Extending System F with Abstraction over Erasable Coercions

Julien Cretin, Didier Rémy

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

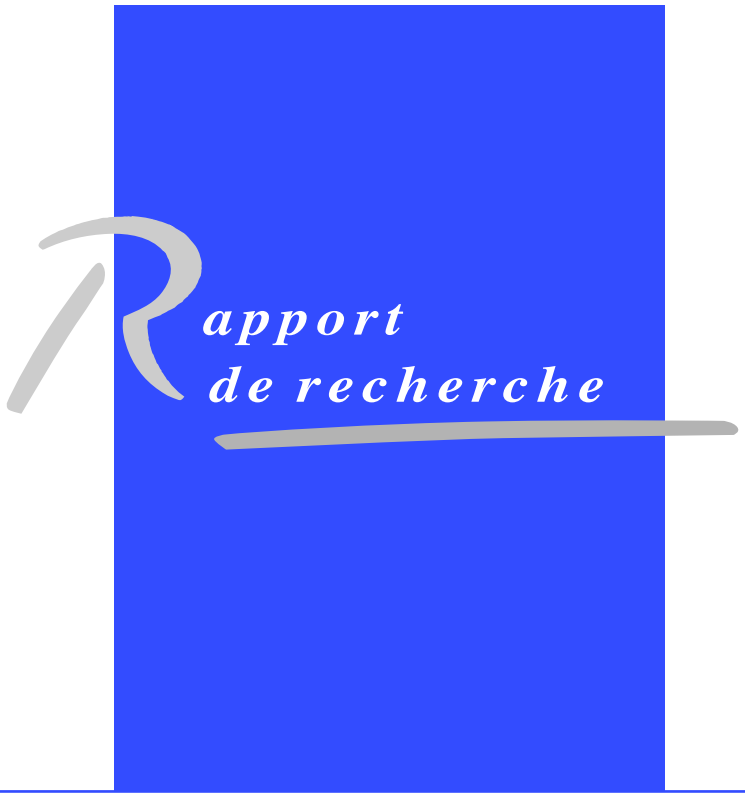
Extending System F with Abstraction over Erasable Coercions

Julien Cretin — Didier Rémy

N° 7587

April 2011

Domaine 2



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

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
PARIS - ROCQUENCOURT

Extending System F with Abstraction over Erasable Coercions

Julien Cretin , Didier Rémy

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

Rapport de recherche n° 7587 — April 2011 — 42 pages

Abstract: Erasable coercions in System F_η, also known as retyping functions, are well-typed η-expansions of the identity. They may change the type of terms without changing their behavior and can thus be erased before reduction. However, F_η does not allow abstraction over retyping functions, which limits its expressiveness—and the use of retyping functions. In a naive generalization of System F with abstraction over coercions, abstract coercions may block the reduction and thus not be erasable. Erasability can be recovered by choosing a weak reduction strategy or by enforcing abstract coercions to be parametric in either their domain type or their return type. Although limited, the latter solution already subsumes both $x\mathsf{MLF}$ and $\mathsf{F}_{<:}$. We discuss a more general approach introducing coercion destructors to move abstract coercions out of redexes.

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

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

Extension du Système F avec des coercions effaçables abstraites

Résumé : Les coercions effaçables dans le Système F_η, aussi connues sous le nom de fonctions de retypage, sont des η-expansions de l'identité. Elles peuvent changer le type des termes sans en changer leur comportement et peuvent donc être effacées avant la réduction. Cependant, F_η ne permet pas d'abstraire sur des fonctions de retypage, ce qui limite son expressivé—et l'usage des fonctions de retypage. Une généralisation naïve du Système F avec la possibilité d'abstraire sur les coercions introduit des variables de coercion qui peuvent bloquer la réduction et donc ne pas être effaçables. Le caractère effaçable peut être retrouvé en choisissant une stratégie de réduction faible ou en forçant les coercions abstraites à être paramétriques dans leur domaine ou dans leur type de retour. Bien que limitée, cette dernière solution est déjà une généralisation de $x\mathsf{MLF}$ et de $\mathsf{F}_{<:}$. Nous décrivons une approche plus générale en introduisant des destructeurs de coercions qui permettent de déplacer les coercions abtraites à l'extérieur des redex.

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

Contents

1 Introduction **4**

2 System F and F_η **8**
2.1 System F 8
2.2 System F_η 10

3 Abstraction over coercions: F_ι **14**
3.1 Definition of F_ι 14
3.2 Soundness 16
3.3 Termination of reduction 20
3.4 Confluence 21
3.5 Forward simulation 21

4 Weak F_ι **21**

5 Parametric F_ι **25**
5.1 Syntax changes 25
5.2 Adjustments to the semantics 26
5.3 Properties 28

6 Expressiveness of Parametric F_ι **31**

7 Towards a more general setting? **33**

8 Related work **35**

9 Conclusions **35**

A Delayed Proofs **37**

1 Introduction

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

As a result, type systems are becoming quite sophisticated but also extremely complex: they commonly include first-class parametric polymorphism and type abstraction; some form of subtyping; they may also include type equalities, GADTs, contracts, capabilities, ownership, etc. Moreover, while these features can be studied independently, they often need to be combined together to be useful in practice: for instance, complex invariants of a data-structure provided as a library must be presented to clients with an abstract interface, which requires type abstraction; type abstraction may benefit from subtyping so that hiding can be done a posteriori with only one type constraint; first-class existential types also calls for first-class universal types so that abstract types do not leak outside of their scope; *etc.* Not to mention the use of capabilities and ownership for a better typechecking of memory management, which requires so many features to be used simultaneously [Charguéraud and Pottier, 2008].

The rise of coercions

When combined together, these constructs induce non trivial forms of type conversions. A type conversion between τ_1 and τ_2 allows treating a value of type τ_1 as a value of type τ_2. Several notions of type conversions may coexist together. For instance, they may originate from subtyping, an underlying notion of type-equivalence, or implicit type-instantiation, *etc.* Type conversions may be inferred during type checking, uniquely determined from the context, or written explicitly inside source expressions. Different strategies may be used for the different notions of type-conversions. However type-conversions are introduced, their validity must still be checked, which may be hard if not undecidable. In such cases, source expressions must also contain witnesses for type conversion proofs, typically using a small language of *coercions* to represent full or partial derivations of type-conversion judgments. Such an approach is not new [Crary, 2000], but it is used more and more often and in various contexts, both theoretical [Pottier, 2011] and practical [Weirich et al., 2011].

The essence of coercions is perhaps to be found in System F_η [Mitchell, 1988], which can be seen as the closure of System F by η-reduction or, equivalently, as an extension of System F with *type containment*, a relation that allows deep type specialization of terms, strengthening the domain of functions or weakening their codomain, *a posteriori*. Although in Mitchell's view, type containment is implicit, it can be made explicit either using *retyping functions*, which are terms of System F that are η-convertible to the identity, or reflecting type-containment derivations inside terms. For instance, we may build a coercion $G_1 \to G_2$ of coercion type $(\tau_1' \to \tau_2) \triangleright (\tau_1 \to \tau_2')$ in F_η from coercions G_1 and G_2 of respective coercion types $\tau_1 \triangleright \tau_1'$ and $\tau_2 \triangleright \tau_2'$.

The reduction semantics manipulates coercions explicitly during reduction. For example, $(G_1 \to G_2)@(\lambda(x : \tau_1')\ M)$ reduces to $\lambda(x : \tau_1)\ G_2@(M[x \leftarrow G_1@x])$ where $G@M$ is the application of a coercion G to a term M.

However, coercions do not really contribute to the reduction: they just surround it. This can be stated formally by defining a projection of source terms into the λ-calculus, called their *erasure*, and splitting reduction in the source language between the essential β-reduction steps and administrative ι-steps so that their is a bisimulation between reduction of source terms and reduction of their erasure, up to ι-steps.

A language with explicit coercions generalizes type instantiation and generalization of church-style System F: indeed, type specialization of a value of type $\forall\alpha.\tau$ into one of type $\forall\beta.\tau[\alpha \leftarrow \sigma]$,

where β is not free in $\forall\alpha.\tau$ can be seen as a coercion function $\lambda(x : \forall\alpha.\tau)\ \lambda\beta\, x\, \sigma$. More generally, F_η allows the coercion of a function of type τ_1 into one of type τ_2 provided τ_1 can be *converted* to τ_2, which is written $\tau_1 \triangleright \tau_2$. However, conversion proofs are left implicit in F_η, while coercions are fully explicit and provide a constructive proof of $\tau_1 \triangleright \tau_2$.

Explicit coercions may be used to simplify the meta-theoretical study of the language, since *reasoning* about type-conversion derivations, which are mathematical objects, can be replaced by *computing* over coercions, which are concrete objects. For example, xMLF [Rémy and Yakobowski, 2010], the internal language of MLF [Le Botlan and Rémy, 2009] has been introduced to maintain terms well-typed during reduction: type soundness became a routine proof in xMLF while it was extremely involved in the surface language even after type inference made all types explicit—but not their transformations. In FC_2 [Weirich et al., 2011], the internal language of Haskell, coercions are also used to maintain types during the first compilation passes, since they are later needed for optimizations and elimination of type classes.

The rise of coercions in programming languages raises the obvious question: do all these uses of coercions have enough in common to be described as several instances of a general framework, so that the main concepts and properties can be introduced once for all?

About non-erasable coercions

All coercions we have considered so far are erasable. Coercions may also refer to terms with some computational content. For example, copying a record into one with fewer fields (or just projecting a record to one of its field) may be treated as a coercion from one type the other. More generally, one could even allow user-definable coercions whose code could be an arbitrary function between given types, and with at most one such coercion for each pair of types; then, the coercion from type τ_1 to type τ_2 could be used as a canonical way to transform a value of type τ_1 into one of type τ_2. Coercions with computational content can also be restricted to a subset of expressions so that their behavior can be analyzed. In some contexts, one may wish to allow several coercions with the same type, provided they are operationally equivalent. See, for instance, [Breazu-Tannen et al., 1991, Swamy et al., 2009]. Haskell type classes can perhaps be also seen—at a low level—as the insertion of implicit non-erasable coercions.

Whether coercions have computational content may actually depend on the underlying runtime. For instance, record subtyping may be an erasable operation if records carry headers from which the domain of the record and placement of fields can be extracted, but it must have to change the representation of records if these are just implemented as an heterogeneous array (*e.g.* as the restriction of a module by a signature with fewer fields does in ML).

Hence, coercions with computational content are also quite useful. However, we restrict our attention to erasable coercions and leave non-erasable coercions for future studies. Hopefully, a good understanding of erasable coercions will also benefit to coercions with computational content, even if these also raise their own additional problems.

