho$ *holds, then:*

1. *If* $\Gamma, x : \rho \vdash M : \tau$ *holds, then* $\Gamma \vdash M[x \leftarrow N] : \tau$ *holds.*
2. *If* $\Gamma, x : \rho \vdash G : \varphi$ *holds, then* $\Gamma \vdash G[x \leftarrow N] : \varphi$ *holds.*
3. *If* $\Gamma, x : \rho \vdash \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*

(Proof p. 50)

Lemma 7 (Coercion Substitution)**.** *If* $\Gamma \vdash G' : \varphi'$ *holds, then:*

1. *If* $\Gamma, c : \varphi' \vdash M : \tau$ *holds, then* $\Gamma \vdash M[c \leftarrow G'] : \tau$ *holds.*
2. *If* $\Gamma, c : \varphi' \vdash G : \varphi$ *holds, then* $\Gamma \vdash G[c \leftarrow G'] : \varphi$ *holds.*
3. *If* $\Gamma, c : \varphi' \vdash \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*

(Proof p. 50)

Proposition 8 (Subject Reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \rightsquigarrow_{\beta\iota} N$ *hold, then* $\Gamma \vdash N : \tau$ *holds.*

(Proof p. 50)

The proof of progress is standard, using the classification lemma to determine the shape of values from the shape of their types. Under a strong reduction strategy, the classification of values is stated as follows:

Lemma 9 (Classification)**.** *If* $\Gamma \vdash v : \tau$ *holds, then either* v *is a prevalue* p *or:*

1. *If* τ *is of the form* $\tau_1 \to \tau_2$*, then* v *is of the form* $\lambda(x : \tau')\ v'$*.*
2. *If* τ *is of the form* $(\tau_1 * \tau_2)$*, then* v *is of the form* (v_1, v_2)*.*
3. *If* τ *is of the form* $\forall\alpha.\tau'$*, then* v *is of the form* $\lambda\alpha\ v'$*.*
4. *If* τ *is of the form* $\varphi \Rightarrow \tau'$*, then* v *is of the form* $\lambda(c : \varphi')\ v'$*.*
5. *If* τ *is of the form* $\top$*, then* v *is of the form* $\mathtt{Top}^\tau\langle v'\rangle$*.*

(Proof p. 51)

Proposition 10 (Progress)**.** *If* $\Gamma \vdash M : \tau$ *holds, then either* M *is a value or* M *reduces.*

(Proof p. 51)

3.2 Termination of reduction

The termination of reduction for F_ι can be piggybacked on the termination of reduction in System F: following Manzonetto and Tranquilli [2010], we show a forward simulation between F_ι and System F, by translating F_ι into System F so that every reduction step in F_ι is simulated by at least one reduction step in System F.

3.3 Reification of F_ι in System F

There is indeed a natural translation of F_ι into System F obtained by reifying coercions as actual computation steps: even though we ultimately erase ι-steps, we do not actually need to do so, and on the contrary, we may see them as computation steps in System F.

Reification is described on Figure 9. We write $\lceil M \rceil$ for the reification of M. Coercions of coercion type $\tau \triangleright \sigma$ are reified as functions of type $\tau \to \sigma$. Hence, a coercion abstraction $\lambda(c : \tau \triangleright \sigma)\ M$ is reified as a higher-order function $\lambda(x_c : \lceil\tau\rceil \to \lceil\sigma\rceil)\ \lceil M \rceil$. A coercion variable c is reified as a term variable x_c (we assume an injective mapping of coercion variables to reserved term variables). Thus, the type $(\tau \triangleright \sigma) \Rightarrow \rho$ of a term abstracted over a coercion is translated into the type $(\lceil\tau\rceil \to \lceil\sigma\rceil) \to \lceil\rho\rceil$ of a higher-order function. Other type expressions are reified homomorphically. The application of a coercion to a term and the application of a term to a coercion are both reified as applications.

The remaining cases are the translation of coercions G, which are all done in two steps: we first translate G into some F_ι-term performing η-expansions to transform a coercion from τ to σ into a function from τ to σ. For atomic coercions (variables, identity, or distributivity), the result of this step is in the System-F subset of F_ι. However, for complex coercions, the result still contains inner coercions. Hence, in the second step, we recursively translate the result of the first step. This translates types and residual coercions. Notice that the first step may introduce applications of coercions to terms, which are then turned into applications of terms to terms during the second step.

The translation of $P[G]$ covers five subcases, one for each form of P. Here as in the reduction rules, the two occurrences of P are different abstract nodes since P is a coercion on the left-hand side and a term on the right-hand side.

The translation uses an auxiliary predicate dom that computes the domain of a coercion: the domain of a coercion G in environment Γ is the unique type τ such that $\Gamma \vdash G : \tau \triangleright \sigma$ for some type σ. This cannot be computed locally. Hence, we assume that terms of F_ι have been previously typechecked and all coercions have been annotated with their domain type. Alternatively, we can define the reification as a translation of typing derivations. We actually use such a translation to show that reification preserves well-typedness.

Proposition 11 (Well-typedness of reification)**.** *The following assertions hold:*

1. *If* $\Gamma \vdash M : \tau$ *holds, then* $\lceil\Gamma\rceil \vdash \lceil M\rceil : \lceil\tau\rceil$ *holds.*
2. *If* $\Gamma \vdash G : \varphi$ *holds, then* $\lceil\Gamma\rceil \vdash \lceil G\rceil : \lceil\varphi\rceil$ *holds.*
3. *If* $\Gamma \vdash \tau$ *holds, then* $\Gamma \vdash \lceil\tau\rceil$ *holds.*
4. *If* $\Gamma \vdash ok$ *holds, then* $\lceil\Gamma\rceil \vdash ok$ *holds.*

Proof. The translation of typing derivations can be easily deduced from Figure 9 since we are explicitly typed. To prove each assertion we proceed by induction on its judgment. For each typing rule we just verify that the translated derivation is valid in System F using induction hypothesis. □

It is easy to verify that reduction in F_ι can be simulated in the translation, which implies the termination of reduction in F_ι.

Lemma 12 (Forward simulation)**.** *If* $\Gamma \vdash M : \tau$ *holds, then:*

1. *If* $M \rightsquigarrow_\beta N$*, then* $\lceil M\rceil \rightsquigarrow \lceil N\rceil$*;*
2. *If* $M \rightsquigarrow_\iota N$*, then* $\lceil M\rceil \rightsquigarrow^+ \lceil N\rceil$*.*

(Proof p. 53)

Corollary 13 (Termination)**.** *Reduction in* F_ι *is terminating.*

Proof. Assume that M is well-typed in F_ι. By Lemma 11, $\lceil M\rceil$ is well-typed in System F, hence the length of reduction sequences starting with $\lceil M\rceil$ in System F is bounded by some integer N. By Lemma 12, N is also a bound to the length of reduction sequence starting with M in F_ι. □

3.4 Confluence

Reduction in F_ι is allowed in any term-context. Since coercions do not contain terms and coercions are never reduced alone, we may equivalently allow reduction in all coercion contexts, since no rule will ever apply. Hence, reduction in F_ι is a rewriting system.

An analysis of reduction rules in F_ι shows that there are no critical pairs. Hence, the reduction is weakly confluent. Since reduction is also terminating, it is confluent.

Corollary 14 (Confluence)**.** *Reduction in* F_ι *is confluent.*

Proof. There is no critical pairs in F_ι. Rule RedCoerFill cannot be part of a critical pair because its left-hand side is of the form $P[W]\langle M\rangle$ and reduction contexts do not allow reduction in contexts of the form $[]\langle M\rangle$. The left-hand sides of all other rules start with a destructor and do not contain any other destructor underneath— destructors are term, type, and coercion applications ($M\ M$, $M\,\tau$, and $M\{G\}$), term projections ($M.1$ and $M.2$), and term coercion ($G\langle M\rangle$). Thus, there is no opportunity for superposition.

Because reduction is permitted in any term context, it is a rewriting system. Hence, the reduction in F_ι is locally confluent. Because it is terminating (Lemma 13), it is confluent (Newman's lemma). □

In fact, the relation $\rightsquigarrow_\iota$ alone is confluent.

Lemma 15. *If both $M \rightsquigarrow_\iota^\star M_1$ and $M \rightsquigarrow_\iota^\star M_2$ hold, then there exists a term N such that $M_1 \rightsquigarrow_\iota^\star N$ and $M_2 \rightsquigarrow_\iota^\star N$.*

Moreover, the reduction $\rightsquigarrow_\beta$ and $\rightsquigarrow_\iota^\star$ commute:

Lemma 16. *If $M \rightsquigarrow_\beta M_1$ and $M \rightsquigarrow_\iota M_2$ hold, then there is a term N such that $M_1 \rightsquigarrow_\iota^\star N$ and $M_2 \rightsquigarrow_\beta N$.*

3.5 Forward simulation

Coercion erasure sends terms of F_ι into the (untyped) λ-calculus. It also induces a simulation from the reduction in F_ι by the reduction in the λ-calculus, where ι-steps becomes equalities.

Lemma 17 (Forward simulation)**.** *If $\Gamma \vdash M : \tau$ holds, then:*

1. *If $M \rightsquigarrow_\beta N$, then $\lfloor M\rfloor \rightsquigarrow \lfloor N\rfloor$.*
2. *If $M \rightsquigarrow_\iota N$, then $\lfloor M\rfloor = \lfloor N\rfloor$.*

(Proof p. 54)

Unfortunately, the backward simulation fails. The wedge $\lambda(c : \tau \to \tau \triangleright \tau \to \tau)\,\lambda(y : \tau)\ c\langle\lambda(x : \tau)\ x\rangle\ y$ is a well-typed closed value in F_ι while its erasure $\lambda y.(\lambda x.x)\,y$ β-reduces to $\lambda y.y$.

To recover bisimulation, the definition of the language must be adjusted so that wedge configurations cannot appear in a reduction context. This observation leads to two opposite solutions, which we present in §5 and §7.

4 Coercions as retyping functions: F_ι^λ

While the reification of F_ι into System F carries good intuitions about what coercions really are, it lacks the ability to distinguish coercions from expressions with computational content. There is an alternative presentation of F_ι, called F_ι^λ, that maintains the distinction between coercions and expressions while remaining closer to the reified form of coercions: F_ι^λ is mainly a coercion decoration of System F. In this sense, it can be seen as an explicit version (with pairs and coercion abstraction) of Mitchell's presentation of F_η as System F with retyping functions.

The main difference is that coercions are directly built as retyping functions in F_ι^λ, using the constructs of the λ-calculus instead of the combinator-like coercion language of F_ι. This makes it easier to write coercions manually and it is thus more appealing from a practical point of view.

The reification of F_ι into System F can be redefined as the composition of a translation from F_ι to F_ι^λ that keeps the distinction between coercions and terms and the final erasing of this differenceobtained by mapping all abstraction-like nodes and application-like nodes of F_ι^λ to term abstractions and applications of System F. The first part, along with its inverse, define translations between F_ι and F_ι^λ that preserves well-typedness and coercion erasure. Although we have not proved it, F_ι and F_ι^λ should be the same up to their representation of coercions.

Unfortunately, typechecking in F_ι^λ is more involved than in F_ι, as we need to typecheck coercions as *binding* expressions.

$$
\begin{array}{cl}
x, y & \text{term} \\
c & \text{coercion} \\
\alpha, \beta & \text{type} \\
\phi & \text{hole}
\end{array}
$$

$$
\begin{array}{rcll}
M, N & ::= & x \mid \lambda(x:\tau)\, M \mid M\, M \mid (M, M) \mid M.1 \mid M.2 & \text{terms} \\
& \mid & H\, M \mid \lambda\alpha\, M \mid M\, \tau \mid \mathtt{Top}^{\tau}\, M & \\
& \mid & \lambda(c:\varphi)\, M \mid M\, H & \\
& \mid & \lambda(\phi:\tau)\, G\, \{\phi \leftarrow M\, G\} \mid (G, G)\{\phi, \phi \leftarrow M\} & \\
G & ::= & \phi \mid H\, G \mid \lambda\alpha\, G \mid G\, \tau \mid \mathtt{Top}^{\tau}\, G & \text{open coercions} \\
& \mid & \lambda(c:\varphi)\, G \mid G\, H & \\
& \mid & \lambda(\phi:\tau)\, G\, \{\phi \leftarrow G\, G\} \mid (G, G)\{\phi, \phi \leftarrow G\} & \\
H & ::= & c \mid \lambda(\phi:\tau)\, G & \text{close coercions} \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid (\tau * \tau) \mid \forall\alpha.\tau \mid \varphi \Rightarrow \tau \mid \top & \text{types} \\
\varphi & ::= & \tau \triangleright \tau & \text{coercion types} \\
\Gamma & ::= & \emptyset \mid \Gamma, x:\tau \mid \Gamma, \alpha \mid \Gamma, c:\varphi & \text{expression environments} \\
\Delta & ::= & \emptyset \mid \alpha, \Delta \mid c:\varphi, \Delta & \text{coercion type environments} \\
Z & ::= & \Box \mid \Delta \star (\phi:\tau) & \text{coercion types}
\end{array}
$$

Figure 10: System $\mathsf{F}_{\iota}^{\lambda}$: syntax

The reason is that coercions are not *exactly* λ-expressions. Having coercions as λ-expressions would require an even more elaborated type system, as it would have to ensure that coercions are η-expansions, which means maintaining a stack of the currently η-expanded variables to remember closing them. For example, consider typechecking the retyping context $\lambda(x:\tau)\ \lambda\alpha\ []\,\alpha\ x$ that permutes term abstraction and type abstraction (known as distributivity): when typechecking the subterm $\lambda\alpha\ []\,\alpha\ x$, we must verify that it is the body of an η-expansion with the variable x. We initially followed this approach and it was cumbersome; moreover, it did not scale to products as the type system must also ensure that two sub-derivation trees have the same coercion erasure.

Instead, we make the η-expansion of a term M an atomic construct, namely $\lambda(\phi_1:\tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\}$ where ϕ's stand for *hole* variables. This can be interpreted as $\lambda(x:\tau)\ G_2'[M\ G_1'[x]]$ which is the η-expansion of M (*i.e.* $\lambda x.M\,x$) using coercion G_1 (interpreted as G_1') around the argument and coercion G_2 (interpreted as G_2') around the result. Here G_2 may bind coercion or hole variables that are used inside M and G_1. Hence, the type system must keep track of those variables with their types when typechecking G_2 and extend the typing environment accordingly when typechecking M and G_1.

We presented F_{ι} rather than its more intuitive version $\mathsf{F}_{\iota}^{\lambda}$ to avoid the additional complexity in the type system; moreover, it is not obvious how to extend $\mathsf{F}_{\iota}^{\lambda}$ with projectors, as discussed in §9.

4.1 Definition of $\mathsf{F}_{\iota}^{\lambda}$

Syntax The syntax of $\mathsf{F}_{\iota}^{\lambda}$ is described on Figure 10. The main differences between F_{ι} and $\mathsf{F}_{\iota}^{\lambda}$ is the replacement of distributivity and both arrow and product congruence rules with more general constructs based on η-expansion (last two forms of expressions). However, there are also a few changes in the presentation. We introduce a new kind of variables, called hole variables and written ϕ, to name the hole of coercions—when seen as retyping contexts. The main reason for naming the holes in $\mathsf{F}_{\iota}^{\lambda}$ is to bring the representation of coercions closer to their reified form, which is abstracted over their unique hole. This induces changes in the syntax of expressions. We also distinguish between *close* and *open* coercions, respectively written with letter H and G, which can also be respectively understood as retyping *functions* and retyping *contexts*.

$$\textsc{LExprTermVar}\ \frac{\vdash \Gamma \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{LExprTermLam}\ \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau)\, M : \tau \to \sigma} \qquad \textsc{LExprTermApp}\ \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M\, N : \sigma}$$

$$\textsc{LExprTermPair}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M, N) : (\tau * \sigma)} \qquad \textsc{LExprTermFst}\ \frac{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.1 : \tau} \qquad \textsc{LExprTermSnd}\ \frac{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.2 : \sigma} \qquad \textsc{LExprHoleVar}\ \frac{\Gamma \vdash \tau}{\Gamma; \emptyset \star (\phi : \tau) \vdash \phi : \tau}$$

$$\textsc{LExprHoleApp}\ \frac{\Gamma \vdash H : \tau \triangleright \sigma \qquad \Gamma; Z \vdash W : \tau}{\Gamma; Z \vdash H\, W : \sigma} \qquad \textsc{LExprTypeLam}\ \frac{\Gamma, \alpha; Z \vdash W : \tau}{\Gamma; (\alpha, \emptyset), Z \vdash \lambda\alpha\, W : \forall\alpha.\tau} \qquad \textsc{LExprTypeApp}\ \frac{\Gamma; Z \vdash W : \forall\alpha.\sigma \qquad \Gamma \vdash \tau}{\Gamma; Z \vdash W\, \tau : \sigma[\alpha \leftarrow \tau]}$$

$$\textsc{LExprTop}\ \frac{\Gamma; Z \vdash W : \tau}{\Gamma; Z \vdash \mathsf{Top}^{\tau}\, W : \top} \qquad \textsc{LExprCoerLam}\ \frac{\Gamma, c : \varphi; Z \vdash W : \rho}{\Gamma; (c : \varphi, \emptyset), Z \vdash \lambda(c : \varphi)\, W : \varphi \Rightarrow \rho} \qquad \textsc{LExprCoerApp}\ \frac{\Gamma; Z \vdash W : \varphi \Rightarrow \rho \qquad \Gamma \vdash H : \varphi}{\Gamma; Z \vdash W\, H : \rho}$$

$$\textsc{LExprEtaArr}\ \frac{\Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \qquad \Gamma, \Delta; Z \vdash W : \tau' \to \sigma' \qquad \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \qquad \Gamma \vdash \tau}{\Gamma; \Delta, Z \vdash \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow W\, G_1\} : \tau \to \sigma}$$

$$\textsc{LExprEtaPrd}\ \frac{\Gamma; \Delta \star (\phi_1 : \tau') \vdash G_1 : \tau \qquad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \qquad \Gamma, \Delta; Z \vdash W : (\tau' * \sigma')}{\Gamma; \Delta, Z \vdash (G_1, G_2)\{\phi_1, \phi_2 \leftarrow W\} : (\tau * \sigma)}$$

$$\textsc{LExprCoerVar}\ \frac{\vdash \Gamma \qquad \Gamma(c) = \varphi}{\Gamma \vdash c : \varphi} \qquad \textsc{LExprHoleLam}\ \frac{\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \lambda(\phi : \tau)\, G : \tau \triangleright \sigma}$$

Figure 11: System $\mathsf{F}_{\iota}^{\lambda}$: typing rules

Terms are presented in four groups (each one on a separate line). The first group corresponds to constructs of the λ-calculus. The second group corresponds to coercion application, System-F coercion constructs, and coercion to top. Terms of these two groups are as in F_{ι} up to minor differences. The third group corresponds to coercion abstraction. The last group describes the new η-expansion coercion forms: $\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\}$ for arrows and $(G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}$ for products. The brace notation suggests the substitution of hole variables by an expression. In both terms variables ϕ_1 is bound in G_1 (but not in G_2) and variable ϕ_2 is bound in G_2 (but not in G_1). As suggested by the notation these are let-like bindings except for ϕ_1 in $\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\}$, which is λ-bound.

Coercions are in three groups. The first group corresponds to reflexivity, transitivity, System-F coercion constructs, and coercion to top. The third group corresponds to coercion abstraction. The last group describes the new η-expansion coercion forms.

Notice that there are four kinds of λ-abstractions, which can be immediately distinguished by the syntactic class of the variable they bind: x is used for term abstraction, α for type abstraction, c for coercion abstraction, and ϕ is used for hole abstraction. Correspondingly, there are four kind of applications. There are all syntactically written by juxtaposition but can be distinguished as follows: type application is $M\, \tau$; coercion application is $W\, H$; hole application $H\, W$; and term application $M\, M$ being the default, where H is syntactically recognizable as it is either a coercion variable c or a coercion abstraction $\lambda(\phi : \tau)\, G$.

We introduce a subset of environments, called a coercion type environment and written Δ, that does not bind term variables, which is used in the static semantics to allow coercions to bind. Notice however that Δ is extended on the left, while Γ is extended on the right.

$$\textsc{LTypeVar}\ \frac{\vdash \Gamma \qquad \Gamma(\alpha)}{\Gamma \vdash \alpha} \qquad \textsc{LTypeArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau \to \sigma} \qquad \textsc{LTypeProduct}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash (\tau * \sigma)} \qquad \textsc{LTypeForall}\ \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\tau} \qquad \textsc{LTypeCoer}\ \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \rho}{\Gamma \vdash \varphi \Rightarrow \rho}$$

$$\textsc{LTypeTop}\ \frac{\vdash \Gamma}{\Gamma \vdash \top}$$

$$\textsc{LEnvEmpty}\ \vdash \emptyset \qquad \textsc{LEnvExpr}\ \frac{\Gamma \vdash \tau \qquad x \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \textsc{LEnvType}\ \frac{\vdash \Gamma \qquad \alpha \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, \alpha} \qquad \textsc{LEnvCoer}\ \frac{\Gamma \vdash \varphi \qquad c \notin \mathsf{dom}(\Gamma)}{\vdash \Gamma, c : \varphi}$$

$$\textsc{LDEnvEmpty}\ \vdash \emptyset \qquad \textsc{LDEnvType}\ \frac{\vdash \Delta \qquad \alpha \notin \mathsf{dom}(\Delta)}{\vdash \alpha, \Delta} \qquad \textsc{LDEnvCoer}\ \frac{\emptyset, \Delta \vdash \varphi \qquad c \notin \mathsf{dom}(\Delta)}{\vdash c : \varphi, \Delta}$$

Figure 12: System F_ι^λ: well-formedness rules

Typing The static semantics, defined on Figure 11, is quite similar to the one of F_ι. To factor out typing rules, we use a new judgment $\Gamma; Z \vdash W : \sigma$ where Z is either $\Box$ or $\Delta \star (\phi : \tau)$: when Z is $\Box$, then W is a term M of type σ under Γ, whereas when Z is $\Delta \star (\phi : \tau)$ then W is a coercion G that may bind, retyping a term (bound to ϕ) of type τ under Γ, Δ into one of type σ under Γ. For conciseness, we just write $\Gamma \vdash M : \tau$ instead of $\Gamma; \Box \vdash M : \tau$. The notation Δ, Z, defined on Figure 13, is used for optional extension of coercion type bindings. These judgments could be expanded into several more basic judgments by eliminating disjunctions on Z. We also define $\Gamma \vdash H : \varphi$ to type close coercions.

