

Extending System F-eta with Abstraction over Erasable Coercions

Julien Cretin, Didier Rémy

► **To cite this version:**

Julien Cretin, Didier Rémy. Extending System F-eta with Abstraction over Erasable Coercions. [Research Report] RR-7587, 2011, pp.64. inria-00582570v2

HAL Id: inria-00582570

<https://hal.inria.fr/inria-00582570v2>

Submitted on 13 Jul 2011 (v2), last revised 12 Dec 2011 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Extending System F_η with Abstraction over Erasable Coercions

Julien Cretin — Didier Rémy

N° 7587

July 2011

Domaine 2

 *Rapport
de recherche*

Extending System F_η with Abstraction over Erasable Coercions

Julien Cretin , Didier Rémy

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

Rapport de recherche n° 7587 — July 2011 — 61 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 form 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 , $F_{<}$, and MLF in a unified framework.

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

Étension du Système F_η avec des coercions effaçables abstraites

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 ne permet pas de raisonner sous des hypothèses de sous-typage ni certaines formes d'instanciation retardée. 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 , $xMLF$ et $F_{<}$ 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	12
2.4	Examples	14
3	Properties of F_ι	15
3.1	Soundness	16
3.2	Termination of reduction	18
3.3	Reification of F_ι in System F	18
3.4	Confluence	19
3.5	Forward simulation	19
4	Coercions as retyping functions: F_ι^λ	20
4.1	Definition of F_ι^λ	20
4.2	Soundness	24
4.3	Confluence	25
4.4	Reification into System F	26
4.5	Completeness	26
4.6	Soundness	27
4.7	Bisimulation between F_ι and F_ι^λ	33
5	Parametric F_ι	33
5.1	Syntax changes	33
5.2	Adjustments to the semantics	34
5.3	Properties	35
6	Expressiveness of Parametric F_ι	38
7	Weak F_ι	44
8	Related work	47
9	Discussion and future work	48
A	Delayed Proofs	51

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 $F_{<}$. [Cardelli, 1993]. However, this is just one (relatively easy) spot in the design space. In fact, much work in the 90's has been dedicated 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 present in the language F_{η} , the closure of System F by η -conversion proposed by Mitchell [Mitchell, 1988]. In F_{η} , a retyping context from τ to σ is a one hole context G such that $\lambda(x : \tau) G(x)$ has type $\tau \rightarrow \sigma$ in System F and is an η -expansion of the identity. For now, we write $G\langle M \rangle$ for filling a context G with a term M and \diamond^{τ} for the empty context of type τ . By definition of F_{η} , if G is a retyping context from τ to σ and M is a term of type τ , then $G\langle M \rangle$ is a term of type σ . Contravariance is induced by η -expansion as follows: if G_i is a retyping context from τ_i to τ'_i for i in $\{1, 2\}$, then $\lambda(x : \tau_1) G_2\langle \diamond^{\tau'_1 \rightarrow \tau_2} (G_1\langle x \rangle) \rangle$ is a retyping context from type $\tau'_1 \rightarrow \tau_2$ to $\tau_1 \rightarrow \tau'_2$.

Besides contravariance, η -expansion also introduces opportunities for inserting type abstractions and type applications, which may change polymorphism a posteriori. For instance, $\lambda(x : \forall \alpha. \tau) \lambda \beta \diamond^{\forall \alpha. \tau \rightarrow \sigma} \rho (x \rho)$ is a retyping context from $\forall \alpha. \tau \rightarrow \sigma$ to $(\forall \alpha. \tau) \rightarrow (\forall \beta. \sigma[\alpha \leftarrow \rho])$ provided β do not appear free in $\forall \alpha. \sigma$. Such retypings are not supported in $F_{<}$: where polymorphism can only be introduced and eliminated explicitly at the topmost part of expressions.

Conversely, $F_{<}$ allows reasoning under subtyping assumptions, which F_{η} does not support. Indeed, bounded quantification $\Lambda(\alpha <: \tau) M$ of $F_{<}$ 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.

Is there a language that supersedes both F_{η} and $F_{<}$? 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 (since F_{η} does not support it). Bounded type-instantiation has also similarities with bounded quantification of $F_{<}$, 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} , $F_{<}$, 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 simultaneously combined together. This question has in fact already been raised in some of the works on MLF [Rémy and Jakobowski, 2010, Manzonetto and Tranquilli, 2010].

Our contributions We answer positively by introducing a language F_l^p that extends F_η with abstraction over retyping functions, combining all features simultaneously in a unified framework (§ 5). The language F_l^p subsumes F_η , $F_{<:}$, and MLF (§6). The language F_l^p also fixes and extends a previous language of coercions designed for modeling MLF alone [Manzonetto and Tranquilli, 2010]. Our subset of F_l^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_l of which F_l^p is a (drastic) restriction (§2). The language F_l is well-behaved. We show that F_l can be simulated into System F. Hence, reduction rules in F_l are just particular instances of β -reduction (§4). F_l 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 if we choose a weak reduction strategy (§7), but this is not quite satisfactory either. Hence, other restrictions or extensions of F_l with erasable coercions are still to be found. We believe that F_l is a solid ground for understanding and further studies of erasable coercions (§9).

System F as the origin All languages we consider are second order calculi which have System F at their origin. 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.* equal); and conversely, if the erasure of a term M β -reduces to a term m' , then M also reduces by a sequence of ι -reductions followed by a single β -reduction to a term whose erasure is 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 during compilation for efficiency of program execution after its compilation, and also for providing the source language with an untyped semantics which is generally simpler to understand and deal with. (We may argue that when the source language does not have an untyped semantics, it should first be compiled in a type-dependent way to an intermediate language that itself has an untyped semantics.)

From types to type conversions Our approach to coercions is similar because we focus here on retyping functions that we wish to be erasable. In some circumstances, one may use *other* forms of coercions that may, *e.g.*, change the representation of values and thus have some computational content and not be erasable. As for types, we should then compile source expressions into an intermediate language where remaining coercions, if any, are erasable, and 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 to understand that types do not determine the semantics of their programs. At least, we should give them an intermediate representation in which this is true.

Coercions may also be introduced a posteriori to make type conversions explicit inside source terms, in which case they must be erasable not to alter the semantics of the language. 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 which are only objects of the mathematical discourse.

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 prior 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 while, conversely, properties for weak reduction strategies do not say much about strong reduction strategies.

The two faces of \mathbf{F}_η Let us first return to the definition of \mathbf{F}_η , which in Mitchell’s original presentation is given in curry style. It thus suffices to add to curry-style System F a *type containment* rule that allows to convert a term m of type τ to one of type σ provided the existence of a retyping context from type τ to σ , written $\vdash \tau \triangleright \sigma$. This judgment, called type containment, is equivalent to the existence of a (closed) retyping function m' of System F such that $\vdash m' : \tau \rightarrow \sigma$. 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, $G_1 \rightarrow G_2$ is a coercion that, given a function M , returns a function that coerces its argument with G_1 , pass it to M , and coerces the result with G_2 —using the contravariance of type containment. (A full presentation of coercions appears in §2 where \mathbf{F}_η is described as a subset of \mathbf{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—and 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. For 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 \mathbf{F}_ι To abstract over coercion functions, we introduce a new form $\lambda(c : \tau \triangleright \sigma) M$ in \mathbf{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 \mathbf{F}_ι and of their erasure in the untyped λ -calculus. Moreover, \mathbf{F}_ι has the subject reduction property and is strongly normalizing. Still, coercions cannot be erased in \mathbf{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 \mathbf{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. 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 $(\lambda(x : \tau) [M]) [N]$ can be reduced, immediately. Hence, the existence of wedges in evaluation contexts prevents erasability.

Taming coercions in \mathbf{F}_ι^p An obvious solution to recover erasability is to make wedge configurations ill-typed—so that they will never appear during the reduction of well-typed programs. One interesting restriction, called \mathbf{F}_ι^p (read parametric \mathbf{F}_ι), is to request that coercion parameters

x, y		variables	
$m ::= x \mid \lambda x.m \mid m m \mid (m, m) \mid m.1 \mid m.2$		terms	
$e ::= \lambda x.\square \mid \square m \mid m \square \mid (\square, m) \mid (m, \square) \mid \square.1 \mid \square.2$		evaluation contexts	
$\frac{\text{REDCONTEXT}}{m \rightsquigarrow m'}$ $\frac{}{e[m] \rightsquigarrow e[m']}$	$\frac{\text{REDBETA}}{(\lambda x.m) m' \rightsquigarrow m[x \leftarrow m']}$	$\frac{\text{REDPROJFIRST}}{(m_1, m_2).1 \rightsquigarrow m_1}$	$\frac{\text{REDPROJSECOND}}{(m_1, m_2).2 \rightsquigarrow m_2}$

Figure 1: λ -calculus: syntax and semantics

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 an evaluation context any more. This variant is called F_ι^w (read weak F_ι).

As neither restriction is quite satisfactory, 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 remind the definition of the (untyped) λ -calculus on Figure 1, so as to be self-contained. 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 type variables, ranged over by letters x and y . Untyped terms, written m , include variables, abstractions $\lambda x.m$, applications $m m'$, pairs (m, m') , and projections $m.1$ and $m.2$. The semantics of untyped λ -terms is given by a small-step strong reduction relation. Evaluation contexts of the λ -calculus are all one-hole contexts, written e . We now write $e[m]$ for the term obtained by filling the hole of e with m and $m[x \leftarrow m']$ for the capture avoiding substitution of m' for x in m and reserved the notation used in the introduction for the concrete syntax of coercion application. 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 $\tau \rightarrow \tau$, 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.

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 , abstraction $\lambda(x : \tau) M$, application MM , pair (M, M) , projection $M.1$ and $M.2$, type abstraction $\lambda \alpha M$, and type application $M \tau$. 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

$\tau, \sigma ::= \alpha \mid \tau \rightarrow \tau \mid (\tau * \tau) \mid \forall \alpha. \tau \mid \top$	Types
$\mid \varphi \Rightarrow \tau$	<i>coercion abstraction</i>
$\varphi ::= \tau \triangleright \tau$	coercion type
$M, N ::= x \mid \lambda(x : \tau) M \mid M M \mid (M, M) \mid M.1 \mid M.2$	Terms
$\mid \lambda \alpha M \mid M \tau$	<i>type abs ℰ app</i>
$\mid G \langle M \rangle$	<i>term coercion</i>
$\mid \lambda(c : \varphi) M \mid M \{G\}$	<i>coercion abs ℰ app</i>
$G ::= c \mid \text{Top}^\tau \mid \diamond^\tau \mid G \xrightarrow{\tau} G \mid (G * G)$	Coercions
$\mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha.}$	<i>distributivity</i>
$\mid \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow}$	<i>type abs ℰ app</i>
$\mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha.}$	<i>coercion coercion</i>
$\mid \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow}$	<i>coercion abs ℰ app</i>
$\mid \lambda \alpha G \mid G \tau$	<i>type abs ℰ app</i>
$\mid G \langle G \rangle$	<i>coercion coercion</i>
$\mid \lambda(c : \varphi) G \mid G \{G\}$	<i>coercion abs ℰ app</i>
$\Gamma ::= \emptyset \mid \Gamma, \alpha \mid \Gamma, x : \tau \mid \Gamma, c : \varphi$	Typing environments

Figure 2: Syntax of F_L .

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 justifies 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.

The coercion variable c just stands for the coercion it will be bound to. The opaque coercion Top^τ is a downgraded version of existential types; it turns a term of any type into an opaque term of type \top that can only be used abstractly. The empty coercion \diamond^τ stands for the empty retyping context and witnesses reflexivity of type containment. The arrow coercion $G_1 \xrightarrow{\tau} G_2$ stands for $\lambda(x : \tau) G_2 \langle \lambda (G_1 \langle x \rangle) \rangle$ and witnesses contravariance of the arrow type constructor. The distributivity coercion $\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha.}$ stands for $\lambda(x : \tau) \lambda \alpha \langle \alpha x \rangle$ and permutes a type abstraction with a term abstraction: assuming the hole has type $\forall \alpha. \tau \rightarrow \sigma$ where α does not appear free in τ , it returns a term of type $\tau \rightarrow \forall \alpha. \sigma$. For instance, if the term is a polymorphic function $\lambda \alpha \lambda(y : \tau) N$, it makes it look like if it had been defined as $\lambda(y : \tau) \lambda \alpha N$ (and this is what it will reduce to). The other distributivity coercion $\text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow}$, which stands for $\lambda(x : \tau) \lambda(c : \varphi) (\langle \{c\} x \rangle)$ is similar but permutes a coercion abstraction with a term abstraction.

The product coercion $(G_1 * G_2)$ stands for $(G_1 \langle \cdot \rangle.1, G_2 \langle \cdot \rangle.2)$ and allows congruence on the product type constructor. The distributivity coercion $\text{Dist}_{(\tau * \sigma)}^{\forall \alpha.}$ stands for $(\lambda \alpha (\langle \alpha \rangle.1), \lambda \alpha (\langle \alpha \rangle.2))$ 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 $\text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow}$, which stands for $(\lambda(c : \varphi) (\langle \{c\} \rangle.1), \lambda(c : \varphi) (\langle \{c\} \rangle.2))$ is similar but permutes a coercion abstraction with a pair construct.

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 $\langle \cdot \rangle \rightarrow \langle \cdot \rangle$, the positive (covariant) hole is the right one, while the negative (contravariant) one is the left one. When b is $(\langle \cdot \rangle * \langle \cdot \rangle)$, both holes are positive. This is why we get $(a \langle \cdot \rangle * a \langle \cdot \rangle)$. This heavy-weighted distributivity

$$\begin{aligned}
[x] &= x \\
[\lambda(x : \tau) M] &= \lambda x. [M] & [\lambda \alpha M] &= [M] \\
[M N] &= [M] [N] & [M \tau] &= [M] \\
[(M, N)] &= ([M], [N]) & [G\langle M \rangle] &= [M] \\
[M.1] &= [M].1 & [\lambda(c : \varphi) M] &= [M] \\
[M.2] &= [M].2 & [M\{G\}] &= [M]
\end{aligned}$$

Figure 3: Coercion erasure

$$\begin{array}{c}
\text{TERMVAR} \\
\frac{\Gamma \vdash ok \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\\
\text{TERMPAIR} \\
\frac{\Gamma \vdash M_1 : \tau_1 \quad \Gamma \vdash M_2 : \tau_2}{\Gamma \vdash (M_1, M_2) : (\tau_1 * \tau_2)} \\
\\
\text{TERMTYPEAPP} \\
\frac{\Gamma \vdash M : \forall \alpha. \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash M \sigma : \tau[\alpha \leftarrow \sigma]} \\
\\
\text{TERMTERMLAM} \\
\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma} \\
\\
\text{TERMTERMAPP} \\
\frac{\Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M N : \sigma} \\
\\
\text{TERMFIRST} \\
\frac{\Gamma \vdash M : (\tau_1 * \tau_2)}{\Gamma \vdash M.1 : \tau_1} \\
\\
\text{TERMSECOND} \\
\frac{\Gamma \vdash M : (\tau_1 * \tau_2)}{\Gamma \vdash M.2 : \tau_2} \\
\\
\text{TERMCOER} \\
\frac{\Gamma \vdash G : \tau \triangleright \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash G\langle M \rangle : \sigma} \\
\\
\text{TERMCOERLAM} \\
\frac{\Gamma, c : \varphi \vdash M : \tau}{\Gamma \vdash \lambda(c : \varphi) M : \varphi \Rightarrow \tau} \\
\\
\text{TERMCOERAPP} \\
\frac{\Gamma \vdash M : \varphi \Rightarrow \tau \quad \Gamma \vdash G : \varphi}{\Gamma \vdash M\{G\} : \tau}
\end{array}$$

Figure 4: System F_i : term typings

mechanism might in the end overcome the difficulties about binding coercions in F_i^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, coercion variables and 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 coercions forms which become vacuous.

