On the Power of Coercion Abstraction

Julien Cretin Didier Rémy
INRIA
{julien.cretin,didier.remy}@inria.fr

Abstract

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

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Design, Languages, Theory

Keywords Type, System F, F-eta, Polymorphism, Coercion, Conversion, Retyping functions, Type containment, Subtyping, Bounded Polymorphism.

1. Introduction

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

Contravariance, the key ingredient of subtyping polymorphism, is already modeled in the language F_η proposed by Mitchell [1988]. On the one-hand, F_η is the closure of System F by η-conversion. On the other-hand it is a language of coercions: a retyping context from τ to σ in F_η is a one hole context G such that $\lambda(x:\tau)\ G\langle x\rangle$ is an η-expansion of the identity and has type $\tau \to \sigma$ in System F. For now, we write $G\langle M\rangle$ for filling a context G with a term M and $\Diamond^\tau$ 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 $\lambda(x:\tau_1)\ G_2\langle\Diamond^{\tau_1'\to\tau_2}\ (G_1\langle x\rangle)\rangle$ is a retyping context from type $\tau_1' \to \tau_2$ to $\tau_1 \to \tau_2'$.

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

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

Is there a language that supersedes both F_η and $\mathsf{F}_{<:}$? 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 $\mathsf{F}_{<:}$, but the two also differ significantly, since for instance, type conversion is not congruent on arrow types in MLF.

Surprisingly, among the three languages F_η, $\mathsf{F}_{<:}$, and MLF, any combination of two have features in common that the other one lacks! Hence, the challenge becomes whether all their features can be simultaneously combined together. This question has in fact already been raised in previous work on MLF [Rémy and Yakobowski 2010].

Our contributions We answer positively by introducing a language F_ι^p that extends F_η with abstraction over retyping functions, combining all features simultaneously in a unified framework (§5). The language F_ι^p subsumes F_η, $\mathsf{F}_{<:}$, and MLF (§6); it also fixes

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

and extends a previous language of coercions designed for modeling MLF alone [Manzonetto and Tranquilli 2010]. Our subset of F_ι^p that coincides with MLF is *well-behaved*: it satisfies the subject reduction and progress lemmas and strongly normalizes. It also has an untyped semantics.

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

System F 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.* equality); and conversely, if the erasure of a term M β-reduces to a term $\mathcal{M}'$, then M also reduces by a sequence of ι-reductions followed by a single β-reduction to a term whose erasure is $\mathcal{M}'$.

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

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

Erasability also means that the dynamic semantics of our language is ultimately that of the underlying λ-calculus—possibly enriched with more constructs. Therefore the semantics only depends on the reduction strategy we choose and not on the typechecking details nor on the coercions we may use. Types are useful for programmers to understand their programs. It is also useful for programmers that types do not determine the semantics of their programs. At least, we should 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 F_η Let us first return to the definition of F_η, which in Mitchell's original presentation is given in Curry-style. It is defined by adding to System F a *type containment* rule that allows to convert a term $\mathcal{M}$ of type τ to one of type σ whenever there exists a retyping context from type τ to σ, which we write $\vdash \tau \triangleright \sigma$. This judgment, called type containment, is equivalent to the existence of a (closed) retyping function $\mathcal{M}'$ of System F such that $\vdash \mathcal{M}' : \tau \to \sigma$. Interestingly, Mitchell gave another characterization of type containment, exhibiting a proof system for the judgment $\vdash \tau \triangleright \sigma$, which can be read back as the introduction of a language of coercions whose expressions G witness type containment derivations. Then, we write $\vdash G : \tau \triangleright \sigma$ where G fully determines the typing derivation (much as a Church-style System-F term M fully determines its typing derivation). For example, $G_1 \to G_2$ is a coercion that, given a function M, returns a function that coerces its argument with G_1, passes it to M, and coerces the result with G_2—using the contravariance of type containment. (A full presentation of coercions appears in §2 where F_η is described as a subset of F_ι.)

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

In Church-style System F, the use of a coercion G around a term M is witnessed explicitly as $G\langle M\rangle$. (We may continue seeing a coercion G as a retyping context and reading this as filling the hole of G or, equivalently, see G as a retyping function and read this as an application of a coercion to a term.) Reduction rules are added to reduce such applications when both G and M have been sufficiently evaluated—in a way depending on the form of both—so that a coercion G is never stuck in the middle of a (well-typed) redex as in $(G\langle\lambda(x : \tau)\ M\rangle)\ N$. The type system ensures that G is of a certain shape for which a reduction exists. 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 F_ι To abstract over coercion functions, we introduce a new form $\lambda(c : \tau \triangleright \sigma)\ M$ in F_ι, where the parameter c stands for a coercion function that can be used inside M to convert an expression of type τ to one of type σ. This abstraction can be typed as $(\tau \triangleright \sigma) \Rightarrow \rho$ where ρ is the type of M. Correspondingly, we need a new application form $M\{G\}$ to pass a coercion G to a coercion abstraction, *i.e.* a term M of type $(\tau \triangleright \sigma) \Rightarrow \rho$.