The anatomy of erasable coercions

Let us look more closely at several existing coercion systems and see what features they implement. First, System F has a superficial type instantiation mechanism, which allows to see a term of polymorphic type $\forall\alpha.\tau$ as if it were typed $\tau[\alpha \leftarrow \sigma]$. Reciprocally, we can see a term typed τ under Γ, α as if it were typed $\forall\alpha.\tau$ under Γ. In System F, only toplevel polymorphism can be instantiated. System F_η extends System F by allowing deeper instantiations. For instance, a term of type $\tau_1 \to (\forall\alpha.\tau_2)$ can be coerced into one of type $\tau_1 \to (\forall\beta.\tau_2[\alpha \leftarrow \sigma])$ provided β does not appear free in $\forall\alpha.\tau_2$. System F_η also introduces structural propagation of coercions, covariantly under universal quantifier and on the right of arrow types, but contravariantly on the left of arrow types. For instance, we can build a coercion typed $\forall\alpha.(\tau' \to \sigma) \triangleright \forall\alpha.(\tau \to \sigma')$, using smaller coercions typed $\tau \triangleright \tau'$ and $\sigma \triangleright \sigma'$.

By contrast, xMLF does not allow the automatic propagation of coercions according to the type structure, as found in F_η; however, it allows abstraction over coercion functions. For example, one

can write a term $\lambda(\alpha, c : \triangleleft\tau)\,\lambda(x : \alpha)\ \mathsf{choose}\ x\ (c\ y)$ that abstracts over some type α and some coercion c from τ to α and later specializes it by passing a type τ' together with a coercion G from τ to τ'. In $x\mathsf{MLF}$, this transforms a term of type $\forall(\alpha \geq \tau).\alpha \to \alpha$ into one of type $\tau' \to \tau'$ provided τ can be coerced to τ'. As F_η, $x\mathsf{MLF}$ permits deep instantiation of quantifiers.

The language $\mathsf{F}_{<:}$ [Cardelli et al., 1994, Cardelli, 1993] can also be seen as a language of coercions. Interestingly, $\mathsf{F}_{<:}$ has yet another combination of features that in a way makes it a complement of F_η and $x\mathsf{MLF}$. As F_η, it propagates coercions structurally; as $x\mathsf{MLF}$, it allows abstraction over coercions (but of opposite direction). However, contrary to both of them, it disallows deep instantiation. This is a significant restriction, as it forces types in relation to have exactly the same structure of quantifiers except maybe at the leaves.

In face of this triangular combination of features, we may wonder whether all features can be joined together. In other words, is there a language with coercions that simultaneously allows deep instantiation, contravariance, and a general form of abstraction over coercions, and of which F_η, $x\mathsf{MLF}$, or $\mathsf{F}_{<:}$ are all particular cases? The problematic of combining F_η and $x\mathsf{MLF}$ has already been pointed out and left as future work in [Rémy and Yakobowski, 2010].

Summary of our contributions

We explore the design space for an extension of System F with erasable coercions. Erasability is a strong requirement on coercions, which implies that the reduction of coercions should be such that they surround the computation without actually driving the reduction: coercions around expressions *may* be transformed during the reduction into coercions about subexpressions; coercions *must* also be transformed in such a way that they never get a redex stuck. We call *wedging* a configuration where a coercion G appears in the middle of a redex as in $(G \mathbin{@} (\lambda(x : \tau)\ M))\ N$. When wedging appears in an evaluation context, its erasure $(\lambda x.M)\ N$ is reducible, and therefore, the wedging configuration must also be reducible. Ensuring the absence of stuck wedging configurations is a key to erasability and the main design issue.

The language F_η We first present F_η as a language with coercions. This serves as a good introduction to coercions. Although a fully explicitly presentation of F_η is folklore knowledge, we think that putting it on paper has an intrinsic interest. F_η does not allow abstraction over coercion functions, so wedging coercions are always concrete and are easily reducible. For instance, when G is $G_1 \to G_2$, it can be deconstructed into G_1 and G_2 that can be placed around the redex. Bisimulation trivially hold for F_η.

The language F_ι We then present a simple and natural extension of F_η with abstraction over coercion functions. The definition of the language is almost straightforward. Type soundness can be easily established by subject reduction and progress. We also show termination by simulating F_ι in System F, reifying coercions as retyping functions. Although this is primarily a technique for proving termination, it is interesting as it gives a good understanding of what coercions are and how they must be reduced.

Unfortunately, due to the introduction of coercion variables in F_ι, irreducible wedging configurations may appear in evaluation contexts. Hence, coercions are not erasable in F_ι. This is a serious problem—that must be solved. The definition of wedging gives us three directions:

- Change the semantics so that stuck wedging configurations cannot appear in an evaluation contexts. This leads to a version of F_ι with a weak evaluation strategy.
- Restrict source terms, so that stuck wedging configurations cannot appear anywhere. This leads to another restriction of F_ι, called parametric F_ι, where abstraction over coercions is only allowed when coercions are parametric in either their domain or their return type.
- Augment the language with new terms and new reduction rules, so that previously stuck wedging configurations can now be reduced. This suggests a more general setting with

$$
\begin{array}{rcll}
x, y & & & \text{variables} \\
a, b & ::= & x \mid \lambda x.a \mid a\,a \mid (a, a) \mid a.1 \mid a.2 & \text{terms} \\
e & ::= & \lambda x.[] \mid [] \, a \mid a \, [] \mid ([], a) \mid (a, []) \mid [].1 \mid [].2 & \text{evaluation contexts}
\end{array}
$$

$$
\frac{a \rightsquigarrow b}{e[a] \rightsquigarrow e[b]}\ \textsc{RedContext} \qquad
(\lambda x.a)\, b \rightsquigarrow a[x \leftarrow b]\ \textsc{RedBeta} \qquad
(a, b).1 \rightsquigarrow a\ \textsc{RedFirst} \qquad
(a, b).2 \rightsquigarrow b\ \textsc{RedSecond}
$$

Figure 1: λ-calculus: syntax and semantics

coercion destructors that can be applied to coercion variables to break them apart and progress.

Weak F_ι In F_ι, wedging configurations are stuck when the wedge G contains a coercion variable c. To avoid such configurations, it suffices to disallow coercion variables to appear under an evaluation context, which may be achieved using a weak evaluation strategy. At first, it seems that only abstraction over coercions need to be weakly evaluated. However, things are slightly more subtle as this restriction will force another one: abstraction over both values and coercions must be weakly evaluated while reduction under type abstraction is still permitted.

There is of course a counterpart to choosing a weak reduction strategy: to preserve erasability, abstraction over coercions must be restricted to terms whose erasure are values.

Thus, Weak F_ι is a strict restriction of F_ι: it can type strictly fewer programs; reduction paths are also a subset of reduction paths in F_ι. We verify that properties of F_ι are preserved in weak F_ι and that, as expected, erasability is recovered.

Unfortunately, Weak F_ι restricts the use of abstraction over coercions. Moreover, excluding reduction under abstract coercions is also a limitation in the first place, while it may be useful in practice. For instance, this is at the heart of FC_2 [Weirich et al., 2011]. Indeed, reduction under abstract coercions is better modeled by strong reduction strategies. We also believe that strong reduction strategies provide better guidelines for understanding the essence of constructs, even if in practice, a weak reduction strategy will be chosen.

Parametric F_ι Parametric F_ι, say F_ι^p, is a restriction of F_ι that avoids wedging configurations with coercion variables by typing, requiring that coercion variables are parametric in either their domain or codomain. This allows a coercion variable c to appear in either $((c @ x)\ M)$ or $c @ (\lambda(x : \tau)\ M)$ but not in $(c @ (\lambda(x : \tau)\ M))\ N$ any more. We verify that properties of F_ι are preserved in F_ι^p and that, as expected, erasability is recovered.

Parametric F_ι is an interesting point in the design space as it supersedes in a unified framework three existing languages, namely F_η, $x\mathsf{MLF}$, and $\mathsf{F}_{<:}$, while remaining relatively simple. Interestingly, it naturally encodes F-bounded polymorphism [Canning et al., 1989] which supersedes $\mathsf{F}_{<:}$. By symmetry, this suggests an extension of $x\mathsf{MLF}$ where variables may appear in their bounds.

Still, Parametric F_ι is limited by design. It shares with $\mathsf{F}_{<:}$ the inability for type variables to have several upper bounds or more generally to encode arbitrary subtyping constraints, *e.g.* as in ML extended with subtyping. In this sense, $\mathsf{F}_{<:}$ and F_ι^p are a little ad hoc, or at least just particular points in the design space.

Coercions in a more general setting? Hence, we should also study extensions of F_ι that supersedes both Weak F_ι and Parametric F_ι, allowing reduction under abstract coercions and more general forms of coercions.

To prevent wedging, we introduce coercion destructors to break coercions into smaller parts and move them around redexes—so that the underlying redexes can eventually be exposed and fire. However, this raises several new problems that we discuss in §7.

$$
\begin{array}{rcll}
x, y & & & \text{term variables} \\
\alpha, \beta & & & \text{type variables} \\
W, M, N, G & ::= & x \mid \lambda(x:\tau)\ M \mid M\ M \mid (M, M) \mid M.\mathtt{1} \mid M.\mathtt{2} & \text{expressions} \\
& \mid & \lambda\alpha\, W \mid W\,\tau & \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid (\tau * \tau) \mid \forall\alpha.\tau & \text{types} \\
\Gamma & ::= & \emptyset \mid \Gamma, B & \text{environments} \\
B & ::= & \alpha \mid (x:\tau) & \text{bindings}
\end{array}
$$

Figure 2: System F: syntax

2 System F and F_η

The coercion languages we consider are all based on System F. Coercions are used to enrich the typing relation with new forms of type conversions. Erasing coercions from source terms should not change the dynamic semantics of the language, but it breaks typability. Therefore, we use the untyped λ-calculus as the target language. We inlcude pairs and projections both to have non trivial errors (otherwise, even untyped terms cannot be stuck) and to have more interesting forms of subtyping. Its definition is recalled on Figure 1. The semantics of untyped λ-terms is given by a small-step strong reduction relation. Term variables are written x or y. Untyped terms, written a or b, include variables, abstractions $\lambda x.a$, applications $a\,b$, pairs (a, b), and projections $a.\mathtt{1}$ and $a.\mathtt{2}$. Evaluation contexts of the λ-calculus are all one-hole contexts, written e. As usual, we write $e[a]$ for the term obtained by filling the hole of e with a and $a[x \leftarrow b]$ for the capture avoiding substitution of b for x in a. Expressions are considered equal up to the renaming of bound variables, which are defined in the usual way. This applies to the λ-calculus, as well as to all typed languages presented below.

In the remaining of this section, we recall the church-style presentations of both systems F and F_η.

2.1 System F

The syntax of System F is described in Figure 2. Term variables are written as in the λ-calculus. Expressions, written W, M, N, or G, contain variables, abstractions $\lambda(x:\tau)\ M$, applications $M\ N$, pairs (M, N), projections $M.\mathtt{1}$ and $M.\mathtt{2}$, type abstractions $\lambda\alpha\, W$, and type applications $W\,\tau$. The reason to use letter W rather than M in type abstractions and applications will appear in future extensions of System F where an expression W may be either a term M or a coercion G. In System F, there is no difference between M and W as there is no coercions yet and all expressions are terms.

Type variables are written α or β. Types, written τ, σ, or ρ, contain variables, function types $\tau \to \sigma$, product types $(\tau * \sigma)$, and universal types $\forall\alpha.\tau$.