The coercion erasure, written $[\cdot]$, defined on Figure 3, is as expected: type annotations on function parameters and coercions are erased, 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 .

$$\begin{array}{c}
\text{COERDOT} \\
\frac{\Gamma \vdash \tau}{\Gamma \vdash \diamond^\tau : \tau \triangleright \tau} \\
\\
\text{COERFORGET} \\
\frac{\Gamma \vdash \tau}{\Gamma \vdash \text{Top}^\tau : \tau \triangleright \top} \\
\\
\text{COERARROW} \\
\frac{\Gamma \vdash G_1 : \tau_1 \triangleright \tau'_1 \quad \Gamma \vdash G_2 : \tau_2 \triangleright \tau'_2}{\Gamma \vdash G_1 \xrightarrow{\tau_3} G_2 : (\tau'_1 \rightarrow \tau_2) \triangleright (\tau_1 \rightarrow \tau'_2)} \\
\\
\text{COERDISTTYPEARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha} : \forall \alpha. (\tau \rightarrow \sigma) \triangleright \tau \rightarrow \forall \alpha. \sigma} \\
\\
\text{COERDISTCOERARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \varphi \quad \Gamma \vdash \sigma}{\Gamma \vdash \text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow} : (\varphi \Rightarrow (\tau \rightarrow \sigma)) \triangleright (\tau \rightarrow (\varphi \Rightarrow \sigma))} \\
\\
\text{COERPAIR} \\
\frac{\Gamma \vdash G_1 : \tau_1 \triangleright \tau'_1 \quad \Gamma \vdash G_2 : \tau_2 \triangleright \tau'_2}{\Gamma \vdash (G_1 * G_2) : (\tau_1 * \tau_2) \triangleright (\tau'_1 * \tau'_2)} \\
\\
\text{COERDISTTYPEPROD} \\
\frac{\Gamma, \alpha \vdash \tau \quad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \text{Dist}_{(\tau * \sigma)}^{\forall \alpha} : \forall \alpha. (\tau * \sigma) \triangleright (\forall \alpha. \tau * \forall \alpha. \sigma)} \\
\\
\text{COERDISTCOERPROD} \\
\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow} : (\varphi \Rightarrow (\tau * \sigma)) \triangleright ((\varphi \Rightarrow \tau) * (\varphi \Rightarrow \sigma))} \\
\\
\text{COERTYPELAM} \\
\frac{\Gamma \vdash \tau \quad \Gamma, \alpha \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda \alpha G : \tau \triangleright \forall \alpha. \sigma} \\
\\
\text{COERTYPEAPP} \\
\frac{\Gamma \vdash G : \tau' \triangleright \forall \alpha. \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash G \tau : \tau' \triangleright \sigma[\alpha \leftarrow \tau]} \\
\\
\text{COERCOER} \\
\frac{\Gamma \vdash G : \tau \triangleright \sigma \quad \Gamma \vdash G' : \rho \triangleright \tau}{\Gamma \vdash G(G') : \rho \triangleright \sigma} \\
\\
\text{COERCOERLAM} \\
\frac{\Gamma, c : \varphi \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda(c : \varphi) G : \tau \triangleright (\varphi \Rightarrow \sigma)} \\
\\
\text{COERCOERAPP} \\
\frac{\Gamma \vdash G' : \tau \triangleright (\varphi \Rightarrow \sigma) \quad \Gamma \vdash G : \varphi}{\Gamma \vdash G'\{G\} : \tau \triangleright \sigma} \\
\\
\text{COERVAR} \\
\frac{\Gamma \vdash ok \quad c : \varphi \in \Gamma}{\Gamma \vdash c : \varphi}
\end{array}$$

Figure 5: System F_c : coercion typings

$$\begin{array}{c}
\text{TYPEVAR} \\
\frac{\Gamma \vdash ok \quad \alpha \in \text{dom}(\Gamma)}{\Gamma \vdash \alpha} \\
\\
\text{TYPEARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \rightarrow \sigma} \\
\\
\text{COERTYPE} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \triangleright \sigma} \\
\\
\text{TYPEPROD} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash (\tau * \sigma)} \\
\\
\text{TYPEFORALL} \\
\frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha. \tau} \\
\\
\text{TYPECOERARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \varphi}{\Gamma \vdash \varphi \Rightarrow \tau} \\
\\
\text{ENVCOER} \\
\frac{c \notin \text{dom}(\Gamma) \quad \Gamma \vdash \varphi}{\Gamma, (c : \varphi) \vdash ok} \\
\\
\text{ENVEMPTY} \\
\emptyset \vdash ok \\
\\
\text{ENVTYPE} \\
\frac{\Gamma \vdash ok \quad \alpha \notin \text{dom}(\Gamma)}{\Gamma, \alpha \vdash ok} \\
\\
\text{ENVTERM} \\
\frac{\Gamma \vdash \tau \quad x \notin \text{dom}(\Gamma)}{\Gamma, (x : \tau) \vdash ok}
\end{array}$$

Figure 6: System F : well-formedness rules

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 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 $\tau \rightarrow \sigma$. 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 . Hence, 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 $\tau'_1 \rightarrow \tau_2$ into an arrow type $\tau_1 \rightarrow \tau'_2$, provided G_i coerces type τ_i into τ'_i for i in $\{1, 2\}$. The distributivity coercion $\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha}$ turns an expression of type $\forall \alpha. \tau \rightarrow \sigma$ into one of type $\tau \rightarrow \forall \alpha. \sigma$ 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 $\text{Dist}_{(\tau * \sigma)}^{\forall \alpha}$ 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 and not a polymorphic coercion. 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 $\varphi \triangleright (\tau_1 \triangleright \tau_2)$ as one could naively expect—which would be an ill-formed type. Type and coercion applications (rules `COERTYPEAPP` and `COERCOERAPP`) are typed accordingly.

The typing rules for F_η are obtained by removing `TERMCOERLAM` and `TERMCOERAPP` for terms and their counterparts `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_l is in church-style. Curry-style F_l is the image of F_l by coercion erasure. That is, it is the subset of terms of the untyped λ -calculus that are the erasure of a term of

$p ::= x \mid p v \mid p.1 \mid p.2 \mid p \tau \mid p\{G\} \mid c\langle v \rangle$	Prevalues
$\mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha.} \langle p \rangle \mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha.} \langle \lambda \alpha p \rangle \mid (G \xrightarrow{\tau} G) \langle p \rangle$ $\mid \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow} \langle p \rangle \mid \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow} \langle \lambda(c : \varphi) p \rangle$ $\mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha.} \langle p \rangle \mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha.} \langle \lambda \alpha p \rangle \mid (G * G) \langle p \rangle$ $\mid \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow} \langle p \rangle \mid \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow} \langle \lambda(c : \varphi) p \rangle$	
$v ::= p \mid \lambda(x : \tau) v \mid (v, v) \mid \lambda \alpha v \mid \lambda(c : \varphi) v \mid \text{Top}^\tau \langle v \rangle$	Values
$E ::= \lambda(x : \tau) \square \mid \square M \mid M \square \mid (\square, M) \mid (M, \square) \mid \square.1 \mid \square.2 \mid P$	Evaluation contexts
$P ::= \lambda \alpha \square \mid \square \tau \mid G \langle \square \rangle \mid \lambda(c : \varphi) \square \mid \square \{G\}$	Retyping contexts

Figure 7: System F_l : values and evaluation contexts

$\frac{\text{REDCONTEXTBETA} \quad M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}$	$\frac{\text{REDCONTEXTIOTA} \quad M \rightsquigarrow_l N}{E[M] \rightsquigarrow_l E[N]}$	$\frac{\text{REDTERM} \quad (\lambda(x : \tau) M) N \rightsquigarrow_\beta M[x \leftarrow N]}{}$	$\frac{\text{REDFIRST} \quad (M, N).1 \rightsquigarrow_\beta M}{}$
$\frac{\text{REDSECOND} \quad (M, N).2 \rightsquigarrow_\beta N}{}$	$\frac{\text{REDTYPE} \quad (\lambda \alpha M) \tau \rightsquigarrow_l M[\alpha \leftarrow \tau]}{}$	$\frac{\text{REDCOER} \quad (\lambda(c : \varphi) M)\{G\} \rightsquigarrow_l M[c \leftarrow G]}{}$	
$\frac{\text{REDCOERARROW} \quad (G_1 \xrightarrow{\tau} G_2) \langle \lambda(x : \sigma) M \rangle \rightsquigarrow_l \lambda(x : \tau) G_2 \langle M[x \leftarrow G_1 \langle x \rangle] \rangle}{}$			
$\frac{\text{REDCOERDISTTYPEARROW} \quad \text{Dist}_{\tau' \rightarrow \sigma'}^{\forall \alpha.} \langle \lambda \alpha \lambda(x : \tau) M \rangle \rightsquigarrow_l \lambda(x : \tau) \lambda \alpha M}{}$			
$\frac{\text{REDCOERDISTCOERARROW} \quad \text{Dist}_{\tau' \rightarrow \sigma'}^{\varphi' \Rightarrow} \langle \lambda(c : \varphi) \lambda(x : \tau) M \rangle \rightsquigarrow_l \lambda(x : \tau) \lambda(c : \varphi) M}{}$	$\frac{\text{REDCOERPROD} \quad (G_1 * G_2) \langle (M, N) \rangle \rightsquigarrow_l (G_1 \langle M \rangle, G_2 \langle N \rangle)}{}$		
$\frac{\text{REDCOERDISTTYPEPROD} \quad \text{Dist}_{(\tau' * \sigma')}^{\forall \alpha.} \langle \lambda \alpha (M, N) \rangle \rightsquigarrow_l (\lambda \alpha M, \lambda \alpha N)}{}$			
$\frac{\text{REDCOERDISTCOERPROD} \quad \text{Dist}_{(\tau' * \sigma')}^{\varphi' \Rightarrow} \langle \lambda(c : \varphi) (M, N) \rangle \rightsquigarrow_l (\lambda(c : \varphi) M, \lambda(c : \varphi) N)}{}$	$\frac{\text{REDCOERDOT} \quad \diamond^\tau \langle M \rangle \rightsquigarrow_l M}{}$		
$\frac{\text{REDCOERFILL} \quad (P[G]) \langle M \rangle \rightsquigarrow_l P[G \langle M \rangle]}{}$			

Figure 8: Reduction rules for F_l

church-style System F_l . We write $\Gamma \vdash m : \tau$ to mean that there exists M such that $\Gamma \vdash M : \tau$ and $\llbracket M \rrbracket$ is m .

2.3 Dynamic semantics

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

A value is an abstraction of a value, a pair of values, an opaque value, or a prevalue. A prevalue is a variable, an application of a prevalue 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. Evaluation contexts E are all one-hole term contexts. However, for convenience, we have distinguished a subset of evaluation contexts P , called *retyping evaluation contexts*: a term M placed in a retyping evaluation 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_{β} and \rightsquigarrow_{ι} .

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.

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, becoming a term applied to a coercion, so that rule REDCOER 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 $\text{Top}^{\tau}\langle v \rangle$ 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 $\text{Dist}_{\tau' \rightarrow \sigma'}^{\forall \alpha}$, and M is a polymorphic function $\lambda \alpha \lambda(x : \tau) M$, Rule REDCOERDISTTYPEARROW reduces the application by exchanging the type and value parameter to $\lambda(x : \tau) \lambda \alpha M$, which is sound since α cannot be free in τ .
- When G is a distributivity coercion $\text{Dist}_{\tau' \rightarrow \sigma'}^{\varphi' \Rightarrow}$, and M is a coercion abstraction followed by a value abstraction $\lambda(c : \varphi) \lambda(x : \tau) M$, Rule REDCOERDISTCOERARROW similarly exchanges the parameters to $\lambda(x : \tau) \lambda(c : \varphi) M$.

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, $\text{Dist}_{(\tau' * \sigma')}^{\forall \alpha}$, 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 $\text{Dist}_{(\tau' * \sigma')}^{\varphi' \Rightarrow}$ 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:

$$\begin{array}{ll}
(\lambda \alpha G)\langle M \rangle \rightsquigarrow_{\iota} \lambda \alpha (G\langle M \rangle) & \text{REDCOERTYPELAM} \\
(G \tau)\langle M \rangle \rightsquigarrow_{\iota} (G\langle M \rangle) \tau & \text{REDCOERTYPEAPP} \\
(G_2\langle G_1 \rangle)\langle M \rangle \rightsquigarrow_{\iota} G_2\langle G_1\langle M \rangle \rangle & \text{REDCOERCOER} \\
(\lambda(c : \varphi) G)\langle M \rangle \rightsquigarrow_{\iota} \lambda(c : \varphi) (G\langle M \rangle) & \text{REDCOERCOERLAM} \\
(G_1\{G_2\})\langle M \rangle \rightsquigarrow_{\iota} (G_1\langle M \rangle)\{G_2\} & \text{REDCOERCOERAPP}
\end{array}$$

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:

$$\begin{aligned} (\text{Dist}_{\tau' \rightarrow \sigma'}^{\forall \alpha.} \langle M \rangle) N \rho &\rightsquigarrow_\iota M \rho N & (\text{Dist}_{\tau' \rightarrow \sigma'}^{\forall \alpha.} \langle \lambda \alpha M \rangle) N &\rightsquigarrow_\iota \lambda \alpha M N \\ (\text{Dist}_{\tau' \rightarrow \sigma'}^{\varphi'} \langle M \rangle) N G &\rightsquigarrow_\iota M G N & (\text{Dist}_{\tau' \rightarrow \sigma'}^{\varphi'} \langle \lambda(c : \varphi) M \rangle) N &\rightsquigarrow_\iota \lambda(c : \varphi) M N \\ \text{REDCOERARROWAPP} & & & \\ ((G_1 \xrightarrow{\tau} G_2)\langle M \rangle) N &\rightsquigarrow_\iota G_2 \langle M (G_1 \langle N \rangle) \rangle & & \\ ((G_1 * G_2)\langle M \rangle).1 &\rightsquigarrow_\iota G_1 \langle M.1 \rangle & ((G_1 * G_2)\langle M \rangle).2 &\rightsquigarrow_\iota G_2 \langle M.2 \rangle \\ (\text{Dist}_{(\tau' * \sigma')}^{\forall \alpha.} \langle M \rangle).1 \rho &\rightsquigarrow_\iota (M \rho).1 & (\text{Dist}_{(\tau' * \sigma')}^{\forall \alpha.} \langle M \rangle).2 \rho &\rightsquigarrow_\iota (M \rho).2 \\ (\text{Dist}_{(\tau' * \sigma')}^{\forall \alpha.} \langle \lambda \alpha M \rangle).1 &\rightsquigarrow_\iota \lambda \alpha M.1 & (\text{Dist}_{(\tau' * \sigma')}^{\forall \alpha.} \langle \lambda \alpha M \rangle).2 &\rightsquigarrow_\iota \lambda \alpha M.2 \\ (\text{Dist}_{(\tau' * \sigma')}^{\varphi'} \langle M \rangle).1 \{G\} &\rightsquigarrow_\iota (M \{G\}).1 & (\text{Dist}_{(\tau' * \sigma')}^{\varphi'} \langle M \rangle).2 \{G\} &\rightsquigarrow_\iota (M \{G\}).2 \\ (\text{Dist}_{(\tau' * \sigma')}^{\varphi'} \langle \lambda(c : \varphi) M \rangle).1 &\rightsquigarrow_\iota \lambda(c : \varphi) M.1 & (\text{Dist}_{(\tau' * \sigma')}^{\varphi'} \langle \lambda(c : \varphi) M \rangle).2 &\rightsquigarrow_\iota \lambda(c : \varphi) M.2 \end{aligned}$$

However, this narrows the set of values and reestablishing progress would require *binding coercions*, as for F_ι^λ described in §4, which are technically more involved. The current presentation has fewer, but sufficiently many, reduction paths, for sake of simplicity. To better understand why we need binding coercions, let's forget about pairs and focus on arrows. The rule REDCOERARROWAPP is telling us that $(G_1 \xrightarrow{\tau} G_2)\langle M \rangle$ behaves like a arrow constructor, since when it is under the arrow destructor (which is the application node) they reduce. This morally means that $(G_1 \xrightarrow{\tau} G_2)\langle M \rangle$ 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 $(G_1 \xrightarrow{\tau} G_2)\langle M \rangle$ everywhere we previously had a $\lambda(x : \tau) M'$, and in particular in REDCOERDIST. So we have to define the reduction of $\text{Dist}_{\tau'' \rightarrow \sigma''}^{\forall \alpha.} \langle \lambda \alpha (G_1 \xrightarrow{\tau} G_2)\langle M \rangle \rangle$ where $\Gamma, \alpha \vdash M : \tau' \rightarrow \sigma'$, $\Gamma, \alpha \vdash G_1 : \tau \triangleright \tau'$, and $\Gamma, \alpha \vdash G_2 : \sigma' \triangleright \sigma$. We would like to say it reduces to $(G_1 \xrightarrow{\tau} (\lambda \alpha G_2))\langle M \rangle$ 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 allows 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:

$$\begin{aligned} \forall \alpha. \forall \beta. \forall \gamma. (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma \\ (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\forall \alpha. \alpha \rightarrow \alpha) \end{aligned}$$

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:

$$\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \text{List } \beta \quad (1)$$

It can also be given the type

$$(\forall \alpha. \alpha \rightarrow D \alpha) \rightarrow \forall \alpha. \text{List } (D \alpha) \rightarrow \text{List } (D(D \alpha)) \quad (2)$$

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

$$\langle (\diamond \rightarrow \lambda \alpha \diamond (D \alpha)) \langle \text{Dist}^\forall \rangle \langle \lambda \alpha (\diamond \alpha \rightarrow \diamond) \langle \diamond \alpha (D \alpha) \rangle \rangle \rangle$$

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

$$\forall \alpha. (\forall \alpha. \alpha \rightarrow D \alpha) \rightarrow \text{List } (\alpha) \rightarrow \text{List } (D \alpha) \quad (3)$$

which in turn $\langle \diamond \rightarrow \lambda \alpha \diamond (D \alpha) \rangle \langle \text{Dist}^\forall \rangle$ 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 next example illustrates coercion abstraction; Tu veux parler de l'exemple $(a1^*(a2^*(..(an*b)..)) \triangleright (a1^*(a2^*(..(an*Top)..)))$? J'avais du le manquer. Le probleme de cet exemple c'est que l'on ne peut pas ecrire de coercions polymorphe dans Fiota. En revanche pour chaque ensemble de types $t1, \dots, tn$ et s on a une coercion de la chaine de tuple $t1^*..*tn$ terminée par s en la chaine de tuple $t1^*..*tn$ terminée par top . Voila une coercion en question :

$$\langle \diamond^{\tau_1} * (\diamond^{\tau_2} * (\dots (\diamond^{\tau_n} * \text{Top}^\sigma) \dots)) \rangle$$

de type :

$$\langle \tau_1 * (\tau_2 * (\dots (\tau_n * \sigma) \dots)) \rangle \triangleright \langle \tau_1 * (\tau_2 * (\dots (\tau_n * \top) \dots)) \rangle$$

Je ne comprends pas ce que tu veux dire par "on ne peut pas ecrire de coercion polymorphe". On peut bien ecrire une coercion de type

$$\forall \beta_1. \forall \beta_2. \forall \alpha. (\alpha \triangleright (\beta_1 * (\beta_2 * \top))) \Rightarrow \alpha \rightarrow \beta_1 \triangleright \forall \beta_1. \forall \alpha. (\alpha \triangleright (\beta_1 * \top)) \Rightarrow \alpha \rightarrow \beta_1$$

C'est

$$\lambda \alpha_1 \lambda \alpha (\diamond \alpha_1 \top \alpha) \langle (\beta_1 * \text{Top}^{(\beta_2 * \top)}) \rangle$$

Es-tu en train de dire qu'il y a un probleme avec l'exemple?