By typing constraints, coercion abstractions can only be instantiated with coercions, which by construction are erasable. Thus, intuitively, coercions do not really contribute to the computation.

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

$$
\textsc{RedContext}\ \frac{\mathcal{M} \rightsquigarrow \mathcal{M}'}{\mathcal{C}[\mathcal{M}] \rightsquigarrow \mathcal{C}[\mathcal{M}']} \qquad \textsc{RedBeta}\ (\lambda x.\mathcal{M})\,\mathcal{M}' \rightsquigarrow \mathcal{M}[x \leftarrow \mathcal{M}']
$$

Figure 1. λ-calculus: syntax and semantics

Is this enough to erase them? Formally, we can exhibit a forward simulation between reduction of terms in F_ι and of their erasure in the untyped λ-calculus. Moreover, F_ι has the subject reduction property and is strongly normalizing. Still, coercions cannot be erased in F_ι, since although they do not create new evaluation paths, they may block existing evaluation paths: a subterm may be stuck while its erasure could proceed. Since coercions are erasable in F_η, this can only be due to the use of a coercion variable. Indeed, a coercion variable c may appear in the middle of a β-redex as in $(c\langle\lambda(x : \tau)\ M\rangle)\ N$. This is irreducible because reduction of coercion applications $G\langle M\rangle$ depends simultaneously on the shapes of G and M so that no rule fires when G is unknown. More generally, we call a wedge an irreducible term of the form $(G\langle\lambda(x : \tau)\ M\rangle)\ N$. Notice that the erasure of a wedge $(\lambda(x : \tau)\ \lfloor M\rfloor)\ \lfloor N\rfloor$ can be reduced, immediately. Hence, the existence of wedges in reduction contexts prevents erasability.

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

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

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

2. The language F_ι

The language F_ι generalizes F_η with abstraction over coercions. We recall the definition of the (untyped) λ-calculus on Figure 1. We normally 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. However, we omit them in this summary for conciseness of exposition. We occasionally refer to them informally to explain how the definitions generalize to pairs. We assume an enumerable collection of term variables, ranged over by letters x and y. Untyped terms, written $\mathcal{M}$, include variables, abstractions $\lambda x.\mathcal{M}$, and applications $\mathcal{M}\,\mathcal{M}'$. The semantics of untyped λ-terms is given by a small-step strong reduction relation. Reduction contexts of the λ-calculus are all one-hole contexts, written $\mathcal{C}$. We now write $\mathcal{C}[\mathcal{M}]$ for the term obtained by filling the hole of $\mathcal{C}$ with $\mathcal{M}$ and $\mathcal{M}[x \leftarrow \mathcal{M}']$ for the capture avoiding substitution of $\mathcal{M}'$ for x in $\mathcal{M}$. Expressions are considered equal up to the renaming of bound variables, which are defined in the usual way. This convention applies to the λ-calculus, as well as to all typed languages presented below.

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

Figure 2. Syntax of F_ι.

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 α and β. Types are type variables, arrow types $\tau \to \tau$, 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, abstractions $\lambda(x : \tau)\ M$, applications MM, type abstractions $\lambda\alpha\ M$, and type applications $M\,\tau$. A construct already present in F_η is the use of the application $G\langle M\rangle$ of a coercion G to a term M. There are two new constructs specific to F_ι and not present in F_η: coercion abstraction $\lambda(c : \varphi)\ M$ which is annotated with the coercion type φ; and coercion application $M\{G\}$ that passes a coercion G to a term M—and should not be confused with the earlier construct $G\langle M\rangle$ of F_η that places a coercion G around a term M.

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

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

$$\lfloor x \rfloor = x \qquad \lfloor \lambda\alpha\ M \rfloor = \lfloor M\,\tau \rfloor = \lfloor M \rfloor$$
$$\lfloor \lambda(x:\tau)\ M \rfloor = \lambda x.\lfloor M \rfloor \qquad \lfloor G\langle M \rangle \rfloor = \lfloor M \rfloor$$
$$\lfloor M\ N \rfloor = \lfloor M \rfloor \lfloor N \rfloor \qquad \lfloor \lambda(c:\varphi)\ M \rfloor = \lfloor M\{G\} \rfloor = \lfloor M \rfloor$$

Figure 3. Coercion erasure

makes it appear as if it had been defined as $\lambda(y:\tau)\ \lambda\alpha\ N$—which is actually what it will reduce to. The other distributivity coercion $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}$, which stands for $\lambda(x:\tau)\ \lambda(c:\varphi)\ ([]\{c\}\ x)$, is similar but permutes a coercion abstraction with a term abstraction.

We may need more distributivity coercions when extending the language of terms. Hence, the notation $\mathsf{Dist}^{a}_{b\to c}$ uses the following mnemonic: the superscript a and the subscript $b \to c$ indicate the kind of the first and second type constructs, respectively. They can be combined into $ab \to c$ to form the type of the hole, or recombined into $b \to ac$ to form the type of the coerced term. For example, $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ is a coercion from $\forall\alpha.\,(\tau \to \sigma)$ to $\tau \to (\forall\alpha.\,\sigma)$, while $\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}$ is a coercion from $\varphi \Rightarrow (\tau \to \sigma)$ to $\tau \to (\varphi \Rightarrow \sigma)$.