Typing rules Typing environments, written Γ, are lists of bindings where bindings, written B, are either type variables or expression variables along with their type (Figure 2). We write $\Gamma \vdash M : \tau$ if M has type τ under Γ. The definition of this judgment is standard and recalled on Figure 3. It uses auxiliary well-formedness judgments for types and typing contexts: we write $\Gamma \vdash ok$ to mean that typing environment Γ is well-formed and $\Gamma \vdash \tau$ to mean that type τ is well-formed in Γ. Both judgments are recursively defined. 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 on Figure 4). This restriction allows us to see Γ as a partial function from term, coercion, or type variables to their types if they have ones.

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

$$\textsc{TermPair}\ \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M, N) : (\tau * \sigma)} \qquad \textsc{TermFirst}\ \frac{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.\mathtt{1} : \tau} \qquad \textsc{TermSecond}\ \frac{\Gamma \vdash M : (\tau * \sigma)}{\Gamma \vdash M.\mathtt{2} : \sigma} \qquad \textsc{TermTypeLam}\ \frac{\Gamma, \alpha \vdash M : \tau}{\Gamma \vdash \lambda\alpha\, M : \forall\alpha.\tau}$$

$$\textsc{TermTypeApp}\ \frac{\Gamma \vdash M : \forall\alpha.\tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash M\,\sigma : \tau[\alpha \leftarrow \sigma]}$$

Figure 3: System F: typing rules

$$\textsc{TypeVar}\ \frac{\Gamma \vdash ok \qquad \alpha \in \mathrm{dom}(\Gamma)}{\Gamma \vdash \alpha} \qquad \textsc{TypeArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau \to \sigma} \qquad \textsc{TypeProduct}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash (\tau * \sigma)} \qquad \textsc{TypeForall}\ \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall\alpha.\tau}$$

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

Figure 4: System F: well-formedness rules

$$\begin{array}{rcll} p & ::= & x \mid p\ v \mid p\,\tau \mid p.\mathtt{1} \mid p.\mathtt{2} & \text{prevalues} \\ v & ::= & p \mid \lambda(x:\tau)\ v \mid \lambda\alpha\, v \mid (v, v) & \text{values} \\ E & ::= & \lambda(x:\tau)\ [] \mid [] \ M \mid M \ [] \mid ([], M) \mid (M, []) \mid [].\mathtt{1} \mid [].\mathtt{2} \mid P & \text{evaluation contexts} \\ P & ::= & \lambda\alpha\, [] \mid [] \, \tau & \text{retyping contexts} \end{array}$$

Figure 5: System F: values and evaluation contexts

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

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

Figure 6: System F: reduction rules

Dynamic semantics The dynamic semantics of System F is given by a standard small-step strong reduction relation. The syntax of values and evaluation contexts are recalled on Figure 5. Values are either abstractions of values, pairs of values, or prevalues where prevalues are variables, an application of a prevalue to either a value or a typeor projections of prevalues. Evaluation contexts E are all one-hole contexts. However, for convenience, we have distinguished a subset of contexts P, called *retyping contexts*: a term M placed in a retyping 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 6. 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 $\rightsquigarrow_\iota$.

$$\lfloor x \rfloor = x \qquad \lfloor \lambda(x:\tau)\ M \rfloor = \lambda x.\lfloor M \rfloor \qquad \lfloor M\ N \rfloor = \lfloor M \rfloor\ \lfloor N \rfloor \qquad \lfloor (M,N) \rfloor = (\lfloor M \rfloor, \lfloor N \rfloor)$$
$$\lfloor M.1 \rfloor = \lfloor M \rfloor.1 \qquad \lfloor M.2 \rfloor = \lfloor M \rfloor.2 \qquad \lfloor P[M] \rfloor = \lfloor M \rfloor$$

Figure 7: Coercion erasure: the coercion-erasure function

Coercion erasure The erasure of terms, written $\lfloor \cdot \rfloor$, is defined on Figure 7, as expected: type annotations on function parameters and coercions are erased, other constructs are projected on their equivalent constructs in the untyped λ-calculus.

Curry-style System F is the image of church-style System F 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. We write $\Gamma \vdash a : \tau$ to mean that there exists M such that $\Gamma \vdash M : \tau$ and $\lfloor M \rfloor$ is a.

Bisimulation Coercions are *erasable* if the semantics of terms coincides with the semantics of their erasure, so that coercions can be erased prior to the execution. Formally, this amounts to establishing a bisimulation between the reduction of source terms and the reduction of their erasure, up to ι-steps. This can be decomposed into the conjunction of three properties: a β-step in the coercion language erases on a reduction step in the untyped λ-calculus; both sides of a ι-step erase on the same untyped term; and finally, when the erasure of a term contains a redex, it must reduce in a final number of ι-steps to reveal this redex in the source.

Definition 1 (Erasability)**.** *Coercions are erasable if for any well-typed term M the following properties hold:*

1. *If $M \rightsquigarrow_\beta N$, then $\lfloor M \rfloor \rightsquigarrow \lfloor N \rfloor$.*
2. *If $M \rightsquigarrow_\iota N$, then $\lfloor M \rfloor = \lfloor N \rfloor$.*
3. *If $\lfloor M \rfloor \rightsquigarrow a$, then $M \rightsquigarrow_\iota^\star \rightsquigarrow_\beta N$ such that $\lfloor N \rfloor = a$.*

Part 1 and 2 define the forward simulation up to erasure which means that coercions do not contribute to the computation. Part 3 is the backward simulation which means that coercions never block the computation. The backward simulation is usually the hardest to establish.

Coercions of System F are erasable, indeed.

2.2 System F_η

Mitchell defined System F_η [Mitchell, 1988] in curry-style as the closure of System F by η-reduction: a term M is typable and has type τ in F_η, if and only if there is a term N that η-reduces on M and has type τ in System F. We recall the definition of η-reduction:

$$\lambda x.a\,x \rightsquigarrow_\eta a \qquad\qquad \frac{a_1 \rightsquigarrow_\eta a_2}{e[a_1] \rightsquigarrow_\eta e[a_2]}$$

Hence, F_η could be defined by adding the following typing rule on the left-hand side to the definition of typing judgments of System F:

$$\frac{\Gamma \vdash a' : \tau \qquad a' \rightsquigarrow_\eta a}{\Gamma \vdash a : \tau} \qquad\qquad \frac{\Gamma \vdash \lambda x.a\,x : \tau}{\Gamma \vdash a : \tau}$$

In fact, Mitchell gave the simpler rule on the right-hand side—for a curry-style presentation where parameters of functions do not carry type annotations. This rule only performs a toplevel η-expansion. However, since the type system is compositional, the previous rule becomes admissible, so both presentations are equivalent. These presentations are still unusual and indirect: they amount to type a well-chosen η-expansion of the term a instead of a itself. One could think of

$$
\begin{array}{rcll}
W & ::= & \ldots \mid \diamond^{\tau} \mid G \mathbin{@} W \mid G \xrightarrow{\tau} G \mid \mathrm{Dist}^{\tau \to \forall\alpha.\tau} \mid (G * G) \mid \mathrm{Dist}^{\forall\alpha.(\tau*\tau)} \mid \mathtt{Top}^{\tau} & \text{expressions} \\
\tau & ::= & \ldots \mid \top & \text{types}
\end{array}
$$

Figure 8: System F_η: new syntax *wrt* System F

$$
\begin{array}{rcll}
p & ::= & \ldots \mid (G \xrightarrow{\tau} G) \mathbin{@} p \mid \mathrm{Dist}^{\tau \to \forall\alpha.\tau} \mathbin{@} p \mid \mathrm{Dist}^{\tau \to \forall\alpha.\tau} \mathbin{@} (\lambda\alpha\, p) & \text{prevalues} \\
 & & \mid (G * G) \mathbin{@} p \mid \mathrm{Dist}^{\forall\alpha.(\tau*\tau)} \mathbin{@} p \mid \mathrm{Dist}^{\forall\alpha.(\tau*\tau)} \mathbin{@} (\lambda\alpha\, p) & \\
v & ::= & \ldots \mid \mathtt{Top}^{\tau} \mathbin{@} v & \text{values} \\
P & ::= & \ldots \mid G \mathbin{@} [] & \text{retyping contexts}
\end{array}
$$

Figure 9: System F_η: new values and contexts *wrt* System F

saturating the source term with η-expansions, but η-expansion (of yet ill-typed terms) does not terminate.

Mitchell also gave an alternative presentation of F_η that does not require η-expansion of source terms but instead uses a type conversion relation $\tau \subseteq \sigma$ called *type containment* [Mitchell, 1988], and adding the following typing rule (and rules defining type-containment):

$$\frac{\Gamma \vdash a : \tau \qquad \tau \subseteq \sigma}{\Gamma \vdash a : \sigma}$$

Mitchell proved that both presentations are equivalent.

In both presentations, η-expansions or type conversions are left implicit and must be rebuilt during type checking. We instead present an explicit version of F_η where all type related information are part of source terms—which is the precise point of introducing coercions. We also extend F_η with pairs. The η-reduction rule for pairs is given on the left-hand side. And we give on the right-hand side what a Mitchell-style typing rule for coercing pairs would have been.

$$(a.1, a.2) \rightsquigarrow_\eta a \qquad\qquad \frac{\Gamma \vdash (a.1, a.2) : \tau}{\Gamma \vdash a : \tau}$$

However, since our approach is to have explicit coercions, we will not use these implicit rules neither. We define F_η as an extension of System F.

Syntax The syntax of F_η is given on Figure 8. Since it extends System F, we only give the new constructs, using the ellipsis "$\ldots$" where the corresponding definition of System F should be unfolded. We do not repeat syntactic categories that are left unmodified. And we only write one letter on the left-hand side of syntactic categories.

Expressions of F_η are an extension of expressions of System F with new forms of expressions called *coercions* that are used to witness type conversions in F_η. The letter G formally ranges over expressions, but we use it in practice to denote expressions that are *coercions* and to distinguish them from expressions that are *terms*, written with letters M or N, while letter W is used to range over either form. The choice of letter is only an informal indication to help the reader. The formal distinction within expressions between terms and coercions will be given by typing judgments.