Non, il n'y a pas de problème avec l'exemple, vu que c'est une coercion monomorphe entre type polymorphes. it is inspired from $F_<$: and is not typable in F_η . It coerces a tuple of length greater or equal to n to a tuple of length n . In $F_<$, the first projection, say first , can be defined as $\lambda \beta \lambda \alpha \lambda (c : \alpha \triangleright (\beta * \top)) \lambda (x : \alpha) (c(x)).1$ of type $\forall \beta. \forall \alpha. (\alpha \triangleright (\beta * \top)) \Rightarrow \alpha \rightarrow \beta$, say τ , and can be applied to any non-zero tuple by passing the appropriate coercion.

Finally, passing first to a function choose of type $\forall \gamma. \gamma \rightarrow \gamma \rightarrow \gamma$, e.g. that picks one of its two arguments randomly, can be defined as $\lambda \gamma \lambda (c : \tau \triangleright \gamma) \text{choose } \gamma (c\{\text{first}\})$ of type $\forall \gamma. (\tau \triangleright \gamma) \Rightarrow \gamma \rightarrow \gamma$. We have delayed the choice of whether and how to instantiate the type of the first , since we can recover the different resulting types a posteriori by choosing appropriate coercions. For example, it can be coerced to type $\forall \alpha. (\sigma \triangleright \alpha) \Rightarrow \alpha \rightarrow \alpha$ where σ is $\forall \beta. \forall \beta'. \forall \alpha. (\alpha \triangleright \beta * \beta' * \top) \Rightarrow \alpha \rightarrow \beta$.

3 Properties of $F_<$

In this section, we show that $F_<$ is well-behaved: it has the subject reduction property and strongly normalizes; moreover, there is a forward simulation between terms of $F_<$ and their coercion erasure.

$$\begin{array}{l}
\llbracket \alpha \rrbracket = \alpha \\
\llbracket \tau \rightarrow \sigma \rrbracket = \llbracket \tau \rrbracket \rightarrow \llbracket \sigma \rrbracket \\
\llbracket (\tau * \sigma) \rrbracket = (\llbracket \tau \rrbracket * \llbracket \sigma \rrbracket) \\
\llbracket \forall \alpha. \tau \rrbracket = \forall \alpha. \llbracket \tau \rrbracket \\
\llbracket \varphi \Rightarrow \tau \rrbracket = \llbracket \varphi \rrbracket \rightarrow \llbracket \tau \rrbracket \\
\llbracket \top \rrbracket = \forall \alpha. (\forall \beta. \beta \rightarrow \alpha) \rightarrow \alpha \\
\llbracket \tau \triangleright \sigma \rrbracket = \llbracket \tau \rrbracket \rightarrow \llbracket \sigma \rrbracket \\
\llbracket \lambda \alpha M \rrbracket = \lambda \alpha \llbracket M \rrbracket \\
\llbracket M \tau \rrbracket = \llbracket M \rrbracket \llbracket \tau \rrbracket \\
\llbracket G \langle M \rangle \rrbracket = \llbracket G \rrbracket \llbracket M \rrbracket \\
\llbracket \lambda (c : \varphi) M \rrbracket = \lambda (x_c : \llbracket \varphi \rrbracket) \llbracket M \rrbracket \\
\llbracket M \{ G \} \rrbracket = \llbracket M \rrbracket \llbracket G \rrbracket \\
\llbracket c \rrbracket = \llbracket x_c \rrbracket \\
\llbracket \diamond \tau \rrbracket = \llbracket \lambda (x : \tau) x \rrbracket \\
\llbracket \text{Top}^\tau \rrbracket = \llbracket \lambda (y : \tau) \lambda \alpha \lambda (x : \forall \beta. \beta \rightarrow \alpha) x \tau y \rrbracket \\
\llbracket G_1 \xrightarrow{\tau} G_2 \rrbracket = \llbracket \lambda (y : \text{dom}(G_1 \xrightarrow{\tau} G_2)) \lambda (x : \tau) G_2 \langle y (G_1 \langle x \rangle) \rangle \rrbracket \\
\llbracket \text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha.} \rrbracket = \llbracket \lambda (y : \text{dom}(\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha.})) \lambda (x : \tau) \lambda \alpha y \alpha x \rrbracket \\
\llbracket \text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow} \rrbracket = \llbracket \lambda (y : \text{dom}(\text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow})) \lambda (x : \tau) \lambda (c : \varphi) y \{ c \} x \rrbracket \\
\llbracket (G_1 * G_2) \rrbracket = \llbracket \lambda (y : \text{dom}(G_1 * G_2)) (G_1 \langle y.1 \rangle, G_2 \langle y.2 \rangle) \rrbracket \\
\llbracket \text{Dist}_{(\tau * \sigma)}^{\forall \alpha.} \rrbracket = \llbracket \lambda (y : \text{dom}(\text{Dist}_{(\tau * \sigma)}^{\forall \alpha.})) (\lambda \alpha (y \alpha).1, \lambda \alpha (y \alpha).2) \rrbracket \\
\llbracket \text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow} \rrbracket = \llbracket \lambda (y : \text{dom}(\text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow})) (\lambda (c : \varphi) (y \{ c \}).1, \lambda (c : \varphi) (y \{ c \}).2) \rrbracket \\
\llbracket P[G] \rrbracket = \llbracket \lambda (x : \text{dom}(G)) P[G \langle x \rangle] \rrbracket \\
\llbracket x \rrbracket = x \\
\llbracket \lambda (x : \tau) M \rrbracket = \lambda (x : \llbracket \tau \rrbracket) \llbracket M \rrbracket \\
\llbracket M N \rrbracket = \llbracket M \rrbracket \llbracket N \rrbracket \\
\llbracket (M, N) \rrbracket = (\llbracket M \rrbracket, \llbracket N \rrbracket) \\
\llbracket M.1 \rrbracket = \llbracket M \rrbracket.1 \\
\llbracket M.2 \rrbracket = \llbracket M \rrbracket.2 \\
\llbracket \emptyset \rrbracket = \emptyset \\
\llbracket \Gamma, B \rrbracket = \llbracket \Gamma \rrbracket, \llbracket B \rrbracket \\
\llbracket \alpha \rrbracket = \alpha \\
\llbracket (x : \tau) \rrbracket = (x : \llbracket \tau \rrbracket) \\
\llbracket (c : \varphi) \rrbracket = (x_c : \llbracket \varphi \rrbracket)
\end{array}$$

Figure 9: Reification of F_t into System F

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_t 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 a the retyping function view of coercions, which is much closer to the reified approach in System F. We may regain this disjunction in F_t^λ , which the other presentation of F_t as retyping functions (§4). Terms of F_t^λ may still be reified into System F, but the reification is a so simple transformation that F_t^λ can be seen almost (but not quite) as an annotated version of System F.

3.1 Soundness

Type soundness of F_t follows as usual from the preservation and progress lemmas. The preservation lemma comes as usual with 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.*
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. 51)

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 $\beta \in \text{dom}(\Gamma, \alpha, \Gamma')$ holds and $\alpha \neq \beta$, then $\beta \in \text{dom}(\Gamma, \Gamma'\theta)$ holds.
4. If $\Gamma, \alpha, \Gamma' \vdash \text{ok}$ holds, then $\Gamma, \Gamma'\theta \vdash \text{ok}$ 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. 51)

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. 51)

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. 51)

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. 52)

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

(Proof p. 52)

The progress lemma is proved as usual by induction on terms, using typing to reject cases that are not values and do not reduce. 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 $\tau_1 \rightarrow \tau_2$, 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 $\text{Top}^\tau \langle v' \rangle$.

(Proof p. 52)

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

(Proof p. 53)

3.2 Termination of reduction

The termination of reduction for F_ι can be piggybacked on the termination of reduction in System F, as in [Manzonetto and Tranquilli, 2010]. For that purpose, it suffices to show a forward simulation between F_ι and System F, that is, to translate F_ι into System F so that every reduction step in F_ι is simulated in System F by at least one reduction step.

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 wish to 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 typed as $\tau \triangleright \sigma$ are reified as functions of type $\tau \rightarrow \sigma$. Hence, a coercion abstraction $\lambda(c : \tau \triangleright \sigma) M$ is reified as a higher-order function $\lambda(x_c : \lceil \tau \rceil \rightarrow \lceil \sigma \rceil) \lceil M \rceil$. A coercion variable c is reified to 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 $(\lceil \tau \rceil \rightarrow \lceil \sigma \rceil) \rightarrow \lceil \rho \rceil$ 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. 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 will be 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 $\lceil \Gamma \rceil \vdash \lceil \tau \rceil$ holds.
4. If $\Gamma \vdash \text{ok}$ holds, then $\lceil \Gamma \rceil \vdash \text{ok}$ 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. \square

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 $\llbracket M \rrbracket \rightsquigarrow^+ \llbracket N \rrbracket$.

(Proof p. 54)

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

Proof. Assume that M is well-typed in F_ι . By Lemma 11, $\llbracket M \rrbracket$ is well-typed in System F , hence the length of reduction sequences starting with $\llbracket M \rrbracket$ 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 . \square

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 evaluation contexts do not allow reduction in contexts of the form $\llbracket \langle M \rangle \rrbracket$. 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). \square

In fact, the relation \rightsquigarrow_ι alone is confluent.

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

Moreover, the reduction \rightsquigarrow_β and \rightsquigarrow_ι^* commute:

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

(Proof p. 55)

Unfortunately, the backward simulation fails. The wedge $\lambda(c : \tau \rightarrow \tau \triangleright \tau \rightarrow \tau) \lambda(y : \tau) c \langle \lambda(x : \tau) x \rangle y$ 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 an evaluation context. This observation invites to different solutions. We present two of them in sections §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 a 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 difference obtained by mapping all abstraction-like nodes and application-like nodes of the coercion sublanguage to term abstractions and applications of System F.

The first part defines a translation from F_ι to F_ι^λ that preserves well-typedness and coercion erasure. There is also an inverse translation from F_ι^λ to F_ι with the same property. Although we have not verified the details formally, each translation should also define a bisimulation up to ι -steps, showing F_ι and F_ι^λ are really 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.

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 the typechecking of $\lambda(x : \tau) \lambda\alpha \ [] \alpha x$ which is the retyping context which 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 $\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow M G_1\}$. 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. By looking at the interpretation, it should be obvious that G_2 may bind 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 F_ι^λ to avoid the additional complexity in the type system.

4.1 Definition of F_ι^λ

Syntax The syntax of F_ι^λ is described on Figure 10. The main differences between F_ι and F_ι^λ 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 F_ι^λ 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*.

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

	x, y	term
	c	coercion
	α, β	type
	ϕ	hole
$M, N ::= x \mid \lambda(x : \tau) M \mid M M \mid (M, M) \mid M.1 \mid M.2$		terms
$\mid H M \mid \lambda\alpha M \mid M \tau \mid \text{Top}^\tau M$		
$\mid \lambda(c : \varphi) M \mid M H$		
$\mid \lambda(\phi : \tau) G \{\phi \leftarrow M G\} \mid (G, G)\{\phi, \phi \leftarrow M\}$		
$G ::= \phi \mid H G \mid \lambda\alpha G \mid G \tau \mid \text{Top}^\tau G$		open coercions
$\mid \lambda(c : \varphi) G \mid G H$		
$\mid \lambda(\phi : \tau) G \{\phi \leftarrow G G\} \mid (G, G)\{\phi, \phi \leftarrow G\}$		
$H ::= c \mid \lambda(\phi : \tau) G$		close coercions
$\tau, \sigma, \rho ::= \alpha \mid \tau \rightarrow \tau \mid (\tau * \tau) \mid \forall\alpha.\tau \mid \varphi \Rightarrow \tau \mid \top$		types
$\varphi ::= \tau \triangleright \tau$		coercion types
$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha \mid \Gamma, c : \varphi$		expression environments
$\Delta ::= \emptyset \mid \alpha, \Delta \mid c : \varphi, \Delta$		coercion type environments
$Z ::= \square \mid \Delta \star (\phi : \tau)$		coercion types

Figure 10: System F_ι^λ : syntax

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.

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 \square or $\Delta \star (\phi : \tau)$: when Z is \square , 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 $\Gamma; \square \vdash M : \tau$. 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.

$$\begin{array}{c}
\frac{\text{LEXPRTERMVAR}}{\vdash \Gamma \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\text{LEXPRTERMLAM}}{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma} \quad \frac{\text{LEXPRTERMAPP}}{\Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} \\
\\
\frac{\text{LEXPRTERMPAIR}}{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M, N) : (\tau * \sigma)} \quad \frac{\text{LEXPRTERMFST}}{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.1 : \tau} \quad \frac{\text{LEXPRTERMSND}}{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.2 : \sigma} \quad \frac{\text{LEXPRHOLEVAR}}{\Gamma \vdash \tau}{\Gamma; \emptyset * (\phi : \tau) \vdash \phi : \tau} \\
\\
\frac{\text{LEXPRHOLEAPP}}{\Gamma \vdash H : \tau \triangleright \sigma \quad \Gamma; Z \vdash W : \tau}{\Gamma; Z \vdash HW : \sigma} \quad \frac{\text{LEXPRTYPELAM}}{\Gamma, \alpha; Z \vdash W : \tau}{\Gamma; (\alpha, \emptyset), Z \vdash \lambda \alpha W : \forall \alpha. \tau} \quad \frac{\text{LEXPRTYPEAPP}}{\Gamma; Z \vdash W : \forall \alpha. \sigma \quad \Gamma \vdash \tau}{\Gamma; Z \vdash W \tau : \sigma[\alpha \leftarrow \tau]} \\
\\
\frac{\text{LEXPRTOP}}{\Gamma; Z \vdash W : \tau}{\Gamma; Z \vdash \text{Top}^\tau W : \top} \quad \frac{\text{LEXPRCOERLAM}}{\Gamma, c : \varphi; Z \vdash W : \rho}{\Gamma; (c : \varphi, \emptyset), Z \vdash \lambda(c : \varphi) W : \varphi \Rightarrow \rho} \quad \frac{\text{LEXPRCOERAPP}}{\Gamma; Z \vdash W : \varphi \Rightarrow \rho \quad \Gamma \vdash H : \varphi}{\Gamma; Z \vdash WH : \rho} \\
\\
\frac{\text{LEXPRETAARR}}{\Gamma; \Delta * (\phi_2 : \sigma') \vdash G_2 : \sigma \quad \Gamma, \Delta; Z \vdash W : \tau' \rightarrow \sigma' \quad \Gamma, \Delta; \Delta' * (\phi_1 : \tau) \vdash G_1 : \tau' \quad \Gamma \vdash \tau}{\Gamma; \Delta, Z \vdash \lambda(\phi_1 : \tau) G_2 \{ \phi_2 \leftarrow W G_1 \} : \tau \rightarrow \sigma} \\
\\
\frac{\text{LEXPRETAPRD}}{\Gamma; \Delta * (\phi_1 : \tau') \vdash G_1 : \tau \quad \Gamma; \Delta * (\phi_2 : \sigma') \vdash G_2 : \sigma \quad \Gamma, \Delta; Z \vdash W : (\tau' * \sigma')}{\Gamma; \Delta, Z \vdash (G_1, G_2) \{ \phi_1, \phi_2 \leftarrow W \} : (\tau * \sigma)} \\
\\
\frac{\text{LEXPRCOERVAR}}{\vdash \Gamma \quad \Gamma(c) = \varphi}{\Gamma \vdash c : \varphi} \quad \frac{\text{LEXPRHOLELAM}}{\Gamma; \Delta * (\phi : \tau) \vdash G : \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash \lambda(\phi : \tau) G : \tau \triangleright \sigma}
\end{array}$$