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

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

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

2.2 Typing rules

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

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

As usual, we require that typing contexts do not bind twice the same variable, which is not restrictive as all expressions are considered equal up to renaming of bound variables (details can be found in the extended version).

Typing rules for terms are described in Figure 4. Rules TermVar, TermTermLam, TermTermApp, 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 $\tau \to \sigma$. Rule TermCoerLam is similar to TermTermLam, except that the parameter c stands for a coercion of coercion type φ instead of a term of type σ: the result is a coercion abstraction of type $\varphi \Rightarrow \tau$.

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

Figure 6. System F_ι: values and reduction contexts

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 $\mathtt{Top}^{\tau}$ converts an expression of type τ to the top type, provided τ is well-formed. The arrow coercion $G_1 \xrightarrow{\tau_1} G_2$ turns an arrow type $\tau_1' \to \tau_2$ into an arrow type $\tau_1 \to \tau_2'$, provided G_i coerces type τ_i into τ_i' for i in $\{1,2\}$. The distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ turns an expression of type $\forall\alpha.\,\tau \to \sigma$ into one of type $\tau \to \forall\alpha.\,\sigma$ provided τ is well-formed in the current environment, which prevents α from appearing free in τ, and σ is well-formed in the current environment extended with α. Finally, Rule CoerDistCoerArrow is similar to CoerDistTypeArrow, but swaps a coercion abstraction and a term abstraction.

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

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

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

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

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

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

Figure 4. System F_ι: term typings

$$\textsc{CoerDot}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \Diamond^\tau : \tau \triangleright \tau} \qquad \textsc{CoerForget}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathsf{Top}^\tau : \tau \triangleright \top} \qquad \textsc{CoerArrow}\ \frac{\Gamma \vdash G_1 : \tau_1 \triangleright \tau_1' \qquad \Gamma \vdash G_2 : \tau_2 \triangleright \tau_2'}{\Gamma \vdash G_1 \xrightarrow{\tau_1} G_2 : (\tau_1' \to \tau_2) \triangleright (\tau_1 \to \tau_2')} \qquad \textsc{CoerDistTypeArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\forall\alpha.}_{\tau \to \sigma} : \forall\alpha.\, (\tau \to \sigma) \triangleright \tau \to \forall\alpha.\, \sigma}$$

$$\textsc{CoerDistCoerArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \varphi \qquad \Gamma \vdash \sigma}{\Gamma \vdash \mathsf{Dist}^{\varphi \Rightarrow}_{\tau \to \sigma} : (\varphi \Rightarrow (\tau \to \sigma)) \triangleright (\tau \to (\varphi \Rightarrow \sigma))} \qquad \textsc{CoerTypeLam}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda\alpha\ G : \tau \triangleright \forall\alpha.\, \sigma} \qquad \textsc{CoerTypeApp}\ \frac{\Gamma \vdash G : \tau' \triangleright \forall\alpha.\, \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash G\,\tau : \tau' \triangleright \sigma[\alpha \leftarrow \tau]}$$

$$\textsc{CoerCoer}\ \frac{\Gamma \vdash G : \tau \triangleright \sigma \qquad \Gamma \vdash G' : \rho \triangleright \tau}{\Gamma \vdash G\langle G'\rangle : \rho \triangleright \sigma} \qquad \textsc{CoerCoerLam}\ \frac{\Gamma, c : \varphi \vdash G : \tau \triangleright \sigma}{\Gamma \vdash \lambda(c : \varphi)\ G : \tau \triangleright (\varphi \Rightarrow \sigma)} \qquad \textsc{CoerCoerApp}\ \frac{\Gamma \vdash G' : \tau \triangleright (\varphi \Rightarrow \sigma) \qquad \Gamma \vdash G : \varphi}{\Gamma \vdash G'\{G\} : \tau \triangleright \sigma} \qquad \textsc{CoerVar}\ \frac{\Gamma \vdash ok \quad c : \varphi \in \Gamma}{\Gamma \vdash c : \varphi}$$

Figure 5. System F_ι: coercion typings

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

$$\textsc{RedTerm}\ (\lambda(x : \tau)\ M)\ N \rightsquigarrow_\beta M[x \leftarrow N] \qquad \textsc{RedType}\ (\lambda\alpha\ M)\,\tau \rightsquigarrow_\iota M[\alpha \leftarrow \tau]$$

$$\textsc{RedCoer}\ (\lambda(c : \varphi)\, M)\{G\} \rightsquigarrow_\iota M[c \leftarrow G]$$

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

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

$$\textsc{RedCoerDistCoerArrow}\ \mathsf{Dist}^{\varphi' \Rightarrow}_{\tau' \to \sigma'}\langle\lambda(c : \varphi)\ \lambda(x : \tau)\ M\rangle \rightsquigarrow_\iota \lambda(x : \tau)\ \lambda(c : \varphi)\ M$$

$$\textsc{RedCoerDot}\ \Diamond^\tau\langle M\rangle \rightsquigarrow_\iota M \qquad \textsc{RedCoerFill}\ (P[G])\langle M\rangle \rightsquigarrow_\iota P[G\langle M\rangle]$$

Figure 7. Reduction rules for F_ι

2.3 Dynamic semantics

The dynamic semantics of System F_ι is given by a standard small-step strong reduction relation. The syntax of values and reduction contexts is recalled on Figure 6.