For example, LExprTypeLam can be seen as the union of the two following rules (the first is for terms and the second for coercions):

$$\frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \lambda \alpha\, M : \forall \alpha.\tau} \qquad \frac{\Gamma, \alpha; \Delta \star (\phi : \varphi) \vdash G : \tau}{\Gamma; \alpha, \Delta \star (\phi : \varphi) \vdash \lambda \alpha\, G : \forall \alpha.\tau}$$

However, the use of Δ cannot be eliminated in typing rules. Consider for example LExprEtaArr, which becomes when Z is $\Box$:

$$\textsc{LExprEtaArr}\ \frac{\Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \qquad \Gamma, \Delta \vdash M : \tau' \to \sigma' \qquad \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \qquad \Gamma \vdash \tau}{\Gamma \vdash \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\} : \tau \to \sigma}$$

The coercion G_2 retypes a term ϕ_2 of type σ' under Γ, Δ into a term G_2 of type σ under Γ; this allows M and G_1 to use both type and coercion variables that are bound in G_2. We need this expressiveness to encode distributivity. For example, $\mathsf{Dist}^{\forall \alpha.}_{\tau \to \sigma}$ in F_ι is translated to $\lambda(\phi_1 : \tau)\,(\lambda \alpha\ \phi_2)\,\{\phi_2 \leftarrow (\phi\, \alpha)\, \phi_1\}$ in F_ι^λ, which is only well-typed (and in particular well-scoped) if we allow coercions to bind as described above.

Rules LExprEtaArr and LExprEtaProd are the two new typing rules in F_ι^λ—for each of the two new language constructs $\lambda(z_1 : \tau)\, G_2\, \{z_2 \leftarrow W\, G_1\}$ and $(G_1, G_2)\{z_1, z_2 \leftarrow W\}$, which apply coercions in the η-expansions of terms (λ-abstractions and pairs, respectively). For instance, $\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\}$ can be understood as $\lambda(x : \tau)\ G_2[\phi_2 \leftarrow M\ G_1[\phi_1 \leftarrow x]]$ which is the coerced η-expansion of M, and similarly $(G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}$ can be understood as $(G_1[\phi_1 \leftarrow M.1], G_2[\phi_2 \leftarrow M.2])$. However, while in the later case the η-expansion duplicates M, the primitive construct does not—it only shares the typechecking of G_1 and G_2 with the same coercion typing context Δ.

Well-formedness rules for types and environments are defined in the obvious way on Figure 12.

Operational semantics Values and reduction contexts are defined on Figure 14; reduction rules are given on both Figure 15, which are as in F_ι, and Figure 16, which are just β-reduction

$$\Delta, \Box = \Box$$
$$\Delta, (\Delta' \star (\phi : \tau)) = (\Delta, \Delta') \star (\phi : \tau)$$

Figure 13: System F_ι^λ: coercion type extension

$$\begin{array}{rcll}
p & ::= & x \mid p\,v \mid p.\mathtt{1} \mid p.\mathtt{2} \mid c\,v \mid p\,\tau \mid p\,H & \text{prevalues} \\
v & ::= & p \mid \lambda(x:\tau)\,v \mid (v,v) \mid \lambda\alpha\,v \mid \lambda(c:\varphi)\,v \mid \mathtt{Top}^\tau\,v & \text{values} \\
 & \mid & \lambda(\phi:\tau)\,G\,\{\phi \leftarrow p\,G\} \mid (G,G)\{\phi,\phi \leftarrow p\} & \\
C & ::= & \lambda(x:\tau)\,[] \mid []\,M \mid M\,[] \mid ([],M) \mid (M,[]) \mid []\mathtt{.1} \mid []\mathtt{.2} \mid P & \text{evaluation contexts} \\
P & ::= & H\,[] \mid \lambda\alpha\,[] \mid []\,\tau \mid \lambda(c:\varphi)\,[] \mid []\,H \mid \mathtt{Top}^\tau\,[] & \text{retyping contexts} \\
 & \mid & \lambda(\phi:\tau)\,G\,\{\phi \leftarrow []\,G\} \mid (G,G)\{\phi,\phi \leftarrow []\} &
\end{array}$$

Figure 14: System F_ι^λ: values and reduction contexts

$$\textsc{LRedContextBeta}\ \frac{M \rightsquigarrow_\beta N}{C[M] \rightsquigarrow_\beta C[N]} \qquad \textsc{LRedContextIota}\ \frac{M \rightsquigarrow_\iota N}{C[M] \rightsquigarrow_\iota C[N]}$$

$$\textsc{LRedTerm}\ (\lambda(x:\tau)\,M)\,N \rightsquigarrow_\beta M[x \leftarrow N] \qquad \textsc{LRedFirst}\ (M,N).\mathtt{1} \rightsquigarrow_\beta M$$

$$\textsc{LRedSecond}\ (M,N).\mathtt{2} \rightsquigarrow_\beta N \qquad \textsc{LRedType}\ (\lambda\alpha\,M)\,\tau \rightsquigarrow_\iota M[\alpha \leftarrow \tau] \qquad \textsc{LRedCoer}\ (\lambda(c:\varphi)\,M)\,H \rightsquigarrow_\iota M[c \leftarrow H]$$

$$\textsc{LRedHole}\ (\lambda(\phi:\tau)\,G)\,M \rightsquigarrow_\iota G[\phi \leftarrow M]$$

Figure 15: System F_ι^λ: usual reduction rules

for the new η-expansion constructs. This can be seen by reification into System F (defined in Figure 17 below). For example, the reification of Rule LRedEtaArrApp is:

$$(\lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil[\phi_2 \leftarrow \lceil M\rceil\ \lceil G_1\rceil[\phi_1 \leftarrow x]])\ \lceil N\rceil \rightsquigarrow \lceil G_2\rceil[\phi_2 \leftarrow \lceil M\rceil\ \lceil G_1\rceil[\phi_1 \leftarrow \lceil N\rceil]]$$

Another minor difference is that $(\lambda(\phi:\tau)\,G)\,M$ is now reduced in one big step to $G[\phi \leftarrow M]$ in rule LRedHole, instead of a sequence of smaller step-by-step reductions as in F_ι. The advantage is that after reification, this will be a usual β-reduction, hence the mathematical substitution in one big step. Notice that this substitution $G[\phi \leftarrow M]$ has to convert all the P nodes between the occurence of ϕ in G and the root of G. This was already happening in F_ι, but in small step. Now, because the hole substitution happens in one big step, we also rewrite the nodes along the path to the hole in one big step.

Rules LRedEtaArrApp, LRedEtaArrEtaArr, LRedEtaPrdFst, LRedEtaPrdSnd, and LRedEtaPrdEtaPrd are the missing rules of F_ι.

4.2 Soundness

In order to prove subject reduction, we need the usual weakening and substitution lemmas, as usual. We will only give the substitution lemma for hole variables since Δ plays a particular role. Other substitution lemmas and weakening lemmas are as usual. Their proofs are routine.

Lemma 18 (Hole substitution)**.** *If* $\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma$ *and* $\Gamma, \Delta \vdash M : \tau$ *hold, then* $\Gamma \vdash G[\phi \leftarrow M] : \sigma$.

As usual, soundness follows from subject reduction and progress lemmas.

Proposition 19 (Subject Reduction)**.** *If* $\Gamma \vdash M : \tau$ *and* $M \rightsquigarrow_{\beta\iota} N$ *hold, then* $\Gamma \vdash N : \tau$ *holds.*

(Proof p. 54)

Progress lemmas uses the following classification of values.

$$\textsc{LRedEtaArrApp}\quad (\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\})\, N \rightsquigarrow_\iota G_2[\phi_2 \leftarrow M\, G_1[\phi_1 \leftarrow N]]$$

$$\textsc{LRedEtaArrLam}\quad \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(x : \tau')\, M)\, G_1\} \rightsquigarrow_\iota \lambda(x : \tau)\, G_2[\phi_2 \leftarrow M[x \leftarrow G_1[\phi_1 \leftarrow x]]]$$

$$\textsc{LRedEtaArrEtaArr}\quad \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(\phi_1' : \tau')\, G_2'\, \{\phi_2' \leftarrow M\, G_1'\})\, G_1\} \rightsquigarrow_\iota \lambda(\phi_1 : \tau)\, G_2[\phi_2 \leftarrow G_2']\, \{\phi_2' \leftarrow M\, G_1'[\phi_1' \leftarrow G_1]\}$$

$$\textsc{LRedEtaPrdFst}\quad ((G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}).1 \rightsquigarrow_\iota G_1[\phi_1 \leftarrow M.1] \qquad \textsc{LRedEtaPrdSnd}\quad ((G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}).2 \rightsquigarrow_\iota G_2[\phi_2 \leftarrow M.2]$$

$$\textsc{LRedEtaPrdPair}\quad (G_1, G_2)\{\phi_1, \phi_2 \leftarrow (M, N)\} \rightsquigarrow_\iota (G_1[\phi_1 \leftarrow M], G_2[\phi_2 \leftarrow N])$$

$$\textsc{LRedEtaPrdEtaPrd}\quad (G_1, G_2)\{\phi_1, \phi_2 \leftarrow ((G_1', G_2')\{\phi_1', \phi_2' \leftarrow M\})\} \rightsquigarrow_\iota (G_1[\phi_1 \leftarrow G_1'], G_2[\phi_2 \leftarrow G_2'])\{\phi_1', \phi_2' \leftarrow M\}$$

Figure 16: System F_ι^λ: eta-related reduction rules

Lemma 20 (Classification of values)**.** *If $\Gamma \vdash v : \tau$ holds, then either v is a prevalue p or:*

1. *If τ is of the form $\tau_1 \to \tau_2$, then v is either of the form $\lambda(x : \tau_1)\, v'$ or of the form $\lambda(\phi_1 : \tau_1)\, G_2\, \{\phi_2 \leftarrow p\, G_1\}$.*
2. *If τ is of the form $(\tau_1 * \tau_2)$, then v is either of the form (v_1, v_2) or $(G_1, G_2)\{\phi_1, \phi_2 \leftarrow p\}$.*
3. *If τ is of the form $\forall\alpha.\tau'$, then v is of the form $\lambda\alpha\, v'$.*
4. *If τ is of the form $\varphi \Rightarrow \tau'$, then v is of the form $\lambda(c : \phi)\, v'$.*
5. *If τ is of the form $\top$, then v is of the form $\mathsf{Top}^{\tau'}\, v'$.*

Proposition 21 (Progress)**.** *If $\Gamma \vdash M : \tau$ holds, then either M is a value or M reduces.*

(Proof p. 54)

4.3 Confluence

The new η-reduction rules introduce several critical pairs in F_ι^λ each originating from one of the following configurations:

$$(\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(x : \tau')\ M)\, G_1\})\ N$$

$$(\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(\phi_1' : \tau')\, G_2'\, \{\phi_2' \leftarrow M\, G_1'\})\, G_1\})\ N$$

$$\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(\phi_1' : \tau')\, G_2'\, \{\phi_2' \leftarrow (\lambda(x : \tau'')\ M)\, G_1'\})\, G_1\}$$

$$\lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow (\lambda(\phi_1' : \tau')\, G_2'\, \{\phi_2' \leftarrow (\lambda(\phi_1'' : \tau'')\, G_2''\, \{\phi_2'' \leftarrow M\, G_1''\})\, G_1'\})\, G_1\}$$

$$(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G_1', G_2')\{\phi_1', \phi_2' \leftarrow (M, N)\}\}$$

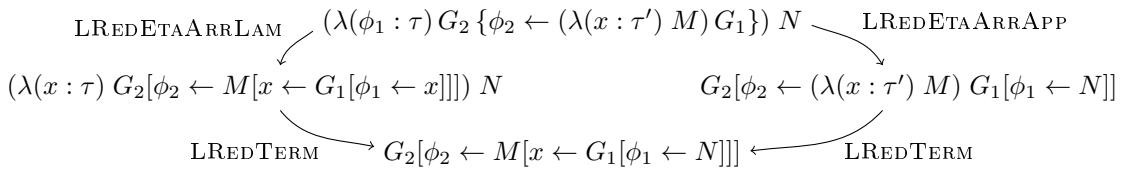
$$(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (M, N)\}.1 \qquad (G_1, G_2)\{\phi_1', \phi_2' \leftarrow (M, N)\}.2$$

$$(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G_1', G_2')\{\phi_1', \phi_2' \leftarrow M\}\}.1 \qquad (G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G_1', G_2')\{\phi_1', \phi_2' \leftarrow M\}\}.2$$

$$(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G_1', G_2')\{\phi_1', \phi_2' \leftarrow (G_1'', G_2'')\{\phi_1'', \phi_2'' \leftarrow M\}\}\}$$

For example, the first configuration can be reduced with either LRedEtaArrLam or LRedEtaArrApp leading to the following critical pair, which however converges in one step as described

below:



Other configurations are similar.

4.4 Reification into System F

The reification, which is mainly decoration erasure, is defined on Figure 17. We reify a well-formed type of $\mathsf{F}^{\lambda}_{\iota}$ to a well-formed type of System F. And we reify a well-typed expression to a well-typed term in System F.

We have the following properties:

Lemma 22. *The following assertions hold:*

- *If* $\Gamma \vdash M : \tau$ *holds, then* $\lceil\Gamma\rceil \vdash \lceil M\rceil : \lceil\tau\rceil$ *holds.*
- *If* $\Gamma \vdash H : \tau \triangleright \sigma$ *holds, then* $\lceil\Gamma\rceil \vdash \lceil H\rceil : \lceil\tau\rceil \to \lceil\sigma\rceil$ *holds.*
- *If both* $\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma$ *and* $\lceil\Gamma, \Delta\rceil \vdash M : \lceil\tau\rceil$ *hold, then* $\lceil\Gamma\rceil \vdash \lceil G\rceil[x_\phi \leftarrow M] : \lceil\sigma\rceil$ *holds.*

Notice that Δ is absent in the last assertions. This is because Δ is the environment when typing ϕ (which becomes M) and is not used to type G or σ, but only ϕ and τ.

Lemma 23. *If* $M \rightsquigarrow_{\beta\iota} N$ *holds in* $\mathsf{F}^{\lambda}_{\iota}$*, then* $\lceil M\rceil \rightsquigarrow^{+} \lceil N\rceil$ *holds.*

One step is not translated to exactly one step because products η-expansion duplicates terms. Using some parallel reduction, we might have a step to step translation.

4.5 Completeness

We show that F_{ι} is complete *wrt* $\mathsf{F}^{\lambda}_{\iota}$ by defining a translation from $\mathsf{F}^{\lambda}_{\iota}$ expressions to F_{ι} expressions, that preserves coercion-erasure and typing—see figures 18 and 19 for the translation of terms and coercions, respectively. Types and environments remain the same. We use two translation judgments: one for terms $\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau$ where the left side is a judgment in $\mathsf{F}^{\lambda}_{\iota}$ and the right side is a judgment in F_{ι}; and one for coercions $\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \forall\Delta.\, \tau \triangleright \sigma$. We write $\forall\Delta.\, \tau$ and $\lambda\Delta\ \tau$, and $W\ \Delta$ in F_{ι} for the folding of universal quantification, λ-abstraction and application over Δ, defined as follows:

$$\begin{array}{rr} \forall\emptyset.\, \tau = \tau & \lambda\emptyset\ \tau = \tau \\ \forall(\alpha, \Delta).\, \tau = \forall\alpha.\, \forall\Delta.\, \tau & \lambda(\alpha, \Delta)\ \tau = \lambda\alpha\ \lambda\Delta\ \tau \\ \forall((c : \varphi), \Delta).\, \tau = \varphi \Rightarrow \forall\Delta.\, \tau & \lambda((c : \varphi), \Delta)\ \tau = \lambda(c : \varphi)\ \lambda\Delta\ \tau \end{array}$$

$$\begin{array}{r} W\ \emptyset = W \\ W\ (\alpha, \Delta) = (W\ \alpha)\ \Delta \\ W\ ((c : \varphi), \Delta) = (W\{c\})\ \Delta \end{array}$$

We also omit type annotations when they can easily be rebuilt from context, in particular for reflexivity $\Diamond$ and arrow congruence $G_1 \to G_2$. We use two helper functions $\mathsf{Dist}^{\forall\Delta.}_{\tau\to\sigma}$ and $\mathsf{Dist}^{\forall\Delta.}_{(\tau*\sigma)}$ to build sequences of distributivity coercions from a coercion type environment Δ, defined as follows:

$$\begin{array}{rl} \mathsf{Dist}^{\forall\emptyset.}_{\tau\to\sigma} & = \Diamond^{\tau\to\sigma} \\ \mathsf{Dist}^{\forall(\alpha,\Delta).}_{\tau\to\sigma} & = \mathsf{Dist}^{\forall\alpha.}_{\tau\to\forall\Delta.\,\sigma}\langle\lambda\alpha\ \mathsf{Dist}^{\forall\Delta.}_{\tau\to\sigma}\langle\Diamond\ \alpha\rangle\rangle \\ \mathsf{Dist}^{\forall((c:\varphi),\Delta).}_{\tau\to\sigma} & = \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\forall\Delta.\,\sigma}\langle\lambda(c : \varphi)\ \mathsf{Dist}^{\forall\Delta.}_{\tau\to\sigma}\langle\Diamond\{c\}\rangle\rangle \end{array}$$

$$
\begin{aligned}
\lceil \alpha \rceil &= \alpha \\
\lceil \tau \to \sigma \rceil &= \lceil \tau \rceil \to \lceil \sigma \rceil \\
\lceil (\tau * \sigma) \rceil &= (\lceil \tau \rceil * \lceil \sigma \rceil) \\
\lceil \forall \alpha . \tau \rceil &= \forall \alpha . \lceil \tau \rceil \\
\lceil (\tau \triangleright \sigma) \Rightarrow \rho \rceil &= (\lceil \tau \rceil \to \lceil \sigma \rceil) \to \lceil \rho \rceil \\
\lceil \top \rceil &= \forall \alpha .\ (\forall \beta .\ \beta \to \alpha) \to \alpha
\end{aligned}
$$

$$
\begin{aligned}
\lceil x \rceil &= x \\
\lceil \lambda (x : \tau)\, M \rceil &= \lambda (x : \lceil \tau \rceil)\ \lceil M \rceil \\
\lceil M\, N \rceil &= \lceil M \rceil\ \lceil N \rceil \\
\lceil (M, N) \rceil &= (\lceil M \rceil, \lceil N \rceil) \\
\lceil M.1 \rceil &= \lceil M \rceil .1 \\
\lceil M.2 \rceil &= \lceil M \rceil .2
\end{aligned}
\qquad\qquad
\begin{aligned}
\lceil \phi \rceil &= x_\phi \\
\lceil \lambda (\phi : \tau)\, G \rceil &= \lambda (x_\phi : \lceil \tau \rceil)\ \lceil G \rceil \\
\lceil H\, W \rceil &= \lceil H \rceil\ \lceil W \rceil \\
\lceil \lambda \alpha\, W \rceil &= \lambda \alpha\ \lceil W \rceil \\
\lceil W\, \tau \rceil &= \lceil W \rceil\ \lceil \tau \rceil
\end{aligned}
$$

$$
\begin{aligned}
\lceil c \rceil &= x_c \\
\lceil \lambda (c : \tau \triangleright \sigma)\, W \rceil &= \lambda (x_c : \lceil \tau \rceil \to \lceil \sigma \rceil)\ \lceil W \rceil \\
\lceil W\, H \rceil &= \lceil W \rceil\ \lceil H \rceil \\
\lceil \mathsf{Top}^\tau\, W \rceil &= \lambda \alpha\ \lambda (x : \forall \beta .\ \beta \to \alpha)\ x\, \lceil \tau \rceil\ \lceil W \rceil \\
\lceil \lambda (\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow W\, G_1\} \rceil &= \lceil \lambda (x : \tau)\, G_2 [\phi_2 \leftarrow W\, G_1 [\phi_1 \leftarrow x]] \rceil \\
\lceil (G_1, G_2) \{\phi_1, \phi_2 \leftarrow W\} \rceil &= \lceil (G_1 [\phi_1 \leftarrow W], G_2 [\phi_2 \leftarrow W]) \rceil
\end{aligned}
$$

Figure 17: System F_ι^λ: reification

and

$$
\begin{aligned}
\mathsf{Dist}^{\forall \emptyset .}_{(\tau * \sigma)} &= \Diamond^{(\tau * \sigma)} \\
\mathsf{Dist}^{\forall (\alpha, \Delta) .}_{(\tau * \sigma)} &= \mathsf{Dist}^{\forall \alpha .}_{(\forall \Delta .\ \tau * \forall \Delta .\ \sigma)} \langle \lambda \alpha\ \mathsf{Dist}^{\forall \Delta .}_{(\tau * \sigma)} \langle \Diamond\ \alpha \rangle \rangle \\
\mathsf{Dist}^{\forall ((c : \varphi), \Delta) .}_{(\tau * \sigma)} &= \mathsf{Dist}^{\varphi \Rightarrow}_{(\forall \Delta .\ \tau * \forall \Delta .\ \sigma)} \langle \lambda (c : \varphi)\ \mathsf{Dist}^{\forall \Delta .}_{(\tau * \sigma)} \langle \Diamond \{c\} \rangle \rangle
\end{aligned}
$$

All the rules but CTermEtaArr, CCoerEtaArr, CTermEtaProd, and CCoerEtaProd are quite easy to understand and retype. We prove that CTermEtaArr and CCoerEtaArr produce a well-typed F_ι term and coercion. The other cases are similar.

Lemma 24. *Assume that* $\Gamma \vdash \tau$ *and* $\Gamma \vdash G_2 : \forall \Delta .\, \sigma' \triangleright \sigma$ *and* $\Gamma, \Delta \vdash M : \tau' \to \sigma'$ *and* $\Gamma, \Delta \vdash G_1 : \forall \Delta' .\, \tau \triangleright \tau'$ *hold. Then,* $\Gamma \vdash (\Diamond \to G_2) \langle \mathit{Dist}^{\forall \Delta .}_{\tau \to \sigma'} \langle \lambda \Delta\ ((G_1 \langle \lambda \Delta'\ \Diamond \rangle) \to \Diamond) \langle M \rangle \rangle \rangle : \tau \to \sigma$ *also holds.*

(Proof p. 55)

Lemma 25. *Assume that* $\Gamma \vdash G_2 : \forall \Delta .\, \sigma' \triangleright \sigma$ *and* $\Gamma, \Delta \vdash G : \forall \Delta'' .\, \rho \triangleright \tau' \to \sigma'$ *and* $\Gamma, \Delta \vdash G_1 : \forall \Delta' .\, \tau \triangleright \tau'$ *hold. Then* $\Gamma \vdash (\Diamond \to G_2) \langle \mathit{Dist}^{\forall \Delta .}_{\tau \to \sigma'} \langle \lambda \Delta\ ((G_1 \langle \lambda \Delta'\ \Diamond \rangle) \to \Diamond) \langle G [\diamond \leftarrow \Diamond\ \Delta] \rangle \rangle \rangle : \forall (\Delta, \Delta'') .\, \rho \triangleright \tau \to \sigma$ *also holds.*

(Proof p. 55)

Remark The proofs of both lemmas rely on the fact that the notation $\mathsf{Dist}^{\forall \Delta .}_{\tau \to \sigma}$ is total on Δ. This is indeed the case in F_ι. However, this means that if we were to extend F_ι with an erasable binder, we should add a distributivity coercion between this binder and any other type construct (arrow and product until now) in order to preserve completeness. Without this guideline, it would be quite easy to propose extensions of the language where completeness is lost.