Coercions may also be thought of as type-conversion proof terms: $G \mathbin{@} W$ witnesses transitivity of type conversion when W is another coercion G'. Otherwise, when W is a term M it means the application of the coercion G to M. The hole $\diamond^{\tau}$ witnesses reflexivity; the arrow coercion $G_1 \xrightarrow{\tau} G_2$ builds a coercion between function types given a coercion G_1 for the domain and a coercion G_2 for the codomain; the coercion $\mathrm{Dist}^{\tau \to \forall\alpha.\sigma}$ pushes a quantifier on the domain of a function type; the coercion $(G_1 * G_2)$ builds a coercion between product types given two coercions, one for each side of the product; and the coercion $\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)}$ pushes a quantifier below a product type. These two last constructs are not present in the original presentation of F_η, which does not contain pairs. Finally, we introduce a coercion $\mathtt{Top}^{\tau}$ that coerces any expression of type τ to one of type $\top$.

$$\textsc{TermCoer}\ \frac{\Gamma \vdash G : \tau \rhd \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash G \mathbin{@} M : \sigma} \qquad \textsc{CoerTrans}\ \frac{\Gamma \vdash G_2 : \sigma \rhd \rho \qquad \Gamma \vdash G_1 : \tau \rhd \sigma}{\Gamma \vdash G_2 \mathbin{@} G_1 : \tau \rhd \rho} \qquad \textsc{CoerTypeLam}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash G : \tau \rhd \sigma}{\Gamma \vdash \lambda\alpha\, G : \tau \rhd \forall\alpha.\sigma}$$

$$\textsc{CoerTypeApp}\ \frac{\Gamma \vdash \rho \qquad \Gamma \vdash G : \tau \rhd \forall\alpha.\sigma}{\Gamma \vdash G\,\rho : \tau \rhd \sigma[\alpha \leftarrow \rho]} \qquad \textsc{CoerArrow}\ \frac{\Gamma \vdash G_1 : \tau \rhd \tau' \qquad \Gamma \vdash G_2 : \sigma \rhd \sigma'}{\Gamma \vdash G_1 \xrightarrow{\tau} G_2 : \tau' \to \sigma \rhd \tau \to \sigma'}$$

$$\textsc{CoerDist}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\tau \to \forall\alpha.\sigma} : \forall\alpha.(\tau \to \sigma) \rhd \tau \to \forall\alpha.\sigma} \qquad \textsc{CoerProduct}\ \frac{\Gamma \vdash G_1 : \tau \rhd \tau' \qquad \Gamma \vdash G_2 : \sigma \rhd \sigma'}{\Gamma \vdash (G_1 * G_2) : (\tau * \sigma) \rhd (\tau' * \sigma')}$$

$$\textsc{CoerDistTypeProduct}\ \frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\forall\alpha.(\tau * \sigma)} : \forall\alpha.(\tau * \sigma) \rhd (\forall\alpha.\tau * \forall\alpha.\sigma)} \qquad \textsc{CoerDot}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \diamond^{\tau} : \tau \rhd \tau} \qquad \textsc{CoerForget}\ \frac{\Gamma \vdash \tau}{\Gamma \vdash \mathtt{Top}^{\tau} : \tau \rhd \top}$$

Figure 10: System F_η: new typing rules *wrt* System F

Types are just extended with the new constant type $\top$. This constant is not part of the original definition of F_η either, but it would be encodable if we extended F_η with existential types.

Static semantics The static semantics of F_η is described in Figure 10 as an extension of the static semantics of System F. The typing judgment $\Gamma \vdash M : \tau$ of System F (given in Figure 3) is extended with a new rule TermCoer for applying a coercion G to a term M in $G \mathbin{@} M$. This rule uses a new typing judgment for typing coercions $\Gamma \vdash G : \tau \rhd \sigma$, stating that in environment Γ, coercion G transforms a term of type τ into one of type σ. To ease reading of coercion typing rules, we also say that G has *coercion type* $\tau \rhd \sigma$—but a coercion type is not a type.

The form $G \mathbin{@} W$ may also be used when W is a coercion G' to mean the composition of coercions, as described by Rule CoerTrans. If G_1 and G_2 have coercion types $\tau \rhd \sigma$ and $\sigma \rhd \rho$ respectively, then $G_2 \mathbin{@} G_1$ has coercion type $\tau \rhd \rho$. Notice that $G_2 \mathbin{@} G_1$ is a coercion while $G \mathbin{@} M$ is a term.

Similarly, type abstractions and type applications build both terms and coercions, as described by rules TermTypeLam and TermTypeApp for terms and their analog rules CoerTypeLam and CoerTypeApp for coercions. Rule CoerTypeLam says that $\lambda\alpha\, G$ has coercion type $\tau \rhd \forall\alpha.\sigma$ in environment Γ if G has coercion type $\tau \rhd \sigma$ in environment Γ extended with α. Conversely, Rule CoerTypeApp says that $G\,\rho$ has coercion type $\tau \rhd \sigma[\alpha \leftarrow \rho]$ if G has coercion type $\tau \rhd \forall\alpha.\sigma$.

CoerArrow builds a coercion between functions given two coercions, one for the argument used contravariantly and one for the result used covariantly. It can be seen as a congruence rule for arrows. CoerDist pushes a universally bound variable on the right-hand side of an arrow when it does not appear free on the left-hand side. CoerProduct is the analog of CoerArrow for product types. It is a congruence rule that coerces pairs given two subcoercions. And CoerDistTypeProduct is the analog of CoerDist. Rule CoerDot is for the identity coercion, that leaves the type of its argument unchanged. Finally, CoerForget states that $\mathtt{Top}^{\tau}$ coerces any well-typed term into one of type $\top$.

The typing judgments $\Gamma \vdash M : \tau$ and $\Gamma \vdash G : \tau \rhd \sigma$ also classify expressions between *terms* and *coercions*, respectively. In particular, an expression W can be either a term or a coercion, but cannot be both, simultaneously.

Notice that the typing environment Γ is only used for well-formedness in coercion judgments $\Gamma \vdash G : \tau \rhd \sigma$. Hence, it induces a binary type-conversion relation $\tau \rhd \sigma$ defined as the existence of a typing environment Γ and of a coercion G such that $\Gamma \vdash G : \tau \rhd \sigma$. Type conversion is a preorder, which is witnessed by the identity and transitivity coercions. Type conversion is called *type containment* in F_η [Mitchell, 1988]. Type containment has been show undecidable. This makes F_η a difficult system to use, as coercions cannot be entirely left implicit in source terms.

$$\frac{\Gamma \vdash \mathit{ok}}{\Gamma \vdash \top}\ \textsc{TypeTop}$$

Figure 11: System F_η: new well-formedness rules *wrt* F

$$(G_1 \xrightarrow{\tau} G_2) \mathbin{@} (\lambda(x:\sigma)\ M) \rightsquigarrow_\iota \lambda(x:\tau)\ G_2 \mathbin{@} M[x \leftarrow G_1 \mathbin{@} x] \quad \textsc{RedTransArrow}$$

$$\mathrm{Dist}^{\tau' \to \forall\alpha.\sigma'} \mathbin{@} (\lambda\alpha\, \lambda(x:\tau)\ M) \rightsquigarrow_\iota \lambda(x:\tau)\ \lambda\alpha\, M \quad \textsc{RedTransDist}$$

$$(G_1 * G_2) \mathbin{@} (M, N) \rightsquigarrow_\iota (G_1 \mathbin{@} M, G_2 \mathbin{@} N) \quad \textsc{RedTransProduct}$$

$$\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)} \mathbin{@} (\lambda\alpha\, (M, N)) \rightsquigarrow_\iota (\lambda\alpha\, M, \lambda\alpha\, N) \quad \textsc{RedTransDistTypeProduct}$$

$$\diamond^\tau \mathbin{@} M \rightsquigarrow_\iota M \quad \textsc{RedTransDot}$$

$$P[G] \mathbin{@} M \rightsquigarrow_\iota P[G \mathbin{@} M] \quad \textsc{RedTransCoer}$$

Figure 12: System F_η: new reduction rules *wrt* System F

The undecidability seems to come from the mixing of distributivity with contravariance in arrow coercions. In particular, if we remove distributivity, then the system becomes decidable as then both sides of type containment share the same structure.

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 whenever they can be unambiguously reconstructed from the context.

Well-formedness judgments are extended with the rule of Figure 11

Operational semantics The operational semantics of F_η is defined in Figure 12 as an extension of the semantics of System F. Among the new nodes, the application of a coercion to a term $G \mathbin{@} M$ plays the role of a destructor: it must be reduced when M is sufficiently evaluated, except for the particular case were G is $\mathtt{Top}^\tau$ since $\mathtt{Top}^\tau\ v$ is a value—one we cannot destruct but only pass around. The other new nodes are all constructors. To allow reduction in the application of a coercion to a term, we extend the grammar of retyping contexts P with $G \mathbin{@} []$. We then have six reduction rules, one for each possible shape of G.

- When G is an arrow coercion $G_1 \xrightarrow{\tau} G_2$ and M is a function $\lambda(x:\sigma)\ M$, Rule RedTransArrow 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 $\mathrm{Dist}^{\tau \to \forall\alpha.\sigma}$ and M is a polymorphic function $\lambda\alpha\, \lambda(x:\tau)\ M$, Rule RedTransDist reduces the application by exchanging the type and value parameter to $\lambda(x:\tau)\ \lambda\alpha\, M$, which is sound since α cannot be free in τ.
- When G is a product coercion $(G_1 * G_2)$ and M is a pair (M, N), Rule RedTransProduct reduces the application by pushing G_1 on M and G_2 on N.
- When G is a distributivity coercion from a forall on a product, $\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)}$, and M is a polymorphic product $\lambda\alpha\, (M, N)$, Rule RedTransDistTypeProduct reduces the application by exchanging the type and pair construct to $(\lambda\alpha\, M, \lambda\alpha\, N)$.
- When G is the identity, Rule RedTransDot just drops it.
- When G is of the form $P[G']$, the Rule RedTransCoer reduces $G \mathbin{@} M$ to $P[G' \mathbin{@} M]$.

$$
\begin{array}{rll}
c & & \text{coercion variables} \\
W & ::= \ldots \mid c \mid \lambda(c : \tau \rhd \tau)\, W \mid W \,\$\, G \mid \mathrm{Dist}^{\tau \to (\tau \rhd \tau) \Rightarrow \tau} \mid \mathrm{Dist}^{(\tau \rhd \tau) \Rightarrow (\tau * \tau)} & \text{expressions} \\
\tau & ::= \ldots \mid (\tau \rhd \tau) \Rightarrow \tau & \text{types} \\
B & ::= \ldots \mid (c : \tau \rhd \tau) & \text{bindings}
\end{array}
$$

Figure 13: System F_ι: new syntax *wrt* F_η

Rule RedTransCoer is actually a meta-rule: expanding the definition of P, it unfolds to the following three rules:

$$
\begin{array}{rr}
(\lambda\alpha\, G) \mathbin{@} M \rightsquigarrow_\iota \lambda\alpha\, (G \mathbin{@} M) & \textsc{RedTransTypeLam} \\
(G\,\tau) \mathbin{@} M \rightsquigarrow_\iota (G \mathbin{@} M)\,\tau & \textsc{RedTransTypeApp} \\
(G_2 \mathbin{@} G_1) \mathbin{@} M \rightsquigarrow_\iota G_2 \mathbin{@} (G_1 \mathbin{@} M) & \textsc{RedTransTrans}
\end{array}
$$

All these rules are ι-reductions (their erasure is an equality): they wrap coercions around terms so that β-redexes can be exposed and eventually fire. Although we say that they do not actively contribute to the underlying computation, they enable it to continue while maintaining well-typedness.