A value is an abstraction of a value, an opaque value $\mathsf{Top}^\tau\langle v\rangle$, or a prevalue. A prevalue is a variable, a prevalue applied to a value, type, or coercion, a value coerced by a coercion variable, or a partial application of a distributivity coercion. Reduction contexts C are all one-hole term contexts. For convenience, we have distinguished a subset of reduction contexts P, called *retyping reduction contexts*: a term M placed in a retyping reduction context is just a retyping of M, *i.e.* a term that behaves as M but possibly with another type.

Reduction rules are defined on Figure 7. We have indexed the reduction rules so as to distinguish between β-steps with computational content (RedTerm), that are preserved after erasure, and ι-steps (RedType) that become equalities after erasure. We write $\rightsquigarrow_{\beta\iota}$ for the union of $\rightsquigarrow_\beta$ and $\rightsquigarrow_\iota$.

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

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

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

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

- When G is an arrow coercion $G_1 \xrightarrow{\tau} G_2$ and M is a function $\lambda(x : \sigma)\ M$, Rule RedCoerArrow reduces the application by pushing G_1 on all occurrences of x in M and G_2 outside of M. This changes the type of the parameter x from σ to τ, hence the need for the annotation τ on arrow coercions.
- When G is a distributivity coercion $\mathsf{Dist}^{\forall\alpha.}_{\tau' \to \sigma'}$ and M is a polymorphic function $\lambda\alpha\ \lambda(x : \tau)\ M$, Rule RedCoerDistTypeArrow reduces the application to $\lambda(x : \tau)\ \lambda\alpha\ M$ by exchanging the type and value parameters; this is sound since α cannot be free in τ.
- When G is a distributivity coercion $\mathsf{Dist}^{\varphi' \Rightarrow}_{\tau' \to \sigma'}$ and M is a coercion abstraction followed by a value abstraction $\lambda(c : \varphi)\ \lambda(x : \tau)\ M$, Rule RedCoerDistCoerArrow reduces the application to $\lambda(x : \tau)\ \lambda(c : \varphi)\ M$ by exchanging the parameters.

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:

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

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, and RedCoerDistCoerArrow.

Optional reduction rules Our presentation of F_ι could be extended with additional reduction rules for arrow and distributivity coercions such as $((G_1 \xrightarrow{\tau} G_2)\langle M\rangle)\ N \rightsquigarrow_\iota G_2\langle M\ (G_1\langle N\rangle)\rangle$. 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. For sake of simplicity, the current presentation has fewer, but sufficiently many, reduction paths.

2.4 Examples

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

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

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

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

It can also be given the type

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

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

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

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

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

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

The next example, which is inspired from $\mathsf{F}_{<:}$ and is not typable in F_η, illustrates coercion abstraction. It defines a function first that implements the first projection for non-empty tuples of arbitrary length. Tuples are encoded as chained pairs ending with $\top$. (We assume pairs have been added to F_ι and write left and right for the projections.) The function first

$$\lambda\beta\ \lambda\alpha\ \lambda(c:\alpha\triangleright(\beta*\top))\,\lambda(x:\alpha)\ \mathsf{left}\ (c\langle x\rangle)$$

of type $\forall\beta.\,\forall\alpha.\,(\alpha\triangleright(\beta*\top))\Rightarrow\alpha\to\beta$, say τ, abstracts over a coercion c from arbitrary tuples to the singleton tuple. It can be applied to any non-zero tuple by passing the appropriate coercion.

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

3. Properties of F_ι

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

3.1 Soundness

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

Proposition 1 (Subject Reduction)**.** *If $\Gamma\vdash M:\tau$ and $M\rightsquigarrow_{\beta\iota}N$ hold, then $\Gamma\vdash N:\tau$ holds.*

Proposition 2 (Progress)**.** *If $\Gamma\vdash M:\tau$ holds, then either M is a value or M reduces.*

3.2 Termination of reduction

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

3.3 Reification of F_ι in System F

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

Reification is described on Figure 8. We write $\lceil M\rceil$ for the reification of M. Coercions of coercion type $\tau\triangleright\sigma$ are reified as functions of type $\tau\to\sigma$. Hence, a coercion abstraction $\lambda(c:\tau\triangleright\sigma)\ M$ is reified as a higher-order function $\lambda(x_c:\lceil\tau\rceil\to\lceil\sigma\rceil)\ \lceil M\rceil$. A coercion variable c is reified as a term variable x_c (we assume an injective mapping of coercion variables to reserved term variables). Thus, the type $(\tau\triangleright\sigma)\Rightarrow\rho$ of a term abstracted over a coercion is translated into the type $(\lceil\tau\rceil\to\lceil\sigma\rceil)\to\lceil\rho\rceil$ of a higher-order function. Other type expressions are reified homomorphically. The application of a coercion to a term and the application of a term to a coercion are both reified as applications.