Figure 11: System F_t^λ : typing rules

$$\begin{array}{c}
\text{LTYPEVAR} \\
\frac{\vdash \Gamma \quad \Gamma(\alpha)}{\Gamma \vdash \alpha} \\
\\
\text{LTYPEARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \rightarrow \sigma} \\
\\
\text{LTYPEPRODUCT} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash (\tau * \sigma)} \\
\\
\text{LTYPEFORALL} \\
\frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha. \tau} \\
\\
\text{LTYPECOER} \\
\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \rho}{\Gamma \vdash \varphi \Rightarrow \rho} \\
\\
\text{LTYPE TOP} \\
\frac{\vdash \Gamma}{\Gamma \vdash \top} \\
\\
\text{LENVEMPTY} \\
\vdash \emptyset \\
\\
\text{LENVEXPR} \\
\frac{\Gamma \vdash \tau \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \\
\\
\text{LENVTYPE} \\
\frac{\vdash \Gamma \quad \alpha \notin \text{dom}(\Gamma)}{\vdash \Gamma, \alpha} \\
\\
\text{LENVCOER} \\
\frac{\Gamma \vdash \varphi \quad c \notin \text{dom}(\Gamma)}{\vdash \Gamma, c : \varphi} \\
\\
\text{LDENVEMPTY} \\
\vdash \emptyset \\
\\
\text{LDENVTYPE} \\
\frac{\vdash \Delta \quad \alpha \notin \text{dom}(\Delta)}{\vdash \alpha, \Delta} \\
\\
\text{LDENVCOER} \\
\frac{\emptyset, \Delta \vdash \varphi \quad c \notin \text{dom}(\Delta)}{\vdash c : \varphi, \Delta}
\end{array}$$

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

$$\begin{aligned}
\Delta, \square &= \square \\
\Delta, (\Delta' \star (\phi : \tau)) &= (\Delta, \Delta') \star (\phi : \tau)
\end{aligned}$$

Figure 13: System F_t^λ : coercion type extension

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 \square :

$$\frac{\text{LEXPRETAARR} \quad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \quad \Gamma, \Delta \vdash M : \tau' \rightarrow \sigma' \quad \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \quad \Gamma \vdash \tau}{\Gamma \vdash \lambda(\phi_1 : \tau) G_2 \{ \phi_2 \leftarrow M G_1 \} : \tau \rightarrow \sigma}$$

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, $\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha}$ in F_t is translated to $\lambda(\phi_1 : \tau) (\lambda \alpha \phi_2) \{ \phi_2 \leftarrow (\phi \alpha) \phi_1 \}$ in F_t^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_t^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 evaluation contexts are defined on Figure 14; reduction rules are given on both Figure 15, which are as in F_t , and Figure 16, which are just β -reduction

$p ::= x \mid pv \mid p.1 \mid p.2 \mid cv \mid p\tau \mid pH$	prevalues
$v ::= p \mid \lambda(x : \tau)v \mid (v, v) \mid \lambda\alpha v \mid \lambda(c : \varphi)v \mid \mathbf{Top}^\tau v$	values
$\quad \mid \lambda(\phi : \tau)G \{\phi \leftarrow pG\} \mid (G, G)\{\phi, \phi \leftarrow p\}$	
$E ::= \lambda(x : \tau)\square \mid \square M \mid M\square \mid (\square, M) \mid (M, \square) \mid \square.1 \mid \square.2 \mid P$	evaluation contexts
$P ::= H\square \mid \lambda\alpha\square \mid \square\tau \mid \lambda(c : \varphi)\square \mid \square H \mid \mathbf{Top}^\tau\square$	retyping contexts
$\quad \mid \lambda(\phi : \tau)G \{\phi \leftarrow \square G\} \mid (G, G)\{\phi, \phi \leftarrow \square\}$	

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

$\frac{\text{LREDCONTEXTBETA} \quad M \rightsquigarrow_\beta N}{E[M] \rightsquigarrow_\beta E[N]}$	$\frac{\text{LREDCONTEXTIOTA} \quad M \rightsquigarrow_\iota N}{E[M] \rightsquigarrow_\iota E[N]}$	$\text{LREDTERM} \quad (\lambda(x : \tau)M)N \rightsquigarrow_\beta M[x \leftarrow N]$
$\text{LREDFIRST} \quad (M, N).1 \rightsquigarrow_\beta M$	$\text{LREDSECOND} \quad (M, N).2 \rightsquigarrow_\beta N$	$\text{LREDTYPE} \quad (\lambda\alpha M)\tau \rightsquigarrow_\iota M[\alpha \leftarrow \tau]$
$\text{LREDCOER} \quad (\lambda(c : \varphi)M)H \rightsquigarrow_\iota M[c \leftarrow H]$	$\text{LREDHOLE} \quad (\lambda(\phi : \tau)G)M \rightsquigarrow_\iota G[\phi \leftarrow M]$	

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:

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

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 occurrence 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 LREDETAPRDETAAPRD 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. 55)

Progress lemmas uses the following classification of values.

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 $\tau_1 \rightarrow \tau_2$, then v is either of the form $\lambda(x : \tau_1)v'$ or of the form $\lambda(\phi_1 : \tau_1)G_2 \{\phi_2 \leftarrow pG_1\}$.*

$$\begin{array}{c}
\text{LREDETAARRAPP} \\
(\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow M G_1\}) N \rightsquigarrow_\iota G_2[\phi_2 \leftarrow M G_1[\phi_1 \leftarrow N]] \\
\\
\text{LREDETAARRLAM} \\
\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(x : \tau') M) G_1\} \rightsquigarrow_\iota \lambda(x : \tau) G_2[\phi_2 \leftarrow M[x \leftarrow G_1[\phi_1 \leftarrow x]]] \\
\\
\text{LREDETAARRETAARR} \\
\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(\phi'_1 : \tau') G'_2 \{\phi'_2 \leftarrow M G'_1\}) G_1\} \rightsquigarrow_\iota \lambda(\phi_1 : \tau) G_2[\phi_2 \leftarrow G'_2 \{\phi'_2 \leftarrow M G'_1[\phi'_1 \leftarrow G_1]\}] \\
\\
\text{LREDETAPRDFST} \qquad \qquad \qquad \text{LREDETAPRDSND} \\
((G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}).1 \rightsquigarrow_\iota G_1[\phi_1 \leftarrow M.1] \qquad ((G_1, G_2)\{\phi_1, \phi_2 \leftarrow M\}).2 \rightsquigarrow_\iota G_2[\phi_2 \leftarrow M.2] \\
\\
\text{LREDETAPRDPAIR} \\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (M, N)\} \rightsquigarrow_\iota (G_1[\phi_1 \leftarrow M], G_2[\phi_2 \leftarrow N]) \\
\\
\text{LREDETAPRDETAPRD} \\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow ((G'_1, G'_2)\{\phi'_1, \phi'_2 \leftarrow M\})\} \rightsquigarrow_\iota (G_1[\phi_1 \leftarrow G'_1], G_2[\phi_2 \leftarrow G'_2])\{\phi'_1, \phi'_2 \leftarrow M\}
\end{array}$$

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

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 $\lambda(c : \varphi) v'$.
5. If τ is of the form \top , then v is of the form $\text{Top}^{\tau'} v'$.

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

(Proof p. 56)

4.3 Confluence

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

$$\begin{array}{c}
(\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(x : \tau') M) G_1\}) N \\
\\
(\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(\phi'_1 : \tau') G'_2 \{\phi'_2 \leftarrow M G'_1\}) G_1\}) N \\
\\
\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(\phi'_1 : \tau') G'_2 \{\phi'_2 \leftarrow (\lambda(x : \tau'') M) G'_1\}) G_1\} \\
\\
\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow (\lambda(\phi'_1 : \tau') G'_2 \{\phi'_2 \leftarrow (\lambda(\phi''_1 : \tau'') G''_2 \{\phi''_2 \leftarrow M G''_1\}) G'_1\}) G_1\} \\
\\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G'_1, G'_2)\{\phi'_1, \phi'_2 \leftarrow (M, N)\}\} \\
\\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (M, N)\}.1 \qquad (G_1, G_2)\{\phi'_1, \phi'_2 \leftarrow (M, N)\}.2 \\
\\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G'_1, G'_2)\{\phi'_1, \phi'_2 \leftarrow M\}\}.1 \qquad (G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G'_1, G'_2)\{\phi'_1, \phi'_2 \leftarrow M\}\}.2 \\
\\
(G_1, G_2)\{\phi_1, \phi_2 \leftarrow (G'_1, G'_2)\{\phi'_1, \phi'_2 \leftarrow (G''_1, G''_2)\{\phi''_1, \phi''_2 \leftarrow M\}\}\}
\end{array}$$

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:

$$\begin{array}{ccc}
& \text{LREDETAARRLAM} & (\lambda(\phi_1 : \tau) G_2 \{ \phi_2 \leftarrow (\lambda(x : \tau') M) G_1 \}) N & \text{LREDETAARRAPP} \\
& \swarrow & & \searrow \\
(\lambda(x : \tau) G_2 [\phi_2 \leftarrow M [x \leftarrow G_1 [\phi_1 \leftarrow x]]]) N & & & G_2 [\phi_2 \leftarrow (\lambda(x : \tau') M) G_1 [\phi_1 \leftarrow N]] \\
& \searrow & & \swarrow \\
& \text{LREDTERM} & G_2 [\phi_2 \leftarrow M [x \leftarrow G_1 [\phi_1 \leftarrow N]] & \text{LREDTERM}
\end{array}$$

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 F_l^λ 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 $[\Gamma] \vdash [M] : [\tau]$ holds.
- If $\Gamma \vdash H : \tau \triangleright \sigma$ holds, then $[\Gamma] \vdash [H] : [\tau] \rightarrow [\sigma]$ holds.
- If both $\Gamma; \Delta \star (\phi : \tau) \vdash G : \sigma$ and $[\Gamma, \Delta] \vdash M : [\tau]$ hold, then $[\Gamma] \vdash [G][x_\phi \leftarrow M] : [\sigma]$ 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 $M \rightsquigarrow_{\beta_l} N$ holds in F_l^λ , then $[M] \rightsquigarrow^+ [N]$ 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_l is complete wrt F_l^λ by defining a translation from F_l^λ expressions to F_l 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 F_l^λ and the right side is a judgment in F_l ; 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_l for the folding of universal quantification, λ -abstraction and application over Δ , defined as follows:

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

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

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

$$\begin{array}{l}
\text{Dist}_{\tau \rightarrow \sigma}^{\forall \emptyset.} = \diamond \tau \rightarrow \sigma \\
\text{Dist}_{\tau \rightarrow \sigma}^{\forall (\alpha, \Delta).} = \text{Dist}_{\tau \rightarrow \forall \Delta. \sigma}^{\forall \alpha.} \langle \lambda \alpha \text{Dist}_{\tau \rightarrow \sigma}^{\forall \Delta.} \langle \diamond \alpha \rangle \rangle \\
\text{Dist}_{\tau \rightarrow \sigma}^{\forall ((c : \varphi), \Delta).} = \text{Dist}_{\tau \rightarrow \forall \Delta. \sigma}^{\varphi \Rightarrow} \langle \lambda (c : \varphi) \text{Dist}_{\tau \rightarrow \sigma}^{\forall \Delta.} \langle \diamond \{c\} \rangle \rangle
\end{array}$$

$$\begin{array}{l}
[\alpha] = \alpha \\
[\tau \rightarrow \sigma] = [\tau] \rightarrow [\sigma] \\
[(\tau * \sigma)] = ([\tau] * [\sigma]) \\
[\forall \alpha. \tau] = \forall \alpha. [\tau] \\
[(\tau \triangleright \sigma) \Rightarrow \rho] = ([\tau] \rightarrow [\sigma]) \rightarrow [\rho] \\
[\top] = \forall \alpha. (\forall \beta. \beta \rightarrow \alpha) \rightarrow \alpha \\
\\
[x] = x \\
[\lambda(x : \tau) M] = \lambda(x : [\tau]) [M] \\
[M N] = [M] [N] \\
[(M, N)] = ([M], [N]) \\
[M.1] = [M].1 \\
[M.2] = [M].2 \\
\\
[\phi] = x_\phi \\
[\lambda(\phi : \tau) G] = \lambda(x_\phi : [\tau]) [G] \\
[H W] = [H] [W] \\
[\lambda \alpha W] = \lambda \alpha [W] \\
[W \tau] = [W] [\tau] \\
\\
[c] = x_c \\
[\lambda(c : \tau \triangleright \sigma) W] = \lambda(x_c : [\tau] \rightarrow [\sigma]) [W] \\
[W H] = [W] [H] \\
[\text{Top}^\tau W] = \lambda \alpha \lambda(x : \forall \beta. \beta \rightarrow \alpha) x [\tau] [W] \\
[\lambda(\phi_1 : \tau) G_2 \{\phi_2 \leftarrow W G_1\}] = [\lambda(x : \tau) G_2 \{\phi_2 \leftarrow W G_1 [\phi_1 \leftarrow x]\}] \\
[(G_1, G_2) \{\phi_1, \phi_2 \leftarrow W\}] = [(G_1 [\phi_1 \leftarrow W], G_2 [\phi_2 \leftarrow W])]
\end{array}$$

Figure 17: System F_l^λ : reification

and

$$\begin{array}{l}
\text{Dist}_{(\tau * \sigma)}^{\forall \emptyset.} = \diamond(\tau * \sigma) \\
\text{Dist}_{(\tau * \sigma)}^{\forall(\alpha, \Delta).} = \text{Dist}_{(\forall \Delta. \tau * \forall \Delta. \sigma)}^{\forall \alpha.} \langle \lambda \alpha \text{Dist}_{(\tau * \sigma)}^{\forall \Delta.} \langle \diamond \alpha \rangle \rangle \\
\text{Dist}_{(\tau * \sigma)}^{\forall(c : \varphi, \Delta).} = \text{Dist}_{(\forall \Delta. \tau * \forall \Delta. \sigma)}^{\varphi \Rightarrow} \langle \lambda(c : \varphi) \text{Dist}_{(\tau * \sigma)}^{\forall \Delta.} \langle \diamond \{c\} \rangle \rangle
\end{array}$$

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_l 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 $\Gamma, \Delta \vdash M : \tau' \rightarrow \sigma'$ and $\Gamma, \Delta \vdash G_1 : \forall \Delta'. \tau \triangleright \tau'$ hold. Then, $\Gamma \vdash (\diamond \rightarrow G_2) \langle \text{Dist}_{\tau \rightarrow \sigma'}^{\forall \Delta.} \langle \lambda \Delta ((G_1 \langle \lambda \Delta' \diamond \rangle) \rightarrow \diamond) \langle M \rangle \rangle \rangle : \tau \rightarrow \sigma$ also holds.*

(Proof p. 56)

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

(Proof p. 56)

Remark The proofs of both lemmas rely on the fact that the notation $\text{Dist}_{\tau \rightarrow \sigma}^{\forall \Delta.}$ is total on Δ . This is indeed the case in F_l . However, this means that if we were to extend F_l 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_l is sound $wrt F_l^\lambda$ by defining a translation from F_l expressions to F_l^λ expressions that preserves typings and coercion-erasure—see figures 20, 21 and 22 for the translation of terms and coercions, respectively.

$$\begin{array}{c}
\text{CTERMTERMVAR} \\
\frac{\Gamma \vdash ok \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau \rightsquigarrow \Gamma \vdash x : \tau} \\
\\
\text{CTERMTERMLAM} \\
\frac{\Gamma, (x : \tau) \vdash M : \sigma \rightsquigarrow \Gamma, (x : \tau) \vdash \hat{M} : \sigma}{\Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma \rightsquigarrow \Gamma \vdash \lambda(x : \tau) \hat{M} : \tau \rightarrow \sigma} \\
\\
\text{CTERMTERMAPP} \\
\frac{\Gamma \vdash M : \tau \rightarrow \sigma \rightsquigarrow \Gamma \vdash \hat{M} : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau \rightsquigarrow \Gamma \vdash \hat{N} : \tau}{\Gamma \vdash M N : \sigma \rightsquigarrow \Gamma \vdash \hat{M} \hat{N} : \sigma} \\
\\
\text{CTERMTERMPAIR} \\
\frac{\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau \quad \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)} \\
\\
\text{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} \\
\\
\text{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} \\
\\
\text{CTERMHOLEAPP} \\
\frac{\Gamma \vdash H : \tau \sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma \quad \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} \\
\\
\text{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} \\
\\
\text{CTERMTYPEAPP} \\
\frac{\Gamma \vdash M : \forall \alpha. \sigma \rightsquigarrow \Gamma \vdash \hat{M} : \forall \alpha. \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash M \tau : \sigma[\alpha \leftarrow \tau] \rightsquigarrow \Gamma \vdash \hat{M} \tau : \sigma[\alpha \leftarrow \tau]} \\
\\
\text{CTERMTOP} \\
\frac{\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau}{\Gamma \vdash \text{Top}^\tau M : \top \rightsquigarrow \Gamma \vdash \text{Top}^\tau \langle \hat{M} \rangle : \top} \\
\\
\text{CTERMETAARR} \\
\frac{\Gamma \vdash \tau \quad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta. \sigma' \triangleright \sigma \quad \Gamma, \Delta \vdash M : \tau' \rightarrow \sigma' \rightsquigarrow \Gamma, \Delta \vdash \hat{M} : \tau' \rightarrow \sigma' \quad \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \rightsquigarrow \Gamma, \Delta \vdash \hat{G}_1 : \forall \Delta'. \tau \triangleright \tau'}{\Gamma \vdash (\diamond \rightarrow \hat{G}_2) \langle \text{Dist}_{\tau \rightarrow \sigma'}^{\forall \Delta'} ((\lambda \Delta ((\hat{G}_1 \langle (\lambda \Delta' \diamond))) \rightarrow \diamond) \langle \hat{M} \rangle)) \rangle : \tau \rightarrow \sigma} \\
\\
\text{CTERMETAPROD} \\
\frac{\Gamma; \Delta \star (\phi_1 : \tau') \vdash G_1 : \tau \rightsquigarrow \Gamma \vdash \hat{G}_1 : \forall \Delta. \tau' \triangleright \tau \quad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta. \sigma' \triangleright \sigma \quad \Gamma, \Delta \vdash M : (\tau' * \sigma') \rightsquigarrow \Gamma, \Delta \vdash \hat{M} : (\tau' * \sigma')}{\Gamma \vdash (G_1, G_2) \{ \phi_1, \phi_2 \leftarrow M \} : (\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{G}_1 * \hat{G}_2) \langle \text{Dist}^{\forall \Delta. (\tau' * \sigma')} ((\lambda \Delta \hat{M})) \rangle : (\tau * \sigma)} \\
\\
\text{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} \\
\\
\text{CTERMCOERAPP} \\
\frac{\Gamma \vdash M : (\tau \triangleright \sigma) \Rightarrow \varphi \rightsquigarrow \Gamma \vdash \hat{M} : (\tau \triangleright \sigma) \Rightarrow \varphi \quad \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}
\end{array}$$