4.6 Soundness

We show that F_ι is sound *wrt* F_ι^λ by defining a translation from F_ι expressions to F_ι^λ expressions that preserves typings and coercion-erasure—see figures 20, 21 and 22 for the translation of terms and coercions, respectively.

$$\textsc{CTermTermVar}\ \frac{\Gamma \vdash ok \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow \Gamma \vdash x : \tau} \qquad \textsc{CTermTermLam}\ \frac{\Gamma, (x:\tau) \vdash M : \sigma \rightsquigarrow \Gamma, (x:\tau) \vdash \hat{M} : \sigma}{\Gamma \vdash \lambda(x:\tau)\ M : \tau \to \sigma \rightsquigarrow \Gamma \vdash \lambda(x:\tau)\ \hat{M} : \tau \to \sigma}$$

$$\textsc{CTermTermApp}\ \frac{\Gamma \vdash M : \tau \to \sigma \rightsquigarrow \Gamma \vdash \hat{M} : \tau \to \sigma \qquad \Gamma \vdash N : \tau \rightsquigarrow \Gamma \vdash \hat{N} : \tau}{\Gamma \vdash M\ N : \sigma \rightsquigarrow \Gamma \vdash \hat{M}\ \hat{N} : \sigma} \qquad \textsc{CTermTermPair}\ \frac{\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau \qquad \Gamma \vdash N : \sigma \rightsquigarrow \Gamma \vdash \hat{N} : \sigma}{\Gamma \vdash (M, N) : (\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{M}, \hat{N}) : (\tau * \sigma)}$$

$$\textsc{CTermTermFst}\ \frac{\Gamma \vdash M : (\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M} : (\tau * \sigma)}{\Gamma \vdash M.1 : \tau \rightsquigarrow \Gamma \vdash \hat{M}.1 : \tau} \qquad \textsc{CTermTermSnd}\ \frac{\Gamma \vdash M : (\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M} : (\tau * \sigma)}{\Gamma \vdash M.2 : \sigma \rightsquigarrow \Gamma \vdash \hat{M}.2 : \sigma}$$

$$\textsc{CTermHoleApp}\ \frac{\Gamma \vdash H : \tau\sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma \qquad \Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau}{\Gamma \vdash H\ M : \sigma \rightsquigarrow \Gamma \vdash \hat{H}\langle \hat{M} \rangle : \sigma} \qquad \textsc{CTermTypeLam}\ \frac{\Gamma, \alpha \vdash M : \tau \rightsquigarrow \Gamma, \alpha \vdash \hat{M} : \tau}{\Gamma \vdash \lambda\alpha\ M : \forall\alpha.\tau \rightsquigarrow \Gamma \vdash \lambda\alpha\ \hat{M} : \forall\alpha.\tau}$$

$$\textsc{CTermTypeApp}\ \frac{\Gamma \vdash M : \forall\alpha.\sigma \rightsquigarrow \Gamma \vdash \hat{M} : \forall\alpha.\sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash M\,\tau : \sigma[\alpha \leftarrow \tau] \rightsquigarrow \Gamma \vdash \hat{M}\,\tau : \sigma[\alpha \leftarrow \tau]} \qquad \textsc{CTermTop}\ \frac{\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau}{\Gamma \vdash \mathsf{Top}^{\tau}\ M : \top \rightsquigarrow \Gamma \vdash \mathsf{Top}^{\tau}\langle \hat{M} \rangle : \top}$$

$$\textsc{CTermEtaArr}\ \frac{\begin{array}{c}\Gamma \vdash \tau \\ \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall\Delta.\,\sigma' \triangleright \sigma \\ \Gamma, \Delta \vdash M : \tau' \to \sigma' \rightsquigarrow \Gamma, \Delta \vdash \hat{M} : \tau' \to \sigma' \\ \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \rightsquigarrow \Gamma, \Delta \vdash \hat{G}_1 : \forall\Delta'.\,\tau \triangleright \tau'\end{array}}{\begin{array}{l}\Gamma \vdash \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow M\, G_1\} : \tau \to \sigma \qquad \rightsquigarrow \\ \Gamma \vdash (\Diamond \to \hat{G}_2)\langle \mathsf{Dist}^{\forall\Delta.}_{\tau \to \sigma'} \langle (\lambda\Delta\ ((\hat{G}_1 \langle (\lambda\Delta'\ \Diamond)\rangle) \to \Diamond)\langle \hat{M} \rangle)\rangle\rangle : \tau \to \sigma\end{array}}$$

$$\textsc{CTermEtaProd}\ \frac{\begin{array}{c}\Gamma; \Delta \star (\phi_1 : \tau') \vdash G_1 : \tau \rightsquigarrow \Gamma \vdash \hat{G}_1 : \forall\Delta.\,\tau' \triangleright \tau \\ \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall\Delta.\,\sigma' \triangleright \sigma \\ \Gamma, \Delta \vdash M : (\tau' * \sigma') \rightsquigarrow \Gamma, \Delta \vdash \hat{M} : (\tau' * \sigma')\end{array}}{\Gamma \vdash (G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\} : (\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{G}_1 * \hat{G}_2)\langle \mathsf{Dist}^{\forall\Delta.\,(\tau' * \sigma')} \langle (\lambda\Delta\ \hat{M})\rangle\rangle : (\tau * \sigma)}$$

$$\textsc{CTermCoerLam}\ \frac{\Gamma, (c : \tau \triangleright \sigma) \vdash M : \varphi \rightsquigarrow \Gamma, (c : \tau \triangleright \sigma) \vdash \hat{M} : \varphi}{\Gamma \vdash \lambda(c : \tau \triangleright \sigma)\, M : (\tau \triangleright \sigma) \Rightarrow \varphi \rightsquigarrow \Gamma \vdash \lambda(c : \tau \triangleright \sigma)\, \hat{M} : (\tau \triangleright \sigma) \Rightarrow \varphi}$$

$$\textsc{CTermCoerApp}\ \frac{\Gamma \vdash M : (\tau \triangleright \sigma) \Rightarrow \varphi \rightsquigarrow \Gamma \vdash \hat{M} : (\tau \triangleright \sigma) \Rightarrow \varphi \qquad \Gamma \vdash H : \tau\sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma}{\Gamma \vdash M\, H : \varphi \rightsquigarrow \Gamma \vdash \hat{M}\{\hat{H}\} : \varphi}$$

Figure 18: System $\mathsf{F}^{\lambda}_{\iota}$: completeness for terms

$$\textsc{CCoerHoleVar}\ \frac{\Gamma \vdash \tau}{\Gamma; \emptyset \star (\phi : \tau) \vdash \phi : \tau \rightsquigarrow \Gamma \vdash \Diamond^{\tau} : \tau \triangleright \tau} \qquad \textsc{CCoerHoleLam}\ \frac{\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta.\, \tau \triangleright \sigma}{\Gamma \vdash \lambda(\phi : \tau)\, G : \tau\sigma \rightsquigarrow \Gamma \vdash \hat{G}\langle(\lambda \Delta\ \Diamond^{\tau})\rangle : \tau \triangleright \sigma}$$

$$\textsc{CCoerHoleApp}\ \frac{\Gamma \vdash H : \tau\sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma \qquad \Gamma; \Delta \star (\phi : \varphi) \vdash G : \tau \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta.\, \varphi \triangleright \tau}{\Gamma; \Delta \star (\phi : \varphi) \vdash H\, G : \sigma \rightsquigarrow \Gamma \vdash \hat{H}\langle \hat{G} \rangle : \forall \Delta.\, \varphi \triangleright \sigma}$$

$$\textsc{CCoerTypeLam}\ \frac{\Gamma, \alpha; \Delta \star (\phi : \varphi) \vdash G : \tau \rightsquigarrow \Gamma, \alpha \vdash \hat{G} : \forall \Delta.\, \varphi \triangleright \tau}{\Gamma; (\alpha, \Delta) \star (\phi : \varphi) \vdash \lambda \alpha\, G : \forall \alpha.\, \tau \rightsquigarrow \Gamma \vdash \lambda \alpha\, \hat{G}[\diamond \leftarrow \Diamond\, \alpha] : \forall (\alpha, \Delta).\, \varphi \triangleright \forall \alpha.\, \tau}$$

$$\textsc{CCoerTypeApp}\ \frac{\Gamma; \Delta \star (\phi : \varphi) \vdash G : \forall \alpha.\, \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta.\, \varphi \triangleright \forall \alpha.\, \sigma \qquad \Gamma \vdash \tau}{\Gamma; \Delta \star (\phi : \varphi) \vdash G\, \tau : \sigma[\alpha \leftarrow \tau] \rightsquigarrow \Gamma \vdash \hat{G}\, \tau : \forall \Delta.\, \varphi \triangleright \sigma[\alpha \leftarrow \tau]}$$

$$\textsc{CCoerTop}\ \frac{\Gamma; \Delta \star (\phi : \sigma) \vdash G : \tau \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta.\, \sigma \triangleright \tau}{\Gamma; \Delta \star (\phi : \sigma) \vdash \mathsf{Top}^{\tau}\, G : \top \rightsquigarrow \Gamma \vdash \mathsf{Top}^{\tau}\langle \hat{G} \rangle : \forall \Delta.\, \sigma \triangleright \top} \qquad \textsc{CCoerCoerVar}\ \frac{\vdash \Gamma \qquad \Gamma(c) = \tau \triangleright \sigma}{\Gamma \vdash c : \tau\sigma \rightsquigarrow \Gamma \vdash c : \tau \triangleright \sigma}$$

$$\textsc{CCoerCoerLam}\ \frac{\Gamma, (c : \tau \triangleright \sigma); \Delta \star (\phi : \rho') \vdash G : \varphi \rightsquigarrow \Gamma, (c : \tau \triangleright \sigma) \vdash \hat{G} : \forall \Delta.\, \rho' \triangleright \varphi}{\begin{array}{c}\Gamma; (c : \tau \triangleright \sigma), \Delta \star (\phi : \rho') \vdash \lambda(c : \tau \triangleright \sigma)\, G : (\tau \triangleright \sigma) \Rightarrow \varphi \qquad \rightsquigarrow \\ \Gamma \vdash \lambda(c : \tau \triangleright \sigma)\, \hat{G}[\diamond \leftarrow \Diamond\{c\}] : \forall ((c : \tau \triangleright \sigma), \Delta).\, \rho' \triangleright (\tau \triangleright \sigma) \Rightarrow \varphi\end{array}}$$

$$\textsc{CCoerCoerApp}\ \frac{\begin{array}{c}\Gamma; \Delta \star (\phi : \rho') \vdash G : (\tau \triangleright \sigma) \Rightarrow \varphi \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta.\, \rho' \triangleright (\tau \triangleright \sigma) \Rightarrow \varphi \\ \Gamma \vdash H : \tau\sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma\end{array}}{\Gamma; \Delta \star (\phi : \rho') \vdash G\, H : \varphi \rightsquigarrow \Gamma \vdash \hat{G}\{\hat{H}\} : \forall \Delta.\, \rho' \triangleright \varphi}$$

$$\textsc{CCoerEtaArr}\ \frac{\begin{array}{c}\Gamma \vdash \tau \\ \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta.\, \sigma' \triangleright \sigma \\ \Gamma, \Delta; \Delta'' \star (\phi : \varphi) \vdash G : \tau' \to \sigma' \rightsquigarrow \Gamma, \Delta \vdash \hat{G} : \forall \Delta''.\, \varphi \triangleright \tau' \to \sigma' \\ \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \rightsquigarrow \Gamma, \Delta \vdash \hat{G}_1 : \forall \Delta'.\, \tau \triangleright \tau'\end{array}}{\begin{array}{c}\Gamma; \Delta, \Delta'' \star (\phi : \varphi) \vdash \lambda(\phi_1 : \tau)\, G_2\, \{\phi_2 \leftarrow G\, G_1\} : \tau \to \sigma \qquad \rightsquigarrow \\ \Gamma \vdash (\Diamond \to \hat{G}_2)\langle \mathsf{Dist}^{\forall \Delta.}_{\tau \to \sigma'}\langle (\lambda \Delta\ ((\hat{G}_1\langle(\lambda \Delta'\ \Diamond)\rangle) \to \Diamond)\langle \hat{G}[\diamond \leftarrow \Diamond\ \Delta]\rangle)\rangle\rangle : \forall (\Delta, \Delta'').\, \varphi \triangleright \tau \to \sigma\end{array}}$$

$$\textsc{CExprEtaProd}\ \frac{\begin{array}{c}\Gamma; \Delta \star (\phi_1 : \tau') \vdash G_1 : \tau \rightsquigarrow \Gamma \vdash \hat{G}_1 : \forall \Delta.\, \tau' \triangleright \tau \\ \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta.\, \sigma' \triangleright \sigma \\ \Gamma, \Delta; \Delta' \star (\phi : \varphi) \vdash G : (\tau' * \sigma') \rightsquigarrow \Gamma, \Delta \vdash \hat{G} : \forall \Delta'.\, \varphi \triangleright (\tau' * \sigma')\end{array}}{\begin{array}{c}\Gamma; \Delta, \Delta' \star (\phi : \varphi) \vdash (G_1, G_2)\{\phi_1, \phi_2 \leftarrow G\} : (\tau * \sigma) \qquad \rightsquigarrow \\ \Gamma \vdash (\hat{G}_1 * \hat{G}_2)\langle \mathsf{Dist}^{\forall \Delta.\, (\tau' * \sigma')}\langle (\lambda \Delta\ \hat{G}[\diamond \leftarrow \Diamond\ \Delta])\rangle\rangle : \forall (\Delta, \Delta').\, \varphi \triangleright (\tau * \sigma)\end{array}}$$

Figure 19: System $\mathsf{F}^{\lambda}_{\iota}$: completeness for coercions

$$\textsc{sTermVar}\ \frac{\Gamma \vdash ok \qquad (x:\tau) \in \Gamma}{\Gamma \vdash x:\tau \rightsquigarrow \Gamma \vdash x:\tau} \qquad \textsc{sTermTermLam}\ \frac{\Gamma, (x:\tau) \vdash M:\sigma \rightsquigarrow \Gamma, x:\tau \vdash \hat{M}:\sigma}{\Gamma \vdash \lambda(x:\tau)\, M : \tau \to \sigma \rightsquigarrow \Gamma \vdash \lambda(x:\tau)\, \hat{M} : \tau \to \sigma}$$

$$\textsc{sTermTermApp}\ \frac{\Gamma \vdash M:\tau \to \sigma \rightsquigarrow \Gamma \vdash M:\tau \to \sigma \qquad \Gamma \vdash N:\tau \rightsquigarrow \Gamma \vdash N:\tau}{\Gamma \vdash M\ N:\sigma \rightsquigarrow \Gamma \vdash \hat{M}\,\hat{N}:\sigma} \qquad \textsc{sTermPair}\ \frac{\Gamma \vdash M:\tau \rightsquigarrow \Gamma \vdash \hat{M}:\tau \qquad \Gamma \vdash N:\sigma \rightsquigarrow \Gamma \vdash \hat{N}:\sigma}{\Gamma \vdash (M,N):(\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{M},\hat{N}):(\tau * \sigma)}$$

$$\textsc{sTermFirst}\ \frac{\Gamma \vdash M:(\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M}:(\tau * \sigma)}{\Gamma \vdash M.\mathsf{1}:\tau \rightsquigarrow \Gamma \vdash \hat{M}.\mathsf{1}:\tau} \qquad \textsc{sTermSecond}\ \frac{\Gamma \vdash M:(\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M}:(\tau * \sigma)}{\Gamma \vdash M.\mathsf{2}:\sigma \rightsquigarrow \Gamma \vdash \hat{M}.\mathsf{2}:\sigma}$$

$$\textsc{sTermTypeLam}\ \frac{\Gamma, \alpha \vdash M:\tau \rightsquigarrow \Gamma, \alpha \vdash \hat{M}:\tau}{\Gamma \vdash \lambda\alpha\, M : \forall\alpha.\tau \rightsquigarrow \Gamma \vdash \lambda\alpha\, \hat{M} : \forall\alpha.\tau} \qquad \textsc{sTermTypeApp}\ \frac{\Gamma \vdash M:\forall\alpha.\tau \rightsquigarrow \Gamma \vdash \hat{M}:\forall\alpha.\tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash M\,\sigma : \tau[\alpha \leftarrow \sigma] \rightsquigarrow \Gamma \vdash \hat{M}\,\sigma : \tau[\alpha \leftarrow \sigma]}$$

$$\textsc{sTermCoer}\ \frac{\Gamma \vdash G:\tau \triangleright \sigma \rightsquigarrow \Gamma \vdash \hat{G}:\tau\sigma \qquad \Gamma \vdash M:\tau \rightsquigarrow \Gamma \vdash \hat{M}:\tau}{\Gamma \vdash G\langle M\rangle : \sigma \rightsquigarrow \Gamma \vdash \hat{G}\,\hat{M}:\sigma} \qquad \textsc{sTermCoerLam}\ \frac{\Gamma, (c:\varphi \triangleright \rho') \vdash M:\sigma \rightsquigarrow \Gamma, c:\varphi\rho' \vdash \hat{M}:\sigma}{\Gamma \vdash \lambda(c:\varphi \triangleright \rho')\, M : (\varphi \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \lambda(c:\varphi)\, \rho'\hat{M} : \varphi \Rightarrow \rho'\sigma}$$

$$\textsc{sTermCoerApp}\ \frac{\Gamma \vdash G:\varphi \triangleright \rho' \rightsquigarrow \Gamma \vdash \hat{G}:\varphi\rho' \qquad \Gamma \vdash M:(\varphi \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \hat{M}:\varphi \Rightarrow \rho'\sigma}{\Gamma \vdash M\{G\}:\sigma \rightsquigarrow \Gamma \vdash \hat{M}\,\hat{G}:\sigma}$$

Figure 20: System $\mathsf{F}^{\lambda}_{\iota}$: soundness for terms

$$\textsc{SCoerTrans}\quad \frac{\Gamma \vdash G_2 : \sigma \triangleright \varphi \rightsquigarrow \Gamma \vdash \hat{G}_2 : \sigma\varphi \qquad \Gamma \vdash G_1 : \tau \triangleright \sigma \rightsquigarrow \Gamma \vdash \hat{G}_1 : \tau\sigma}{\Gamma \vdash G_2\langle G_1\rangle : \tau \triangleright \varphi \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \hat{G}_2\, (\hat{G}_1\, \phi) : \tau\varphi}$$

$$\textsc{SCoerTypeLam}\quad \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash G : \tau \triangleright \sigma \rightsquigarrow \Gamma, \alpha \vdash \hat{G} : \tau\sigma}{\Gamma \vdash \lambda\alpha\, G : \tau \triangleright \forall\alpha.\, \sigma \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \lambda\alpha\, \hat{G}\, \phi : \tau \forall\alpha.\, \sigma}$$

$$\textsc{SCoerTypeApp}\quad \frac{\Gamma \vdash \varphi \qquad \Gamma \vdash G : \tau \triangleright \forall\alpha.\, \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \tau \forall\alpha.\sigma}{\Gamma \vdash G\, \varphi : \tau \triangleright \sigma[\alpha \leftarrow \varphi] \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \hat{G}\, \phi\, \varphi : \tau\sigma[\alpha \leftarrow \varphi]}$$

$$\textsc{SCoerDot}\quad \frac{\Gamma \vdash \tau}{\Gamma \vdash \Diamond^\tau : \tau \triangleright \tau \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \phi : \tau\tau}$$

$$\textsc{SCoerForget}\quad \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathtt{Top}^\tau : \tau \triangleright \top \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \mathtt{Top}^\tau\, \phi : \tau\top}$$

$$\textsc{SCoerVar}\quad \frac{\Gamma \vdash ok \qquad (c : \tau \triangleright \sigma) \in \Gamma}{\Gamma \vdash c : \tau \triangleright \sigma \rightsquigarrow \Gamma \vdash c : \tau\sigma}$$

$$\textsc{SCoerCoerLam}\quad \frac{\Gamma, (c : \rho \triangleright \rho') \vdash G : \tau \triangleright \sigma \rightsquigarrow \Gamma, c : \varphi\rho' \vdash \hat{G} : \tau\sigma}{\Gamma \vdash \lambda(c : \rho \triangleright \rho')\, G : \tau \triangleright (\rho \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \lambda(c : \varphi)\, \rho'\hat{G}\, \phi : \tau(\rho \triangleright \rho') \Rightarrow \sigma}$$

$$\textsc{SCoerCoerApp}\quad \frac{\Gamma \vdash G' : \rho \triangleright \rho' \rightsquigarrow \Gamma \vdash \hat{G}' : \varphi\rho' \qquad \Gamma \vdash G : \tau \triangleright (\rho \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \tau(\rho \triangleright \rho') \Rightarrow \sigma}{\Gamma \vdash G\{G'\} : \tau \triangleright \sigma \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau)\, \hat{G}\, \phi\, \hat{G}' : \tau\sigma}$$

Figure 21: System F_ι^λ: soundness for coercions 1/2

4.7 Bisimulation between F_ι and F_ι^λ

Of course, these two translations (soundness and completeness) preserve typing and coercion erasure. But they also preserve wedging configurations. This means that a wedging configuration is translated to a wedging configuration and reciprocally, only wedging configurations are translated to wedging configurations.

Lemma 26. *For M in F_ι in ι-normal form, if the ι-normal form M' of its translation $\hat{M}$ β-reduces to N', then $\hat{M}$ β-reduces to N which is ι-equivalent to N'.*

(Proof p. 55)

Lemma 27. *For M_0 in F_ι in ι-normal form, if the ι-normal form of its translation $\hat{M}_0$ β-reduces to N_1, then there is a M_2 such that $M_0 \rightsquigarrow_\iota^\star \rightsquigarrow_\beta M_2$ and $\hat{M}_2$ is ι-equivalent to M_2.*

(Proof p. 55)

5 Parametric F_ι

Parametric F_ι, written F_ι^p, restricts the language so as to rule out wedge configurations by means of typechecking. The restriction is on the type φ of coercion abstractions $\lambda(c : \varphi)\ M$, *i.e.* on the type of coercion variables. Observe that a coercion variable appearing in a wedge position $c\langle\lambda(x : \tau)\ M\rangle\ N$ has a coercion type $\sigma \triangleright \rho$ where σ and ρ are both arrow types. To prevent this situation from happening in F_ι^p, we require that either the domain or the codomain of the type of a coercion parameter be a variable. Hence, we only allow $\lambda(c : \alpha \triangleright \rho)\ M$ or $\lambda(c : \sigma \triangleright \alpha)\ M$.