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

RedTransArrowApp
$$((G_1 \xrightarrow{\tau} G_2) \mathbin{@} M)\ N \rightsquigarrow_\iota G_2 \mathbin{@} (M\ (G_1 \mathbin{@} N)) \qquad (\mathrm{Dist}^{\tau \to \forall\alpha.\sigma} \mathbin{@} M)\ N\,\rho \rightsquigarrow_\iota M\,\rho\ N$$

$$(\mathrm{Dist}^{\tau \to \forall\alpha.\sigma} \mathbin{@} (\lambda\alpha\, M))\ N \rightsquigarrow_\iota \lambda\alpha\, M\ N \qquad ((G_1 * G_2) \mathbin{@} M).\mathtt{1} \rightsquigarrow_\iota G_1 \mathbin{@} (M.\mathtt{1})$$

$$((G_1 * G_2) \mathbin{@} M).\mathtt{2} \rightsquigarrow_\iota G_2 \mathbin{@} (M.\mathtt{2})$$

However, doing so would narrow the set of values, and reestablishing progress would require to allow coercions to bind, as described in §7, which is something we prefer to avoid. To better understand why we need binding coercions, let's forget about pairs and focus on arrows. The rule RedTransArrowApp is telling us that $(G_1 \xrightarrow{\tau} G_2) \mathbin{@} M$ behaves like a arrow constructor, since when it is under the arrow destructor (which is the application node) they reduce. This morally means that $(G_1 \xrightarrow{\tau} G_2) \mathbin{@} M$ behaves like $\lambda(x : \tau)\ M'$—actually this can be easily understood after reification (Section 3.3). This correspondence implies that we should define an additional reduction rule with $(G_1 \xrightarrow{\tau} G_2) \mathbin{@} M$ everywhere we previously had a $\lambda(x : \tau)\ M'$, and in particular in RedTransDist. So we have to define the reduction of $\mathrm{Dist}^{\tau \to \forall\alpha.\sigma} \mathbin{@} (\lambda\alpha\, (G_1 \xrightarrow{\tau} G_2) \mathbin{@} M)$ where $\Gamma, \alpha \vdash M : \tau' \to \sigma'$, $\Gamma, \alpha \vdash G_1 : \tau \rhd \tau'$, and $\Gamma, \alpha \vdash G_2 : \sigma' \rhd \sigma$. We would like to say it reduces to $(G_1 \xrightarrow{\tau} (\lambda\alpha\, G_2)) \mathbin{@} M$ but we need the $\lambda\alpha$ to bind additionally in both G_1 and M.

3 Abstraction over coercions: F_ι

In this section we present F_ι, a natural extension of F_η with abstraction over coercions. We present the syntax, static and dynamic semantics of the language, and show that it is well-behaved: it is normalizing, confluent and sound. Although it satisfies the forward simulation, coercions may appear in wedging position during the reduction and are thus not erasable.

3.1 Definition of F_ι

Syntax of F_ι The syntax of F_ι is presented on Figure 13 as an extension of the syntax of F_η. The syntax of expressions is extended with coercion variables, written with letter c, abstractions over coercions $\lambda(c : \tau \rhd \sigma)\, W$, applications to a coercion $W \,\$\, G$, and distributivity of coercion abstraction on arrows $\mathrm{Dist}^{\tau \to (\rho \rhd \rho') \Rightarrow \sigma}$ and products $\mathrm{Dist}^{(\rho \rhd \rho') \Rightarrow (\tau * \sigma)}$.

$$\textsc{CoerVar}\ \frac{\Gamma \vdash ok \qquad (c : \tau \rhd \sigma) \in \Gamma}{\Gamma \vdash c : \tau \rhd \sigma} \qquad \textsc{TermCoerLam}\ \frac{\Gamma, (c : \rho \rhd \rho') \vdash M : \sigma}{\Gamma \vdash \lambda(c : \rho \rhd \rho')\, M : (\rho \rhd \rho') \Rightarrow \sigma}$$

$$\textsc{CoerCoerLam}\ \frac{\Gamma, (c : \rho \rhd \rho') \vdash G : \tau \rhd \sigma}{\Gamma \vdash \lambda(c : \rho \rhd \rho')\, G : \tau \rhd (\rho \rhd \rho') \Rightarrow \sigma} \qquad \textsc{TermCoerApp}\ \frac{\Gamma \vdash G : \rho \rhd \rho' \qquad \Gamma \vdash M : (\rho \rhd \rho') \Rightarrow \sigma}{\Gamma \vdash M \,\$\, G : \sigma}$$

$$\textsc{CoerCoerApp}\ \frac{\Gamma \vdash G' : \rho \rhd \rho' \qquad \Gamma \vdash G : \tau \rhd (\rho \rhd \rho') \Rightarrow \sigma}{\Gamma \vdash G \,\$\, G' : \tau \rhd \sigma}$$

$$\textsc{CoerDistCoerArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma \vdash \rho \qquad \Gamma \vdash \rho' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\tau \to (\rho \rhd \rho') \Rightarrow \sigma} : (\rho \rhd \rho') \Rightarrow (\tau \to \sigma) \rhd \tau \to (\rho \rhd \rho') \Rightarrow \sigma}$$

$$\textsc{CoerDistCoerProduct}\ \frac{\Gamma \vdash \rho \qquad \Gamma \vdash \rho' \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{(\rho \rhd \rho') \Rightarrow (\tau * \sigma)} : (\rho \rhd \rho') \Rightarrow (\tau * \sigma) \rhd (((\rho \rhd \rho') \Rightarrow \tau) * ((\rho \rhd \rho') \Rightarrow \sigma))}$$

Figure 14: System F_ι: new typing rules *wrt* F_η

$$\textsc{TypeCoerArrow}\ \frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \rho \qquad \Gamma \vdash \rho'}{\Gamma \vdash (\rho \rhd \rho') \Rightarrow \sigma} \qquad \textsc{EnvCoer}\ \frac{c \notin \mathrm{dom}(\Gamma) \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma, (c : \tau \rhd \sigma) \vdash ok}$$

Figure 15: System F_ι: new well-formedness rules *wrt* F_ι

These two last constructs could be omitted. We do not need them for any of the following results (from soundness to bisimulation). But since they resemble the distributivity rules for type abstraction in F_η and since they are naturally present in an approach with binding coercions (Section 7), we add them in this presentation.

Since the language is explicitly typed, the abstraction over coercion provides the types of both the domain and the codomain of the coercion. Notice that $\tau \rhd \sigma$ is not a type itself, since coercions are not themselves first-class in F_ι: they cannot be stored into data structures—or created by computation. They can just be abstracted over and passed around as soon as they are created. As above, to ease the discourse, we call the pair of types $\tau \rhd \sigma$ a *coercion type*: it describes a coercion that expects a term of type τ and returns one of type σ. Both terms and coercions can abstract over coercions—and therefore be applied to coercions.

Types are extended with coercion arrows $(\tau \rhd \sigma) \Rightarrow \rho$ to describe the type of a term that expects a coercion of type $\tau \rhd \sigma$ and returns a term of type ρ. Bindings in typing environments are also extended with $(c : \tau \rhd \sigma)$ to introduce a coercion c of coercion type $\tau \rhd \sigma$.

Typing The static semantics of F_ι extends that of F_η with new typing rules given on Figure 14. Rules CoerVar, TermCoerLam, and TermCoerApp are similar to rules TermVar, TermTermLam, and TermTermApp, except that the type τ is replaced by a coercion type $\rho \rhd \rho'$. Rule CoerCoerLam is similar to TermCoerLam but for coercion typings: the coercion $\lambda(c : \rho \rhd \rho')\, G$ has the coercion type $\tau \rhd (\rho \rhd \rho') \Rightarrow \sigma$ in Γ if the coercion G has the coercion type $\tau \rhd \sigma$ in $\Gamma, (c : \rho \rhd \rho')$. CoerCoerApp is similar to TermCoerApp but for coercion typings: $G \,\$\, G'$ has coercion type $\tau \rhd \sigma$ if G has coercion type $\tau \rhd (\rho \rhd \rho') \Rightarrow \sigma$ and G' has coercion

$$
\begin{array}{rcll}
p & ::= & \ldots \mid p \,\$\, G \mid c \mathbin{@} v & \text{prevalues} \\
& & \mid \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \mathbin{@} p \mid \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \mathbin{@} (\lambda(c:\tau\rhd\tau)\,p) & \\
& & \mid \mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)} \mathbin{@} p \mid \mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)} \mathbin{@} (\lambda(c:\tau\rhd\tau)\,p) & \\
v & ::= & \ldots \mid \lambda(c:\tau\rhd\tau)\,v & \text{values} \\
P & ::= & \ldots \mid \lambda(c:\tau\rhd\tau)\,[] \mid [] \,\$\, G & \text{retyping contexts}
\end{array}
$$

$$
\textsc{RedCoer}\quad (\lambda(c:\tau\rhd\sigma)\,M) \,\$\, G \rightsquigarrow_\iota M[c\leftarrow G]
$$

$$
\textsc{RedTransDistCoerArrow}\quad \mathrm{Dist}^{\sigma_1\to(\sigma_2\rhd\sigma_3)\Rightarrow\sigma_4} \mathbin{@} (\lambda(c:\rho\rhd\rho')\,\lambda(x:\tau)\;M) \rightsquigarrow_\iota \lambda(x:\tau)\,\lambda(c:\rho\rhd\rho')\,M
$$

$$
\textsc{RedTransDistCoerProduct}\quad \mathrm{Dist}^{(\sigma_1\rhd\sigma_2)\Rightarrow(\sigma_3*\sigma_4)} \mathbin{@} (\lambda(c:\rho\rhd\rho')\,(M,N)) \rightsquigarrow_\iota (\lambda(c:\rho\rhd\rho')\,M, \lambda(c:\rho\rhd\rho')\,N)
$$

Figure 16: System F_ι: new operational semantics *wrt* F_η

type $\rho\rhd\rho'$. CoerDistCoerArrow is similar to CoerDist. It swaps a coercion and a term abstraction. CoerDistCoerProduct is similar to CoerDistTypeProduct. It swaps a coercion abstraction and a pair constructor.

Well-formedness judgments are extended in the obvious way, with the rules of Figure 15.

Operational semantics The operational semantics of F_ι is defined as an extension of the one of F_η in Figure 16. The changes are new forms of values and evaluation contexts. We add two reduction rules RedTransDistCoerArrow and RedTransDistCoerProduct for distributivity, which are very similar to their analogs RedTransDist and RedTransDistTypeProduct. They simply swap coercion abstraction with either term abstraction or pair constructor. There is also one new reduction rule RedCoer for β-reduction of coercion abstraction. One may wonder why this rule only applies to coercion abstraction over terms. The reason is that, as for F_η, we do not reduce coercions alone, but only their application to terms.

The application of a coercion abstraction G_1 to a coercion G_2 is thus only reduced when applied to a term M —as other complex coercions— by Rule RedTransCoer by wrapping elements of G around M as in the two first steps below, so that Rule RedCoer can fire (last step):

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

The new reductions are all ι-steps, indeed.