The remaining cases are the translation of coercions G, which are all done in two steps: we first translate G into some F_ι-term performing η-expansions to transform a coercion from τ to σ into

$$
\begin{array}{rlrlrlrl}
\lceil \alpha \rceil &= \alpha & \lceil x \rceil &= x & \lceil \lambda\alpha\ M \rceil &= \lambda\alpha\ \lceil M \rceil & \lceil \emptyset \rceil &= \emptyset \\
\lceil \tau \to \sigma \rceil &= \lceil \tau \rceil \to \lceil \sigma \rceil & \lceil \lambda(x:\tau)\ M \rceil &= \lambda(x : \lceil \tau \rceil)\ \lceil M \rceil & \lceil M\,\tau \rceil &= \lceil M \rceil\ \lceil \tau \rceil & \lceil \Gamma, B \rceil &= \lceil \Gamma \rceil, \lceil B \rceil \\
\lceil \forall\alpha.\,\tau \rceil &= \forall\alpha.\,\lceil \tau \rceil & \lceil M\ N \rceil &= \lceil M \rceil\ \lceil N \rceil & \lceil G\langle M \rangle \rceil &= \lceil G \rceil\ \lceil M \rceil & \lceil \alpha \rceil &= \alpha \\
\lceil \varphi \Rightarrow \tau \rceil &= \lceil \varphi \rceil \to \lceil \tau \rceil & & & \lceil \lambda(c:\varphi)\ M \rceil &= \lambda(x_c : \lceil \varphi \rceil)\ \lceil M \rceil & \lceil (x:\tau) \rceil &= (x : \lceil \tau \rceil) \\
\lceil \top \rceil &= \forall\alpha.\,(\forall\beta.\,\beta \to \alpha) \to \alpha & & & \lceil M\{G\} \rceil &= \lceil M \rceil\ \lceil G \rceil & \lceil (c:\varphi) \rceil &= (x_c : \lceil \varphi \rceil) \\
\lceil \tau \triangleright \sigma \rceil &= \lceil \tau \rceil \to \lceil \sigma \rceil & & & & & & \\
& & & & \lceil G_1 \xrightarrow{\tau} G_2 \rceil &= \lceil \lambda(y : \mathsf{dom}(G_1 \xrightarrow{\tau} G_2))\ \lambda(x:\tau)\ G_2\langle y\ (G_1\langle x \rangle)\rangle \rceil & & \\
& & & & \lceil \mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}))\ \lambda(x:\tau)\ \lambda\alpha\ y\,\alpha\ x \rceil & & \\
& & & & \lceil \mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma} \rceil &= \lceil \lambda(y : \mathsf{dom}(\mathsf{Dist}^{\varphi\Rightarrow}_{\tau\to\sigma}))\ \lambda(x:\tau)\ \lambda(c:\varphi)\ y\{c\}\ x \rceil & & \\
\lceil [c] \rceil &= \lceil [x_c] \rceil & & & \lceil P[G] \rceil &= \lceil \lambda(x : \mathsf{dom}(G))\ P[G\langle x \rangle] \rceil & & \\
\lceil \Diamond^{\tau} \rceil &= \lceil \lambda(x:\tau)\ x \rceil & & & & & & \\
\lceil \mathsf{Top}^{\tau} \rceil &= \lceil \lambda(y:\tau)\ \lambda\alpha\ \lambda(x : \forall\beta.\,\beta \to \alpha)\ x\,\tau\ y \rceil & & & & & &
\end{array}
$$

Figure 8. Reification of F_ι into System F

a function from τ to σ. For atomic coercions (variables, identity, or distributivity), the result of this step is in the System-F subset of F_ι. However, for complex coercions, the result still contains inner coercions. Hence, in the second step, we recursively translate the result of the first step. This translates types and residual coercions. Notice that the first step may introduce applications of coercions to terms, which are then turned into applications of terms to terms.

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 (easy but lengthy details can be found in the extended version).

Proposition 3 (Well-typedness of reification)**.** *If $\Gamma \vdash M : \tau$ then $\lceil M \rceil$ is well-defined and $\lceil \Gamma \rceil \vdash \lceil M \rceil : \lceil \tau \rceil$.*

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

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

Corollary 5 (Termination)**.** *Reduction in F_ι is terminating.*

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. In fact, the relation $\rightsquigarrow_\iota$ alone is confluent. Moreover, the reduction $\rightsquigarrow_\beta$ and $\rightsquigarrow_\iota^\star$ commute.

Corollary 6 (Confluence)**.** *Reduction in F_ι is confluent.*

Lemma 7. *If $M \rightsquigarrow_\beta M_1$ and $M \rightsquigarrow_\iota M_2$ hold, then there is a term N such that $M_1 \rightsquigarrow_\iota^\star N$ and $M_2 \rightsquigarrow_\beta N$.*

3.5 Forward simulation

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

Lemma 8 (Forward simulation)**.** *If $\Gamma \vdash M : \tau$ holds, then:*

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

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

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

4. Coercions as retyping functions: F_ι^λ

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

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. The first part, along with its inverse, define translations between F_ι and F_ι^λ that preserves well-typedness and coercion erasure. Although we have not proved it, F_ι and F_ι^λ should be the same up to their representation of coercions.