In order to preserve this invariant by reduction, we must request the type variable to be introduced simultaneously. So, we may write $\lambda\alpha\ \lambda(c : \alpha \triangleright \tau)\ M$ but not $\lambda(c : \alpha \triangleright \tau)\ M$ alone. This is a form of parametricity since either the domain or the codomain of c must be treated abstractly

SCoerArrow

$$\frac{\Gamma \vdash G_1 : \tau \triangleright \tau' \rightsquigarrow \Gamma \vdash \hat{G}_1 : \tau\tau' \qquad \Gamma \vdash G_2 : \sigma \triangleright \sigma' \rightsquigarrow \Gamma \vdash \hat{G}_2 : \sigma\sigma'}{\begin{array}{c}\Gamma \vdash G_1 \xrightarrow{\tau} G_2 : \tau' \to \sigma \triangleright \tau \to \sigma' \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : \tau' \to \sigma)\, \lambda(\phi_1 : \tau)\, (\hat{G}_2\, \phi_2) \left\{\phi_2 \leftarrow \phi\, (\hat{G}_1\, \phi_1)\right\} : \tau' \to \sigma\tau \to \sigma'\end{array}}$$

SCoerTransDistTypeArrow

$$\frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\begin{array}{c}\Gamma \vdash \mathsf{Dist}^{\forall\alpha.}_{\tau \to \sigma} : \forall\alpha.\, (\tau \to \sigma) \triangleright \tau \to \forall\alpha.\, \sigma \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : \forall\alpha.\, (\tau \to \sigma))\, \lambda(\phi_1 : \tau)\, (\lambda\alpha\, \phi_2)\, \{\phi_2 \leftarrow (\phi\, \alpha)\, \phi_1\} : \forall\alpha.\, (\tau \to \sigma)\tau \to \forall\alpha.\, \sigma\end{array}}$$

SCoerProd

$$\frac{\Gamma \vdash G_1 : \tau \triangleright \tau' \rightsquigarrow \Gamma \vdash \hat{G}_1 : \tau\tau' \qquad \Gamma \vdash G_2 : \sigma \triangleright \sigma' \rightsquigarrow \Gamma \vdash \hat{G}_2 : \sigma\sigma'}{\begin{array}{c}\Gamma \vdash (G_1 * G_2) : (\tau * \sigma) \triangleright (\tau' * \sigma') \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : (\tau * \sigma))\, (\hat{G}_1\, \phi_1, \hat{G}_2\, \phi_2)\{\phi_1, \phi_2 \leftarrow \phi\} : (\tau * \sigma)(\tau' * \sigma')\end{array}}$$

SCoerDistTypeProd

$$\frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\begin{array}{c}\Gamma \vdash \mathsf{Dist}^{\forall\alpha.}_{(\tau * \sigma)} : \forall\alpha.\, (\tau * \sigma) \triangleright (\forall\alpha.\, \tau * \forall\alpha.\, \sigma) \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : \forall\alpha.\, (\tau * \sigma))\, (\lambda\alpha\, \phi_1, \lambda\alpha\, \phi_2)\{\phi_1, \phi_2 \leftarrow \phi\, \alpha\} : \forall\alpha.\, (\tau * \sigma)(\forall\alpha.\, \tau * \forall\alpha.\, \sigma)\end{array}}$$

SCoerDistCoerArrow

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \varphi \qquad \Gamma \vdash \rho' \qquad \Gamma \vdash \sigma}{\begin{array}{c}\Gamma \vdash \mathsf{Dist}^{\varphi \triangleright \rho' \Rightarrow}_{\tau \to \sigma} : ((\varphi \triangleright \rho') \Rightarrow (\tau \to \sigma)) \;\triangleright\; (\tau \to (\varphi \triangleright \rho') \Rightarrow \sigma) \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : (\varphi \triangleright \rho') \Rightarrow (\tau \to \sigma))\, \lambda(\phi_1 : \tau)\, \lambda(c : \varphi)\, \rho'\phi_2\, \{\phi_2 \leftarrow (\phi\, c)\, \phi_1\} : \\ ((\varphi \triangleright \rho') \Rightarrow (\tau \to \sigma)) \;\triangleright\; (\tau \to (\varphi \triangleright \rho') \Rightarrow \sigma)\end{array}}$$

SCoerDistCoerProd

$$\frac{\Gamma \vdash \varphi \qquad \Gamma \vdash \rho' \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\begin{array}{c}\Gamma \vdash \mathsf{Dist}^{\varphi \triangleright \rho' \Rightarrow}_{(\tau * \sigma)} : ((\varphi \triangleright \rho') \Rightarrow (\tau * \sigma)) \;\triangleright\; (((\varphi \triangleright \rho') \Rightarrow \tau) * ((\varphi \triangleright \rho') \Rightarrow \sigma)) \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\phi : (\varphi \triangleright \rho') \Rightarrow (\tau * \sigma))\, (\lambda(c : \varphi)\, \rho'\phi_1, \lambda(c : \varphi)\, \rho'\phi_2)\{\phi_1, \phi_2 \leftarrow \phi\, c\} : \\ ((\varphi \triangleright \rho') \Rightarrow (\tau * \sigma)) \;\triangleright\; (((\varphi \triangleright \rho') \Rightarrow \tau) * ((\varphi \triangleright \rho') \Rightarrow \sigma))\end{array}}$$

Figure 22: System $\mathsf{F}^{\lambda}_{\iota}$: soundness for coercions 2/2

$$
\begin{array}{rcll}
\Phi & ::= & \lhd \mid \rhd & \text{bounds} \\
\tau & ::= & \ldots \not| \ \varphi \Rightarrow \tau \mid \forall(\alpha \Phi \tau) \Rightarrow \tau & \text{types} \\
M & ::= & \ldots \not| \ \lambda(c:\varphi)\, M \mid \lambda(\alpha \Phi c:\tau)\, M & \text{expressions} \\
 & & \not| \ M\{G\} \mid M\{\tau \Phi G\} & \\
G & ::= & \ldots \not| \ \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau} \mid \mathsf{Dist}^{\forall\alpha\Phi\tau\Rightarrow}_{\tau\to\tau} & \text{coercions} \\
 & & \not| \ \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)} \mid \mathsf{Dist}^{\forall\alpha\Phi\tau\Rightarrow}_{(\tau*\tau)} & \\
 & & \not| \ \lambda(c:\varphi)\, G \mid \lambda(\alpha \Phi c:\tau)\, G & \\
 & & \not| \ G\{G\} \mid G\{\tau \Phi G\} & \\
\Gamma & ::= & \ldots \not| \ \Gamma, c:\varphi \mid \Gamma, \alpha \Phi c:\tau & \text{environments}
\end{array}
$$

Figure 23: Parametric F_ι: syntax restriction *wrt* F_ι

$$\textsc{TermTCoerLam}\ \frac{\Gamma, \alpha \Phi c:\tau \vdash M:\sigma}{\Gamma \vdash \lambda(\alpha \Phi c:\tau)\, M : \forall(\alpha \Phi \tau) \Rightarrow \sigma} \qquad \textsc{TermTCoerApp}\ \frac{\Gamma \vdash M : \forall(\alpha \Phi \tau) \Rightarrow \tau' \qquad \Gamma \vdash G : \sigma \Phi \tau[\alpha \leftarrow \sigma]}{\Gamma \vdash M\{\sigma \Phi G\} : \tau'[\alpha \leftarrow \sigma]}$$

$$\textsc{TCoerVar}\ \frac{\Gamma \vdash ok \qquad \alpha \Phi c:\tau \in \Gamma}{\Gamma \vdash c : \alpha \Phi \tau} \qquad \textsc{CoerTCoerLam}\ \frac{\Gamma, \alpha \Phi c:\tau \vdash G : \sigma \rhd \sigma' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \lambda(\alpha \Phi c:\tau)\, G : \sigma \rhd \forall(\alpha \Phi \tau) \Rightarrow \sigma'}$$

$$\textsc{CoerTCoerApp}\ \frac{\Gamma \vdash G' : \rho \rhd \forall(\alpha \Phi \tau) \Rightarrow \tau' \qquad \Gamma \vdash G : \sigma \Phi \tau[\alpha \leftarrow \sigma]}{\Gamma \vdash G'\{\sigma \Phi G\} : \rho \rhd \tau'[\alpha \leftarrow \sigma]}$$

$$\textsc{CoerDistTCoerArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\forall\alpha\Phi\rho\Rightarrow}_{\tau\to\sigma} : (\forall(\alpha \Phi \rho) \Rightarrow \tau \to \sigma) \rhd (\tau \to \forall(\alpha \Phi \rho) \Rightarrow \sigma)}$$

$$\textsc{CoerDistTCoerProd}\ \frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\forall\alpha\Phi\rho\Rightarrow}_{(\tau*\sigma)} : \forall(\alpha \Phi \rho) \Rightarrow (\tau * \sigma) \rhd (\forall(\alpha \Phi \rho) \Rightarrow \tau * \forall(\alpha \Phi \rho) \Rightarrow \sigma)}$$

Figure 24: Parametric F_ι: typing rules *wrt* F_ι

(and thus not as an arrow type) in M. To enforce this restriction we stick a type abstraction to every coercion abstraction and see $\lambda\alpha\ \lambda(c:\alpha \rhd \tau)\ M$ as a single syntactic node, which we write $\lambda(\alpha \rhd c:\tau)\, M$ to avoid confusion. Although, we modify the syntax of source terms, F^p_ι can still be understood as a syntactic restriction of F_ι.

5.1 Syntax changes

The syntax of Parametric F_ι is defined on Figure 23 as a patch to the syntax of F_ι (we write $\not|$ for removal of a previous grammar form). We replace coercion abstraction $\lambda(c:\tau \rhd \sigma)\, M$ of F_ι by two new constructs $\lambda(\alpha \rhd c:\tau)\, M$ and $\lambda(\alpha \lhd c:\tau)\, M$ to mean $\lambda\alpha\ \lambda(c:\alpha \rhd \tau)\, M$ and $\lambda\alpha\ \lambda(c:\tau \rhd \alpha)\, M$ but atomically. For conciseness, we introduce a mode Φ that ranges over $\rhd$ and $\lhd$. Hence, we write $\lambda(\alpha \Phi c:\tau)\, M$ for either $\lambda(\alpha \rhd c:\tau)\, M$ or $\lambda(\alpha \lhd c:\tau)\, M$. Note that the type variable α is bounded in both τ and M. As a mnemonic device, we can read the type of the coercion variable by moving "$c:$" in front, *i.e.* $\alpha \rhd c:\tau$ becomes $c:\alpha \rhd \tau$ while $\alpha \lhd c:\tau$ becomes $c:\alpha \lhd \tau$ which can also be read $c:\tau \rhd \alpha$. The reason to keep the type variable α before the coercion variable is to preserve the order of the abstractions in F_ι.

We say that $\lambda(\alpha \rhd c:\tau)\, M$ and $\lambda(\alpha \lhd c:\tau)\, M$ are *negative* and *positive* coercion abstractions, respectively. The positive form is parametric on the codomain of the coercion and implements a lower bounded quantification $\tau \rhd \alpha$, as in xMLF. The negative form is parametric on the domain of the coercion and implements an upper bounded quantification $\alpha \rhd \tau$, as in $\mathsf{F}_{<:}$.

$$
\begin{array}{rcll}
p & ::= & \ldots \not\downarrow p\{G\} \mid p\{\tau \diamond G\} & \text{prevalues} \\
& & \not\downarrow \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}\langle p\rangle \mid \mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{\tau\to\tau}\langle p\rangle & \\
& & \not\downarrow \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}\langle \lambda(c:\varphi)\ p\rangle \mid \mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{\tau\to\tau}\langle \lambda(\alpha\diamond c:\tau)\ p\rangle & \\
& & \not\downarrow \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}\langle p\rangle \mid \mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{(\tau*\tau)}\langle p\rangle & \\
& & \not\downarrow \mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}\langle \lambda(c:\varphi)\ p\rangle \mid \mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{(\tau*\tau)}\langle \lambda(\alpha\diamond c:\tau)\ p\rangle & \\
v & ::= & \ldots \not\downarrow \lambda(c:\varphi)\ v \mid \lambda(\alpha\diamond c:\tau)\ v & \text{values} \\
P & ::= & \ldots \not\downarrow \lambda(c:\varphi)\ [] \mid \lambda(\alpha\diamond c:\tau)\ [] \not\downarrow []\{G\} \mid []\{\tau\diamond G\} & \text{retyping contexts}
\end{array}
$$

Figure 25: Parametric F_ι: changes in values *wrt* F_ι

$$
\textsc{TCoerArrow}\ \frac{\Gamma,\alpha\vdash\tau \qquad \Gamma,\alpha\vdash\sigma}{\Gamma\vdash\forall(\alpha\diamond\tau)\Rightarrow\sigma}
\qquad\qquad
\textsc{EnvCoer}\ \frac{\Gamma,\alpha\vdash\tau \qquad \alpha,c\notin\mathsf{dom}(\Gamma)}{\Gamma,\alpha\diamond c:\tau\vdash ok}
$$

Figure 26: Parametric F_ι: well-formedness judgments *wrt* F_ι

Continuing with the definition of F^p_ι, we replace coercion application $M\{G\}$ by $M\{\tau\diamond G\}$ to perform type and coercion applications $(M\,\tau)\{G\}$ atomically. Both positive and negative versions have the same meaning, but different typing rules. Type τ appears before G to remind that the type application is performed before the coercion application in the expanded form. As a mnemonic device, the $\diamond$ is oriented towards the side of the variable it instantiates in the coercion type of M. Hence, if M is $\lambda(\alpha\triangleright c:\sigma)\,N$, we must write $M\{\tau\triangleright G\}$.

We must change types accordingly, replacing coercion types $\varphi\Rightarrow\tau$ by $\forall(\alpha\diamond\tau)\Rightarrow\sigma$, which factors the two forms $\forall(\alpha\triangleright\tau)\Rightarrow\sigma$ and $\forall(\alpha\triangleleft\tau)\Rightarrow\sigma$ whose expansions in F_ι are $\forall\alpha.\,(\alpha\triangleright\tau)\Rightarrow\sigma$ and $\forall\alpha.\,(\tau\triangleright\alpha)\Rightarrow\sigma$, respectively. Typing environments are modified accordingly. Notice that $\Gamma,\alpha\diamond c:\tau$ stands for $\Gamma,\alpha,c:\alpha\diamond\tau$ ($c:\alpha\triangleleft\tau$ should be read as $c:\tau\triangleright\alpha$) and therefore α may appear free in τ—as for coercion abstractions: this allows the encoding of "recursively defined bounds" discussed below.

In the syntax of coercions, we replace coercion abstractions and coercion applications as we did for expressions. We also change the distributivity coercion that exchanges term abstraction with coercion abstraction to reflect the change in coercion types: it must simultaneously permute the term abstraction with the type abstraction and coercion abstraction that are stuck together. The same modifications are also done for product type.

5.2 Adjustments to the semantics

The syntactic changes imply corresponding adjustments to the semantics of the language. Notice that all restrictions are captured syntactically, so no further restriction of typing rules is necessary.

Typing rules Consistently with the change of syntax, we replace the typing rules TermCoerLam and TermCoerApp by rules TermTCoerLam and TermTCoerApp given on Figure 24. The corresponding typing rules CoerCoerLam and CoerCoerApp for coercions are changed similarly. We also replace CoerVar by TCoerVar. Finally, the modified distributivity coercions are typed as described by rules CoerDistTCoerArrow and CoerDistTCoerProd. Notice that $\diamond$ is a meta-variable as M or τ and different occurrences of the same meta-variable can only be instantiated simultaneously all by $\triangleright$ or all by $\triangleleft$. (We use different meta-variables $\diamond_1$ and $\diamond_2$ when we mean to instantiate them independently.)

The new typing rules for F^p_ι are derived from the typing rules of the corresponding nodes in F_ι. For example, TermTCoerLam is just the combination of rules TermCoerLam and TermTypeLam in F_ι.

Well-formedness judgments are adjusted in the obvious way, as described on figures 26.

$$\textsc{RedTCoer}\quad (\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\, M)\{\sigma \mathrel{\triangleleft\!\triangleright} G\} \rightsquigarrow_\iota M[\alpha \leftarrow \sigma][c \leftarrow G]$$

$$\textsc{RedCoerDistTCoerArrow}\quad \mathsf{Dist}^{\forall\alpha\triangleleft\!\triangleright\sigma_1\Rightarrow}_{\sigma_2\to\sigma_3}\langle\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\,\lambda(x : \sigma)\; M\rangle \rightsquigarrow_\iota \lambda(x : \sigma)\;\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\, M$$

$$\textsc{RedCoerDistTCoerProd}\quad \mathsf{Dist}^{\forall\alpha\triangleleft\!\triangleright\sigma_1\Rightarrow}_{(\sigma_2 * \sigma_3)}\langle\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\,(M, N)\rangle \rightsquigarrow_\iota (\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\, M, \lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\, N)$$

Figure 27: Parametric F_ι: new reduction rules *wrt* F_ι

$$\begin{aligned}
(\lambda(\alpha \mathrel{\triangleleft\!\triangleright} c : \tau)\, W)^\circ &= \lambda\alpha\;\lambda(c : \alpha \mathrel{\triangleleft\!\triangleright} \tau^\circ)\; W^\circ\\
(W\{\tau \mathrel{\triangleleft\!\triangleright} G\})^\circ &= (W^\circ\,\tau^\circ)\{G^\circ\}\\
(\mathsf{Dist}^{\forall\alpha\triangleleft\!\triangleright\rho\Rightarrow}_{\tau\to\sigma})^\circ &= \mathsf{Dist}^{\forall\alpha.}_{\tau^\circ\to(\alpha\triangleleft\!\triangleright\rho^\circ)\Rightarrow\sigma^\circ}\langle\lambda\alpha\;\mathsf{Dist}^{\alpha\triangleleft\!\triangleright\rho^\circ\Rightarrow}_{\tau^\circ\to\sigma^\circ}\langle\Diamond\,\alpha\rangle\rangle\\
(\mathsf{Dist}^{\forall\alpha\triangleleft\!\triangleright\rho\Rightarrow}_{(\tau*\sigma)})^\circ &= \mathsf{Dist}^{\forall\alpha.}_{(\alpha\triangleleft\!\triangleright\rho^\circ\Rightarrow\tau^\circ*\alpha\triangleleft\!\triangleright\rho^\circ\Rightarrow\sigma^\circ)}\langle\lambda\alpha\;\mathsf{Dist}^{\alpha\triangleleft\!\triangleright\rho^\circ\Rightarrow}_{(\tau^\circ*\sigma^\circ)}\langle\Diamond\,\alpha\rangle\rangle
\end{aligned}$$

$$\begin{aligned}
\tau \triangleright \sigma^\circ &= \tau^\circ \triangleright \sigma^\circ\\
\tau \triangleleft \sigma^\circ &= \sigma^\circ \triangleright \tau^\circ\\
(\forall(\alpha \mathrel{\triangleleft\!\triangleright} \tau) \Rightarrow \sigma)^\circ &= \forall\alpha.\,(\alpha \mathrel{\triangleleft\!\triangleright} \tau^\circ) \Rightarrow \sigma^\circ\\
(\Gamma, \alpha \mathrel{\triangleleft\!\triangleright} c : \tau)^\circ &= \Gamma, \alpha, c : \alpha \mathrel{\triangleleft\!\triangleright} \tau^\circ
\end{aligned}$$

Figure 28: Translation of F^p_ι into F_ι

Operational semantics The operational semantics is modified in the obvious way. The syntax of values for F^p_ι is defined on Figure 25 as a modification of the syntax of F_ι. The adjustments in the reduction rules are the replacement of RedCoer by RedTCoer, RedCoerDistCoerArrow by RedCoerDistTCoerArrow, RedCoerDistCoerProd by RedCoerDistTCoerProd, and the change of retyping contexts that induces a change in RedCoerFill as described in Figure 25.

5.3 Properties

Since F^p_ι can be seen as a restriction of F_ι where coercion abstraction is always preceded by a type abstraction, some properties of F^p_ι can be derived from those of F_ι. In particular, normalization and subject reduction properties are preserved, just by observing that F^p_ι is syntactically closed by reduction.

This can be formalized by making the correspondence between F^p_ι and F_ι explicit.

Definition 28 (F^p_ι to F_ι traduction)**.** *We define a Translation from F^p_ι to F_ι witnessing the language inclusion on Figure 28. We write M° the translation of M. We only give the translations that do not simply reuse the same construct by calling recursively the translation function on subtrees. The meta-variable W stand for either M or G.*

Lemma 29 (Restriction equivalence)**.** *The following assertions hold.*

1. $\Gamma \vdash M : \tau$ *holds in* F^p_ι *if and only if* $\Gamma^\circ \vdash M^\circ : \tau^\circ$ *holds in* F_ι.
2. $\Gamma \vdash G : \tau \triangleright \sigma$ *holds in* F^p_ι *if and only if* $\Gamma^\circ \vdash G^\circ : \tau^\circ \triangleright \sigma^\circ$ *holds in* F_ι.
3. $\Gamma \vdash \tau$ *holds in* F^p_ι *if and only if* $\Gamma^\circ \vdash \tau^\circ$ *holds in* F_ι.
4. $\Gamma \vdash ok$ *holds in* F^p_ι *if and only if* $\Gamma^\circ \vdash ok$ *holds in* F_ι.

Lemma 30. *The following assertions hold.*

1. *If* $M \rightsquigarrow_\beta N$ *in* F^p_ι, *then* $M^\circ \rightsquigarrow_\beta N^\circ$ *in* F_ι.
2. *If* $M \rightsquigarrow_\iota N$ *in* F^p_ι, *then* $M^\circ \rightsquigarrow^+_\iota N^\circ$ *in* F_ι.

Proposition 31 (Preservation). *If $\Gamma \vdash M : \tau$ and $M \rightsquigarrow_{\beta\iota} N$ hold, then $\Gamma \vdash N : \tau$ holds.*

(Proof p. 55)

Proposition 32 (Termination). *Reduction in F^p_ι is terminating.*

(Proof p. 55)

Confluence and progress must still be verified. For confluence, we observe that there are still no critical pairs (although this does not follow from the absence of critical pairs in F_ι), so weak confluence is still preserved and confluence comes as a corollary.

Corollary 33 (Confluence). *Reduction in F^p_ι is confluent.*

(Proof p. 55)

Progress is a proof on its own, but it is similar to the one in F_ι.

Lemma 34 (Classification). *If $\Gamma \vdash v : \tau$ holds, then either v is a prevalue p or:*

1. *If τ is of the form $\tau \to \tau$, then v is of the form $\lambda(x : \tau)\, v$.*
2. *If τ is of the form $(\tau * \tau)$, then v is of the form (v, v).*
3. *If τ is of the form $\forall\alpha.\tau$, then v is of the form $\lambda\alpha\, v$.*
4. *If τ is of the form $\forall(\alpha \mathrel{\diamond} \tau) \Rightarrow \tau$, then v is of the form $\lambda(\alpha \mathrel{\diamond} c : \tau)\, v$.*
5. *If τ is of the form $\top$, then v is of the form $\mathtt{Top}^\tau\langle v\rangle$.*

(Proof p. 56)

Proposition 35 (Progress). *If $\Gamma \vdash M : \tau$ holds, then either M is a value or M reduces.*

(Proof p. 56)

As expected, coercions are erasable in F^p_ι. Because the new reduction rules are a combination of two ι-rules, and are themselves ι-rules, the forward simulation follows from forward simulation in F_ι. It remains to check the backward simulation.