3.2 Soundness

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

Definition 2 (Valid Extension)**.** *Γ' is a valid extension of Γ iff $\Gamma\subseteq\Gamma'$ and $\Gamma'\vdash ok$.*

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

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

$$
\begin{array}{rcl}
\lceil \alpha \rceil &=& \alpha \\
\lceil \tau \to \sigma \rceil &=& \lceil \tau \rceil \to \lceil \sigma \rceil \\
\lceil (\tau * \sigma) \rceil &=& (\lceil \tau \rceil * \lceil \sigma \rceil) \\
\lceil \forall \alpha.\tau \rceil &=& \forall \alpha.\lceil \tau \rceil \\
\lceil (\tau \rhd \sigma) \Rightarrow \rho \rceil &=& (\lceil \tau \rceil \to \lceil \sigma \rceil) \to \lceil \rho \rceil \\
\lceil \top \rceil &=& \forall \alpha.(\alpha \to \alpha)
\end{array}
\qquad
\begin{array}{rcl}
\lceil x \rceil &=& x \\
\lceil \lambda(x : \tau)\ M \rceil &=& \lambda(x : \lceil \tau \rceil)\ \lceil M \rceil \\
\lceil M\ N \rceil &=& \lceil M \rceil\ \lceil N \rceil \\
\lceil (M, N) \rceil &=& (\lceil M \rceil, \lceil N \rceil) \\
\lceil M.\mathtt{1} \rceil &=& \lceil M \rceil.\mathtt{1} \\
\lceil M.\mathtt{2} \rceil &=& \lceil M \rceil.\mathtt{2}
\end{array}
$$

$$
\begin{array}{rcl}
\lceil \lambda\alpha\, M \rceil &=& \lambda\alpha\, \lceil M \rceil \\
\lceil M\, \tau \rceil &=& \lceil M \rceil\ \lceil \tau \rceil \\
\lceil G \mathbin{@} M \rceil &=& \lceil G \rceil\ \lceil M \rceil \\
\lceil \lambda(c : \tau \rhd \sigma)\, M \rceil &=& \lambda(x_c : \lceil \tau \rceil \to \lceil \sigma \rceil)\ \lceil M \rceil \\
\lceil M \mathbin{\$} G \rceil &=& \lceil M \rceil\ \lceil G \rceil
\end{array}
$$

$$
\begin{array}{rcl}
\lceil c \rceil &=& \lceil x_c \rceil \\
\lceil \diamond^{\tau} \rceil &=& \lceil \lambda(x : \tau)\ x \rceil \\
\lceil G_1 \xrightarrow{\tau} G_2 \rceil &=& \lceil \lambda(y : \mathrm{dom}(G_1 \xrightarrow{\tau} G_2))\ \lambda(x : \tau)\ G_2 \mathbin{@} (y\ (G_1 \mathbin{@} x)) \rceil \\
\lceil \mathrm{Dist}^{\tau \to \forall\alpha.\sigma} \rceil &=& \lceil \lambda(y : \mathrm{dom}(\mathrm{Dist}^{\tau \to \forall\alpha.\sigma}))\ \lambda(x : \tau)\ \lambda\alpha\, y\, \alpha\ x \rceil \\
\lceil \mathrm{Dist}^{\tau \to (\rho \rhd \rho') \Rightarrow \sigma} \rceil &=& \lceil \lambda(y : \mathrm{dom}(\mathrm{Dist}^{\tau \to (\rho \rhd \rho') \Rightarrow \sigma}))\ \lambda(x : \tau)\ \lambda(c : \rho \rhd \rho')\ (y \mathbin{\$} c)\ x \rceil \\
\lceil (G_1 * G_2) \rceil &=& \lceil \lambda(y : \mathrm{dom}(G_1 * G_2))\ (G_1 \mathbin{@} y.\mathtt{1}, G_2 \mathbin{@} y.\mathtt{2}) \rceil \\
\lceil \mathrm{Dist}^{\forall\alpha.(\tau * \sigma)} \rceil &=& \lceil \lambda(y : \mathrm{dom}(\mathrm{Dist}^{\forall\alpha.(\tau * \sigma)}))\ (\lambda\alpha\, (y\, \alpha).\mathtt{1}, \lambda\alpha\, (y\, \alpha).\mathtt{2}) \rceil \\
\lceil \mathrm{Dist}^{(\rho \rhd \rho') \Rightarrow (\tau * \sigma)} \rceil &=& \lceil \lambda(y : \mathrm{dom}(\mathrm{Dist}^{(\rho \rhd \rho') \Rightarrow (\tau * \sigma)}))\ (\lambda(c : \rho \rhd \rho')\ (y \mathbin{\$} c).\mathtt{1}, \lambda(c : \rho \rhd \rho')\ (y \mathbin{\$} c).\mathtt{2}) \rceil \\
\lceil \mathtt{Top}^{\tau} \rceil &=& \lceil \lambda(y : \tau)\ \lambda\alpha\, \lambda(x : \alpha)\ x \rceil \\
\lceil P[G] \rceil &=& \lceil \lambda(x : \mathrm{dom}(G))\ P[G \mathbin{@} x] \rceil
\end{array}
$$

$$
\begin{array}{rcl}
\lceil \emptyset \rceil &=& \emptyset \\
\lceil \Gamma, B \rceil &=& \lceil \Gamma \rceil, \lceil B \rceil \\
\lceil \alpha \rceil &=& \alpha \\
\lceil (x : \tau) \rceil &=& (x : \lceil \tau \rceil) \\
\lceil (c : \tau \rhd \sigma) \rceil &=& (x_c : \lceil \tau \rceil \to \lceil \sigma \rceil)
\end{array}
$$

Figure 17: Reification of F_ι into System F

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

Proof. The proof is standard. □

Lemma 4 (Extract Environment)**.** *If* $\Gamma \vdash M : \tau$, $\Gamma \vdash G : \tau \rhd \sigma$, *or* $\Gamma \vdash \tau$ *holds, then* $\Gamma \vdash ok$.

Lemma 5 (Type Substitution)**.** *If* $\Gamma \vdash \rho$ *holds, then (we write* θ *for* $[\alpha \leftarrow \rho]$*):*

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

Proof. This is a standard proof. □

Lemma 6 (Extract Type)**.** 1. *If* $\Gamma \vdash M : \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*

2. *If* $\Gamma \vdash G : \tau \rhd \sigma$ *holds, then* $\Gamma \vdash \tau$ *and* $\Gamma \vdash \sigma$ *hold.*

Lemma 7 (Term Substitution)**.** *If* $\Gamma \vdash N : \rho$ *holds, then:*

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

Proof. The proof is standard and relies on Lemma 3. □

Lemma 8 (Coercion Substitution)**.** *If* $\Gamma \vdash G : \rho \rhd \rho'$ *holds, then:*

1. *If* $\Gamma, (c : \rho \rhd \rho') \vdash M : \tau$ *holds, then* $\Gamma \vdash M[c \leftarrow G] : \tau$ *holds.*
2. *If* $\Gamma, (c : \rho \rhd \rho') \vdash G' : \tau \rhd \sigma$ *holds, then* $\Gamma \vdash G'[c \leftarrow G] : \tau \rhd \sigma$ *holds.*
3. *If* $\Gamma, (c : \rho \rhd \rho') \vdash \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*

Proof. The proof is standard and relies on Lemma 3. □

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

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

- RedContextBeta and RedContextIota: By induction on the context. The same typing rule works by using the induction hypothesis to build the premise in the hole (other premises stay the same).
- RedTerm: By Lemma 7.
- RedFirst and RedSecond: We use the corresponding subderivation.
- RedCoer: By Lemma 8.
- RedType: By Lemma 5.

- RedTransArrow: We have $\Gamma \vdash G_1 : \tau \rhd \tau'$, $\Gamma \vdash G_2 : \sigma \rhd \sigma'$, and $\Gamma, (x : \tau') \vdash M : \sigma$. We build $\Gamma, (y : \tau) \vdash G_1 : \tau \rhd \tau'$ and $\Gamma, (y : \tau) \vdash G_2 : \sigma \rhd \sigma'$ using Lemma 6 (to show the valid extension) and 3 where y is fresh. We build $\Gamma, (y : \tau), (x : \tau') \vdash M : \sigma$ using Lemma 6 and 3. We can now use Lemma 7 to build $\Gamma, (y : \tau) \vdash M[x \leftarrow G_1 \mathbin{@} y] : \sigma'$. The result simply follows.
- RedTransDist: We have $\Gamma, \alpha, (x : \tau) \vdash M : \sigma$. We build $\Gamma, (x : \tau), \alpha \vdash M : \sigma$ using Lemma 3. We use $\Gamma \vdash \tau$ to show that $\Gamma, (x : \tau), \alpha$ is a valid extension of $\Gamma, \alpha, (x : \tau)$.
- RedTransDistCoerArrow: We have $\Gamma, (c : \rho \rhd \rho'), (x : \tau) \vdash M : \sigma$. We build $\Gamma, (x : \tau), (c : \rho \rhd \rho') \vdash M : \sigma$ using Lemma 3. We show $\Gamma \vdash \tau$ using Lemma 8.
- RedTransProduct: Obvious.
- RedTransDistTypeProduct: We have $\Gamma, \alpha \vdash M : \tau$ and $\Gamma, \alpha \vdash M : \sigma$. The result is obvious.
- RedTransDistCoerProduct: We have $\Gamma, (c : \rho \rhd \rho') \vdash M : \tau$ and $\Gamma, (c : \rho \rhd \rho') \vdash M : \sigma$. The result is obvious.
- RedTransDot: Obvious.
- RedTransCoer: By induction on P.
  - $\lambda\alpha\,[]$: We have $\Gamma, \alpha \vdash G : \tau \rhd \sigma$ and $\Gamma \vdash M : \tau$. We use Lemma 3 to build $\Gamma, \alpha \vdash M : \tau$ (and Lemma 4 to show $\Gamma, \alpha \vdash ok$).
  - $[]\,\tau$: Easy.
  - $G \mathbin{@} []$: Easy.
  - $\lambda(c : \rho \rhd \rho')\,[]$: We have $\Gamma, (c : \rho \rhd \rho') \vdash G : \tau \rhd \sigma$ and $\Gamma \vdash M : \tau$. We use Lemma 3 to build $\Gamma, (c : \rho \rhd \rho') \vdash M : \tau$ (and Lemma 4 to show $\Gamma, (c : \rho \rhd \rho') \vdash ok$).
  - $[]\,\$\,G$: Easy.

□

The progress lemma also works by induction on terms, as usual, using typing to reject cases that are not values and cannot reduce. And we use a value classification lemma, which in strong reduction states like this:

Lemma 10 (Classification)**.** *If $\Gamma \vdash v : \tau$ holds, then either v is a prevalue p or:*

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

Proof. This holds with a simple induction on v. □

Proposition 11 (Progress)**.** *If $\Gamma \vdash M : \tau$ holds, then either M is a value or M reduces.*

(Proof p. 37)

3.3 Termination of reduction

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

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

Reification is described on Figure 17. We write $\lceil M \rceil$ for the reification of M. Coercions typed as $\tau \rhd \sigma$ are reified as functions of type $\tau \to \sigma$. Hence, a coercion abstraction $\lambda(c : \tau \rhd \sigma)\, M$ is reified as a higher-order function $\lambda(x_c : \lceil\tau\rceil \to \lceil\sigma\rceil)\, M$. A coercion variable c is reified to a term variable x_c (we assume an injective mapping of coercion variables to reserved term variables). Thus, the type $(\tau \rhd \sigma) \Rightarrow \rho$ of a term abstracted over a coercion is translated into the type $(\tau \to \sigma) \to \rho$ of a higher-order function. Other type expressions are reified homomorphically. The application of a coercion to a term and the application of a term to a coercion are both reified as applications.