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

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

Instead, we make the η-expansion of a term M an atomic construct, namely $\lambda(\phi_1 : \tau)\,G_2\,\{\phi_2 \leftarrow M\,G_1\}$. 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 type-

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

Figure 9. Parametric F_ι: syntax restriction *wrt* F_ι

checking G_2 and extend the typing environment accordingly when typechecking M and G_1.

We presented F_ι rather than its more intuitive version F^λ_ι to avoid the additional complexity in the type system; moreover, it is not obvious how to extend F^λ_ι with projectors, as discussed in §9.

5. Parametric F_ι

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

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

5.1 Syntax changes

The syntax of Parametric F_ι is defined on Figure 9 as a patch to the syntax of F_ι (we write $\not|$ for removal of a previous grammar form). We replace coercion abstraction $\lambda(c:\tau\triangleright\sigma)\, M$ of F_ι 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 ranges over $\triangleright$ 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 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_ι.

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 *x*MLF. The negative form is parametric on the domain of the coercion and implements an upper bounded quantification $\alpha\triangleright\tau$, as in $\mathsf{F}_{<:}$.

Continuing with the definition of F^p_ι, we replace coercion application $M\{G\}$ by $M\{\tau\diamond G\}$ to perform type and coercion applications $(M\,\tau)\{G\}$ atomically. Both positive and negative versions

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

Figure 11. Parametric F_ι: changes in values *wrt* F_ι

have the same meaning, but different typing rules. Type τ appears before G to remind that the type application is performed before the coercion application in the expanded form. As a mnemonic device, the $\diamond$ is oriented towards the side of the variable it instantiates in the coercion type of M. Hence, if M is $\lambda(\alpha\triangleright c:\sigma)\, N$, we must write $M\{\tau\triangleright G\}$.

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

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

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 10. The corresponding typing rules COERCOERLAM and COERCOERAPP for coercions are changed similarly. We also replace COERVAR by TCOERVAR. Finally, the modified distributivity coercion is typed as described by Rule COERDISTTCOERARROW. Notice that $\diamond$ is a meta-variable as M or τ and different occurrences of the same meta-variable can only be instantiated simultaneously all by $\triangleright$ or all by $\triangleleft$. (We use different meta-variables $\diamond_1$ and $\diamond_2$ when we mean to instantiate them independently.)

The new typing rules for F^p_ι are derived from the typing rules of the corresponding nodes in F_ι. For example, TERMTCOERLAM is the combination of rules TERMCOERLAM and TERMTYPELAM in F_ι.

Well-formedness judgments are adjusted in the obvious way (details can be found in the extended version).

Operational semantics The operational semantics is modified in the obvious way. The syntax of values for F^p_ι is defined on Figure 11 as a modification of the syntax of F_ι. The adjustments in the reduction rules are the replacement of REDCOER by REDTCOER, REDCOERDISTCOERARROW by REDCOERDISTTCOERARROW, and the change of retyping contexts that induces a change in REDCOERFILL as described in Figure 11.

5.3 Properties

Since F^p_ι can be seen as a restriction of F_ι where coercion abstraction is always preceded by a type abstraction, some properties of F^p_ι can be derived from those of F_ι. In particular, normalization and

$$\textsc{TermTCoerLam}\quad \frac{\Gamma, \alpha \mathrel{\diamond} c : \tau \vdash M : \sigma}{\Gamma \vdash \lambda(\alpha \mathrel{\diamond} c : \tau)\, M : \forall(\alpha \mathrel{\diamond} \tau) \Rightarrow \sigma}$$

$$\textsc{TermTCoerApp}\quad \frac{\Gamma \vdash M : \forall(\alpha \mathrel{\diamond} \tau) \Rightarrow \tau' \qquad \Gamma \vdash G : \sigma \mathrel{\diamond} \tau[\alpha \leftarrow \sigma]}{\Gamma \vdash M\{\sigma \mathrel{\diamond} G\} : \tau'[\alpha \leftarrow \sigma]}$$

$$\textsc{TCoerVar}\quad \frac{\Gamma \vdash ok \qquad \alpha \mathrel{\diamond} c : \tau \in \Gamma}{\Gamma \vdash c : \alpha \mathrel{\diamond} \tau}$$

$$\textsc{CoerTCoerLam}\quad \frac{\Gamma, \alpha \mathrel{\diamond} c : \tau \vdash G : \sigma \triangleright \sigma' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \lambda(\alpha \mathrel{\diamond} c : \tau)\, G : \sigma \triangleright \forall(\alpha \mathrel{\diamond} \tau) \Rightarrow \sigma'}$$

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

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

Figure 10. Parametric F_ι: typing rules *wrt* F_ι

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

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

Figure 12. Parametric F_ι: new reduction rules *wrt* F_ι

subject reduction properties are preserved, just by observing that F_ι^p is syntactically closed by reduction.

Proposition 9 (Preservation)**.** *If $\Gamma \vdash M : \tau$ and $M \rightsquigarrow_{\beta\iota} N$ hold, then $\Gamma \vdash N : \tau$ holds.*

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.