Lemma 36 (Forward simulation). *If $\Gamma \vdash M : \tau$ holds, then:*

1. *If $M \rightsquigarrow_\beta N$, then $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$.*
2. *If $M \rightsquigarrow_\iota N$, then $\lfloor M \rfloor = \lfloor N \rfloor$.*

(Proof p. 57)

Lemma 37 (Classification). *If $\Gamma \vdash Q[\lambda(x : \rho)\, M] : \tau$ (resp. $\Gamma \vdash Q[(M, N)] : \tau$) holds and $Q[\lambda(x : \rho)\, M]$ (resp. $Q[(M, N)]$) is in ι-normal-form, then:*

1. *If τ is $\sigma \to \sigma'$ (resp. $(\sigma * \sigma')$) then Q is $[]$.*
2. *If τ is $\forall\alpha.\sigma$ then Q is $\lambda\alpha\, Q'$.*
3. *If τ is $\forall(\alpha \mathrel{\diamond} \sigma) \Rightarrow \tau'$ then Q is $\lambda(\alpha \mathrel{\diamond} c : \sigma)\, Q'$.*
4. *For all $\alpha \mathrel{\triangleright} c : \sigma$ in Γ, τ is not α.*

Proof. We only do the proof for $Q[\lambda(x : \rho)\, M]$. The proof for $Q[(M, N)]$ is similar. By induction on Q.

- $[]$: Conditions 2 and 3 do not apply. Conditions 1 and 4 hold trivially.
- $\lambda\alpha\, Q'$: Conditions 2 and 4 hold trivially. Other conditions do not apply.
- $Q'\,\tau'$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\, M] : \forall\alpha.\rho'$ such that $\rho'[\alpha \leftarrow \tau'] = \tau$. By induction hypothesis we have Q' of the form $\lambda\alpha\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedType applies.
- $G\langle Q'\rangle$: By induction on G.

  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, and $M.2$: These are refused by typing, because they are terms instead of coercions.
  - $\lambda\alpha\ W$, $W\,\tau$, $G\langle W\rangle$, $\lambda(\alpha \diamond c : \tau)\,W$, and $W\{\tau \diamond G\}$: These are not in ι-normal-form, since RedCoerFill applies.
  - $\Diamond^{\tau}$: It is not in ι-normal-form, since RedCoerDot applies.
  - c when $\alpha \triangleleft c : \tau$: Conditions 1 to 3 do not apply. And 4 holds because α cannot be bounded twice in Γ and it is already present with $\alpha \triangleleft c : \tau$.
  - c when $\alpha \triangleright c : \tau$: This case is rejected by induction hypothesis, since we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \alpha$ with $\alpha \triangleright c : \tau \in \Gamma$.
  - $G \xrightarrow{\tau} G$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \sigma \to \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι-normal-form, since RedCoerArrow applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda\alpha\ \lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedCoerDistTypeArrow applies.
  - $\mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{\tau\to\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda(\alpha \diamond c : \sigma)\,\lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedCoerDistTCoerArrow applies.
  - Top^{τ}, $(G * G)$, $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}$, and $\mathsf{Dist}^{\forall\alpha\diamond\tau\Rightarrow}_{(\tau*\tau)}$: Conditions 1 to 3 do not apply, and condition 4 trivially holds.

- $\lambda(\sigma \diamond c : \alpha)\,Q'$: Conditions 3 and 4 hold trivially. Other conditions do not apply.
- $Q'\{\sigma' \diamond G\}$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \forall(\alpha \diamond \sigma) \Rightarrow \tau$. By induction hypothesis we have Q' of the form $\lambda(\alpha \diamond c : \sigma)\,Q''$, which contradicts the fact that we were in ι-normal-form, since RedTCoer applies.

□

Proposition 38 (Backward simulation)**.** *If* $\Gamma \vdash M : \tau$ *and* $\lfloor M \rfloor \rightsquigarrow \mathcal{M}$*, then* $M \rightsquigarrow_{\iota}^{\star} \rightsquigarrow_{\beta} N$ *such that* $\lfloor N \rfloor = \mathcal{M}$*.*

Proof. The proof schema is not original: following Manzonetto and Tranquilli [2010]. we show that the ι-normal-form of M β-reduces to N with $\lfloor N \rfloor$ equal to $\mathcal{M}$. Since F_{ι} strongly normalizes, we may assume, without lost of generality that M is already in ι-normal-form. Because $\lfloor M \rfloor$ reduces, we can use the reduction derivation to show that it must be of the form $e[(\lambda x.\mathcal{M}_1)\,\mathcal{M}_2]$. By inversion of the coercion-erasure function, we show that M is of the form $C[Q[\lambda(x : \tau)\ M_1]\ M_2]$ where C is a reduction context and Q a retyping context of arbitrary depth, such that C, M_1, and M_2 erase to e, $\mathcal{M}_1$, and $\mathcal{M}_2$ respectively. We show using Lemma 37 that if a ι-normal term of the form $Q[\lambda(x : \tau)\ M]$ has an arrow type, then Q is empty. Hence, M is of the form $C[(\lambda(x : \tau)\ M_1)\ M_2]$ and β-reduces to $C[M_1[x \leftarrow M_2]\,]$ whose erasure is $e[a_1[x \leftarrow a_2]\,]$. A similar proof holds for pairs instead of arrows. □

6 Expressiveness of Parametric F_{ι}

Although it is bridled by-design, F^{p}_{ι} is already an interesting spot in the design space, as it subsumes in a unified framework three known languages: F_{η}, $x\mathsf{MLF}$, and $\mathsf{F}_{<:}$ (in fact, its more expressive version with F-bounded polymorphism [Canning et al., 1989]).

By construction, F_{η} is included (and simulated) in Parametric F_{ι}. In the rest of this section, we show that $x\mathsf{MLF}$ and $\mathsf{F}_{<:}$ are also subsumed by F^{p}_{ι}. In each case, we exhibit a translation of typing judgments from the source language to typing judgments of F^{p}_{ι} so that the coercion erasure of the translation of a source term is equal to the type erasure of this term, and therefore the translation is semantics preserving.

$$\textsc{Kernel-Fsub}\ \frac{\Sigma, \alpha <: A \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A)\ B'} \qquad \textsc{Full-Fsub}\ \frac{\Sigma \vdash A' <: A \qquad \Sigma, \alpha <: A' \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A')\ B'}$$

$$\textsc{F-Bounded}\ \frac{\Sigma, \alpha <: A' \vdash \alpha <: A \qquad \Sigma, \alpha <: A' \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A')\ B'}$$

Figure 29: Bounded polymorphism: variants on the subtyping rule

To avoid confusion between source and target terms, we write T or S for terms, A or B for types, and Σ for typing environments in the source language. Formally, we exhibit a translation of judgments $\Sigma \vdash T : A \rightsquigarrow \Gamma \vdash M : \tau$ that is well-defined, type preserving, and semantics preserving. That is, if $\Sigma \vdash T : A$ then $\Sigma \vdash T : A \rightsquigarrow \Gamma \vdash M : \tau$ holds for some Γ, M, and τ such that $\Gamma \vdash M : \tau$ and $\lfloor T \rfloor = \lfloor M \rfloor$. As a consequence, reduction in the source language terminates, since it is simulated in F_ι^p.

Bounded polymorphism. $\mathsf{F}_{<:}$ is a well-known extension of System F with subtyping. There are several variations on $\mathsf{F}_{<:}$, all sharing the same features, but with different expressiveness due to the way they deal with subtyping of bounded quantification. Bounded quantification $\forall(\alpha <: A)\ B$ restricts types A' that α ranges over to be subtypes of the bound A. The differences lie in when the subtyping judgment $\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A')\ B'$ holds. Different versions of the corresponding subtyping rule are given on Figure 29. In Kernel $\mathsf{F}_{<:}$, the bounds A and A' must be equal, whereas Full $\mathsf{F}_{<:}$ only requires the bound A' to be a subtype of the bound A. Moreover, α cannot appear free in the bounds A or A' in Kernel or Full $\mathsf{F}_{<:}$, while $\mathsf{F}_{\mu<:}$ allows this form of recursion, called F-bounded polymorphism. The most general assumption, $\Sigma, \alpha <: A' \vdash \alpha <: A$, is that of $\mathsf{F}_{\mu<:}$. Perhaps surprisingly, this is a slightly more general rule [Baldan et al., 1999] than the more intuitive one $\Sigma, \alpha <: A' \vdash A' <: A$. In summary, we have Kernel $\mathsf{F}_{<:} \subset$ Full $\mathsf{F}_{<:} \subset \mathsf{F}_{\mu<:}$ where all inclusions are strict.

We show that the most expressive version $\mathsf{F}_{\mu<:}$ is included into F_ι^p. The translation of typing judgments uses auxiliary translations of subtyping judgments $\Sigma \vdash A <: B \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ and well-formedness judgments. Bounded polymorphism $\forall(\alpha <: A)\ B$ is translated into a negative coercion abstraction $\forall(\alpha \triangleright \tau) \Rightarrow \sigma$ which encodes upper bounds. (Positive coercion abstraction $\forall(\alpha \triangleleft \tau) \Rightarrow \sigma$ encodes lower bounds and are never needed in the translation of $\mathsf{F}_{\mu<:}$.)

Translation of expressions is easy. For example, the translation of a type application is a coercion application, as follows:

$$\frac{\Sigma \vdash T : \forall(\alpha <: B)\ B' \rightsquigarrow \Gamma \vdash M : \forall(\alpha \triangleright \sigma) \Rightarrow \sigma' \qquad \Sigma \vdash A <: B[\alpha \leftarrow A] \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma[\alpha \leftarrow \tau]}{\Sigma \vdash T\ A : B'[\alpha \leftarrow A] \rightsquigarrow \Gamma \vdash M\{\tau \triangleright G\} : \sigma'[\alpha \leftarrow \tau]}$$

The most involved part in the translation is for subtyping judgments—in particular, for the bounded-quantification case:

$$\frac{\begin{array}{c}\Sigma, \alpha <: A' \vdash \alpha <: A \rightsquigarrow \Gamma, \alpha \triangleright c : \tau' \vdash G : \alpha \triangleright \tau \quad (\mathbf{1}) \\ \Sigma, \alpha <: A' \vdash B <: B' \rightsquigarrow \Gamma, \alpha \triangleright c : \tau' \vdash G' : \sigma \triangleright \sigma' \quad (\mathbf{2})\end{array}}{\begin{array}{c}\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A')\ B' \rightsquigarrow \\ \Gamma \vdash \lambda(\alpha \triangleright c : \tau')\, G'\langle \Diamond\{\alpha \triangleright G\}\rangle : \forall(\alpha \triangleright \tau) \Rightarrow \sigma \triangleright \forall(\alpha \triangleright \tau') \Rightarrow \sigma'\end{array}}$$

Let us check that the judgment returned by the conclusion holds under the assumptions returned by the premises (1) and (2). The implicit superscript of the hole in the conclusion is the domain of the coercion $\forall(\alpha \triangleright \tau) \Rightarrow \sigma$, say ρ. In environment $\Gamma, \alpha \triangleright c : \tau'$, the coercion $\Diamond\{\alpha \triangleright G\}$ has type $\rho \triangleright \sigma$ by rule CoerTCoerApp and, since G' coerces σ to σ', the coercion $G'\langle \Diamond\{\alpha \triangleright G\}\rangle$ has type $\rho \triangleright \sigma'$. Hence, by rule CoerTCoerLam, the coercion of the conclusion has type $\rho \triangleright \forall(\alpha \triangleright \tau') \Rightarrow \sigma'$, as expected.

$$\begin{array}{c} \alpha^{\flat} = \alpha \\ (T \to S)^{\flat} = T^{\flat} \to S^{\flat} \\ (\forall(\alpha <: T)\ S)^{\flat} = \forall(\alpha \triangleright T^{\flat}) \Rightarrow S^{\flat} \\ \top^{\flat} = \top \end{array} \qquad \begin{array}{c} \epsilon^{\flat} = \emptyset \\ (A, \alpha <: T)^{\flat} = A^{\flat}, (\alpha \triangleright c_{\alpha} : T^{\flat}) \\ (A, x : T)^{\flat} = A^{\flat}, (x : T^{\flat}) \end{array}$$

Figure 30: $\mathsf{F}_{\mu<:}$: type and environment translation

$$\textsc{BTermVar}\ \frac{T = A(x)}{A \vdash x : T \rightsquigarrow A^{\flat} \vdash x : T^{\flat}} \qquad \textsc{BSubs}\ \frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \quad A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma}{A \vdash m : S \rightsquigarrow \Gamma \vdash G\langle M \rangle : \sigma}$$

$$\textsc{BTermLam}\ \frac{A, x : T \vdash m : S \rightsquigarrow \Gamma, (x : \tau) \vdash M : \sigma}{A \vdash \lambda x : T.m : T \to S \rightsquigarrow \Gamma \vdash \lambda(x : \tau)\ M : \tau \to \sigma} \qquad \textsc{BTermApp}\ \frac{A \vdash m : T \to S \rightsquigarrow \Gamma \vdash M : \tau \to \sigma \quad A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau}{A \vdash m(n) : S \rightsquigarrow \Gamma \vdash M\ N : \sigma}$$

$$\textsc{BTypeLam}\ \frac{A, \alpha <: S \vdash m : T \rightsquigarrow \Gamma, (\sigma \triangleleft c_{\alpha} : \alpha) \vdash M : \tau}{A \vdash \Lambda(\alpha <: S)\, m : \forall(\alpha <: S)\ T \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleright c_{\alpha} : \sigma)\ M : \forall(\alpha \triangleright \sigma) \Rightarrow \tau}$$

$$\textsc{BTypeApp}\ \frac{A \vdash m : \forall(\alpha <: T)\ S \rightsquigarrow \Gamma \vdash M : \forall(\alpha \triangleright \tau) \Rightarrow \sigma \quad A \vdash T' <: T[\alpha \leftarrow T'] \rightsquigarrow \Gamma \vdash G : \tau' \triangleright \tau[\alpha \leftarrow \tau']}{A \vdash m\{T'\} : S[\alpha \leftarrow T'] \rightsquigarrow \Gamma \vdash M\{\tau' \triangleleft G\} : \sigma[\alpha \leftarrow \tau']}$$

Figure 31: $\mathsf{F}_{\mu<:}$: term translation

Notice that $\mathsf{F}_{\mu<:}$ is missing type abstraction and type application in coercions, as well as distributivity of the universal on the arrow as in F_{η}. Indeed, $\mathsf{F}_{\mu<:}$ only allows instantiation of quantifiers at the root of types, as in System F and contrary to F_{η}. Hence, the inclusion $\mathsf{F}_{\mu<:} \subset \mathsf{F}_{\iota}^{p}$ is strict.

It is remarkable that F_{ι}^{p} naturally matches the most expressive version $\mathsf{F}_{\mu<:}$. This encourages following a systematic approach and viewing type conversions as erasable coercions as in F_{ι}^{p} rather then a limited subtyping relation. Additionally, F_{ι}^{p} may simplify the proof of type soundness for $\mathsf{F}_{\mu<:}$, as coercions are explicit.

To show that $\mathsf{F}_{\mu<:}$ is included in F_{ι}^{p}, we define a translation for types and environments on Figure 30, for term judgments on Figure 31, and for subtyping rules on Figure 32.

We have the following obvious lemma stating that the translation of term and subtyping judgments respect the translation of types and environments:

Lemma 39. *1. If $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ holds then $\Gamma = A^{\flat}$ and $\tau = T^{\flat}$ hold.*

2. If $A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ holds then $\Gamma = A^{\flat}$, $\tau = T^{\flat}$, and $\sigma = S^{\flat}$ hold.

We show that $\mathsf{F}_{\mu<:}$ is included in F_{ι}^{p} by showing in Proposition 40 that well-typed expressions are included.

Proposition 40. *The following assertions hold:*

1. If $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ holds, then $\Gamma \vdash M : \tau$ holds.

2. If $A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ holds, then $\Gamma \vdash G : \tau \triangleright \sigma$ holds.

(Proof p. 57)

$$\textsc{BId}\quad A \vdash T <: T \rightsquigarrow \Gamma \vdash \Diamond^{T^\flat} : T^\flat \triangleright T^\flat$$

$$\textsc{BTrans}\quad \frac{A \vdash T <: S \rightsquigarrow \Gamma \vdash G_1 : \tau \triangleright \sigma \qquad A \vdash S <: U \rightsquigarrow \Gamma \vdash G_2 : \sigma \triangleright \rho}{A \vdash T <: U \rightsquigarrow \Gamma \vdash G_2\langle G_1\rangle : \tau \triangleright \rho}$$

$$\textsc{BCoerVar}\quad \frac{\alpha <: T \in A}{A \vdash \alpha <: T \rightsquigarrow \Gamma \vdash c_\alpha : \alpha \triangleright \tau}$$

$$\textsc{BTop}\quad A \vdash T <: \top \rightsquigarrow \Gamma \vdash \mathtt{Top}^\tau : \tau \triangleright \top$$

$$\textsc{BArrow}\quad \frac{A \vdash T' <: T \rightsquigarrow \Gamma \vdash G_1 : \tau' \triangleright \tau \qquad A \vdash S <: S' \rightsquigarrow \Gamma \vdash G_2 : \sigma \triangleright \sigma'}{A \vdash T \to S <: T' \to S' \rightsquigarrow \Gamma \vdash G_1 \xrightarrow{\tau'} G_2 : \tau \to \sigma \triangleright \tau' \to \sigma'}$$

$$\textsc{BForall}\quad \frac{A, \alpha <: T' \vdash \alpha <: T \rightsquigarrow \Gamma, (\alpha \triangleright c_\alpha : \tau') \vdash G : \alpha \triangleright \tau \qquad A, \alpha <: T' \vdash S <: S' \rightsquigarrow \Gamma, (\alpha \triangleright c_\alpha : \tau') \vdash G' : \sigma \triangleright \sigma'}{A \vdash \forall(\alpha <: T)\ S <: \forall(\alpha <: T')\ S' \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleright c : \tau')\ G'\langle\Diamond\{\alpha \triangleleft G\}\rangle : \forall(\alpha \triangleright \tau) \Rightarrow \sigma \triangleright \forall(\alpha \triangleright \tau') \Rightarrow \sigma'}$$

Figure 32: $\mathsf{F}_{\mu<:}$: subtyping translation

$$\begin{array}{rcll}
\tau & ::= & \ldots \not| \ \forall(\alpha \triangleright \tau) \Rightarrow \tau & \text{types} \\
M & ::= & \not| \ \lambda(\alpha \triangleright c : \tau)\, M \not| \ M\{\tau \triangleright G\} & \text{expressions} \\
G & ::= & \ldots \not| \ G \xrightarrow{\tau} G \not| \ \mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau} \not| \ \mathsf{Dist}^{\forall\alpha\triangleright\tau\Rightarrow}_{\tau\to\tau} & \text{coercions} \\
 & & \not| \ (G * G) \not| \ \mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)} \not| \ \mathsf{Dist}^{\forall\alpha\triangleright\tau\Rightarrow}_{(\tau*\tau)} & \\
 & & \not| \ \lambda(\alpha \triangleright c : \tau)\, G' \not| \ G'\{\tau \triangleright G\} &
\end{array}$$

Figure 33: $\mathsf{F}^{\mu x}_\iota$: syntax and notations

Instance-bounded polymorphism. The language xMLF [Rémy and Yakobowski, 2010] is the internal language of MLF which is itself an extension of System F with *instance-bounded polymorphism*. Instance-bounded polymorphism is a mechanism to delay type instantiation of System F; it is a key to performing type inference in MLF and keeping principal types—given optional type annotations of function parameters. As our current concern is not type inference but expressiveness, we use xMLF rather than MLF for comparison with F^p_ι. By lack of space, we cannot formally present xMLF. Instead, we identify a subset F^x_ι of F^p_ι and explains how it closely relates to xMLF without giving all the details of xMLF.

We first define the subset $\mathsf{F}^{\mu x}_\iota$ of F^p_ι by removing negative coercion abstractions (in types, terms, and coercions), arrow coercions $G \xrightarrow{\tau} G$, and distributivity coercions from the syntax of terms. Of course, we remove typing rules and reduction rules for these constructs, accordingly.

We then define F^x_ι as the restriction of $\mathsf{F}^{\mu x}_\iota$ where a type variable cannot appear in its instance bound, *i.e.* α is not free in τ in $\forall(\alpha \triangleleft \tau) \Rightarrow \sigma$. Both restrictions are closed by reduction, so they preserve the properties of F^p_ι.

We claim that xMLF is equivalent to F^x_ι. Unsurprisingly, the translation of instance-bounded polymorphism $\forall(\alpha \geq A).B$ is a positive coercion abstraction $\forall(\alpha \triangleleft \tau) \Rightarrow \sigma$ where τ and σ are the translation of A and B. The translation of expressions and type instantiations is then routine. The proof for the direct inclusion is similar to one by Manzonetto and Tranquilli [2010]. The proof for the reverse inclusion is new but not much more difficult.

In summary, we have $x\mathsf{MLF} \approx \mathsf{F}^x_\iota \subset \mathsf{F}^{\mu x}_\iota \subset \mathsf{F}^p_\iota$. It is interesting that the natural restriction of F_ι that resembles xMLF allows variables to appear in their instance bounds, much as with F-bounded polymorphism. This suggests an extension to xMLF with recursively defined bounds. However, we do not know whether this extension could still permit partial type inference in MLF.

$$\begin{array}{c} \alpha^{\mathtt{x}} = \alpha \\ (T \to S)^{\mathtt{x}} = T^{\mathtt{x}} \to S^{\mathtt{x}} \\ (\forall(\alpha \geq T).S)^{\mathtt{x}} = \forall(\alpha \lhd T^{\mathtt{x}}) \Rightarrow S^{\mathtt{x}} \\ \bot^{\mathtt{x}} = \forall\alpha.\,\alpha \end{array} \qquad \begin{array}{c} \emptyset^{\mathtt{x}} = \emptyset \\ (A, (\alpha \geq T))^{\mathtt{x}} = A^{\mathtt{x}}, (\alpha \lhd c_\alpha : T^{\mathtt{x}}) \\ (A, (x : T))^{\mathtt{x}} = A^{\mathtt{x}}, (x : T^{\mathtt{x}}) \end{array}$$

Figure 34: $x\mathsf{MLF}$: type and environment translation

$$\text{XVar}\ \frac{x : T \in A}{A \vdash x : T \rightsquigarrow A^{\mathtt{x}} \vdash x : T^{\mathtt{x}}} \qquad \text{XLet}\ \frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \qquad A, (x : T) \vdash n : S \rightsquigarrow \Gamma, (x : \tau) \vdash N : \sigma}{A \vdash \mathsf{let}\ x = m\ \mathsf{in}\ n : S \rightsquigarrow \Gamma \vdash (\lambda(x : \tau)\ N)\ M : \sigma}$$

$$\text{XApp}\ \frac{A \vdash m : T \to S \rightsquigarrow \Gamma \vdash M : \tau \to \sigma \qquad A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau}{A \vdash m\,n : S \rightsquigarrow \Gamma \vdash M\ N : \sigma}$$

$$\text{XAbs}\ \frac{A, (x : T) \vdash m : S \rightsquigarrow \Gamma, (x : \tau) \vdash M : \sigma}{A \vdash \lambda(x : T)m : T \to S \rightsquigarrow \Gamma \vdash \lambda(x : \tau)\ M : \tau \to \sigma}$$

$$\text{XTAbs}\ \frac{A, (\alpha \geq S) \vdash m : T \rightsquigarrow \Gamma, (\alpha \lhd c_\alpha : \sigma) \vdash M : \tau \qquad \alpha \notin \mathsf{ftv}(A, S)}{A \vdash \Lambda(\alpha \geq S)\,m : \forall(\alpha \geq S).T \rightsquigarrow \Gamma \vdash \lambda(\alpha \lhd c_\alpha : \sigma)\ M : \forall(\alpha \lhd \sigma) \Rightarrow \tau}$$

$$\text{XTApp}\ \frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \qquad A \vdash \phi : T \leq S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma}{A \vdash m\,\phi : S \rightsquigarrow \Gamma \vdash G\langle M \rangle : \sigma}$$

Figure 35: $x\mathsf{MLF}$: term translation

Moreover, reduction in $x\mathsf{MLF}$ is simulated in F^x_ι. This implies termination of reduction in $x\mathsf{MLF}$ (a result already proved by Manzonetto and Tranquilli [2010]).