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

The translation uses an auxiliary predicate dom that computes the domain of a coercion. 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. The domain of a coercion G in environment Γ is the unique type τ such that $\Gamma \vdash G : \tau \rhd \sigma$ for some type σ.

Alternatively, we could define the translation as a translation of typing derivations. Indeed, we need such a translation to show that translation is well-typed:

Proposition 12 (Well-typedness)**.** *The following assertions hold:*

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

Proof. The translation of typing derivations can be easily deduced from Figure 17 since we are explicitly typed. To prove each assertion we proceed by induction on its judgment. For each typing rule we just verify that the translated derivation is valid in System F using induction hypothesis. □

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

Lemma 13 (Forward simulation)**.** *If* $\Gamma \vdash M : \tau$ *holds, then:*

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

(Proof p. 39)

Corollary 14 (Termination)**.** *Reduction in* F_ι *is terminating.*

Proof. Assume that M is well-typed and has a non-terminating reduction path. Then $\lceil M \rceil$ is well-typed in System F using Lemma 12. We can build a non-terminating reduction path for $\lceil M \rceil$ in System F using Lemma 13. However this violates System F strong normalization result, hence the result. □

Reification could be enriched to decorate reified expressions so as to distinguish in System F between coercions and terms, *i.e.* between parts that could be dropped and those that should be kept when erasing to λ-calculus. Actually, our first presentation of F_ι was just such a decorated version of System F. However, type-checking decorated terms requires coercions to bind, as described in §7, which introduces more complex machinery. By keeping coercions as proof terms, we can avoid this in F_ι. However, as a counter-part, coercions are less intuitive—and reification should be kept in mind to understand the meaning of coercions and their reduction rules.

3.4 Confluence

An analysis of reduction rules in F_ι shows that there is no critical pairs. This is because F_ι does not reduce coercions alone, but only when they are applied to terms. In particular $[] \mathbin{@} M$ is not an evaluation context. If we allowed such reduction (and indeed simplifications inside coercions could occur), then there would be critical pairs to be considered for proving local confluence.

In the absence of critical pairs, reduction is locally confluent, and as it is terminating, it is also confluent.

Corollary 15 (Confluence)**.** *Reduction in F_ι is confluent.*

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

Hence, the reduction in F_ι is locally confluent. Because it is terminating (Lemma 14), it is confluent (Newman's lemma). □

3.5 Forward simulation

We prove the forward simulation for F_ι by case on the reduction rule.

Lemma 16 (Forward simulation)**.** *If $\Gamma \vdash M : \tau$ holds, then:*

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

Proof. The proof is by simple computation. □

Unfortunately, the backward simulation does not hold. The wedge $\lambda(c : \tau \to \tau \rhd \tau \to \tau)\, \lambda(y : \tau)\ (c \mathbin{@} (\lambda(x : \tau)\ x))\ y$ 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, one adjust the definition of the language so that wedging configuration cannot appear in an evaluation context. This observation invites to three different solutions.

4 Weak F_ι

Weak F_ι is defined so that coercion variables cannot appear under evaluation contexts. At first, it seems to suffice to use weak reduction on coercion abstraction. Indeed, if a coercion variable cannot appear under an evaluation context, it cannot appear in a wedging configuration. However, since

$$
\begin{array}{rcll}
W & ::= & \ldots \not| \ \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \not| \ \lambda(c:\tau\rhd\tau)\,M \mid \lambda(c:\tau\rhd\tau)\,u & \text{expressions} \\
v & ::= & \lambda(x:\tau)\ M \mid \lambda\alpha\, v \mid \lambda(c:\tau\rhd\tau)\,u \mid (v,v) \mid \mathtt{Top}^{\tau} \mathbin{@} v & \text{values} \\
u & ::= & v \mid G \mathbin{@} u & \text{value forms} \\
E & ::= & []\ M \mid M\ [] \mid ([],M) \mid (M,[]) \mid []\mathtt{.1} \mid []\mathtt{.2} & \text{evaluation contexts} \\
 & \mid & \lambda\alpha\,[] \mid []\,\tau \mid G \mathbin{@} [] \mid [] \mathbin{\$} G & \\
 & & \textsc{RedTransCoerLam} & \\
 & & (\lambda(c:\tau\rhd\sigma)\,G) \mathbin{@} u \rightsquigarrow_\iota \lambda(c:\tau\rhd\sigma)\,G \mathbin{@} u &
\end{array}
$$

Figure 18: Weak F_ι: syntax and semantics *wrt* F_ι

$\lambda(c:\tau\rhd\sigma)\,M$ 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, $\lambda(c:\tau\rhd\sigma)\,M$ would only be allowed when M is fully evaluated, which would considerably limit the interest of abstracting over c.

Therefore, we choose a weak strategy for both coercions and terms. Keeping strong reduction on types is optional and independent.

Syntactic restrictions The syntax of Weak F_ι is defined on Figure 18 as a restriction of the syntax of F_ι (we write $\not|$ for removal of a previous grammar form). We remove distributivity of coercion abstraction on term abstraction $\mathrm{Dist}^{\tau\to(\rho\rhd\rho')\Rightarrow\sigma}$ in order to preserve the value restriction during reduction. We replace $\lambda(c:\tau\rhd\tau)\,M$ in terms by $\lambda(c:\tau\rhd\tau)\,u$ where u are value form. A *value form* is a term that erases to a value, *i.e.* a value or an application of a coercion G to a value form. A value is any form of abstraction whose subterm is an arbitrary term for a term abstraction, a value for a type abstraction (because we may evaluate under type abstractions), or a value form for a coercion abstraction.

The static semantics of Weak F_ι and F_ι are the same.

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

$$
\begin{array}{l}
\textsc{RedTransCoerLam} \\
(\lambda(c:\tau\rhd\sigma)\,G) \mathbin{@} M \rightsquigarrow_\iota \lambda(c:\tau\rhd\sigma)\,G \mathbin{@} M
\end{array}
$$

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

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

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

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

Lemma 17 (Value restriction preservation)**.** *If M is syntactically correct and $M \rightsquigarrow_{\beta\iota} N$ holds, then N is syntactically correct.*

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

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

□

Lemma 18. *If $\Gamma \vdash M : \tau$, $\Gamma \vdash G : \tau \rhd \sigma$, $\Gamma \vdash \tau$, $\Gamma \vdash ok$, or $M \rightsquigarrow_{\beta\iota} N$ holds in Weak F_ι, then it also holds in F_ι.*

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

Lemma 19. *If $\Gamma \vdash M : \tau$ holds in F_ι and M is syntactically correct in Weak F_ι, then $\Gamma \vdash M : \tau$ holds in Weak F_ι.*

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

Proposition 20 (Preservation)**.** *If $\Gamma \vdash M : \tau$ and $M \rightsquigarrow_{\beta\iota} N$ hold, then $\Gamma \vdash N : \tau$ holds.*

Proof. Using Lemma 18, we can call Lemma 9 in F_ι, and then use Lemma 19 to come back in Weak F_ι. □

Proposition 21 (Termination)**.** *Reduction in Weak F_ι is terminating.*

Proof. Assume we have an infinite reduction path starting from M. Then using Lemma 18, we build an infinite reduction path in F_ι. However it contradicts Corollary 14. □

Lemma 22 (Classification)**.** *If $\Gamma \vdash v : \tau$ holds, then:*

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

Proof. This holds with a simple induction on v. □

Lemma 23 (Progress)**.** *If $\overrightarrow{\alpha} \vdash M : \tau$ holds, then either M is a value or it reduces.*

(Proof p. 40)

Confluence in Weak F_ι must be proved separately because its statement uses reduction both in premises and conclusion. However, the absence of critical pairs in F_ι *is* preserved in weak F_ι, from which local confluence and confluence immediately follow.

Corollary 24 (Confluence)**.** *Reduction in Weak F_ι is confluent.*

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

Lemma 25 (Forward simulation)**.** *If $\Gamma \vdash M : \tau$ holds, then:*

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

Proof. We simply use Lemma 18 to call Lemma 16. □

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

Lemma 26 (Classification)**.** *If $\overrightarrow{\alpha} \vdash Q[\lambda(x : \rho)\ M] : \tau$ (resp. $\overrightarrow{\alpha} \vdash Q[(M, N)] : \tau$) holds and $Q[\lambda(x : \rho)\ M]$ (resp. $Q[(M, N)]$) is in ι-normal-form, then:*

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

Proof. By induction on Q.

- $[]$: Only the first case is possible by typing. And it has the right form.
- $\lambda\alpha\, Q'$: Only the second case is possible by typing. And it has the right form.
- $Q'\,\tau'$: By typing we have $\overrightarrow{\alpha} \vdash Q'[\lambda(x : \rho)\ M] : \forall\alpha.\rho'$ (resp. $\overrightarrow{\alpha} \vdash Q'[(M, N)] : \forall\alpha.\rho'$) such that $\rho'[\alpha \leftarrow \tau'] = \tau$. By induction hypothesis we have Q' of the form $\lambda\alpha\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedType applies.
- $G \mathbin{@} Q'$: By induction on G.
  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, and $M.2$: These are refused by typing, because they are terms instead of coercions.
  - $\lambda\alpha\, W$, $W\,\tau$, $G \mathbin{@} W$, $\lambda(c : \tau \rhd \tau)\, W$, and $W \mathbin{\$} G$: These are not in ι-normal-form, since RedTransCoer applies.
  - $\diamond^\tau$: It is not in ι-normal-form, since RedTransDot applies.
  - c: This is refused by typing, since we only have type variables in the environment.
  - $\mathtt{Top}^\tau$: No case apply.
  - $G \xrightarrow{\tau} G$: By typing we have $\overrightarrow{\alpha} \vdash Q'[\lambda(x : \rho)\ M] : \sigma \to \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι-normal-form, since RedTransArrow applies.
  - $\mathrm{Dist}^{\tau \to \forall\alpha.\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda\alpha\, \lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedTransDist applies.
  - $(G * G)$, $\mathrm{Dist}^{\forall\alpha.(\tau * \tau)}$, and $\mathrm{Dist}^{(\tau \rhd \tau) \Rightarrow (\tau * \tau)}$: No case apply.
  - $G \xrightarrow{\tau} G$ and $\mathrm{Dist}^{\tau \to \forall\alpha.\tau}$ (resp.): No case apply.
  - $(G * G)$ (resp.): By induction hypothesis, RedTransProduct applies.
  - $\mathrm{Dist}^{\forall\alpha.(\tau * \tau)}$ (resp.): By induction hypothesis used twice, RedTransDistTypeProduct applies.