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

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

As expected, coercions are erasable in F_ι^p. 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.

Proposition 11 (Backward simulation)**.** *If $\Gamma \vdash M : \tau$ and $\lfloor M \rfloor \rightsquigarrow \mathcal{M}$, then $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ such that $\lfloor N \rfloor = \mathcal{M}$.*

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

6. Expressiveness of Parametric F_ι

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

By construction, F_η is included (and simulated) in Parametric F_ι. In the rest of this section, we show that $x\mathsf{MLF}$ and $\mathsf{F}_{<:}$ are also subsumed by F_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 is equal to its type erasure, 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 $\lfloor T \rfloor = \lfloor M \rfloor$. As a consequence, reduction in the source language terminates, since it is simulated in F_ι^p.

Bounded polymorphism. $\mathsf{F}_{<:}$ is a well-known extension of System F with subtyping. There are several variations on $\mathsf{F}_{<:}$, all sharing the same features, but with different expressiveness due to the way they deal with subtyping of bounded quantification. Bounded quantification $\forall(\alpha <: A)\ B$ restricts types A' that α ranges over to be subtypes of the bound A. The differences lie in when the subtyping judgment $\Sigma \vdash \forall(\alpha <: A)\ B <: \forall(\alpha <: A')\ B'$ holds. Different versions of the corresponding subtyping rule are given on Figure 13. In Kernel $\mathsf{F}_{<:}$, the bounds A and A' must be equal, whereas Full $\mathsf{F}_{<:}$ only requires the bound A' to be a subtype of the bound A. Moreover, α cannot appear free in the bounds A or A' in Kernel or Full $\mathsf{F}_{<:}$, while $\mathsf{F}_{\mu<:}$ allows this form of recursion, called F-bounded polymorphism. The most general assumption, $\Gamma, \alpha <: A' \vdash \alpha <: A$, is that of $\mathsf{F}_{\mu<:}$. Perhaps surprisingly, this is a slightly more general rule [Baldan et al. 1999] than the more intuitive one $\Gamma, \alpha <: A' \vdash A' <: A$. In summary, we have Kernel $\mathsf{F}_{<:} \subset$ Full $\mathsf{F}_{<:} \subset \mathsf{F}_{\mu<:}$ where all inclusions are strict.

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

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

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

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

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

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

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

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

Figure 13. Bounded polymorphism: variants on the subtyping rule

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

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

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

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

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

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

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

Moreover, reduction in xMLF is simulated in F^x_ι. This implies termination of reduction in xMLF (a result already proved by Manzonetto and Tranquilli [2010]).

Summary Features of F^p_ι and its variants are summed up on Figure 14. The expressiveness of F_η, xMLF, and $\mathsf{F}_{<:}$ can be compared by checking which feature is present in one language and not in the others. Deep instantiation corresponds to the $\lambda\alpha\ G$ and $G\,\tau$ constructs, allowed in F_η and xMLF, but not in $\mathsf{F}_{<:}$. Upper bounds are used in $\mathsf{F}_{<:}$ and lower bounds are used xMLF. They correspond to coercion abstraction $\lambda(\alpha \diamond c : \tau)\ G$ and $G\{\tau \diamond G\}$ when $\diamond$ is $\triangleright$ or $\triangleleft$, respectively. F_η allows neither. Arrow congruence is the $G \xrightarrow{\tau} G$ construct, allowed in F_η and $\mathsf{F}_{<:}$. Distributivity $\mathsf{Dist}^{\forall\alpha.}_{\tau\to\sigma}$ is used in F_η. The other form $\mathsf{Dist}^{\forall\alpha\diamond\rho\Rightarrow}_{\tau\to\sigma}$ is only used in F^p_ι since it involves coercion abstraction.

Extension of System F	F_η	$\mathsf{F}_{<:}$	xMLF	F^p_ι
Deep instantiation	✓	-	✓	✓
Arrow congruence	✓	✓	-	✓
Permutation of $\forall$ and $\to$	✓	-	-	✓
Upper bounds	-	✓	-	✓
Lower bounds	-	-	✓	✓

Figure 14. Language and feature comparison

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

All examples of §2.4 are actually typable in F^p_ι—with some syntactic adjustment of course. For instance, the last example becomes $\lambda(\gamma \triangleleft c : \tau)\ \mathsf{choose}\ \{\gamma \triangleleft (c\{\mathsf{first}\})\}$ of type $\forall(\gamma \triangleleft \tau) \Rightarrow \gamma \to \gamma$ where τ is $\forall\beta.\forall(\alpha \triangleright \beta * \top) \Rightarrow \alpha \to \beta$. For instance, it can be coerced to the type $\forall(\gamma \triangleleft \sigma) \Rightarrow \gamma \to \gamma$ where σ is the type $\forall\beta.\forall\beta'.\forall(\alpha \triangleright \beta * (\beta' * \top)) \Rightarrow \alpha \to \beta$. This uses the coercion $\lambda(\gamma \triangleleft c : \sigma)\ \Diamond\{\gamma \triangleleft c\langle G\rangle\}$ where G is the coercion from τ to σ equal to $\lambda\beta\ \lambda\beta'\ \lambda(\alpha \triangleright c : \beta * (\beta' * \top))\ (\Diamond\ \beta)\{\alpha \triangleright (\Diamond^\beta * \mathsf{Top}^{\beta' * \top})\langle c\rangle\}$.