To show that $x\mathsf{MLF}$ is included into F^x_ι, we define a translation for types and environments on Figure 34, for term judgments on Figure 35, and for instance judgments on Figure 36.

We have the following obvious lemma stating that the translation of term and instance judgments respect the translation of types and environments:

Lemma 41. *1. If $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ holds then $\Gamma = A^{\mathtt{x}}$ and $\tau = T^{\mathtt{x}}$ hold.*

2. If $A \vdash \phi : T \leq S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ holds then $\Gamma = A^{\mathtt{x}}$, $\tau = T^{\mathtt{x}}$, and $\sigma = S^{\mathtt{x}}$ hold.

We show that $x\mathsf{MLF}$ is included in F^x_ι by showing in Proposition 42 that well-typed expressions are included and in Proposition 43 that the reduction relation is included.

Proposition 42. *The following assertions hold:*

1. *If $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ holds, then $\Gamma \vdash M : \tau$ holds and $\lfloor m \rfloor = \lfloor M \rfloor$.*
2. *If $A \vdash \phi : T \leq S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ holds, then $\Gamma \vdash G : \tau \triangleright \sigma$ holds.*

(Proof p. 57)

Proposition 43. *If $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$, $A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau$, and $m \rightsquigarrow n$ hold, then $M \rightsquigarrow^+_{\iota\beta} N$ holds.*

(Proof p. 57)

To show that F^x_ι is included into $x\mathsf{MLF}$, we define a translation for types and environments on Figure 37 and for expressions on Figure 38.

We show that F^x_ι is included in $x\mathsf{MLF}$ by showing in Proposition 44 that well-typed expressions are included in terms and instances and in Proposition 45 that the reduction relation is included.

$$\textsc{XIBot}\quad A \vdash T : \bot \leq T \rightsquigarrow \Gamma \vdash \Diamond^{\forall\alpha.\,\alpha}\,\tau : \forall\alpha.\,\alpha \triangleright \tau$$

$$\textsc{XIAbs}\quad \frac{\alpha \geq T \in A}{A \vdash !\alpha : T \leq \alpha \rightsquigarrow \Gamma \vdash c_\alpha : \tau \triangleright \alpha}$$

$$\textsc{XIUnder}\quad \frac{A, (\alpha \geq T) \vdash \phi : T_1 \leq T_2 \rightsquigarrow \Gamma, (\alpha \triangleleft c_\alpha : \tau) \vdash G : \tau_1 \triangleright \tau_2}{\begin{array}{l} A \vdash \forall(\alpha \geq)\phi : \forall(\alpha \geq T).T_1 \leq \forall(\alpha \geq T).T_2 \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\alpha \triangleleft c_\alpha : \tau)\; G\langle \Diamond^{\forall(\alpha \triangleleft \tau) \Rightarrow \tau_1}\{\alpha \triangleright c_\alpha\}\rangle : \forall(\alpha \triangleleft \tau) \Rightarrow \tau_1 \triangleright \forall(\alpha \triangleleft \tau) \Rightarrow \tau_2 \end{array}}$$

$$\textsc{XIInside}\quad \frac{A \vdash \phi : T_1 \leq T_2 \rightsquigarrow \Gamma \vdash G : \tau_1 \triangleright \tau_2}{\begin{array}{l} A \vdash \forall(\geq \phi) : \forall(\alpha \geq T_1).T \leq \forall(\alpha \geq T_2).T \quad \rightsquigarrow \\ \Gamma \vdash \lambda(\alpha \triangleleft c_\alpha : \tau_2)\; \Diamond^{\forall(\alpha \triangleleft \tau_1) \Rightarrow \tau}\{\alpha \triangleright c_\alpha\langle G\rangle\} : \forall(\alpha \triangleleft \tau_1) \Rightarrow \tau \triangleright \forall(\alpha \triangleleft \tau_2) \Rightarrow \tau \end{array}}$$

$$\textsc{XIIntro}\quad \frac{\alpha \notin \mathsf{ftv}(T)}{A \vdash \mathfrak{Y} : T \leq \forall(\alpha \geq \bot).T \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleleft c_\alpha : \forall\alpha.\,\alpha)\; \Diamond^{\tau} : \tau \triangleright \forall(\alpha \triangleleft \forall\alpha.\,\alpha) \Rightarrow \tau}$$

$$\textsc{XIComp}\quad \frac{A \vdash \phi_1 : T_1 \leq T_2 \rightsquigarrow \Gamma \vdash G_1 : \tau_1 \triangleright \tau_2 \qquad A \vdash \phi_2 : T_2 \leq T_3 \rightsquigarrow \Gamma \vdash G_2 : \tau_2 \triangleright \tau_3}{A \vdash \phi_1; \phi_2 : T_1 \leq T_3 \rightsquigarrow \Gamma \vdash G_2\langle G_1\rangle : \tau_1 \triangleright \tau_3}$$

$$\textsc{XIElim}\quad A \vdash \,\& : \forall(\alpha \geq T).S \leq S[\alpha \leftarrow T] \rightsquigarrow \Gamma \vdash \Diamond^{\forall(\alpha \triangleleft \tau) \Rightarrow \sigma}\{\tau \triangleright \Diamond^{\tau}\} : \forall(\alpha \triangleleft \tau) \Rightarrow \sigma \triangleright \sigma[\alpha \leftarrow \tau]$$

$$\textsc{XIId}\quad A \vdash \mathbf{1} : T \leq T \rightsquigarrow \Gamma \vdash \Diamond^{\tau} : \tau \triangleright \tau$$

Figure 36: xMLF: instance translation

$$\begin{array}{rlrl} \alpha^{\mathrm{y}} &= \alpha & \emptyset^{\mathrm{y}} &= \emptyset \\ (\tau \to \sigma)^{\mathrm{y}} &= \tau^{\mathrm{y}} \to \sigma^{\mathrm{y}} & (\Gamma, (\alpha \triangleleft c : \tau))^{\mathrm{y}} &= \Gamma^{\mathrm{y}}, (\alpha \geq \tau^{\mathrm{y}}) \\ (\forall\alpha.\,\tau)^{\mathrm{y}} &= \forall(\alpha \geq \bot).\tau^{\mathrm{y}} & (\Gamma, (x : \tau))^{\mathrm{y}} &= \Gamma^{\mathrm{y}}, (x : \tau^{\mathrm{y}}) \\ (\forall(\alpha \triangleleft \tau) \Rightarrow \sigma)^{\mathrm{y}} &= \forall(\alpha \geq \tau^{\mathrm{y}}).\sigma^{\mathrm{y}} & (\Gamma, \alpha)^{\mathrm{y}} &= \Gamma^{\mathrm{y}}, (\alpha \geq \bot) \end{array}$$

Figure 37: xMLF: type and environment reverse translation

$$\begin{array}{rlrl} x^{\mathrm{y}} &= x & (\Diamond^{\tau})^{\mathrm{y}} &= \mathbf{1} \\ (\lambda(x : \tau)\; M)^{\mathrm{y}} &= \lambda(x : \tau^{\mathrm{y}})M^{\mathrm{y}} & c^{\mathrm{y}} &= !(\mathsf{codom}(\tau)) \\ (M\; N)^{\mathrm{y}} &= M^{\mathrm{y}}\, N^{\mathrm{y}} & (\lambda\alpha\; G)^{\mathrm{y}} &= \mathfrak{Y}; \forall(\alpha \geq) G^{\mathrm{y}} \\ (\lambda\alpha\; M)^{\mathrm{y}} &= \Lambda(\alpha \geq \bot)\, M^{\mathrm{y}} & (G\,\tau)^{\mathrm{y}} &= G^{\mathrm{y}}; \forall(\geq \tau^{\mathrm{y}}); \& \\ (M\,\tau)^{\mathrm{y}} &= M^{\mathrm{y}}\, (\forall(\geq \tau^{\mathrm{y}}); \&) & (\lambda(\alpha \triangleleft c : \tau)\; G)^{\mathrm{y}} &= \mathfrak{Y}; \forall(\geq \tau^{\mathrm{y}}); \forall(\alpha \geq) G^{\mathrm{y}} \\ (\lambda(\alpha \triangleleft c : \tau)\; M)^{\mathrm{y}} &= \Lambda(\alpha \geq \tau^{\mathrm{y}})\, M^{\mathrm{y}} & (G\{\tau \triangleright G'\})^{\mathrm{y}} &= G^{\mathrm{y}}; \forall(\geq G'^{\mathrm{y}}); \& \\ (M\{\tau \triangleright G\})^{\mathrm{y}} &= M^{\mathrm{y}}\, (\forall(\geq G^{\mathrm{y}}); \&) & (G\langle G'\rangle)^{\mathrm{y}} &= G'^{\mathrm{y}}; G^{\mathrm{y}} \\ (G\langle M\rangle)^{\mathrm{y}} &= M^{\mathrm{y}}\, G^{\mathrm{y}} & & \end{array}$$

Figure 38: xMLF: expression reverse translation

Extension of System F	F_η	$\mathsf{F}_{<:}$	*x*MLF	F_ι^p
Deep instantiation	√	-	√	√
Arrow congruence	√	√	-	√
Permutation of $\forall$ and $\to$	√	-	-	√
Upper bounds	-	√	-	√
Lower bounds	-	-	√	√

Figure 39: Language and feature comparison

Proposition 44. *The following assertions hold.*

1. *If* $\Gamma \vdash M : \tau$ *holds, then* $\Gamma^{\mathrm{y}} \vdash M^{\mathrm{y}} : \tau^{\mathrm{y}}$ *holds and* $\lfloor M^{\mathrm{y}} \rfloor = \lfloor M \rfloor$.
2. *If* $\Gamma \vdash G : \tau \triangleright \sigma$ *holds, then* $\Gamma^{\mathrm{y}} \vdash G^{\mathrm{y}} : \tau^{\mathrm{y}} \leq \sigma^{\mathrm{y}}$ *holds.*

(Proof p. 58)

Proposition 45. *If* $M \rightsquigarrow_{\beta\iota} N$ *holds, then* $M^{\mathrm{y}} \rightsquigarrow^{+} N^{\mathrm{y}}$ *holds.*

(Proof p. 58)

Summary Features of F_ι^p and its variants are summed up on Figure 39. The expressiveness of F_η, *x*MLF, and $\mathsf{F}_{<:}$ can be compared by checking which feature is present in one language and not in the others. Deep instantiation corresponds to the $\lambda\alpha\ G$ and $G\,\tau$ constructs, allowed in F_η and *x*MLF, but not in $\mathsf{F}_{<:}$. Upper bounds are used in $\mathsf{F}_{<:}$ and lower bounds are used *x*MLF. They correspond to coercion abstraction $\lambda(\alpha \diamond c : \tau)\, G$ and $G\{\tau \diamond G\}$ when $\diamond$ is $\triangleright$ or $\triangleleft$, respectively. F_η allows neither. Arrow congruence is the $G \xrightarrow{\tau} G$ construct, allowed in F_η and $\mathsf{F}_{<:}$. Distributivity $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ is used in F_η. The other form $\mathsf{Dist}^{\forall\alpha\diamond\rho\Rightarrow}_{\tau\to\sigma}$ is only used in F_ι^p since it involves coercion abstraction.

Notice that *x*MLF and $\mathsf{F}_{<:}$ only have coercion abstraction in common, but with opposite polarities. Each of them share a different feature with F_η. None of them uses distributivity as it only makes sense when deep instantiation and arrow congruence are available simultaneously.

All examples of §2.4 are actually typable in F_ι^p—with some syntactic adjustment of course. For instance, the last example becomes $\lambda(\gamma \lhd c : \sigma_{ch})\, \mathsf{choose}\ \{\gamma \lhd (c\{\mathsf{choose}\})\}$ of type $\forall(\gamma \lhd \sigma_{ch}) \Rightarrow \gamma \to \gamma$. For instance, it can be coerced to the type $\forall(\gamma \lhd \sigma_{plus}) \Rightarrow \gamma \to \gamma$. This uses the coercion $\lambda(\gamma \lhd c : \sigma_{plus})\, \Diamond\{\gamma \lhd c\langle \Diamond^{\sigma_{ch}}\, \mathsf{int}\rangle\}$.

7 Weak F_ι

Another solution to recover erasability is to prevent wedges from appearing in a reduction context.

At first, it seems to suffice to use weak reduction on coercion abstraction. Indeed, if a coercion variable cannot appear under a reduction context, it cannot appear in a wedging configuration. However, since $\lambda(c : \varphi)\ M$ is irreducible, its erasure $\lfloor M \rfloor$ should also be irreducible, *i.e.* a value. If we choose strong reduction for term abstraction, we must also choose strong reduction in the λ-calculus used as the target, hence $\lfloor M \rfloor$ must be a value for strong reduction. That is, $\lambda(c : \varphi)\ M$ would only be allowed when M is fully evaluated, which would considerably limit the interest of abstracting over c. Therefore, we choose a weak strategy for both coercions and terms. Keeping strong reduction on types is optional and independent.

Syntactic restrictions The syntax of Weak F_ι, written F_ι^w, is defined on Figure 40 as a restriction of the syntax of F_ι. We remove distributivity of coercion abstraction on term abstraction $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}$ in order to preserve the value restriction during reduction. We replace $\lambda(c : \varphi)\ M$ in terms by $\lambda(c : \varphi)\ u$ where u is a value form. A *value form* is a term that erases to a value, *i.e.* a

$$
\begin{array}{rcll}
M & ::= & \ldots \not| \ \lambda(c:\varphi)\ M \mid \lambda(c:\varphi)\ u & \text{expressions} \\
G & ::= & \ldots \not| \ \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma} & \text{coercions} \\
v & ::= & \lambda(x:\tau)\ M \mid \lambda\alpha\ v \mid \lambda(c:\varphi)\ u \mid (v,v) \mid \mathsf{Top}^{\tau}\langle v\rangle & \text{values} \\
u & ::= & v \mid G\langle u\rangle & \text{value forms} \\
C & ::= & []\ M \mid M\ [] \mid ([],M) \mid (M,[]) \mid [].1 \mid [].2 \mid \lambda\alpha\ [] \mid []\,\tau \mid G\langle[]\rangle \mid []\{G\} & \text{reduction ctx}
\end{array}
$$

$$
(\lambda(c:\varphi)\ G)\langle u\rangle \overset{\textsc{RedCoerCoerLam}^{w}}{\rightsquigarrow_{\iota}} \lambda(c:\varphi)\ G\langle u\rangle
$$

Figure 40: Weak F_ι: syntax and semantics *wrt* F_ι

value or an application of a coercion G to a value form. A value is any form of abstraction whose subterm is an arbitrary term for a term abstraction, a value for a type abstraction (because we may evaluate under type abstractions), or a value form for a coercion abstraction.

The static semantics of F^w_ι and F_ι are the same.

Changes in the dynamic semantics The reduction relation of F^w_ι is a subrelation of the reduction relation of F_ι that prevents evaluation under term and coercion abstractions and preserves the value restriction. Reduction contexts are modified accordingly: $\lambda(x:\tau)\ []$ and $\lambda(c:\varphi)\ []$ are removed. Rule RedCoerCoerLam (the coercion abstraction part of RedCoerFill) is restricted to make it call-by-value. Indeed, keeping the F_ι rule:

$$
(\lambda(c:\varphi)\ G)\langle M\rangle \rightsquigarrow_{\iota} \lambda(c:\varphi)\ G\langle M\rangle
$$

would place the arbitrary term M under a coercion abstraction. We also completely remove RedCoerDistCoerArrow as it can never happen.

Preservation of properties By construction, a well-typed term of Weak F_ι is also a well-typed term of F_ι, since the syntax of Weak F_ι is a subset of that of F_ι and the typing rules are the same.

Additionally, the reduction relation of Weak F_ι is included into the reduction relation of F_ι. This is the case because we restricted the reduction contexts to implement weak reduction and the RedCoerCoerLam rule to preserve the value restriction.

As a consequence, subject reduction and normalization properties of F_ι are preserved in Weak F_ι. The progress lemma cannot be lifted in a similar way, because the reduction relation of Weak F_ι is strictly included into the one of F_ι and we changed the values. Because we do strong reduction only on types, we state the progress lemma for terms that are typed in an environment containing only type variables. The proof is done as usual. But prior to this, we prove that reduction stays in the language.

Lemma 46 (Value restriction preservation)**.** *If M is syntactically correct and $M \rightsquigarrow_{\beta\iota} N$ holds, then N is syntactically correct.*

(Proof p. 58)

Lemma 47. *If $\Gamma \vdash M : \tau$, $\Gamma \vdash G : \tau \triangleright \sigma$, $\Gamma \vdash \tau$, $\Gamma \vdash ok$, or $M \rightsquigarrow_{\beta\iota} N$ holds in Weak F_ι, then it also holds in F_ι.*

(Proof p. 58)

Lemma 48. *If $\Gamma \vdash M : \tau$ holds in F_ι and M is syntactically correct in Weak F_ι, then $\Gamma \vdash M : \tau$ holds in Weak F_ι.*

(Proof p. 58)

Proposition 49 (Preservation)**.** *If $\Gamma \vdash M : \tau$ and $M \rightsquigarrow_{\beta\iota} N$ hold, then $\Gamma \vdash N : \tau$ holds.*

(Proof p. 59)

Proposition 50 (Termination)**.** *Reduction in Weak F_ι is terminating.*

(Proof p. 59)

Lemma 51 (Classification)**.** *If $\Gamma \vdash v : \tau$ holds, then:*

1. *If τ is of the form $\tau \to \tau$, then v is of the form $\lambda(x : \tau)\, v$.*
2. *If τ is of the form $(\tau * \tau)$, then v is of the form (v, v).*
3. *If τ is of the form $\forall\alpha.\tau$, then v is of the form $\lambda\alpha\, v$.*
4. *If τ is of the form $(\tau \triangleright \tau) \Rightarrow \tau$, then v is of the form $\lambda(c : \tau \triangleright \tau)\, v$.*
5. *If τ is of the form $\top$, then v is of the form $\mathsf{Top}^\tau\langle v\rangle$.*

(Proof p. 59)

Lemma 52 (Progress)**.** *If $\overrightarrow{\alpha} \vdash M : \tau$ holds, then either M is a value or it reduces.*

(Proof p. 59)

Confluence in Weak F_ι must be proved separately because its statement uses reduction both in premises and conclusion. However, since the only source of non-determinism if the order of evaluation between the function and the argument of an application, confluence is easy to establish.

Corollary 53 (Confluence)**.** *Reduction in Weak F_ι is confluent.*

Bisimulation It remains to check that coercions are erasable in Weak F_ι, *i.e.* to establish a bisimulation with λ-calculus. Of course, this is when λ-calculus is also equipped with a weak evaluation strategy. The forward simulation holds, since it holds in F_ι and the reduction relation is smaller in Weak F_ι, and the erasure of reduction contexts in F_ι^w are Call-by-value reduction contexts in λ-calculus.

Lemma 54 (Forward simulation)**.** *If $\Gamma \vdash M : \tau$ holds, then:*

1. *If $M \rightsquigarrow_\beta N$, then $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$.*
2. *If $M \rightsquigarrow_\iota N$, then $\lfloor M \rfloor = \lfloor N \rfloor$.*

(Proof p. 59)

It remains to check that the backward simulation also holds. Because backward simulation is similar to a progress lemma for ι-reduction, we first show a classification lemma on ι-normal-forms. To do so, we define retyping contexts of arbitrary depth Q as a sequence of retyping contexts P. These Q are arbitrary contexts that erase to their hole.

Lemma 55 (Classification)**.** *If $\overrightarrow{\alpha} \vdash Q[\lambda(x : \tau')\, M] : \tau$ (resp. $\overrightarrow{\alpha} \vdash Q[(M, N)] : \tau$) holds and $Q[\lambda(x : \tau')\, M]$ (resp. $Q[(M, N)]$) is in ι-normal form, then:*

1. *If τ is $\sigma \to \sigma'$ (resp. $(\sigma * \sigma')$) then Q is $[]$.*
2. *If τ is $\forall\alpha.\sigma$ then Q is $\lambda\alpha\, Q'$.*
3. *If τ is $\varphi \Rightarrow \sigma$ then Q is $\lambda(c : \varphi)\, Q'$.*

Proof. By induction on Q.

- $[]$: Only the first case is possible by typing. And it has the right form.
- $\lambda\alpha\, Q'$: Only the second case is possible by typing. And it has the right form.
- $Q'\, \tau'$: By typing we have $\overrightarrow{\alpha} \vdash Q'[\lambda(x : \rho)\, M] : \forall\alpha.\rho'$ (resp. $\overrightarrow{\alpha} \vdash Q'[(M, N)] : \forall\alpha.\rho'$) such that $\rho'[\alpha \leftarrow \tau'] = \tau$. By induction hypothesis we have Q' of the form $\lambda\alpha\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedType applies.
- $G\langle Q'\rangle$: By induction on G.