Figure 18: System F_t^λ : completeness for terms

$$\begin{array}{c}
\text{CCoERHOLEVAR} \\
\frac{\Gamma \vdash \tau}{\Gamma; \emptyset \star (\phi : \tau) \vdash \phi : \tau \rightsquigarrow \Gamma \vdash \hat{\diamond}^\tau : \tau \triangleright \tau} \\
\\
\text{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 \hat{\diamond}^\tau) \rangle : \tau \triangleright \sigma} \\
\\
\text{CCoERHOLEAPP} \\
\frac{\Gamma \vdash H : \tau \sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma \quad \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} \\
\\
\text{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 \hat{\diamond} \alpha] : \forall (\alpha, \Delta). \varphi \triangleright \forall \alpha. \tau} \\
\\
\text{CCoERTYPEAPP} \\
\frac{\Gamma; \Delta \star (\phi : \varphi) \vdash G : \forall \alpha. \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \forall \Delta. \varphi \triangleright \forall \alpha. \sigma \quad \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]} \\
\\
\text{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 \text{Top}^\tau G : \top \rightsquigarrow \Gamma \vdash \text{Top}^\tau \langle \hat{G} \rangle : \forall \Delta. \sigma \triangleright \top} \\
\text{CCoERCOERVAR} \\
\frac{\vdash \Gamma \quad \Gamma(c) = \tau \triangleright \sigma}{\Gamma \vdash c : \tau \sigma \rightsquigarrow \Gamma \vdash c : \tau \triangleright \sigma} \\
\\
\text{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 \quad \Gamma; (c : \tau \triangleright \sigma), \Delta \star (\phi : \rho') \vdash \lambda(c : \tau \triangleright \sigma) G : (\tau \triangleright \sigma) \Rightarrow \varphi \rightsquigarrow \Gamma \vdash \lambda(c : \tau \triangleright \sigma) \hat{G} [\diamond \leftarrow \hat{\diamond} \{c\}] : \forall ((c : \tau \triangleright \sigma), \Delta). \rho' \triangleright (\tau \triangleright \sigma) \Rightarrow \varphi}{\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} \\
\\
\text{CCoERCOERAPP} \\
\frac{\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 \quad \Gamma \vdash H : \tau \sigma \rightsquigarrow \Gamma \vdash \hat{H} : \tau \triangleright \sigma}{\Gamma; \Delta \star (\phi : \rho') \vdash G H : \varphi \rightsquigarrow \Gamma \vdash \hat{G} \langle \hat{H} \rangle : \forall \Delta. \rho' \triangleright \varphi} \\
\\
\text{CCoERETAARR} \\
\frac{\Gamma \vdash \tau \quad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta. \sigma' \triangleright \sigma \quad \Gamma, \Delta; \Delta'' \star (\phi : \varphi) \vdash G : \tau' \rightarrow \sigma' \rightsquigarrow \Gamma, \Delta \vdash \hat{G} : \forall \Delta''. \varphi \triangleright \tau' \rightarrow \sigma' \quad \Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau' \rightsquigarrow \Gamma, \Delta \vdash \hat{G}_1 : \forall \Delta'. \tau \triangleright \tau'}{\Gamma; \Delta, \Delta'' \star (\phi : \varphi) \vdash \lambda(\phi_1 : \tau) G_2 \{ \phi_2 \leftarrow G G_1 \} : \tau \rightarrow \sigma \rightsquigarrow \Gamma \vdash (\diamond \rightarrow \hat{G}_2) \langle \text{Dist}_{\tau \rightarrow \sigma'}^{\forall \Delta. (\tau' * \sigma')} \langle (\lambda \Delta ((\hat{G}_1 \langle (\lambda \Delta' \hat{\diamond}^\tau)) \rightarrow \diamond) \langle \hat{G} [\diamond \leftarrow \hat{\diamond} \Delta] \rangle) \rangle \rangle : \forall (\Delta, \Delta''). \varphi \triangleright \tau \rightarrow \sigma} \\
\\
\text{CExprETAProd} \\
\frac{\Gamma; \Delta \star (\phi_1 : \tau') \vdash G_1 : \tau \rightsquigarrow \Gamma \vdash \hat{G}_1 : \forall \Delta. \tau' \triangleright \tau \quad \Gamma; \Delta \star (\phi_2 : \sigma') \vdash G_2 : \sigma \rightsquigarrow \Gamma \vdash \hat{G}_2 : \forall \Delta. \sigma' \triangleright \sigma \quad \Gamma, \Delta; \Delta' \star (\phi : \varphi) \vdash G : (\tau' * \sigma') \rightsquigarrow \Gamma, \Delta \vdash \hat{G} : \forall \Delta'. \varphi \triangleright (\tau' * \sigma')}{\Gamma; \Delta, \Delta' \star (\phi : \varphi) \vdash (G_1, G_2) \{ \phi_1, \phi_2 \leftarrow G \} : (\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{G}_1 * \hat{G}_2) \langle \text{Dist}^{\forall \Delta. (\tau' * \sigma')} \langle (\lambda \Delta \hat{G} [\diamond \leftarrow \hat{\diamond} \Delta] \rangle) \rangle \rangle : \forall (\Delta, \Delta'). \varphi \triangleright (\tau * \sigma)}
\end{array}$$

Figure 19: System F_ι^λ : completeness for coercions

$$\begin{array}{c}
\text{STERMVAR} \\
\frac{\Gamma \vdash ok \quad (x : \tau) \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow \Gamma \vdash x : \tau}
\end{array}
\qquad
\begin{array}{c}
\text{STERMTERMLAM} \\
\frac{\Gamma, (x : \tau) \vdash M : \sigma \rightsquigarrow \Gamma, x : \tau \vdash \hat{M} : \sigma}{\Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma \rightsquigarrow \Gamma \vdash \lambda(x : \tau) \hat{M} : \tau \rightarrow \sigma}
\end{array}$$

$$\begin{array}{c}
\text{STERMTERMAPP} \\
\frac{\Gamma \vdash M : \tau \rightarrow \sigma \rightsquigarrow \Gamma \vdash M : \tau \rightarrow \sigma \quad \Gamma \vdash N : \tau \rightsquigarrow \Gamma \vdash N : \tau}{\Gamma \vdash M N : \sigma \rightsquigarrow \Gamma \vdash \hat{M} \hat{N} : \sigma}
\end{array}
\qquad
\begin{array}{c}
\text{STERMPAIR} \\
\frac{\Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau \quad \Gamma \vdash N : \sigma \rightsquigarrow \Gamma \vdash \hat{N} : \sigma}{\Gamma \vdash (M, N) : (\tau * \sigma) \rightsquigarrow \Gamma \vdash (\hat{M}, \hat{N}) : (\tau * \sigma)}
\end{array}$$

$$\begin{array}{c}
\text{STERMFIRST} \\
\frac{\Gamma \vdash M : (\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M} : (\tau * \sigma)}{\Gamma \vdash M.1 : \tau \rightsquigarrow \Gamma \vdash \hat{M}.1 : \tau}
\end{array}
\qquad
\begin{array}{c}
\text{STERMSECOND} \\
\frac{\Gamma \vdash M : (\tau * \sigma) \rightsquigarrow \Gamma \vdash \hat{M} : (\tau * \sigma)}{\Gamma \vdash M.2 : \sigma \rightsquigarrow \Gamma \vdash \hat{M}.2 : \sigma}
\end{array}$$

$$\begin{array}{c}
\text{STERMTYPELAM} \\
\frac{\Gamma, \alpha \vdash M : \tau \rightsquigarrow \Gamma, \alpha \vdash \hat{M} : \tau}{\Gamma \vdash \lambda \alpha M : \forall \alpha. \tau \rightsquigarrow \Gamma \vdash \lambda \alpha \hat{M} : \forall \alpha. \tau}
\end{array}
\qquad
\begin{array}{c}
\text{STERMTYPEAPP} \\
\frac{\Gamma \vdash M : \forall \alpha. \tau \rightsquigarrow \Gamma \vdash \hat{M} : \forall \alpha. \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash M \sigma : \tau[\alpha \leftarrow \sigma] \rightsquigarrow \Gamma \vdash \hat{M} \sigma : \tau[\alpha \leftarrow \sigma]}
\end{array}$$

$$\begin{array}{c}
\text{STERMCOER} \\
\frac{\Gamma \vdash G : \tau \triangleright \sigma \rightsquigarrow \Gamma \vdash \hat{G} : \tau \sigma \quad \Gamma \vdash M : \tau \rightsquigarrow \Gamma \vdash \hat{M} : \tau}{\Gamma \vdash G \langle M \rangle : \sigma \rightsquigarrow \Gamma \vdash \hat{G} \hat{M} : \sigma}
\end{array}
\qquad
\begin{array}{c}
\text{STERMCOERLAM} \\
\frac{\Gamma, (c : \varphi \triangleright \rho') \vdash M : \sigma \rightsquigarrow \Gamma, c : \varphi \rho' \vdash \hat{M} : \sigma}{\Gamma \vdash \lambda(c : \varphi \triangleright \rho') M : (\varphi \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \lambda(c : \varphi) \rho' \hat{M} : \varphi \Rightarrow \rho' \sigma}
\end{array}$$

$$\begin{array}{c}
\text{STERMCOERAPP} \\
\frac{\Gamma \vdash G : \varphi \triangleright \rho' \rightsquigarrow \Gamma \vdash \hat{G} : \varphi \rho' \quad \Gamma \vdash M : (\varphi \triangleright \rho') \Rightarrow \sigma \rightsquigarrow \Gamma \vdash \hat{M} : \varphi \Rightarrow \rho' \sigma}{\Gamma \vdash M \{G\} : \sigma \rightsquigarrow \Gamma \vdash \hat{M} \hat{G} : \sigma}
\end{array}$$

Figure 20: System F_t^λ : soundness for terms

$$\begin{array}{c}
\text{SCOERTRANS} \\
\frac{\Gamma \vdash G_2 : \sigma \triangleright \varphi \rightsquigarrow \Gamma \vdash \hat{G}_2 : \sigma\varphi \quad \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} \\
\\
\text{SCOERTYPELAM} \\
\frac{\Gamma \vdash \tau \quad \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} \\
\\
\text{SCOERTYPEAPP} \qquad \text{SCOERDOT} \\
\frac{\Gamma \vdash \varphi \quad \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]} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \diamond^\tau : \tau \triangleright \tau \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau) \phi : \tau\tau} \\
\\
\text{SCOERFORGET} \qquad \text{SCOERVAR} \\
\frac{\Gamma \vdash \tau}{\Gamma \vdash \text{Top}^\tau : \tau \triangleright \top \rightsquigarrow \Gamma \vdash \lambda(\phi : \tau) \text{Top}^\tau \phi : \tau\top} \qquad \frac{\Gamma \vdash ok \quad (c : \tau \triangleright \sigma) \in \Gamma}{\Gamma \vdash c : \tau \triangleright \sigma \rightsquigarrow \Gamma \vdash c : \tau\sigma} \\
\\
\text{SCOERCOERLAM} \\
\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} \\
\\
\text{SCOERCOERAPP} \\
\frac{\Gamma \vdash G' : \rho \triangleright \rho' \rightsquigarrow \Gamma \vdash \hat{G}' : \varphi\rho' \quad \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}
\end{array}$$

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

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

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

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

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

$$\begin{array}{c}
\text{SCOERDISTCOERARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma \vdash \varphi \quad \Gamma \vdash \rho' \quad \Gamma \vdash \sigma}{\Gamma \vdash \text{Dist}_{\tau \rightarrow \sigma}^{\varphi \triangleright \rho' \Rightarrow} : ((\varphi \triangleright \rho') \Rightarrow (\tau \rightarrow \sigma)) \triangleright (\tau \rightarrow (\varphi \triangleright \rho') \Rightarrow \sigma) \rightsquigarrow} \\
\Gamma \vdash \lambda(\phi : (\varphi \triangleright \rho') \Rightarrow (\tau \rightarrow \sigma)) \lambda(\phi_1 : \tau) \lambda(c : \varphi) \rho' \phi_2 \{ \phi_2 \leftarrow (\phi c) \phi_1 \} : \\
((\varphi \triangleright \rho') \Rightarrow (\tau \rightarrow \sigma)) \triangleright (\tau \rightarrow (\varphi \triangleright \rho') \Rightarrow \sigma)
\end{array}$$

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

Figure 22: System F_c^λ : soundness for coercions 2/2

$\diamond ::= \triangleleft \mid \triangleright$	bounds
$\tau ::= \dots \not\! / \varphi \Rightarrow \tau \mid \forall(\alpha \diamond \tau) \Rightarrow \tau$	types
$M ::= \dots \not\! / \lambda(c : \varphi) M \mid \lambda(\alpha \diamond c : \tau) M$ $\not\! / M\{G\} \mid M\{\tau \diamond G\}$	expressions
$G ::= \dots \not\! / \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow} \mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha \diamond \tau \Rightarrow}$ $\not\! / \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow} \mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha \diamond \tau \Rightarrow}$ $\not\! / \lambda(c : \varphi) G \mid \lambda(\alpha \diamond c : \tau) G$ $\not\! / G\{G\} \mid G\{\tau \diamond G\}$	coercions
$\Gamma ::= \dots \not\! / \Gamma, c : \varphi \mid \Gamma, \alpha \diamond c : \tau$	environments

Figure 23: Parametric F_l : syntax restriction $wrt F_l$

4.7 Bisimulation between F_l and F_l^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_l 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. 56)

Lemma 27. *For M_0 in F_l 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 $M_0 \rightsquigarrow_l^* \rightsquigarrow_\beta M_2$ and \hat{M}_2 is ι -equivalent to N_1 .*

(Proof p. 57)

5 Parametric F_l

Parametric F_l , written F_l^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 $c(\lambda(x : \tau) M) N$ has a coercion type $\sigma \triangleright \rho$ where σ and ρ are both arrow types. To prevent this situation from happening in F_l^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 of domain or the codomain of c must be treated abstractly (and thus not as an arrow type) in M . To enforce this restriction we stick a type abstraction to every coercion abstraction and see $\lambda \alpha \lambda(c : \alpha \triangleright \tau) M$ as a single syntactic node, which we write $\lambda(\alpha \triangleright c : \tau) M$ to avoid confusion. Although, we modify the syntax of source terms, Parametric F_l can still be understood as a syntactic restriction of F_l .