7. Weak F_ι

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

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

The syntax of Weak F_ι, written F^w_ι, is defined on Figure 15 as a restriction of the syntax of F_ι. 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 F^w_ι and F_ι are the same. The reduction relation of F^w_ι is a subrelation of the reduction relation of F_ι that prevents evaluation under term and coercion abstractions and pre-

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

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

Figure 15. Weak F_ι: syntax and semantics *wrt* F_ι

serves the value restriction. Reduction contexts are modified accordingly: $\lambda(x:\tau)\ []$ and $\lambda(c:\varphi)\ []$ are removed. Rule RedCoerCoerLam (the coercion abstraction part of RedCoerFill) is restricted to make it call-by-value. Indeed, keeping the F_ι rule:

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

would place the arbitrary term M under a coercion abstraction.

It is routine to check that F_ι^w is well-behaved and that coercions are erasable. We refer the reader to the extended version.

8. Related work

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

We have already widely discussed F_η, $\mathsf{F}_{<:}$, and *x*MLF. Parts of F_ι^x is closely related to the work of Manzonetto and Tranquilli [2010] who proposed the first encoding of *x*MLF in a calculus of coercions, but for the main purpose of proving the termination of *x*MLF. They exhibit a type and semantics preserving encoding of *x*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*MLF into F_ι^x by lack of space, but also because it resembles theirs. In fact, their translation of *x*MLF into F_ι^x has itself been inspired by the translation of MLF into System F by Leijen and Löh [2005] and Leijen [2007]. However, Manzonetto and Tranquilli restrict their study to the termination of *x*MLF without any interest in F_η or $\mathsf{F}_{<:}$, 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*MLF.

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

Record subtyping in $\mathsf{F}_{<:}$ may also be compiled away into records without subtyping in plain System F by inserting coercions with computational content [Breazu-Tannen et al. 1991] that change the representation of records whenever subtyping is used. Since these coercions are not erasable and can be inserted in different ways, the soundness of the approach depends on a coherence result to show that the semantics of the translation does not actually depend on the places where coercions are inserted.

Another method for eliminating subtyping has been used by Crary [2000]: bounded polymorphism $\forall(\alpha \le \tau).\,\sigma$ is compiled away into an intersection type $\forall\alpha.\,\sigma[\alpha \leftarrow \alpha\cap\tau]$ while intersection types are themselves encoded with explicit erasable coercions. This directly relates to our work by their canonization, which is similar to our ι-reduction, and their use of bisimulation up to canonization to show erasability of coercions. Of course, the languages are different, as we do not consider intersection types while they do have neither coercion abstraction nor distributivity and only consider call-by-value reduction. Their work could serve as a reference to extend F_ι with recursive types.

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

Coercions introduced in FC_2 [Weirich et al. 2011], the internal language of Haskell, are interesting because they use coercion projections and cannot be expressed in F_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 be abstracted over as in F_ι and also stored in data-structures. However, as a result of being first-class, coercions may change the termination (hence the semantics) of programs and are not erasable in our terminology. The two languages F_ι and FC_2 follow orthogonal approaches and are thus not easily comparable; combining the features of both would be an interesting challenge.

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

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

9. Discussion and future work

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

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

Instead, F_ι^p restricts the types of coercion parameters and forces them to be polymorphic in either their domain or codomain. The advantage of F_ι^p is to retain a strong reduction relation, which shows that the calculus is really well-behaved. Although restrictive, it already subsumes F_η, *x*MLF, and $\mathsf{F}_{<:}$. We believe it is an interesting point in the design space. It also shows that an extension of

xMLF with subtyping would be possible and beneficial, even if the question of designing the surface language to make type inference possible remains open.

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

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

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

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

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

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

Language extensions Several features of programming languages have also been left out of F_ι. Although products are not included in this short presentation, we have already verified that they can easily be added. Labeled products should work as well.

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

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

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

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

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 *FPCA*, 1989. 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 *ICFP*, 2000. URL `http://doi.acm.org/10.1145/351240.351247`.

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

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

D. Leijen. A type directed translation of MLF to System F. In *ICFP*, Oct. 2007. URL `http://research.microsoft.com/users/daan/download/papers/mlftof.pdf`.

D. Leijen and A. Löh. Qualified types for MLF. In *ICFP*, Sept. 2005. URL `http://murl.microsoft.com/users/daan/download/papers/qmlf.pdf`.

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

G. Manzonetto and P. Tranquilli. Harnessing MLF with the Power of System F. In *MFCS*, volume 6281, 2010. 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), 1988.

D. Rémy and B. Yakobowski. A Church-Style Intermediate Language for MLF. In *FLOPS*, volume 6009, pages 24–39. 2010. URL `http://dx.doi.org/10.1007/978-3-642-12251-4_4`.

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

S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. In *POPL*, 2011. URL `http://doi.acm.org/10.1145/1926385.1926411`.