  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, and $M.2$: These are refused by typing, because they are terms instead of coercions.
  - $\lambda\alpha\ W$, $W\,\tau$, $G\langle W\rangle$, $\lambda(c : \tau \triangleright \tau)\,W$, and $W\{G\}$: These are not in ι-normal-form, since RedCoerCoer applies.
  - $\Diamond^\tau$: It is not in ι-normal-form, since RedCoerDot applies.
  - c: This is refused by typing, since we only have type variables in the environment.
  - Top^τ: No case apply.
  - $G \xrightarrow{\tau} G$: By typing we have $\overrightarrow{\alpha} \vdash Q'[\lambda(x : \rho)\ M] : \sigma \to \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι-normal-form, since RedCoerArrow applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda\alpha\ \lambda(x : \varphi)\ M$, which contradicts the fact that we were in ι-normal-form, since RedCoerDist applies.
  - $(G * G)$, $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}$, and $\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}$: No case apply.
  - $G \xrightarrow{\tau} G$ and $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}$ (resp.): No case apply.
  - $(G * G)$ (resp.): By induction hypothesis, RedCoerProd applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}$ (resp.): By induction hypothesis used twice, RedCoerDistTypeProd applies.
  - $\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}$ (resp.): By induction hypothesis used twice, RedCoerDistCoerProd applies.

- $\lambda(c : \sigma \triangleright \sigma')\,Q'$: Only the third case is possible by typing. And it has the right form.
- $Q'\{G\}$: By typing we have $\overrightarrow{\alpha} \vdash Q'[\lambda(x : \rho)\ M] : (\sigma \triangleright \sigma') \Rightarrow \tau$ (resp. $\overrightarrow{\alpha} \vdash Q'[(M, N)] : (\sigma \triangleright \sigma') \Rightarrow \tau$). By induction hypothesis we have Q' of the form $\lambda(c : \sigma \triangleright \sigma')\,Q''$, which contradicts the fact that we were in ι-normal-form, since RedCoer applies.

□

Lemma 56 (Backward simulation)**.** *If $\overrightarrow{\alpha} \vdash M : \tau$ and $\lfloor M \rfloor \rightsquigarrow a$, then $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ such that $\lfloor N \rfloor = a$.*

Proof. We show that the ι-normal-form of M β-reduces to N with $\lfloor N \rfloor$ equal to a. Since F_ι strongly normalizes, we may assume, without lost of generality, that M is already in ι-normal-form. Because $\lfloor M \rfloor$ reduces, we can use the reduction derivation to show that it must be of the form $e[(\lambda x.a_1)\,a_2]$. By inversion of the coercion-erasure function, we show that M is of the form $C[Q[\lambda(x : \tau)\ M_1]\ M_2]$ where C is a reduction context (we can have neither term abstraction since we do not have them in the λ-calculus, nor coercion abstraction because we have an application node below, and there is no way to have an application node under a coercion abstraction without using a term abstraction) and Q a retyping context of arbitrary depth, such that C, M_1, and M_2 erase to e, a_1, and a_2 respectively. We show using Lemma 55 that if a ι-normal term of the form $Q[\lambda(x : \tau)\ M]$ has an arrow type, then Q is empty. Hence, M is of the form $C[(\lambda(x : \tau)\ M_1)\ M_2]$ and β-reduces to $C[M_1[x \leftarrow M_2]\ ]$ whose erasure is $e[a_1[x \leftarrow a_2]\,]$. A similar proof holds for reduction of pairs. □

8 Related work

Although many type systems could be explained using coercions, since for instance they use a form of subtyping, very few have followed this path and made the connection with coercions explicit.

We have already widely discussed F_η, $\mathsf{F}_{<:}$, and $x\mathsf{MLF}$. Parts of F^x_ι is closely related to the work of Manzonetto and Tranquilli [2010] who proposed the first encoding of $x\mathsf{MLF}$ in a calculus of coercions, but for the main purpose of proving the termination of $x\mathsf{MLF}$. They exhibit a type and

semantics preserving encoding of $x\mathsf{MLF}$ into (their version of) F^x_ι and show a simulation of computation between their F^x_ι and System F. Unfortunately, subject reduction and other properties that depend on it do not hold in their system. Our version of F^x_ι can be seen as a fix to their definition. Hence, there are many resemblances between their development of F^x_ι and our development of F_ι—but the typing rules differ. We omitted the proof of inclusion from $x\mathsf{MLF}$ into F^x_ι by lack of space, but also because it resembles theirs. In fact, their translation of $x\mathsf{MLF}$ into F^x_ι has itself been inspired by the translation of MLF into System F by Leijen and Löh [2005] and Leijen [2007]. However, Manzonetto and Tranquilli restrict their study to the termination of $x\mathsf{MLF}$ without any interest in F_η or $\mathsf{F}_{<:}$, while our main interest is not in F^x_ι, but in F^p_ι and F_ι, *i.e.* a general treatment of abstraction over coercion functions that extends F_η, and as a side result a possible enhancement of $x\mathsf{MLF}$.

Although F_ι subsumes core $\mathsf{F}_{<:}$, we have not included records in F_ι, which are often the first application of $\mathsf{F}_{<:}$. Our formalization includes tuples, and therefore models tuple inclusion. We claim that F_ι can model record subtyping as well. However, our treatment of records in F_ι would be similar to their treatment in $\mathsf{F}_{<:}$ and require an expressive runtime system so that subtyping is erasable.

Record subtyping in $\mathsf{F}_{<:}$ may also be compiled away into records without subtyping in plain System F by inserting coercions with computational content [Breazu-Tannen et al., 1991] 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 [2000]: bounded polymorphism $\forall(\alpha \leq \tau).\,\sigma$ is compiled away into an intersection type $\forall\alpha.\,\sigma[\alpha \leftarrow \alpha \cap \tau]$ 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. Their work could serve as a reference to extend F_ι with recursive types.

Languages with dependent types often split terms with and without computational content using kinds so that parts of terms that contribute only to the static semantics can be dropped at runtime. This is more powerful than our notion of coercions; for instance, it could allow to build coercions by computation—a feature that we would like to have. However, we do not know whether this approach could be applied and benefit to our extension of F_η.

Coercions introduced in FC_2 [Weirich et al., 2011], the internal language of Haskell, are interesting because they use coercion projections and cannot be expressed in F^λ_ι. Although FC_2 uses a weak evaluation strategy, it can declare abstract coercions at the toplevel, which amount to a form of coercion abstraction—hence they need coercion projections to regain erasability. However, coercions in FC_2 are non-oriented, do not have distributivity nor deep instantiation of quantifiers and are thus *structural*, which allows for an easier setting and a simple criteria to be used for consistency checking. A new version of FC_2 [Vytiniotis and Jones, 2011] makes coercions first-class values in an otherwise comparable setting. Coercions can be abstracted over as in F_ι and also stored in data-structures. However, as a result of being first-class, coercions may change the termination (hence the semantics) of programs and are not erasable in our terminology. The two languages F_ι and FC_2 follow orthogonal approaches and are thus not easily comparable; combining the features of both would be an interesting challenge.

Adding coercion projections to F_ι and taking distributivity away, we could obtain a version much closer to FC_2 but where coercions are oriented. Surprisingly few works have consider distributivity and include the power of F_η, apart from theoretical papers on F_η itself.

Retyping functions can also be seen as a way of rearranging typing derivations. Abstraction over coercions is then abstraction over type derivation transformations. There might be interesting connections to establish with expansion variables for $\forall$-quantifiers introduced by Lenglet and Wells [2010].

9 Discussion and future work

The language F_ι extends F_η with abstraction over coercion functions in a general way where coercions are retyping functions, *i.e.* certain terms of the λ-calculus that do not contribute but may block the evaluation. In order to solve this problem and make coercions erasable, we have proposed two restrictions of F_ι.

Weak F_ι restricts the reduction relation by choosing a weak evaluation strategy for both coercions and terms and restrict coercion abstraction to value forms. The main advantage of this solution is its simplicity and its generality. Still, the restriction of coercion abstractions to value forms, which is analogous to value-only polymorphism in languages with side effects, is significant. Moreover, it allows the abstraction over coercions of uninhabited coercion types, which are never applicable, thus leaving the possibility of non-sensible code hidden under coercion abstraction undetected—or at least delaying its detection.

Instead, F_ι^p restricts the types of coercion parameters and forces them to be polymorphic in either their domain or codomain. The advantage of F_ι^p is to retain a strong reduction relation, which shows that the calculus is really well-behaved. Although restrictive, it already subsumes F_η, $x\mathsf{MLF}$, and $\mathsf{F}_{<:}$. We believe it is an interesting point in the design space. It also shows that an extension of $x\mathsf{MLF}$ with subtyping would be possible and beneficial, even if the question of designing the surface language to make type inference possible remains open.

Still, as both solutions are significant and orthogonal restrictions to F_ι, we may explore other possibilities.

Relaxing F_ι^p Relaxing F_ι^p so that it could type more expressions but still prevent wedges from being typable is probably the easiest extension to this work. An obvious but minor generalization is to let $\lambda(\alpha \diamond \bar{c} : \bar{\tau})\, M$ abstract over several coercions simultaneously, but all with the same polarity. Allowing multiple polarities cannot come without further restrictions, as transitivity could then be used to build an abstract coercion between arrow types.

A more ambitious generalization is to replace the local constraint on the type of coercions by a global constraint defined by some auxiliary consistency judgment. We could allow abstractions of the form $\lambda(\bar{\alpha}, \bar{c} : \bar{\tau} \triangleright \bar{\sigma})\ M$ using a side condition on the typing rule to ensure that the combination of coercions in context still prevents the creation of wedges. However, finding a suitable notion of consistency in the presence of distributivity is challenging.

Beyond F_ι So far, we have explored restrictions of F_ι to prevent wedges from appearing in a reduction context. Instead, we could perhaps extend the calculus to allow breaking them apart. Observe that when a coercion variable appears in a wedge, it is always a coercion between arrow types and that any actual coercion that will be passed at runtime will start with an arrow coercion $G_1 \xrightarrow{\tau} G_2$ that can be decomposed into G_1 and G_2 and pushed out of the way. So, we could decompose the abstract coercion as well, by introducing coercion projections $\mathsf{Left}\, G$ and $\mathsf{Right}\, G$ that behaves as G_1 and G_2 whenever G is $G_1 \xrightarrow{\tau} G_2$.

While this idea is intuitively simple, it is actually quite involved as new difficulties appear one after the other when solving them, due to the presence of distributivity. Projectors require both binding coercions as in F_ι^λ and, independently, a notion of structural equivalence to treat coercions up to some rearrangements; unfortunately, the combination of both breaks confluence; a fix to confluence is to reduce coercions themselves, which introduces further problems! Moreover, even assuming that such a calculus can be set up, there will remain to solve a typechecking problem quite similar to (although more flexible than) the one for relaxing F_ι^p with non-local consistency. Indeed, decomposing nonsensical coercions cannot ensure erasability, ι-reduction may either get stuck, being unsound, or loop forever. We leave this exploration for future work.

Here are some hints on the difficulties to add projectors. Since we can abstract over coercions, we can have in context a coercion c between arbitrary arrow types for which they may not be any actual coercion. For example, we may assume c of coercion type $(\mathit{Int} \to \forall\alpha.\,\alpha) \triangleright (\mathit{Char} \to \mathit{Bool})$. Although, we cannot build any coercion of such type in F_ι, it could be considered valid,

semantically: since $Int \to \forall\alpha.\,\alpha$ is empty, any function of such type can also be treated as a function of type $Char \to Bool$. However, $\mathsf{Left}\, c$ would then have coercion type $Char \triangleright Int$, which is semantically nonsensical. We thus need stronger typing rules to rule out these types from which projection could be meaningless. However, because of distributivity it is not obvious how to syntactically do so. (Removing distributivity is a huge source of simplifications, which might be worth exploring when adding projectors, even if it reduces expressiveness accordingly.)

Here is the intuition why we need binding coercions. Consider the wedge M equal to $c\langle\lambda\alpha\ \lambda(n : Int)\ \lambda(x : \alpha)\ x\rangle\ 3$, typed as follows $c : (\forall\alpha.\, Int \to \alpha \to \alpha) \triangleright (Int \to \tau) \vdash M : \tau$ where τ is $\forall\alpha.\,\alpha \to \alpha$. It erases to $(\lambda n.\lambda x.x)\ 3$ which reduces to $\lambda x.x$. The reduction of M should result in something like $(\mathsf{Right}\, c\langle\lambda\alpha\ \Diamond\rangle)\langle\lambda(x : \alpha)\ x\rangle : \forall\alpha.\,\alpha \to \alpha$ where $\mathsf{Right}\, c\langle\lambda\alpha\ \Diamond\rangle$ binds the type variable α.

We also need structural equivalence. In the previous example we need the two following terms to be equivalent:

$$M_1 = c\langle\lambda\alpha\ \lambda(n : Int)\ \lambda(x : \alpha)\ x\rangle \tag{1}$$
$$M_2 = (c\langle\lambda\alpha\ \Diamond\rangle)\langle\lambda(n : Int)\ \lambda(x : \alpha)\ x\rangle \tag{2}$$

M_1 must reduce to M_2 so that the projection reduction can occur. M_2 must also reduce to M_1 to handle all our previous rules about coercions (like RedType or RedCoerDistTypeArrow). However, we cannot add both reductions simultaneously, as the calculus would not terminate. Instead, one solution is to treat M_1 and M_2 as structurally equivalent. (Notice that we just need an equivalence relation on terms, not on coercions.)

Leaving F_η and freezing quantifiers We have added coercion abstraction to the language F_η as it is the reference in the absence of abstraction. However, many of the difficulties in F_ι come from the distributivity rules, which allow coercions to move quantifiers inside types, or more precisely, from the combination of distributivity with contravariance of the arrow constructor—which is already the source of difficulties in F_η, including undecidability of type-containment. This suggests exploring a restriction of F_ι that does not have distributivity, nor type abstraction and type application of coercions, that would not extend F_η, but have a much simpler metatheory.

Language extensions Several features of programming languages have also been left out of F_ι. We have only included pairs in our presentation of F_ι, but labeled products should work similarly.

We do not expect difficulties with tagged unions or iso-recursive types, *e.g.* following Crary [2000] although details are subtle and still need to be checked. We don't foresee any difficulties for adding fix points to the source language.

Some care is needed for existential types, which already raise a problem in System F as they do not have an erasing semantics with a strong evaluation strategy. Therefore, we left them out of F_ι and replaced them by a top type. This is, however, an orthogonal issue.

An interesting extension is to make coercion first-class objects which raises another challenge for erasability: since coercions can then be built by computation, should a computation that just builds coercions be erasable as well? Coercion types are monomorphic in F_ι but between possibly polymorphic types. We do not expect difficulties to have polymorphic coercion types. First-class coercions would naturally bring polymorphic coercion types.

We have studied coercions for second-order polymorphism. We should not expect difficulties with higher-order polymorphism. However, adding coercions to a language with dependent types may be more challenging.

Conclusions

We have explored extensions of System F_η with abstraction over coercion functions. We have proposed a typed calculus F_ι that strongly normalizes. Coercions do not contribute to the reduction but may block it and are thus non erasable.

We have proposed two restrictions of coercion abstraction to ensure erasability: Weak F_ι prevents evaluation under coercion abstraction while Parametric F_ι prevents coercion variables to appear in the middle of redexes. We believe that Parametric F_ι is an interesting point in the design space, as it factors out several known languages in a simple framework.

Still, Parametric F_ι only permits a limited use of coercion abstraction. We have sketched a few other directions for recovering erasability, which we leave for future work.

Finally, we would like to better understand the logical counter-part of erasable coercions. An intriguing question is a better characterization of the expressiveness of F_ι which is more expressive than F_η which is itself already closed by η-expansion.

References

P. Baldan, G. Ghelli, and A. Raffaetà. Basic theory of F-bounded quantification. *Inf. Comput.*, 153: 173–237, September 1999. URL `http://portal.acm.org/citation.cfm?id=320278.320285`.

V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA'89, pages 273–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. URL `http://doi.acm.org/10.1145/99370.99392`.

L. Cardelli. An implementation of FSub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993. URL `http://research.microsoft.com/Users/luca/Papers/SRC-097.pdf`.

K. 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. ISBN 1-58113-202-6. URL `http://doi.acm.org/10.1145/351240.351247`.

K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6):567–600, 2002. URL `http://dx.doi.org/10.1017/S0956796801004282`.

D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207(6):726–785, 2009. ISSN 0890-5401. URL `http://dx.doi.org/10.1016/j.ic.2008.12.006`.

D. Leijen. A type directed translation of MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*. ACM Press, Oct. 2007. URL `http://research.microsoft.com/users/daan/download/papers/mlftof.pdf`.

D. Leijen and A. Löh. Qualified types for MLF. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 144–155, New York, NY, USA, Sept. 2005. ACM Press. ISBN 1-59593-064-7. URL `http://murl.microsoft.com/users/daan/download/papers/qmlf.pdf`.

S. Lenglet and J. B. Wells. Expansion for forall-quantifiers. Available electronically, 2010. URL `http://sardes.inrialpes.fr/~slenglet/papers/systemFs.pdf`.

G. Manzonetto and P. Tranquilli. Harnessing MLF with the Power of System F. In P. Hlinený and A. Kucera, editors, *Mathematical Foundations of Computer Science 2010, 35th International Symposium, (MFCS)*, volume 6281 of *LNCS*, pages 525–536. Springer, 2010. ISBN 978-3-642-15154-5. doi: http://dx.doi.org/10.1007/978-3-642-15155-2_46.

J. C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3 (76):211–249, 1988.

D. Rémy and B. Yakobowski. A Church-Style Intermediate Language for MLF. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin / Heidelberg, 2010. URL `http://dx.doi.org/10.1007/978-3-642-12251-4_4`.

D. Vytiniotis and S. P. Jones. Practical aspects of evidence-based compilation in system FC. Available electronically, 2011. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/`.

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. URL `http://doi.acm.org/10.1145/1926385.1926411`.

A Delayed Proofs

Proof of Lem 3

The proof is standard. It uses Lemma 2 to ensure well-formedness of Γ from the hypothesis. The second and third cases are proved by mutual induction. ■

Proof of Lem 4

This proof is standard. Cases 4 and 5 are proved by mutual induction, as well as for cases 6 and 7. ■

Proof of Lem 5

The proof is standard and uses lemmas 3 and 4. ■

Proof of Lem 6

The proof is standard and uses Lemma 3. ■

Proof of Lem 7

The proof is standard and uses Lemma 3. ■

Proof of Prop 8

By induction on $M_0 \rightsquigarrow_{\beta\iota} N_0$.

- RedContextBeta and RedContextIota: By case analysis on the context. By inversion of typing, only one typing rule match, and we reuse it along with the induction hypothesis to build the premise in the hole (other premises stay the same) since typechecking is compositional.
- RedTerm: By Lemma 6.
- RedFirst and RedSecond: We use the corresponding subderivation.
- RedCoer: By Lemma 7.
- RedType: By Lemma 4.

- RedCoerArrow: By inversion of typing we have $\Gamma \vdash G_1 : \tau \triangleright \tau'$, $\Gamma \vdash G_2 : \sigma \triangleright \sigma'$, and $\Gamma, (x : \tau') \vdash M : \sigma$. We build $\Gamma, (y : \tau) \vdash G_1 : \tau \triangleright \tau'$ and $\Gamma, (y : \tau) \vdash G_2 : \sigma \triangleright \sigma'$ using Lemma 5 (to show the valid extension) and Lemma 3 where y is fresh for Γ and x. We build $\Gamma, (y : \tau), (x : \tau') \vdash M : \sigma$ using Lemma 5 and 3. We can now use Lemma 6 to build $\Gamma, (y : \tau) \vdash M[x \leftarrow G_1\langle y \rangle] : \sigma'$. The result follows by rules TermCoer and TermTermLam.
- RedCoerDistTypeArrow: We have $\Gamma, \alpha, (x : \tau) \vdash M : \sigma$. We build $\Gamma, (x : \tau), \alpha \vdash M : \sigma$ using Lemma 3. We use $\Gamma \vdash \tau$ (which means $\alpha \notin \mathsf{ftv}(\tau)$) to show that $\Gamma, (x : \tau), \alpha$ is a valid extension of $\Gamma, \alpha, (x : \tau)$.
- RedCoerDistCoerArrow: We have $\Gamma, (c : \rho \triangleright \rho'), (x : \tau) \vdash M : \sigma$. We build $\Gamma, (x : \tau), (c : \rho \triangleright \rho') \vdash M : \sigma$ using Lemma 3. We show $\Gamma \vdash \tau$ using Lemma 7.
- RedCoerProd: Obvious.
- RedCoerDistTypeProd: We have $\Gamma, \alpha \vdash M : \tau$ and $\Gamma, \alpha \vdash N : \sigma$. The result is obvious.
- RedCoerDistCoerProd: We have $\Gamma, (c : \rho \triangleright \rho') \vdash M : \tau$ and $\Gamma, (c : \rho \triangleright \rho') \vdash N : \sigma$. The result is obvious.
- RedCoerDot: Obvious.
- RedCoerFill: By cases on P.
  - $\lambda\alpha\,[]$: We have $\Gamma, \alpha \vdash G : \tau \triangleright \sigma$ and $\Gamma \vdash M : \tau$. We use Lemma 3 to build $\Gamma, \alpha \vdash M : \tau$ (and Lemma 2 to show $\Gamma, \alpha \vdash ok$).
  - $[]\,\tau$: Easy.
  - $G\langle[]\rangle$: Easy.
  - $\lambda(c : \rho \triangleright \rho')\,[]$: We have $\Gamma, (c : \rho \triangleright \rho') \vdash G : \tau \triangleright \sigma$ and $\Gamma \vdash M : \tau$. We use Lemma 3 to build $\Gamma, (c : \rho \triangleright \rho') \vdash M : \tau$ (and Lemma 2 to show $\Gamma, (c : \rho \triangleright \rho') \vdash ok$).
  - $[]\{G\}$: Easy.

■

Proof of Lem 9

This is proved by induction on v. At a higher level, we observe that value forms are partitioned and mapped to a partition of types, hence the mapping can be inverted. ■

Proof of Prop 10

This is a standard proof using Lemma 9. By induction on M_0.

- $C[M]$ when M is not a value: The induction hypothesis applies to M, and we use RedContextBeta or RedContextIota to build $C[M] \rightsquigarrow_{\beta\iota} C[N]$.
- x, $\lambda(x : \tau)\,v$, $\lambda\alpha\,v$, $\lambda(c : \tau_1 \triangleright \tau_2)\,v$, and (v_1, v_2): These are values.
- $\Diamond^\tau$, $G_1 \xrightarrow{\tau} G_2$, $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$, $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}$, $(G_1 * G_2)$, $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}$, $\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\sigma)}$, Top^τ, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\ v_2$: Using Lemma 9 on v_1.
  - p: It is a value.
  - $\lambda(x : \tau)\ v_3$: RedTerm applies.
- $v\,\tau$: Using Lemma 9 on v.
  - p: It is a value.

  - $\lambda\alpha\ v'$: RedType applies.