5.1 Syntax changes

The syntax of Parametric F_l is defined on Figure 23 as a patch to the syntax of F_l (we write $\not\! /$ for removal of a previous grammar form). We replace coercion abstraction $\lambda(c : \tau \triangleright \sigma) M$ of F_l by two new constructs $\lambda(\alpha \triangleright c : \tau) M$ and $\lambda(\alpha \triangleleft c : \tau) M$ to mean $\lambda \alpha \lambda(c : \alpha \triangleright \tau) M$ and $\lambda \alpha \lambda(c : \tau \triangleright \alpha) M$ but atomically. For conciseness, we introduce a mode \diamond that range other \triangleright and \triangleleft . Hence, we write $\lambda(\alpha \diamond c : \tau) M$ for either $\lambda(\alpha \triangleright c : \tau) M$ or $\lambda(\alpha \triangleleft c : \tau) M$. 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

$$\begin{array}{c}
\text{TERMTCOERLAM} \\
\frac{\Gamma, \alpha \triangleleft c : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(\alpha \triangleleft c : \tau) M : \forall(\alpha \triangleleft \tau) \Rightarrow \sigma} \\
\\
\text{TCOERVAR} \\
\frac{\Gamma \vdash ok \quad \alpha \triangleleft c : \tau \in \Gamma}{\Gamma \vdash c : \alpha \triangleleft \tau} \\
\\
\text{COERTCOERAPP} \\
\frac{\Gamma \vdash G' : \rho \triangleright \forall(\alpha \triangleleft \tau) \Rightarrow \tau' \quad \Gamma \vdash G : \sigma \triangleleft \tau[\alpha \leftarrow \sigma]}{\Gamma \vdash G' \{\sigma \triangleleft G\} : \rho \triangleright \tau'[\alpha \leftarrow \sigma]} \\
\\
\text{COERDISTTCOERARROW} \\
\frac{\Gamma \vdash \tau \quad \Gamma, \alpha \vdash \rho \quad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha \triangleleft \rho \Rightarrow} : (\forall(\alpha \triangleleft \rho) \Rightarrow \tau \rightarrow \sigma) \triangleright (\tau \rightarrow \forall(\alpha \triangleleft \rho) \Rightarrow \sigma)} \\
\\
\text{COERDISTTCOERPROD} \\
\frac{\Gamma, \alpha \vdash \tau \quad \Gamma, \alpha \vdash \rho \quad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \text{Dist}_{(\tau * \sigma)}^{\forall \alpha \triangleleft \rho \Rightarrow} : \forall(\alpha \triangleleft \rho) \Rightarrow (\tau * \sigma) \triangleright (\forall(\alpha \triangleleft \rho) \Rightarrow \tau * \forall(\alpha \triangleleft \rho) \Rightarrow \sigma)}
\end{array}$$

Figure 24: Parametric F_L : typing rules $wrt F_L$

by moving “ c ” in front, *i.e.* $\alpha \triangleright c : \tau$ becomes $c : \alpha \triangleright \tau$ while $\alpha \triangleleft c : \tau$ becomes $c : \alpha \triangleleft \tau$ which can also be read $c : \tau \triangleright \alpha$. The reason to keep the type variable α before the coercion variable is to preserve the order of the abstractions in F_L .

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

Continuing with the definition of F_L^p , we replace coercion application $M\{G\}$ by $M\{\tau \triangleleft G\}$ 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 \triangleleft 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 $\forall(\alpha \triangleleft \tau) \Rightarrow \sigma$, which factors the two forms $\forall(\alpha \triangleright \tau) \Rightarrow \sigma$ and $\forall(\alpha \triangleleft \tau) \Rightarrow \sigma$ whose expansions in F_L are $\forall \alpha. (\alpha \triangleright \tau) \Rightarrow \sigma$ and $\forall \alpha. (\tau \triangleright \alpha) \Rightarrow \sigma$, respectively. Typing environments are modified accordingly. Notice that $\Gamma, \alpha \triangleleft c : \tau$ stands for $\Gamma, \alpha, c : \alpha \triangleleft \tau$ ($c : \alpha \triangleleft \tau$ should be read as $c : \tau \triangleright \alpha$) and therefore α may appear free in τ —as for coercion abstractions.

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 which 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 $\text{COERDISTTCOERARROW}$ and COERDISTTCOERPROD . Notice that \triangleleft is a meta-

$$\begin{array}{l}
p ::= \dots \not\! / p\{G\} \mid p\{\tau \triangleleft G\} \quad \text{prevalues} \\
\quad \not\! / \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow} \langle p \rangle \mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha \triangleleft \tau \Rightarrow} \langle p \rangle \\
\quad \not\! / \text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow} \langle \lambda(c : \varphi) p \rangle \mid \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha \triangleleft \tau \Rightarrow} \langle \lambda(\alpha \triangleleft c : \tau) p \rangle \\
\quad \not\! / \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow} \langle p \rangle \mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha \triangleleft \tau \Rightarrow} \langle p \rangle \\
\quad \not\! / \text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow} \langle \lambda(\alpha \triangleleft c : \tau) p \rangle \mid \text{Dist}_{(\tau * \tau)}^{\forall \alpha \triangleleft \tau \Rightarrow} \langle \lambda(\alpha \triangleleft c : \tau) p \rangle \\
v ::= \dots \not\! / \lambda(c : \varphi) v \mid \lambda(\alpha \triangleleft c : \tau) v \quad \text{values} \\
P ::= \dots \not\! / \lambda(c : \varphi) [] \mid \lambda(\alpha \triangleleft c : \tau) [] \not\! / []\{G\} \mid []\{\tau \triangleleft G\} \quad \text{retyping contexts}
\end{array}$$

Figure 25: Parametric F_l : changes in values $wrt F_l$

$$\begin{array}{c}
\text{TCOERARROW} \\
\frac{\Gamma, \alpha \vdash \tau \quad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \forall(\alpha \triangleleft \tau) \Rightarrow \sigma} \\
\text{ENVCOER} \\
\frac{\Gamma, \alpha \vdash \tau \quad \alpha, c \notin \text{dom}(\Gamma)}{\Gamma, \alpha \triangleleft c : \tau \vdash \text{ok}}
\end{array}$$

Figure 26: Parametric F_l : well-formedness judgments $wrt F_l$

$$\begin{array}{c}
\text{REDTCOER} \\
(\lambda(\alpha \triangleleft c : \tau) M)\{\sigma \triangleleft G\} \rightsquigarrow_l M[\alpha \leftarrow \sigma][c \leftarrow G] \\
\text{REDCOERDISTTCOERARROW} \\
\text{Dist}_{\sigma_2 \rightarrow \sigma_3}^{\forall \alpha \triangleleft \sigma_1 \Rightarrow} \langle \lambda(\alpha \triangleleft c : \tau) \lambda(x : \sigma) M \rangle \rightsquigarrow_l \\
\lambda(x : \sigma) \lambda(\alpha \triangleleft \tau : c) M \\
\text{REDCOERDISTTCOERPROD} \\
\text{Dist}_{(\sigma_2 * \sigma_3)}^{\forall \alpha \triangleleft \sigma_1 \Rightarrow} \langle \lambda(\alpha \triangleleft c : \tau) (M, N) \rangle \rightsquigarrow_l (\lambda(\alpha \triangleleft c : \tau) M, \lambda(\alpha \triangleleft c : \tau) N)
\end{array}$$

Figure 27: Parametric F_l : new reduction rules $wrt F_l$

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 \triangleleft_1 and \triangleleft_2 when we mean to instantiate them independently.)

The new typing rules for F_l^p are just derived from the typing rules of the corresponding nodes in F_l . For example, TERMTCoERLAM is just the combination of rules TERMCoERLAM and TERMTYPELAM in F_l .

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

Operational semantics The operational semantics is modified in the obvious way. The syntax of values for F_l^p is defined on Figure 25 as a modification of the syntax of F_l . The adjustments in the reduction rules are the replacement of REDCoER by REDTCOER , $\text{REDCoERDISTCoERARROW}$ by $\text{REDCOERDISTTCOERARROW}$, $\text{REDCoERDISTCoERPROD}$ by $\text{REDCOERDISTTCOERPROD}$, and the change of retyping contexts that induces a change in REDCoERFILL as described in Figure 25.

5.3 Properties

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

This can be formalized by making the correspondence between F_l^p and F_l explicit.

Definition 28 (F_l^p to F_l traduction). *We define a Translation from F_l^p to F_l witnessing the language inclusion on Figure 28. We write M° the translation of M . We only give the translations that*

$$\begin{aligned}
(\lambda(\alpha \diamond c : \tau) W)^\circ &= \lambda\alpha \lambda(c : \alpha \diamond \tau^\circ) W^\circ \\
(W\{\tau \diamond G\})^\circ &= (W^\circ \tau^\circ)\{G^\circ\} \\
(\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha \diamond \rho \Rightarrow})^\circ &= \text{Dist}_{\tau^\circ \rightarrow \sigma^\circ}^{\forall \alpha. (\alpha \diamond \rho^\circ) \Rightarrow \sigma^\circ} \langle \lambda\alpha \text{Dist}_{\tau^\circ \rightarrow \sigma^\circ}^{\alpha \diamond \rho^\circ \Rightarrow} \langle \diamond \alpha \rangle \rangle \\
(\text{Dist}_{(\tau * \sigma)}^{\forall \alpha \diamond \rho \Rightarrow})^\circ &= \text{Dist}_{(\alpha \diamond \rho^\circ \Rightarrow \tau^\circ * \alpha \diamond \rho^\circ \Rightarrow \sigma^\circ)}^{\forall \alpha.} \langle \lambda\alpha \text{Dist}_{(\tau^\circ * \sigma^\circ)}^{\alpha \diamond \rho^\circ \Rightarrow} \langle \diamond \alpha \rangle \rangle \\
\tau \triangleright \sigma^\circ &= \tau^\circ \triangleright \sigma^\circ \\
\tau \triangleleft \sigma^\circ &= \sigma^\circ \triangleright \tau^\circ \\
(\forall(\alpha \diamond \tau) \Rightarrow \sigma)^\circ &= \forall \alpha. (\alpha \diamond \tau^\circ) \Rightarrow \sigma^\circ \\
(\Gamma, \alpha \diamond c : \tau)^\circ &= \Gamma, \alpha, c : \alpha \diamond \tau^\circ
\end{aligned}$$

Figure 28: Translation of F_ι^p into F_ι

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_ι^p 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_ι^p 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_ι^p if and only if $\Gamma^\circ \vdash \tau^\circ$ holds in F_ι .
4. $\Gamma \vdash \text{ok}$ holds in F_ι^p if and only if $\Gamma^\circ \vdash \text{ok}$ holds in F_ι .

Lemma 30. *The following assertions hold.*

1. If $M \rightsquigarrow_\beta N$ in F_ι^p , then $M^\circ \rightsquigarrow_\beta N^\circ$ in F_ι .
2. If $M \rightsquigarrow_\iota N$ in F_ι^p , 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. 57)

Proposition 32 (Termination). *Reduction in F_ι^p is terminating.*

(Proof p. 57)

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_ι^p is confluent.*

(Proof p. 57)

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 $\tau \rightarrow \tau$, 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 $\forall(\alpha \diamond \tau) \Rightarrow \tau$, then v is of the form $\lambda(\alpha \diamond c : \tau) v$.
5. If τ is of the form \top , then v is of the form $\text{Top}^\tau \langle v \rangle$.

(Proof p. 57)

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

(Proof p. 57)

As expected, coercions are erasable in Parametric F_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 $\llbracket M \rrbracket \rightsquigarrow \llbracket N \rrbracket$.*
2. *If $M \rightsquigarrow_\iota N$, then $\llbracket M \rrbracket = \llbracket N \rrbracket$.*

(Proof p. 58)

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 $\sigma \rightarrow \sigma'$ (resp. $(\sigma * \sigma')$) then Q is \square .*
2. *If τ is $\forall \alpha. \sigma$ then Q is $\lambda \alpha Q'$.*
3. *If τ is $\forall (\alpha \triangleleft \sigma) \Rightarrow \tau'$ then Q is $\lambda (\alpha \triangleleft c : \sigma) Q'$.*
4. *For all $\alpha \triangleright c : \sigma$ 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 .

- \square : 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.
- $G(Q')$: 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.
 - $\lambda \alpha W, W \tau, G(W), \lambda(\alpha \triangleleft c : \tau) W$, and $W\{\tau \triangleleft G\}$: These are not in ι -normal-form, since REDCOERFILL applies.
 - \diamond^τ : 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 $\Gamma \vdash Q'[\lambda(x : \rho) M] : \sigma \rightarrow \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι -normal-form, since REDCOERARROW applies.
 - $\text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha.}$: 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.
 - $\text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha \triangleleft \tau \Rightarrow}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda(\alpha \triangleleft c : \sigma) \lambda(x : \rho) M$, which contradicts the fact that we were in ι -normal-form, since REDCOERDISTTCOERARROW applies.
 - $\text{Top}^\tau, (G * G), \text{Dist}_{(\tau * \tau)}^{\forall \alpha.}$, and $\text{Dist}_{(\tau * \tau)}^{\forall \alpha \triangleleft \tau \Rightarrow}$: Conditions 1 to 3 do not apply, and condition 4 trivially holds.

$$\begin{array}{c}
\text{KERNEL-FSUB} \\
\frac{\Sigma, \alpha <: A \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A) B <: \forall(\alpha <: A) B'} \\
\\
\text{FULL-FSUB} \\
\frac{\Sigma \vdash A' <: A \quad \Sigma, \alpha <: A' \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A) B <: \forall(\alpha <: A') B'} \\
\\
\text{F-BOUNDED} \\
\frac{\Sigma, \alpha <: A' \vdash \alpha <: A \quad \Sigma, \alpha <: A' \vdash B <: B'}{\Sigma \vdash \forall(\alpha <: A) B <: \forall(\alpha <: A') B'}
\end{array}$$

Figure 29: Bounded polymorphism: variants on the subtyping rule

- $\lambda(\sigma \diamond c : \alpha) Q'$: 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 $[M] \rightsquigarrow m$, then $M \rightsquigarrow_{\iota}^* \rightsquigarrow_{\beta} N$ such that $[N] = m$.*

Proof. The proof schema is not original [Manzonetto and Tranquilli, 2010]. We show that the ι -normal-form of M β -reduces to N with $[N]$ equal to m . Since F_{ι} strongly normalizes, we may assume, without loss of generality that M is already in ι -normal-form. Because $[M]$ reduces, we can use the reduction derivation to show that it must be of the form $e[(\lambda x.m_1) m_2]$. By inversion of the coercion-erasure function, we show that M is of the form $E[Q[\lambda(x : \tau) M_1] M_2]$ where E is an evaluation context and Q a retyping context of arbitrary depth, such that E , M_1 , and M_2 erase to e , m_1 , and 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 $E[(\lambda(x : \tau) M_1) M_2]$ and β -reduces to $E[M_1[x \leftarrow M_2]]$ 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}

Despite it is bridled by-design, Parametric F_{ι} is already an interesting spot in the design space, as it subsumes in a unified framework three known languages: F_{η} , $x\text{MLF}$, and $F_{<}$: (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 $x\text{MLF}$ and $F_{<}$ are also subsumed by F_{ι}^p . In each case, we exhibit a translation of typing judgments from the source language to typing judgments of F_{ι}^p so that the coercion erasure of the translation of a source term m is equal to the type erasure of m , and therefore the translation is semantics preserving.

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 $[T] = [M]$. As a consequence, reduction in the source language terminates, since it is simulated in Parametric F_{ι} .

Bounded polymorphism. $F_{<}$ is a well-known extension of System F with subtyping. There are several variations on $F_{<}$, 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 $F_{<}$, the bounds A and A' must be

$$\begin{array}{ll}
\alpha^b = \alpha & \epsilon^b = \emptyset \\
(T \rightarrow S)^b = T^b \rightarrow S^b & (A, \alpha <: T)^b = A^b, (\alpha \triangleright c_\alpha : T^b) \\
(\forall(\alpha <: T) S)^b = \forall(\alpha \triangleright T^b) \Rightarrow S^b & (A, x : T)^b = A^b, (x : T^b) \\
\top^b = \top &
\end{array}$$

Figure 30: $F_{\mu <}$: type and environment translation

equal, whereas Full $F_{<}$ does 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 $F_{<}$, while $F_{\mu <}$ allows this form of recursion, called F-bounded polymorphism. The most general assumption, $\Gamma, \alpha <: A' \vdash \alpha <: A$, is that of $F_{\mu <}$. Perhaps surprisingly, this is a slightly more general rule [Baldan et al., 1999] than the more intuitive one $\Gamma, \alpha <: A' \vdash A' <: A$. In summary, we have $\text{Kernel } F_{<} \subset \text{Full } F_{<} \subset F_{\mu <}$ where all inclusions are strict.

We show that the most expressive version $F_{\mu <}$ is also included into F_ℓ^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 $\forall(\alpha < \tau) \Rightarrow \sigma$ encodes lower bounds and are never needed in the translation of $F_{\mu <}$.)

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

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

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

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

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.

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

It is remarkable that F_ℓ^p naturally matches the most expressive version $F_{\mu <}$. This encourages to follow a systematic approach viewing type conversions as erasable coercions as in F_ℓ^p rather than an *ad hoc* subtyping relation. Additionally, F_ℓ^p may simplify the proof of type soundness for $F_{\mu <}$, as coercions are explicit.

To show that $F_{\mu <}$ is included in F_ℓ^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 $\Gamma = A^b$ and $\tau = T^b$ hold.
2. If $A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma$ holds then $\Gamma = A^b$, $\tau = T^b$, and $\sigma = S^b$ hold.

$$\begin{array}{c}
\text{BTERMVAR} \\
\frac{T = A(x)}{A \vdash x : T \rightsquigarrow A^b \vdash x : T^b} \\
\\
\text{BTERM LAM} \\
\frac{A, x : T \vdash m : S \rightsquigarrow \Gamma, (x : \tau) \vdash M : \sigma}{A \vdash \lambda x : T. m : T \rightarrow S \rightsquigarrow \Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma} \\
\\
\text{BTYPE LAM} \\
\frac{A, \alpha < : S \vdash m : T \rightsquigarrow \Gamma, (\sigma \triangleleft c_\alpha : \alpha) \vdash M : \tau}{A \vdash \Lambda(\alpha < : S) m : \forall(\alpha < : S) T \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleright c_\alpha : \sigma) M : \forall(\alpha \triangleright \sigma) \Rightarrow \tau} \\
\\
\text{BTYPE APP} \\
\frac{A \vdash m : \forall(\alpha < : T) S \rightsquigarrow \Gamma \vdash M : \forall(\alpha \triangleright \tau) \Rightarrow \sigma \quad A \vdash T' < : T[\alpha \leftarrow T'] \rightsquigarrow \Gamma \vdash G : \tau' \triangleright \tau[\alpha \leftarrow \tau']}{A \vdash m\{T'\} : S[\alpha \leftarrow T'] \rightsquigarrow \Gamma \vdash M\{\tau' \triangleleft G\} : \sigma[\alpha \leftarrow \tau']} \\
\\
\text{BSUBS} \\
\frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \quad A \vdash T < : S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma}{A \vdash m : S \rightsquigarrow \Gamma \vdash G\langle M \rangle : \sigma} \\
\\
\text{BTERM APP} \\
\frac{A \vdash m : T \rightarrow S \rightsquigarrow \Gamma \vdash M : \tau \rightarrow \sigma \quad A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau}{A \vdash m(n) : S \rightsquigarrow \Gamma \vdash M N : \sigma}
\end{array}$$

Figure 31: $F_{\mu <}$: term translation

$$\begin{array}{c}
\text{BID} \\
A \vdash T < : T \rightsquigarrow \Gamma \vdash \diamond^{T^b} : T^b \triangleright T^b \\
\\
\text{BCOERVAR} \\
\frac{\alpha < : T \in A}{A \vdash \alpha < : T \rightsquigarrow \Gamma \vdash c_\alpha : \alpha \triangleright \tau} \\
\\
\text{BTRANS} \\
\frac{A \vdash T < : S \rightsquigarrow \Gamma \vdash G_1 : \tau \triangleright \sigma \quad A \vdash S < : U \rightsquigarrow \Gamma \vdash G_2 : \sigma \triangleright \rho}{A \vdash T < : U \rightsquigarrow \Gamma \vdash G_2\langle G_1 \rangle : \tau \triangleright \rho} \\
\\
\text{BTOP} \\
A \vdash T < : \top \rightsquigarrow \Gamma \vdash \text{Top}^\tau : \tau \triangleright \top \\
\\
\text{BARROW} \\
\frac{A \vdash T' < : T \rightsquigarrow \Gamma \vdash G_1 : \tau' \triangleright \tau \quad A \vdash S < : S' \rightsquigarrow \Gamma \vdash G_2 : \sigma \triangleright \sigma'}{A \vdash T \rightarrow S < : T' \rightarrow S' \rightsquigarrow \Gamma \vdash G_1 \xrightarrow{\tau'} G_2 : \tau \rightarrow \sigma \triangleright \tau' \rightarrow \sigma'} \\
\\
\text{BFORALL} \\
\frac{A, \alpha < : T' \vdash \alpha < : T \rightsquigarrow \Gamma, (\alpha \triangleright c_\alpha : \tau') \vdash G : \alpha \triangleright \tau \quad A, \alpha < : T' \vdash S < : S' \rightsquigarrow \Gamma, (\alpha \triangleright c_\alpha : \tau') \vdash G' : \sigma \triangleright \sigma'}{A \vdash \forall(\alpha < : T) S < : \forall(\alpha < : T') S' \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleright c : \tau') G' \langle \diamond\{\alpha \triangleleft G\} \rangle : \forall(\alpha \triangleright \tau) \Rightarrow \sigma \triangleright \forall(\alpha \triangleright \tau') \Rightarrow \sigma'}
\end{array}$$

Figure 32: $F_{\mu <}$: subtyping translation

$$\begin{array}{ll}
\tau ::= \dots \not\bowtie \forall(\alpha \triangleright \tau) \Rightarrow \tau & \text{types} \\
M ::= \not\bowtie \lambda(\alpha \triangleright c : \tau) M \not\bowtie M\{\tau \triangleright G\} & \text{expressions} \\
G ::= \dots \not\bowtie G \xrightarrow{\tau} G \not\bowtie \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha.} \not\bowtie \text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha \triangleright \tau \Rightarrow} & \text{coercions} \\
& \not\bowtie (G * G) \not\bowtie \text{Dist}_{(\tau * \tau)}^{\forall \alpha.} \not\bowtie \text{Dist}_{(\tau * \tau)}^{\forall \alpha \triangleright \tau \Rightarrow} \\
& \not\bowtie \lambda(\alpha \triangleright c : \tau) G' \not\bowtie G'\{\tau \triangleright G\}
\end{array}$$

Figure 33: $F_{\iota}^{\mu x}$: syntax and notations

$$\begin{array}{ll}
\alpha^x = \alpha & \emptyset^x = \emptyset \\
(T \rightarrow S)^x = T^x \rightarrow S^x & (A, (\alpha \geq T))^x = A^x, (\alpha \triangleleft c_{\alpha} : T^x) \\
(\forall(\alpha \geq T).S)^x = \forall(\alpha \triangleleft T^x) \Rightarrow S^x & (A, (x : T))^x = A^x, (x : T^x) \\
\perp^x = \forall \alpha. \alpha &
\end{array}$$

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

We show that $F_{\mu <}^p$ 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. 58)

Instance-bounded polymorphism. The language $x\text{MLF}$ [Rémy and Jakobowski, 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 type inference in MLF and have principal types—given optional type annotations of function parameters. As our current concern is not type inference but expressiveness, we use $x\text{MLF}$ rather than MLF for comparison with F_{ι}^p . By lack of space, we cannot formally present $x\text{MLF}$. Instead, we identify a subset F_{ι}^x of F_{ι}^p and explains how it closely relates to $x\text{MLF}$ without giving all the details of $x\text{MLF}$, which can be found in the extended version [Cretin and Rémy, 2011].

We first define the subset $F_{\iota}^{\mu x}$ of F_{ι}^p 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_{ι}^x as the restriction of $F_{\iota}^{\mu x}$ 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_{ι}^p .

We claim that $x\text{MLF}$ is equivalent to F_{ι}^x . 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 (see the extended version [Cretin and Rémy, 2011]). The proof for the direct inclusion is similar to the one in [Manzonetto and Tranquilli, 2010]. The proof for the reverse inclusion is new but not much difficult.

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

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

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

$$\begin{array}{c}
\text{XVAR} \\
\frac{x : T \in A}{A \vdash x : T \rightsquigarrow A^\sharp \vdash x : T^\sharp} \\
\\
\text{XLET} \\
\frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \quad A, (x : T) \vdash n : S \rightsquigarrow \Gamma, (x : \tau) \vdash N : \sigma}{A \vdash \text{let } x = m \text{ in } n : S \rightsquigarrow \Gamma \vdash (\lambda(x : \tau) N) M : \sigma} \\
\\
\text{XAPP} \\
\frac{A \vdash m : T \rightarrow S \rightsquigarrow \Gamma \vdash M : \tau \rightarrow \sigma \quad A \vdash n : T \rightsquigarrow \Gamma \vdash N : \tau}{A \vdash mn : S \rightsquigarrow \Gamma \vdash MN : \sigma} \\
\\
\text{XABS} \\
\frac{A, (x : T) \vdash m : S \rightsquigarrow \Gamma, (x : \tau) \vdash M : \sigma}{A \vdash \lambda(x : T)m : T \rightarrow S \rightsquigarrow \Gamma \vdash \lambda(x : \tau) M : \tau \rightarrow \sigma} \\
\\
\text{XTABS} \\
\frac{A, (\alpha \geq S) \vdash m : T \rightsquigarrow \Gamma, (\alpha \triangleleft c_\alpha : \sigma) \vdash M : \tau \quad \alpha \notin \text{ftv}(A, S)}{A \vdash \Lambda(\alpha \geq S) m : \forall(\alpha \geq S). T \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleleft c_\alpha : \sigma) M : \forall(\alpha \triangleleft \sigma) \Rightarrow \tau} \\
\\
\text{XTAPP} \\
\frac{A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau \quad A \vdash \phi : T \leq S \rightsquigarrow \Gamma \vdash G : \tau \triangleright \sigma}{A \vdash m \phi : S \rightsquigarrow \Gamma \vdash G(M) : \sigma}
\end{array}$$

Figure 35: x MLF: term translation

$$\begin{array}{c}
\text{XIBOT} \\
A \vdash T : \perp \leq T \rightsquigarrow \Gamma \vdash \diamond^{\forall \alpha. \alpha} \tau : \forall \alpha. \alpha \triangleright \tau \\
\\
\text{XIABS} \\
\frac{\alpha \geq T \in A}{A \vdash !\alpha : T \leq \alpha \rightsquigarrow \Gamma \vdash c_\alpha : \tau \triangleright \alpha} \\
\\
\text{XIUNDER} \\
\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}{\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} \\
\\
\text{XIINSIDE} \\
\frac{A \vdash \phi : T_1 \leq T_2 \rightsquigarrow \Gamma \vdash G : \tau_1 \triangleright \tau_2}{\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} \\
\\
\text{XIINTRO} \\
\frac{\alpha \notin \text{ftv}(T)}{A \vdash \wp : T \leq \forall(\alpha \geq \perp). T \rightsquigarrow \Gamma \vdash \lambda(\alpha \triangleleft c_\alpha : \forall \alpha. \alpha) \diamond^\tau : \tau \triangleright \forall(\alpha \triangleleft \forall \alpha. \alpha) \Rightarrow \tau} \\
\\
\text{XICOMP} \\
\frac{A \vdash \phi_1 : T_1 \leq T_2 \rightsquigarrow \Gamma \vdash G_1 : \tau_1 \triangleright \tau_2 \quad 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} \\
\\
\text{XIELIM} \\
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] \\
\\
\text{XIID} \\
A \vdash \mathbf{1} : T \leq T \rightsquigarrow \Gamma \vdash \diamond^\tau : \tau \triangleright \tau
\end{array}$$

Figure 36: x MLF: instance translation

$$\begin{array}{ll}
\alpha^y = \alpha & \emptyset^y = \emptyset \\
(\tau \rightarrow \sigma)^y = \tau^y \rightarrow \sigma^y & (\Gamma, (\alpha \triangleleft c : \tau))^y = \Gamma^y, (\alpha \geq \tau^y) \\
(\forall \alpha. \tau)^y = \forall (\alpha \geq \perp). \tau^y & (\Gamma, (x : \tau))^y = \Gamma^y, (x : \tau^y) \\
(\forall (\alpha \triangleleft \tau) \Rightarrow \sigma)^y = \forall (\alpha \geq \tau^y). \sigma^y & (\Gamma, \alpha)^y = \Gamma^y, (\alpha \geq \perp)
\end{array}$$

Figure 37: x MLF: type and environment reverse translation

$$\begin{array}{ll}
x^y = x & (\diamond \tau)^y = \mathbf{1} \\
(\lambda(x : \tau) M)^y = \lambda(x : \tau^y) M^y & c^y = !(\text{codom}(\tau)) \\
(M N)^y = M^y N^y & (\lambda \alpha G)^y = \mathfrak{X}; \forall (\alpha \geq) G^y \\
(\lambda \alpha M)^y = \Lambda(\alpha \geq \perp) M^y & (G \tau)^y = G^y; \forall (\geq \tau^y); \& \\
(M \tau)^y = M^y (\forall (\geq \tau^y); \&) & (\lambda(\alpha \triangleleft c : \tau) G)^y = \mathfrak{X}; \forall (\geq \tau^y); \forall (\alpha \geq) G^y \\
(\lambda(\alpha \triangleleft c : \tau) M)^y = \Lambda(\alpha \geq \tau^y) M^y & (G\{\tau \triangleright G'\})^y = G^y; \forall (\geq G'^y); \& \\
(M\{\tau \triangleright G\})^y = M^y (\forall (\geq G^y); \&) & (G\langle G'\rangle)^y = G'^y; G^y \\
(G\langle M\rangle)^y = M^y G^y &
\end{array}$$

Figure 38: x MLF: expression reverse translation

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 $\Gamma = A^x$ and $\tau = T^x$ hold.

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

We show that x MLF is included in F_t^x 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 $[m] = [M]$.

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. 59)

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. 59)

To show that F_t^x is included into x MLF, we define a translation for types and environments on Figure 37 and for expressions on Figure 38.

We show that F_t^x is included in x 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.

Proposition 44. The following assertions hold.

1. If $\Gamma \vdash M : \tau$ holds, then $\Gamma^y \vdash M^y : \tau^y$ holds and $[M^y] = [M]$.

2. If $\Gamma \vdash G : \tau \triangleright \sigma$ holds, then $\Gamma^y \vdash G^y : \tau^y \leq \sigma^y$ holds.

(Proof p. 59)

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

(Proof p. 59)

Extension of System F	F_η	$F_{<}$	$xMLF$	F_l^p
Deep instantiation	✓	-	✓	✓
Arrow congruence	✓	✓	-	✓
Permutation of \forall and \rightarrow	✓	-	-	✓
Upper bounds	-	✓	-	✓
Lower bounds	-	-	✓	✓

Figure 39: Language and feature comparison

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

Notice that $xMLF$ and $F_{<}$ 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_l^p —with some syntactic adjustment of course. For instance, the last example becomes $\lambda(\gamma \triangleleft c : \tau) \text{choose } \{\gamma \triangleleft (c\{\text{first}\})\}$ of type $\forall(\gamma \triangleleft \tau) \Rightarrow \gamma \rightarrow \gamma$ where τ is $\forall\beta. \forall(\alpha \triangleright \beta * \top) \Rightarrow \alpha \rightarrow \beta$. For instance, it can be coerced to $\forall(\gamma \triangleleft \sigma) \Rightarrow \gamma \rightarrow \gamma$ where σ is the type $\forall\beta. \forall\beta'. \forall(\alpha \triangleright \beta * (\beta' * \top)) \Rightarrow \alpha \rightarrow \beta$, using the coercion $\lambda(\gamma \triangleleft c : \sigma) \diamond \{\gamma \triangleleft c\langle G \rangle\}$ where G is itself the coercion:

$$\lambda\beta \lambda\beta' \lambda(\alpha \triangleright c : \beta * (\beta' * \top)) (\diamond \beta) \{\alpha \triangleright (\diamond^\beta * \text{Top}^{\beta' * \top}) \langle c \rangle\}$$

from τ to σ .

7 Weak F_l

Another solution to recover erasability is to prevent wedges from appearing in an evaluation context.

At first, it seems to suffice to use weak reduction on coercion abstraction. Indeed, if a coercion variable cannot appear under an evaluation context, it cannot appear in a wedging configuration. However, since $\lambda(c : \varphi) M$ is irreducible, its erasure $[M]$ 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 $[M]$ 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_l is defined on Figure 40 as a restriction of the syntax of F_l . We remove distributivity of coercion abstraction on term abstraction $\text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow}$ 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 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 Weak F_l and F_l are the same.

$M ::= \dots \not\mid \lambda(c : \varphi) M \mid \lambda(c : \varphi) u$	expressions
$G ::= \dots \not\mid \mathbf{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow}$	coercions
$v ::= \lambda(x : \tau) M \mid \lambda\alpha v \mid \lambda(c : \varphi) u \mid (v, v) \mid \mathbf{Top}^\tau \langle v \rangle$	values
$u ::= v \mid G \langle u \rangle$	value forms
$E ::= [] M \mid M [] \mid ([], M) \mid (M, []) \mid [] . 1 \mid [] . 2 \mid \lambda\alpha [] \mid [] \tau \mid G \langle [] \rangle \mid [] \{ G \}$	evaluation ctx
$\mathbf{REDCOERCOERLAM}^w$ $(\lambda(c : \varphi) G) \langle u \rangle \rightsquigarrow_l \lambda(c : \varphi) G \langle u \rangle$	

Figure 40: Weak F_l : syntax and semantics $wrt F_l$

Changes in the dynamic semantics The reduction relation of Weak F_l is a subrelation of the reduction relation of F_l that prevents evaluation under term or coercion abstractions and preserves the value restriction. Evaluation contexts are modified accordingly: $\lambda(x : \tau) []$ and $\lambda(c : \varphi) []$ are removed. Rule $\mathbf{REDCOERCOERLAM}$ (the coercion abstraction part of $\mathbf{REDCOERFILL}$) is restricted to make it call-by-value. Indeed, keeping the F_l rule:

$$(\lambda(c : \varphi) G) \langle M \rangle \rightsquigarrow_l \lambda(c : \varphi) G \langle M \rangle$$

would place the arbitrary term M under a coercion abstraction. We also completely remove $\mathbf{REDCOERDISTCOERARROW}$ as it can never happen.

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

Additionally, the reduction relation of Weak F_l is included into the reduction relation of F_l . This is the case because we restricted the evaluation contexts to implement weak reduction and the $\mathbf{REDCOERCOERLAM}$ rule to preserve the value restriction.

As a consequence, subject reduction and normalization properties of F_l are preserved in Weak F_l . The progress lemma cannot be lifted in a similar way, because the reduction relation of Weak F_l is strictly included into the one of F_l 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 $M \rightsquigarrow_{\beta_l} N$ holds, then N is syntactically correct.*

(Proof p. 59)

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

(Proof p. 60)

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

(Proof p. 60)

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

(Proof p. 60)

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

(Proof p. 60)

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

1. *If τ is of the form $\tau \rightarrow \tau$, 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 $\text{Top}^\tau \langle v \rangle$.

(Proof p. 60)

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

(Proof p. 60)

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 is 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 evaluation contexts in F_ι^w are Call-by-value evaluation contexts in λ -calculus.