- $v\{G\}$: Using Lemma 9 on v.
  - p: It is a value.
  - $\lambda(c : \tau_1 \triangleright \tau_2)\, v'$: RedCoer applies.
- $v.1$: Using Lemma 9 on v.
  - p: It is a value.
  - (v_1, v_2): RedFirst applies.
- $v.2$: Using Lemma 9 on v.
  - p: It is a value.
  - (v_1, v_2): RedSecond applies.
- $G\langle v\rangle$: By induction on G.
  - x, $\lambda(x : \tau)\ M$, $M\ N$, (M, N), $M.1$, $M.2$: These expressions are rejected by typing because they are terms and not coercions.
  - c, Top^τ: These are values.
  - $\Diamond^\tau$: RedCoerDot applies.
  - $\lambda\alpha\, W$, $W\,\tau$, $\lambda(c : \tau_1 \triangleright \tau_2)\, W$, $W\{G\}$, and $G\langle W\rangle$: RedCoerFill applies.
  - $G_1 \xrightarrow{\tau} G_2$: Using Lemma 9 on v.
    - p: It is a value.
    - $\lambda(x : \tau)\ v'$: RedCoerArrow applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$: Using Lemma 9 on v.
    - p: It is a value.
    - $\lambda\alpha\ v'$: Using Lemma 9 on v'.
      - p': It is a value.
      - $\lambda(x : \tau)\ v''$: RedCoerDistTypeArrow applies.
  - $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}$: Using Lemma 9 on v.
    - p: It is a value.
    - $\lambda(c : \rho \triangleright \rho')\, v'$: Using Lemma 9 on v'.
      - p': It is a value.
      - $\lambda(x : \tau)\ v''$: RedCoerDistCoerArrow applies.
  - $(G_1 * G_2)$: Using Lemma 9 on v.
    - p: It is a value.
    - (v_1, v_2): RedCoerProd applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}$: Using Lemma 9 on v.
    - p: It is a value.
    - $\lambda\alpha\ v'$: Using Lemma 9 on v'.
      - p': It is a value.
      - (v_1, v_2): RedCoerDistTypeProd applies.
  - $\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\sigma)}$: Using Lemma 9 on v.
    - p: It is a value.
    - $\lambda(c : \rho \triangleright \rho')\, v'$: Using Lemma 9 on v'.
      - p': It is a value.
      - (v_1, v_2): RedCoerDistCoerProd applies.

■

Proof of Lem 12

The proof consists in reducing the translation of every redex. This is just simple computation and all details below could be easily rebuilt by the reader.

1. By induction on $M \rightsquigarrow_\beta N$.
   - RedContextBeta: By case on the context. Reduction contexts are reified on System F contexts, and because we are in strong reduction, all contexts are reduction contexts.
   - RedTerm, RedFirst, and RedSecond: These rules were already in System F and are reified on themselves.
2. By induction on $M \rightsquigarrow_\iota N$.
   - RedContextIota: Same argument as for β-reduction.
   - RedType: This was already a rule of System F and is reified on itself.
   - RedCoerArrow: We have

$$\begin{aligned}
&(\lambda(y : \lceil\tau'\rceil \to \lceil\sigma\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ (y\ (\lceil G_1\rceil\ x)))\ (\lambda(x : \lceil\tau'\rceil)\ \lceil M\rceil)\\
\rightsquigarrow&\lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ ((\lambda(x : \lceil\tau'\rceil)\ \lceil M\rceil)\ (\lceil G_1\rceil\ x))\\
\rightsquigarrow&\lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ M[x \leftarrow \lceil G_1\rceil\ x]
\end{aligned}$$

   - RedCoerDist: We have

$$\begin{aligned}
&(\lambda(y : \forall\alpha.\ \lceil\tau\rceil \to \lceil\sigma\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lambda\alpha\ y\,\alpha\ x)\ (\lambda\alpha\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\\
\rightsquigarrow&\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\ (\lambda\alpha\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\,\alpha\ x\\
\rightsquigarrow&\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\ (\lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x\\
\rightsquigarrow&\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\ \lceil M\rceil
\end{aligned}$$

   - RedCoerDistCoerArrow: We have

$$\begin{aligned}
&(\lambda(y : (\lceil\rho\rceil \to \lceil\rho'\rceil) \to \lceil\tau\rceil \to \lceil\sigma\rceil)\ \lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ y\ x_c\ x)\\
&(\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\\
\rightsquigarrow&\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x_c\ x\\
\rightsquigarrow&\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x\\
\rightsquigarrow&\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lceil M\rceil
\end{aligned}$$

   - RedCoerProd: We have

$$\begin{aligned}
&(\lambda(y : (\lceil\tau\rceil * \lceil\sigma\rceil))\ (\lceil G_1\rceil\ y.\mathtt{1}, \lceil G_2\rceil\ y.\mathtt{2}))\ (\lceil M\rceil, \lceil N\rceil)\\
\rightsquigarrow&(\lceil G_1\rceil\ (\lceil M\rceil, \lceil N\rceil).\mathtt{1}, \lceil G_2\rceil\ (\lceil M\rceil, \lceil N\rceil).\mathtt{2})\\
\rightsquigarrow\rightsquigarrow&(\lceil G_1\rceil\ \lceil M\rceil, \lceil G_2\rceil\ \lceil N\rceil)
\end{aligned}$$

   - RedCoerDistTypeProd: We have

$$\begin{aligned}
&(\lambda(y : \forall\alpha.\ (\tau * \sigma))\ (\lambda\alpha\ (y\,\alpha).\mathtt{1}, \lambda\alpha\ (y\,\alpha).\mathtt{2}))\ (\lambda\alpha\ (\lceil M\rceil, \lceil N\rceil))\\
\rightsquigarrow&(\lambda\alpha\ ((\lambda\alpha\ (\lceil M\rceil, \lceil N\rceil))\,\alpha).\mathtt{1}, \lambda\alpha\ ((\lambda\alpha\ (\lceil M\rceil, \lceil N\rceil))\,\alpha).\mathtt{2})\\
\rightsquigarrow\rightsquigarrow&(\lambda\alpha\ (\lceil M\rceil, \lceil N\rceil).\mathtt{1}, \lambda\alpha\ (\lceil M\rceil, \lceil N\rceil).\mathtt{2})\\
\rightsquigarrow\rightsquigarrow&(\lambda\alpha\ \lceil M\rceil, \lambda\alpha\ \lceil N\rceil)
\end{aligned}$$

- RedCoerDistCoerProd: We have

$$\begin{aligned}
&(\lambda(y : (\lceil\rho\rceil \to \lceil\rho'\rceil) \to (\lceil\tau\rceil * \lceil\sigma\rceil))\\
&\quad (\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (y\ x_c).\mathtt{1}, \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (y\ x_c).\mathtt{2}))\\
&\quad (\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lceil M\rceil, \lceil N\rceil))\\
\rightsquigarrow &(\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ ((\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lceil M\rceil, \lceil N\rceil))\ x_c).\mathtt{1}\\
&, \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ ((\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lceil M\rceil, \lceil N\rceil))\ x_c).\mathtt{2})\\
\rightsquigarrow\rightsquigarrow &(\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lceil M\rceil, \lceil N\rceil).\mathtt{1}\\
&, \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lceil M\rceil, \lceil N\rceil).\mathtt{2})\\
\rightsquigarrow\rightsquigarrow &(\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lceil M\rceil, \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lceil N\rceil)
\end{aligned}$$

- RedCoerDot: We have $(\lambda(x : \lceil\tau\rceil)\ x)\ \lceil M\rceil \rightsquigarrow \lceil M\rceil$
- RedCoerCoer: We have $(\lambda(x : \lceil\tau\rceil)\ P[\lceil G\rceil\ x])\ \lceil M\rceil \rightsquigarrow P[\lceil G\rceil\ \lceil M\rceil]$
- RedCoer: This reifies on a simple RedTerm.

■

Proof of Lem 17

The proof is by simple computation. ■

Proof of Prop 19

The proof is by induction on the reduction and unsurprising. We only detail the most significant cases; other cases are either similarly or easy.

- LRedCoer and LRedHole work with substitution lemmas.
- LRedEtaArrApp: By inversion of typing we have
  - $\Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma$,
  - $\Gamma, \Delta \vdash M : \tau' \to \sigma'$,
  - $\Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau'$,
  - $\Gamma \vdash N : \tau$.

  which leads to the result with weakening, substitution, and LExprTermApp.
- LRedEtaProdFst: By inversion of typing we have $\Gamma; \Delta \star (\phi_1 : \tau) \vdash G_1 : \tau'$ and $\Gamma, \Delta \vdash M : (\tau * \sigma)$.

■

Proof of Prop 21

By induction on the term.

- $C[M]$: If M reduces, then we apply the context rule.
- x, $\lambda(x : \tau)\, v$, (v_1, v_2), $\lambda\alpha\, v$, $\lambda(c : \varphi)\, v$, $\mathtt{Top}^\tau\ v$, and $c\, v$: These are values.
- ϕ: Refused by typing.
- $v_1\ v_2$: By classification of values on v_1.
  - p: This is a value.

  - $\lambda(x:\tau)\,v$: LRedTerm applies.
  - $\lambda(\phi_1:\tau)\,G_2\,\{\phi_2 \leftarrow p\,G_1\}$: LRedEtaArrApp applies.

- $v'.1$ and $v'.2$: By classification of values on v' and LRedFirst, LRedEtaProdFst, LRedSecond, or LRedEtaProdSnd.
- $H\,v'$: By cases on H. If it is a coercion variable c then the whole application is a value, else it is a hole abstraction $\lambda(\phi:\tau)\,G$ and LRedHole applies.
- $v'\,\tau$ and $v'\,H$: By classification of values on v' and LRedType or LRedCoer.
- $\lambda(\phi_1:\tau)\,G_2\,\{\phi_2 \leftarrow p\,G_1\}$ and $(G_1,G_2)\{\phi_1,\phi_2 \leftarrow p\}$: By classification of values on v' and LRedEtaArrLam, LRedEtaArrEtaArr, LRedEtaProdPair, or LRedEtaProdEtaProd. ■

Proof of Lem 24

We first check that G_2, M, and G_1 are used in the correct environments. G_2 is used under Γ, whereas M and G_1 appear under $\lambda\Delta$ which extends their environment to Γ,Δ. Then we check that the term is typed $\tau \to \sigma$ under Γ. We observe that $(G_1\langle\lambda\Delta'\,\Diamond\rangle) \to \Diamond$ is typed $\tau' \to \sigma' \triangleright \tau \to \sigma'$. So its application to M is typed $\tau \to \sigma'$. Abstracting over Δ gives type $\forall\Delta.\,\tau \to \sigma'$. Applying the distributivity returns a term of type $\tau \to \forall\Delta.\,\sigma'$ (this is where we use $\Gamma \vdash \tau$). And finally we apply $\Diamond \to G_2$ which coerces our term into a term of type $\tau \to \sigma$ which was our goal. ■

Proof of Lem 25

The proof is very similar to the previous one but for $G[\diamond \leftarrow \Diamond\ \Delta]$. This allows us to start from type $\forall(\Delta,\Delta'').\,\rho$ instead of $\forall\Delta''.\,\rho$. ■

Proof of Lem 26

Using $M' \rightsquigarrow_\beta N'$, we have that M' is of the form $C[(\lambda(x:\tau)\,M_0)\,M_1]$.

We observe that no ι-reduction rule can reveal a β-redex which was not already present. ■

Proof of Lem 27

Let's call N_0 the ι-normal form of $\hat{M}_0$. ■

Proof of Prop 31

Using Lemma 29 and Lemma 30, we build $\Gamma^\circ \vdash M^\circ : \tau^\circ$ and $M^\circ \rightsquigarrow^+_{\beta\iota} N^\circ$. Using Lemma 8, potentially several times, in F_ι, we build $\Gamma^\circ \vdash N^\circ : \tau^\circ$. Finally, we use Lemma 29 to show $\Gamma \vdash N : \tau$. ■

Proof of Prop 32

Assume we have an infinite reduction path starting from M. Then using Lemma 29 and Lemma 30, we build an infinite reduction path in F_ι. However it contradicts Corollary 13. ■

Proof of Cor 33

There is no critical pairs in F^p_ι. Thus the reduction in F^p_ι is locally confluent. Because it is terminating, it is confluent (Newman). ■

Proof of Lem 34

This holds with a simple induction on v. ■

Proof of Prop 35

This is a standard proof using Lemma 34. The proof is by induction on M. Using RedContextBeta and RedContextIota, we can only consider cases where subterms are values.

- x, $\lambda(x:\tau)\,v$, $\lambda\alpha\,v$, $\lambda(\alpha \Leftrightarrow c:\tau)\,v$, and (v,v): These are values.
- $\Diamond^\tau$, $G \xrightarrow{\tau} G$, $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$, $\mathsf{Dist}^{\forall\alpha\Leftrightarrow\rho\Rightarrow}_{\tau\to\sigma}$, $(G * G)$, $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\sigma)}$, $\mathsf{Dist}^{\alpha\Rightarrow}_{(\rho*\tau)}\sigma$, $\mathtt{Top}^\tau$, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\,v$: Using Lemma 34 on v_1.
  - p: It is a value.
  - $\lambda(x:\tau)\,v$: RedTerm applies.
- $v\,\tau$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda\alpha\,v$: RedType applies.
- $v\{\tau \Leftrightarrow G\}$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda(\alpha \Leftrightarrow c:\tau)\,v$: RedTCoer applies.
- $v.1$: Using Lemma 34 on v.
  - p: It is a value.
  - (v,v): RedFirst applies.
- $v.2$: Using Lemma 34 on v.
  - p: It is a value.
  - (v,v): RedSecond applies.
- $G\langle v\rangle$: Using Lemma 34 on G.
  - x, $\lambda(x:\tau)\,M$, $M\;M$, (M,M), $M.1$, $M.2$: These expressions are rejected by typing because they are terms and not coercions.
  - c, $\mathtt{Top}^\tau$: These are values.
  - $\Diamond^\tau$: RedCoerDot applies.
  - $\lambda\alpha\,W$, $W\,\tau$, $\lambda(\alpha \Leftrightarrow c:\tau)\,W$, $W\{\tau \Leftrightarrow G\}$, and $G\langle W\rangle$: RedCoerFill applies.
  - $G \xrightarrow{\tau} G$: Using Lemma 34 on v.
    - p: It is a value.
    - $\lambda(x:\tau)\,v$: RedCoerArrow applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}$: Using Lemma 34 on v.
    - p: It is a value.
    - $\lambda\alpha\,v$: Using Lemma 34 on v.
      - p: It is a value.
      - $\lambda(x:\tau)\,v$: RedCoerDist applies.

- $\mathsf{Dist}^{\forall\alpha \unlhd \tau \Rightarrow}_{\tau\to\tau}$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda(\alpha \unlhd c : \tau)\, v$: Using Lemma 34 on v.
    - p: It is a value.
    - $\lambda(x : \tau)\, v$: RedCoerDistTCoerArrow applies.
- $(G * G)$: Using Lemma 34 on v.
  - p: It is a value.
  - (v, v): RedCoerProd applies.
- $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda\alpha\, v$: Using Lemma 34 on v.
    - p: It is a value.
    - (v, v): RedCoerDistTypeProd applies.
- $\mathsf{Dist}^{\forall\alpha \unlhd \tau \Rightarrow}_{(\tau*\tau)}$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda(\alpha \unlhd c : \tau)\, v$: Using Lemma 34 on v.
    - p: It is a value.
    - (v, v): RedCoerDistTCoerProd applies.

■

Proof of Lem 36

We simply use Lemma 29 and Lemma 30 to call Lemma 17. ■

Proof of Prop 40

1. By induction on $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$. All the proof are just type-checking.
2. By induction on $A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$. All the proof are just type-checking.

■

Proof of Prop 42

1. By induction on $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$. All the proof are just type-checking.
2. By induction on $A \vdash \phi : T \leq S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$. All the proof are just type-checking.

■

Proof of Prop 43

By induction on $m \rightsquigarrow n$. The context and beta rules are translated on the same rules. Let's consider the remaining rules, which are all the rules involving instantiations. We proceed by inversion of $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ and $A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau$ to get M and N.

- $m\,\mathbf{1} \rightsquigarrow m$: We have $\Diamond^{\tau}\langle M\rangle \rightsquigarrow_{\iota} M$ with RedCoerDot.
- $m\,(\phi;\phi') \rightsquigarrow (m\,\phi)\,\phi'$: We have $(G'\langle G\rangle)\langle M\rangle \rightsquigarrow_{\iota} G'\langle G\langle M\rangle\rangle$ with RedCoerCoer.
- $m\,\Re \rightsquigarrow \Lambda(\alpha \geq \bot)\, m$ (with $\alpha \notin \mathsf{ftv}(m)$): We have $(\lambda(\alpha \lhd c_\alpha : \forall\alpha.\,\alpha)\ \Diamond)\langle M\rangle \rightsquigarrow_{\iota} \lambda(\alpha \lhd c_\alpha : \forall\alpha.\,\alpha)\ M$ with RedCoerCoer.

- $(\Lambda(\alpha \geq T)\, m)\,\& \rightsquigarrow m[!\alpha \leftarrow \mathbf{1}][\alpha \leftarrow T]$: We have with RedCoerCoer and RedCoerPos:

$$\begin{aligned}(\Diamond\{\tau \triangleright \Diamond^{\tau}\})\langle\lambda(\alpha \triangleleft c_\alpha : \tau)\ M\rangle &\rightsquigarrow_\iota (\lambda(\alpha \triangleleft c_\alpha : \tau)\ M)\{\tau \triangleright \Diamond^{\tau}\} \\ &\rightsquigarrow_\iota M[\alpha \leftarrow \tau][c_\alpha \leftarrow \Diamond^{\tau}]\end{aligned}$$

- $(\Lambda(\alpha \geq T)\, m)\,(\forall(\alpha \geq)\phi) \rightsquigarrow \Lambda(\alpha \geq T)\,(m\,\phi)$: We have:

$$\begin{aligned}&(\lambda(\alpha \triangleleft c_\alpha : \tau)\ G\langle\Diamond^{\forall(\alpha \triangleleft \tau) \Rightarrow \tau_1}\{\alpha \triangleright c_\alpha\}\rangle)\langle\lambda(\alpha \triangleleft c_\alpha : \tau)\ M\rangle \\ \rightsquigarrow_\iota &\lambda(\alpha \triangleleft c_\alpha : \tau)\ G\langle(\lambda(\alpha \triangleleft c_\alpha : \tau)\ M)\{\alpha \triangleright c_\alpha\}\rangle \\ \rightsquigarrow_\iota &\lambda(\alpha \triangleleft c_\alpha : \tau)\ G\langle M\rangle\end{aligned}$$

- $(\Lambda(\alpha \geq T)\, m)\,(\forall(\geq \phi)) \rightsquigarrow \Lambda(\alpha \geq T\phi)\, m[!\alpha \leftarrow \phi; !\alpha]$: We have:

$$\begin{aligned}&(\lambda(\alpha \triangleleft c_\alpha : \tau_2)\ \Diamond^{\forall(\alpha \triangleleft \tau_1) \Rightarrow \tau}\{\alpha \triangleright c_\alpha\langle G\rangle\})\langle\lambda(\alpha \triangleleft c_\alpha : \tau)\ M\rangle \\ \rightsquigarrow_\iota &\lambda(\alpha \triangleleft c_\alpha : \tau_2)\ (\lambda(\alpha \triangleleft c_\alpha : \tau)\ M)\{\alpha \triangleright c_\alpha\langle G\rangle\} \\ \rightsquigarrow_\iota &\lambda(\alpha \triangleleft c_\alpha : \tau_2)\ M[c_\alpha \leftarrow c_\alpha\langle G\rangle]\end{aligned}$$

■

Proof of Prop 44

By induction. This is just type-checking. ■

Proof of Prop 45

By induction on $M \rightsquigarrow_{\beta\iota} N$. ■

Proof of Lem 46

By induction on $M \rightsquigarrow_{\beta\iota} N$. We have two syntactic restrictions: first, we removed $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}$; then, we added a value restriction on coercion abstraction. We notice that no rules introduce a $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\tau}$, so we only need to check the preservation of the second restriction:

- RedContextBeta and RedContextIota: By induction on C.
- RedCoerFill: Only the RedCoerCoerLam involves a coercion abstraction and it was modified to work correctly.
- RedCoerDistCoerProd: The pair has to already be a value, so both subterms are values. Hence the resulting term does not break the restriction.
- Remaining rules do not involve coercion abstraction.

■

Proof of Lem 47

Obvious, since the syntax, typing rules, and reduction rules are restrictions of those in F_ι. ■

Proof of Lem 48

The derivation of the judgment in F_ι is a valid derivation in Weak F_ι because typing rules are the same. ■

Proof of Prop 49

Using Lemma 47, we can call Lemma 8 in F_ι, and then use Lemma 48 to come back in Weak F_ι. ■

Proof of Prop 50

Assume we have an infinite reduction path starting from M. Then using Lemma 47, we build an infinite reduction path in F_ι. However it contradicts Corollary 13. ■

Proof of Lem 51

This holds with a simple induction on v. ■

Proof of Lem 52

By induction on M. Using RedContextBeta and RedContextIota, we can only consider cases where subterms are values when reduction contexts allow it. For each reduction context we need to check that it only binds type variables in his hole, which is the case.

- x: This is refused by typing since we only have type variables in the environment.
- $\lambda(x:\tau)\,M$, $\lambda\alpha\,v$, $\lambda(c:\tau\triangleright\tau)\,u$, and (v,v): These are values.
- $\Diamond^\tau$, $G\xrightarrow{\tau}G$, $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$, $(G*G)$, $\mathsf{Dist}^{\forall\alpha.\,(\tau*\sigma)}$, $\mathsf{Dist}^{(\varphi\triangleright\rho')\Rightarrow(\tau*\sigma)}$, Top^τ, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\,v$: Using Lemma 51 on v_1, RedTerm applies.
- $v\,\tau$: Using Lemma 51 on v, RedType applies.
- $v\{G\}$: Using Lemma 51 on v, RedCoer applies.
- $v.1$: Using Lemma 51 on v, RedFirst applies.
- $v.2$: Using Lemma 51 on v, RedSecond applies.
- $G\langle v\rangle$: By induction on G.
  - x, $\lambda(x:\tau)\,M$, $M\;M$, (M,M), $M.1$, $M.2$: These expressions are rejected by typing because they are terms and not coercions.
  - c: This is refused by typing since we only have type variables in the environment.
  - Top^τ: It is a value.
  - $\Diamond^\tau$: RedCoerDot applies.
  - $\lambda\alpha\,W$, $W\,\tau$, $\lambda(c:\tau\triangleright\tau)\,W$, $W\{G\}$, and $G\langle W\rangle$: RedCoerCoer applies.
  - $G\xrightarrow{\tau}G$: Using Lemma 51 on v, RedCoerArrow applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\tau}$: Using Lemma 51 consecutively twice on v, RedCoerDist applies.
  - $(G*G)$: Using Lemma 51 on v, RedCoerProd applies.
  - $\mathsf{Dist}^{\forall\alpha.}_{(\tau*\tau)}$: Using Lemma 51 consecutively twice on v, RedCoerDistTypeProd applies.
  - $\mathsf{Dist}^{\varphi\Rightarrow}_{(\tau*\tau)}$: Using Lemma 51 consecutively twice on v, RedCoerDistCoerProd applies.

■

Proof of Lem 54

We simply use Lemma 47 to call Lemma 17. ■

Centre de recherche INRIA Paris – Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399