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

1. If $M \rightsquigarrow_\beta N$, then $[M] \rightsquigarrow [N]$.
2. If $M \rightsquigarrow_\iota N$, then $[M] = [N]$.

(Proof p. 61)

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

1. If τ is $\sigma \rightarrow \sigma'$ (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 $\vec{\alpha} \vdash Q'[\lambda(x : \rho) M] : \forall\alpha. \rho'$ (resp. $\vec{\alpha} \vdash Q'[(M, N)] : \forall\alpha. \rho'$) 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^τ : 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 $\vec{\alpha} \vdash Q'[\lambda(x : \rho) M] : \sigma \rightarrow \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι -normal-form, since REDCOERARROW applies.
 - $\text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha}$: 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.
 - $(G * G)$, $\text{Dist}_{(\tau * \tau)}^{\forall \alpha}$, and $\text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow}$: No case apply.
 - $G \xrightarrow{\tau} G$ and $\text{Dist}_{\tau \rightarrow \tau}^{\forall \alpha}$ (resp.): No case apply.
 - $(G * G)$ (resp.): By induction hypothesis, REDCOERPROD applies.
 - $\text{Dist}_{(\tau * \tau)}^{\forall \alpha}$ (resp.): By induction hypothesis used twice, $\text{REDCOERDISTTYPEPROD}$ applies.
 - $\text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow}$ (resp.): By induction hypothesis used twice, $\text{REDCOERDISTCOERPROD}$ 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 $\vec{\alpha} \vdash Q'[\lambda(x : \rho) M] : (\sigma \triangleright \sigma') \Rightarrow \tau$ (resp. $\vec{\alpha} \vdash Q'[(M, N)] : (\sigma \triangleright \sigma') \Rightarrow \tau$). 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 $\vec{\alpha} \vdash M : \tau$ and $\llbracket M \rrbracket \rightsquigarrow a$, then $M \rightsquigarrow_{\iota}^* \rightsquigarrow_{\beta} N$ such that $\llbracket N \rrbracket = a$.*

Proof. We show that the ι -normal-form of M β -reduces to N with $\llbracket N \rrbracket$ equal to a . Since F_{ι} strongly normalizes, we may assume, without loss of generality, that M is already in ι -normal-form. Because $\llbracket M \rrbracket$ 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 $E[Q[\lambda(x : \tau) M_1] M_2]$ where E is an evaluation 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 E , 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 $E[(\lambda(x : \tau) M_1) M_2]$ and β -reduces to $E[M_1[x \leftarrow M_2]]$ 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} , $F_{<}$, and $x\text{MLF}$. Parts of F_{ι}^x is closely related to the work of Manzonetto and Tranquilli [Manzonetto and Tranquilli, 2010] who proposed the first encoding of $x\text{MLF}$ in a calculus of coercions, for the main purpose of proving the termination of $x\text{MLF}$. They exhibit a type and semantics preserving encoding of $x\text{MLF}$ into (their version of) F_{ι}^x and show a simulation of computation between their F_{ι}^x and System F. Unfortunately, subject reduction and other properties that depend on it do not hold in their system. Our version of F_{ι}^x can be seen as a fix to their definition. Hence, there are many resemblances between their development of F_{ι}^x and our development of F_{ι} —but the typing rules differ. We omitted the proof of inclusion from

$x\text{MLF}$ into F_ι^x by lack of space, but also because it resembles theirs. In fact, their translation of $x\text{MLF}$ into F_ι^x has itself been inspired by the translation of MLF into System F by Leijen and Löh [Leijen and Löh, 2005, Leijen, 2007]. However, Manzonetto and Tranquilli restrict their study to the termination of $x\text{MLF}$ without any interest in F_η or $F_{<}$, while our main interest is not in F_ι^x , but in F_ι^p 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 $x\text{MLF}$.

Although F_ι subsumes core $F_{<}$, we have not included records in F_ι , which are often the first application of $F_{<}$. Our formalization in the extended version [Cretin and Rémy, 2011] 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 $F_{<}$ and require an expressive runtime system so that subtyping is erasable.

Record subtyping in $F_{<}$ 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 that shows 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 [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 part 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_ι^λ . 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 thus 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 thus not erasable in our terminology. The two languages F_ι and FC_2 have followed orthogonal approaches, and are thus not easily comparable, but 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 considered 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 [SergueïLenglet and Wells].

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, Parametric 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 , $x\text{MLF}$, and $F_{<}$. We believe it is an interesting point in the design space. It also shows that an extension of $x\text{MLF}$ 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 cases but still prevent wedges from being typable is probably the easiest extension to this work. An obvious but minor generalization is to let $\lambda(\alpha \diamond \bar{\tau} : \bar{c}) M$ 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 does not permit building wedges. However, finding a suitable notion of consistency in the presence of distributivity is challenging.

Beyond F_ι We have so far explored restrictions of F_ι to prevent wedges from appearing in an evaluation context. Instead, we could perhaps extend the calculus to allow breaking them. 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 $\text{Left } G$ and $\text{Right } G$ 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 solved. 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 for ever. We leave this exploration for future work.

Leaving F_η and freezing quantifiers We have added coercion abstraction to the language F_η because F_η 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. Preliminary investigations are actually promising.

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 [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, as they already raise a problem in System F where they cannot have an erasing semantics with a strong evaluation strategy. This is the reason why we left them out of F_ι and introduced a top type instead. 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 the 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>.
- J. Cretin and D. Rémy. Extending System F with Abstraction over Erasable Coercions. Technical report, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, Apr. 2011. URL <http://crystal.inria.fr/~remy/coercions/Cretin-Remy:coercions@inria2011.pdf>.
- 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. Löb. 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>.
- 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.
- Sergueï Lenglet and J. B. Wells. Expansion for forall-quantifiers. Available electronically. URL <http://sardes.inrialpes.fr/~slenglet/papers/systemFs.pdf>.
- 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 $\Gamma, (y : \tau) \vdash M[x \leftarrow G_1(y)] : \sigma'$. 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 $\alpha \notin \text{ftv}(\tau)$) to show that $\Gamma, (x : \tau), \alpha$ is a valid extension of $\Gamma, \alpha, (x : \tau)$.
- REDCOERDISTCOERARROW: We have $\Gamma, (c : \varphi \triangleright \rho'), (x : \tau) \vdash M : \sigma$. We build $\Gamma, (x : \tau), (c : \varphi \triangleright \rho') \vdash M : \sigma$ 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 $\Gamma, (c : \varphi \triangleright \rho') \vdash M : \tau$ and $\Gamma, (c : \varphi \triangleright \rho') \vdash N : \sigma$. 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.
 - $\lambda(c : \varphi \triangleright \rho') []$: We have $\Gamma, (c : \varphi \triangleright \rho') \vdash G : \tau \triangleright \sigma$ and $\Gamma \vdash M : \tau$. We use Lemma 3 to build $\Gamma, (c : \varphi \triangleright \rho') \vdash M : \tau$ (and Lemma 2 to show $\Gamma, (c : \varphi \triangleright \rho') \vdash ok$).
 - $[]\{G\}$: 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 .

- $E[M]$ when M is not a value: The induction hypothesis applies to M , and we use REDCONTEXTBETA or REDCONTEXTIOTA to build $E[M] \rightsquigarrow_{\beta\iota} E[N]$.
- $x, \lambda(x : \tau) v, \lambda\alpha v, \lambda(c : \tau_1 \triangleright \tau_2) v$, and (v_1, v_2) : These are values.
- $\diamond^\tau, G_1 \xrightarrow{\tau} G_2, \text{Dist}_{\tau \rightarrow \sigma}^{\forall\alpha}, \text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow}, (G_1 * G_2), \text{Dist}_{(\tau * \sigma)}^{\forall\alpha}, \text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow}, \text{Top}^\tau$, 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^τ : 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.
 - $\text{Dist}_{\tau \rightarrow \sigma}^{\forall\alpha}$: 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.
- $\text{Dist}_{\tau \rightarrow \sigma}^{\varphi \Rightarrow}$: Using Lemma 9 on v .
 - * p : It is a value.
 - * $\lambda(c : \varphi \triangleright \rho') v'$: 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.
- $\text{Dist}_{(\tau * \sigma)}^{\forall \alpha.}$: 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.
- $\text{Dist}_{(\tau * \sigma)}^{\varphi \Rightarrow}$: Using Lemma 9 on v .
 - * p : It is a value.
 - * $\lambda(c : \varphi \triangleright \rho') v'$: 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. Evaluation contexts are reified on System F contexts, and because we are in strong reduction, all contexts are evaluation 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

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

- REDCOERDIST: We have

$$\begin{aligned}
 & (\lambda(y : \forall \alpha. [\tau] \rightarrow [\sigma]) \lambda(x : [\tau]) \lambda \alpha y \alpha x) (\lambda \alpha \lambda(x : [\tau]) [M]) \\
 \rightsquigarrow & \lambda(x : [\tau]) \lambda \alpha (\lambda \alpha \lambda(x : [\tau]) [M]) \alpha x \\
 \rightsquigarrow & \lambda(x : [\tau]) \lambda \alpha (\lambda(x : [\tau]) [M]) x \\
 \rightsquigarrow & \lambda(x : [\tau]) \lambda \alpha [M]
 \end{aligned}$$

- REDCOERDISTCOERARROW: We have

$$\begin{aligned}
& (\lambda(y : ([\rho] \rightarrow [\rho']) \rightarrow [\tau] \rightarrow [\sigma]) \lambda(x : \tau) \lambda(x_c : [\rho] \rightarrow [\rho']) y x_c x) \\
& \quad (\lambda(x_c : [\rho] \rightarrow [\rho']) \lambda(x : [\tau]) [M]) \\
& \rightsquigarrow \lambda(x : \tau) \lambda(x_c : [\rho] \rightarrow [\rho']) (\lambda(x_c : [\rho] \rightarrow [\rho']) \lambda(x : [\tau]) [M]) x_c x \\
& \rightsquigarrow \lambda(x : \tau) \lambda(x_c : [\rho] \rightarrow [\rho']) (\lambda(x : [\tau]) [M]) x \\
& \rightsquigarrow \lambda(x : \tau) \lambda(x_c : [\rho] \rightarrow [\rho']) [M]
\end{aligned}$$

- REDCOERPROD: We have

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

- REDCOERDISTTYPEPROD: We have

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

- REDCOERDISTCOERPROD: We have

$$\begin{aligned}
& (\lambda(y : ([\rho] \rightarrow [\rho']) \rightarrow ([\tau] * [\sigma])) \\
& \quad (\lambda(x_c : [\rho] \rightarrow [\rho']) (y x_c).1, \lambda(x_c : [\rho] \rightarrow [\rho']) (y x_c).2)) \\
& \quad (\lambda(x_c : [\rho] \rightarrow [\rho']) ([M], [N])) \\
& \rightsquigarrow (\lambda(x_c : [\rho] \rightarrow [\rho']) ((\lambda(x_c : [\rho] \rightarrow [\rho']) ([M], [N])) x_c).1 \\
& \quad , \lambda(x_c : [\rho] \rightarrow [\rho']) ((\lambda(x_c : [\rho] \rightarrow [\rho']) ([M], [N])) x_c).2) \\
& \rightsquigarrow (\lambda(x_c : [\rho] \rightarrow [\rho']) ([M], [N]).1 \\
& \quad , \lambda(x_c : [\rho] \rightarrow [\rho']) ([M], [N]).2) \\
& \rightsquigarrow (\lambda(x_c : [\rho] \rightarrow [\rho']) [M], \lambda(x_c : [\rho] \rightarrow [\rho']) [N])
\end{aligned}$$

- REDCOERDOT: We have $(\lambda(x : [\tau]) x) [M] \rightsquigarrow [M]$
- REDCOERCOER: We have $(\lambda(x : [\tau]) P[[G] x]) [M] \rightsquigarrow P[[G] [M]]$
- 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

$$- \Gamma; \Delta * (\phi_2 : \sigma') \vdash G_2 : \sigma,$$

- $\Gamma, \Delta \vdash M : \tau' \rightarrow \sigma'$,
- $\Gamma, \Delta; \Delta' \star (\phi_1 : \tau) \vdash G_1 : \tau'$,
- $\Gamma \vdash N : \tau$.

which leads to the result with weakening, substitution, and `LEXPRTERMAPP`.

- `LREDETAPRODST`: 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.

- $E[M]$: If M reduces, then we apply the context rule.
- $x, \lambda(x : \tau) v, (v_1, v_2), \lambda \alpha v, \lambda(c : \varphi) v, \text{Top}^\tau v$, and cv : 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`, `LREDETAPRODST`, `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 $\tau \rightarrow \sigma$ under Γ . We observe that $(G_1(\lambda\Delta' \diamond)) \rightarrow \diamond$ is typed $\tau' \rightarrow \sigma' \triangleright \tau \rightarrow \sigma'$. So its application to M is typed $\tau \rightarrow \sigma'$. Abstracting over Δ gives type $\forall\Delta. \tau \rightarrow \sigma'$. Applying the distributivity returns a term of type $\tau \rightarrow \forall\Delta. \sigma'$ (this is where we use $\Gamma \vdash \tau$). And finally we apply $\diamond \rightarrow G_2$ which coerces our term into a term of type $\tau \rightarrow \sigma$ 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 $E[(\lambda(x : \tau) M_0) M_1]$.

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_ι^p . Thus the reduction in F_ι^p 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 REDCONTEXT-BETA and REDCONTEXTIOTA, we can only consider cases where subterms are values.

- $x, \lambda(x : \tau) v, \lambda\alpha v, \lambda(\alpha \triangleleft c : \tau) v$, and (v, v) : These are values.
- $\diamond\tau, G \xrightarrow{\tau} G, \text{Dist}_{\tau \rightarrow \sigma}^{\forall\alpha}, \text{Dist}_{\tau \rightarrow \sigma}^{\forall\alpha \triangleleft \rho \Rightarrow}, (G * G), \text{Dist}_{(\tau * \sigma)}^{\forall\alpha}, \text{Dist}_{(\rho * \tau)}^{\alpha \Rightarrow} \sigma, \text{Top}^\tau$, 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.
- $v\{\tau \triangleleft G\}$: Using Lemma 34 on v .
 - p : It is a value.
 - $\lambda(\alpha \triangleleft c : \tau) v$: 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.
- $G(v)$: 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, Top^τ : These are values.
 - \diamond^τ : REDCOERDOT applies.
 - $\lambda\alpha W, W \tau, \lambda(\alpha \diamond c : \tau) W, W\{\tau \diamond G\}$, and $G(W)$: REDCOERFILL applies.
 - $G \xrightarrow{\tau} G$: Using Lemma 34 on v .
 - * p : It is a value.
 - * $\lambda(x : \tau) v$: REDCOERARROW applies.
 - $\text{Dist}_{\tau \rightarrow \tau}^{\forall\alpha}$: 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.
 - $\text{Dist}_{\tau \rightarrow \tau}^{\forall\alpha \diamond \tau \Rightarrow}$: Using Lemma 34 on v .
 - * p : It is a value.
 - * $\lambda(\alpha \diamond c : \tau) v$: 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.
 - $\text{Dist}_{(\tau * \tau)}^{\forall\alpha}$: 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.
 - $\text{Dist}_{(\tau * \tau)}^{\forall\alpha \diamond \tau \Rightarrow}$: Using Lemma 34 on v .
 - * p : It is a value.
 - * $\lambda(\alpha \diamond c : \tau) v$: 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.
- $m \forall \rightsquigarrow \Lambda(\alpha \geq \perp) m$ (with $\alpha \notin \text{ftv}(m)$): We have $(\lambda(\alpha \triangleleft c_\alpha : \forall \alpha. \alpha) \diamond) \langle M \rangle \rightsquigarrow_\iota \lambda(\alpha \triangleleft c_\alpha : \forall \alpha. \alpha) M$ 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 $\text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow}$; then, we added a value restriction on coercion abstraction. We notice that no rules introduce a $\text{Dist}_{\tau \rightarrow \tau}^{\varphi \Rightarrow}$, so we only need to check the preservation of the second restriction:

- REDCONTEXTBETA and REDCONTEXTIOTA: By induction on E .

- **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_l . ■

Proof of Lem 48

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

Proof of Prop 49

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

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_l . 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 evaluation contexts allow it. For each evaluation 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^τ , $G \xrightarrow{\tau} G$, $\text{Dist}_{\tau \rightarrow \sigma}^{\forall \alpha}$, $(G * G)$, $\text{Dist}^{\forall \alpha. (\tau * \sigma)}$, $\text{Dist}^{(\varphi \triangleright \rho') \Rightarrow (\tau * \sigma)}$, 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^τ : 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.
- $\text{Dist}_{\tau \rightarrow \tau}^{\forall\alpha}$: Using Lemma 51 consecutively twice on v , REDCOERDIST applies.
- $(G * G)$: Using Lemma 51 on v , REDCOERPROD applies.
- $\text{Dist}_{(\tau * \tau)}^{\forall\alpha}$: Using Lemma 51 consecutively twice on v , $\text{REDCOERDISTTYPEPROD}$ applies.
- $\text{Dist}_{(\tau * \tau)}^{\varphi \Rightarrow}$: Using Lemma 51 consecutively twice on v , $\text{REDCOERDISTCOERPROD}$ 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