  - $\mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)}$ (resp.): By induction hypothesis used twice, RedTransDistCoerProduct applies.

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

□

Lemma 27 (Backward simulation)**.** *If $\overrightarrow{\alpha}\vdash M:\tau$ and $\lfloor M\rfloor\rightsquigarrow a$, then $M\rightsquigarrow_\iota^\star\ \rightsquigarrow_\beta N$ such that $\lfloor N\rfloor=a$.*

Proof. We show that the ι-normal-form of M β-reduces to N with $\lfloor N\rfloor$ equal to a. Since F_ι strongly normalizes, we may assume *wlog* that M is already in ι-normal-form. Because $\lfloor M\rfloor$ reduces, we can use the reduction derivation to show that it must be of the form $e[(\lambda x.a_1)\,a_2]$. By inversion of the coercion-erasure function, we show that M is of the form $E[Q[\lambda(x:\tau)\ M_1]\ M_2]$ where E is an evaluation context (we can have neither term abstraction since we do not have them in the λ-calculus, nor coercion abstraction because we have an application node below, and there is no way to have an application node under a coercion abstraction without using a term abstraction) and Q a retyping context of arbitrary depth, such that E, M_1, and M_2 erase to e, a_1, and a_2 respectively. We show using Lemma 26 that if a ι-normal term of the form $Q[\lambda(x:\tau)\ M]$ has an arrow type, then Q is empty. Hence, M is of the form $E[(\lambda(x:\tau)\ M_1)\ M_2]$ and β-reduces to $E[M_1[x\leftarrow M_2]\,]$ whose erasure is $e[a_1[x\leftarrow a_2]\,]$. A similar proof holds for reduction of pairs. □

5 Parametric F_ι

Parametric F_ι, written F_ι^p, is another way of avoiding a coercion variable from appearing in a wedging position, in any context, hence allowing reduction under coercion abstractions. In Parametric F_ι, a coercion parameter must be parametric in either its domain or codomain. That is, we only allow coercion abstractions of the form $\lambda\alpha\,\lambda(c:\alpha\rhd\sigma)\,W$ or $\lambda\alpha\,\lambda(c:\sigma\rhd\alpha)\,W$ but not $\lambda(c:\tau\rhd\sigma)\,W$ alone—or even $\lambda(c:\sigma\rhd\alpha)\,W$ alone, since it could reduce to $\lambda(c:\sigma\rhd\tau)\,W$ by type application.

To enforce this restriction we need to glue construction together and thus modify the syntax of source terms. But we should really see Parametric F_ι as a restriction of F_ι.

5.1 Syntax changes

The syntax of Parametric F_ι is defined on Figure 19 as a modification of the syntax of F_ι. We replace coercion abstraction $\lambda(c:\tau\rhd\sigma)\,W$ of F_ι by two new constructs $\lambda(\alpha,c:\rhd\sigma)\,W$ and $\lambda(\alpha,c:\lhd\sigma)\,W$ to mean $\lambda\alpha\,\lambda(c:\alpha\rhd\sigma)\,W$ and $\lambda\alpha\,\lambda(c:\sigma\rhd\alpha)\,W$ but atomically. Note that the type variable α is bounded in both τ and W. As a mnemonic device, we can read the type of the coercion by erasing the comma, the coercion variable, and the colon: $(\alpha,c:\lhd\tau)$ becomes $\alpha\lhd\tau$ or $\tau\rhd\alpha$ with standard notation.

When parametricity is on the codomain, we say that we have a positive coercion abstraction; this implements the coercion abstraction of *x*MLF and represents a lower bounded quantification $\tau\rhd\alpha$. When the parametricity is on the domain, we say that the coercion abstraction is negative; this implements the coercion abstraction of $\mathsf{F}_{<:}$ and is an upper bounded quantification $\alpha\rhd\tau$.

Accordingly, we also replace coercion application $W\,\$\,G$ by $W\,(\tau\rhd G)$ and $W\,(\tau\lhd G)$ to perform type and coercion applications $(W\,\tau)\ \$\ G$ atomically. Both positive and negative versions have the same meaning, but different typing rules. Type τ appears before G to remind that the type application is performed before the coercion application in the expanded form. As a mnemonic device, the triangle is oriented towards the side of the variable it instantiates in the coercion type of W. So positive coercion application is written $W\,(\tau\lhd G)$ because its corresponding abstraction

$$
\begin{array}{rcll}
W & ::= & \ldots \not| \ \lambda(c:\tau \rhd \tau)\, W \not| \ W \,\$\, G \not| \ \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \not| \ \mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)} & \text{expressions} \\
 & & \mid \lambda(\alpha, c:\lhd\tau)\, W \mid W\,(\tau \lhd G) \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\lhd\tau).\tau} \mid \mathrm{Dist}^{\forall(\alpha,\lhd\tau).(\tau*\tau)} & \\
 & & \mid \lambda(\alpha, c:\rhd\tau)\, W \mid W\,(\tau \rhd G) \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\rhd\tau).\tau} \mid \mathrm{Dist}^{\forall(\alpha,\rhd\tau).(\tau*\tau)} & \\
\tau & ::= & \ldots \not| \ (\tau \rhd \tau) \Rightarrow \tau \mid \forall(\alpha, \lhd\tau).\tau \mid \forall(\alpha, \rhd\tau).\tau & \text{types} \\
B & ::= & \ldots \not| \ (c:\tau \rhd \tau) \mid (\alpha, c:\lhd\tau) \mid (\alpha, c:\rhd\tau) & \text{bindings} \\
p & ::= & \ldots \not| \ p \,\$\, G \mid p\,(\tau \lhd G) \mid p\,(\tau \rhd G) & \text{prevalues} \\
 & & \not| \ \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \ @\ p \not| \ \mathrm{Dist}^{\tau\to(\tau\rhd\tau)\Rightarrow\tau} \ @\ (\lambda(c:\tau\rhd\tau)\, p) & \\
 & & \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\lhd\tau).\tau} \ @\ p \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\lhd\tau).\tau} \ @\ (\lambda(\alpha, c:\lhd\tau)\, p) & \\
 & & \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\rhd\tau).\tau} \ @\ p \mid \mathrm{Dist}^{\tau\to\forall(\alpha,\rhd\tau).\tau} \ @\ (\lambda(\alpha, c:\rhd\tau)\, p) & \\
 & & \not| \ \mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)} \ @\ p \not| \ \mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)} \ @\ (\lambda(c:\tau\rhd\tau)\, p) & \\
 & & \mid \mathrm{Dist}^{\forall(\alpha,\lhd\tau).(\tau*\tau)} \ @\ p \mid \mathrm{Dist}^{\forall(\alpha,\lhd\tau).(\tau*\tau)} \ @\ (\lambda(\alpha, c:\lhd\tau)\, p) & \\
 & & \mid \mathrm{Dist}^{\forall(\alpha,\rhd\tau).(\tau*\tau)} \ @\ p \mid \mathrm{Dist}^{\forall(\alpha,\rhd\tau).(\tau*\tau)} \ @\ (\lambda(\alpha, c:\rhd\tau)\, p) & \\
v & ::= & \ldots \not| \ \lambda(c:\tau \rhd \tau)\, v \mid \lambda(\alpha, c:\lhd\tau)\, v \mid \lambda(\alpha, c:\rhd\tau)\, v & \text{values} \\
P & ::= & \ldots \not| \ \lambda(c:\tau \rhd \tau)\, [] \mid \lambda(\alpha, c:\lhd\tau)\, [] \mid \lambda(\alpha, c:\rhd\tau)\, [] & \text{retyping contexts} \\
 & & \not| \ [] \,\$\, G \mid [] \,(\tau \lhd G) \mid [] \,(\tau \rhd G) &
\end{array}
$$

Figure 19: Parametric F_ι: syntax restriction *wrt* F_ι

$$
\textsc{TermCoerLamPos}\ \frac{\Gamma, (\alpha, c:\lhd\tau) \vdash M : \sigma}{\Gamma \vdash \lambda(\alpha, c:\lhd\tau)\, M : \forall(\alpha, \lhd\tau).\sigma}
\qquad
\textsc{TermCoerAppPos}\ \frac{\Gamma \vdash M : \forall(\alpha, \lhd\tau).\sigma \qquad \Gamma \vdash \rho \qquad \Gamma \vdash G : \tau[\alpha \leftarrow \rho] \rhd \rho}{\Gamma \vdash M\,(\rho \lhd G) : \sigma[\alpha \leftarrow \rho]}
$$

$$
\textsc{TermCoerLamNeg}\ \frac{\Gamma, (\alpha, c:\rhd\tau) \vdash M : \sigma}{\Gamma \vdash \lambda(\alpha, c:\rhd\tau)\, M : \forall(\alpha, \rhd\tau).\sigma}
\qquad
\textsc{TermCoerAppNeg}\ \frac{\Gamma \vdash M : \forall(\alpha, \rhd\tau).\sigma \qquad \Gamma \vdash \rho \qquad \Gamma \vdash G : \rho \rhd \tau[\alpha \leftarrow \rho]}{\Gamma \vdash M\,(\rho \rhd G) : \sigma[\alpha \leftarrow \rho]}
$$

Figure 20: Parametric F_ι: typing rules *wrt* F_ι

is written $\lambda(\alpha, c : \lhd\tau)\, W$. We also replace the coercion abstraction distributivity constructs accordingly.

In the grammar of types, we must replace the coercion abstraction type $(\tau \rhd \sigma) \Rightarrow \rho$ of F_ι with two constructs $\forall(\alpha, \lhd\tau).\sigma$ and $\forall(\alpha, \rhd\tau).\sigma$ that glue it with a type abstraction, atomically. They describe terms parametric in a type variable α and abstracted over a coercion of type $\tau \rhd \alpha$ or $\alpha \rhd \tau$, respectively. As in expressions, α may appear in τ. Bindings are changed accordingly. We also update the prevalues, values, and retyping contexts.

5.2 Adjustments to the semantics

The syntactic changes made in F_ι, implies changes in the semantics as well. However, as Parametric F_ι can be considered a subset of F_ι, these changes are just an adjustment to the syntactic changes.

We add four typing rules for terms, given in Figure 20, each one describing a new construct of Parametric F_ι and derived from its expansion in F_ι. For example, TermCoerLamPos is just the combination of rules TermCoerLam and TermTypeLam in F_ι.

Similarly, we add four rules for typing coercions, given in Figure 21, each one describing a new form of coercion and derived from its expansion in F_ι. We also add two new rules for extracting joint bindings $(\alpha, c : \lhd\tau)$ and $(\alpha, c : \rhd\tau)$ introduced by the new type-coercion abstractions. And finally we add four rules about distributivity.

Well-formedness judgments are adjusted in the obvious way, as described on figures 22.

Finally, it remains to adapt the rules for the reduction of coercion abstractions. We replace RedCoer with two rules, defined on Figure 23 in the obvious way, as an atomic application

$$\textsc{CoerVarPos}\ \frac{\Gamma \vdash ok \qquad (\alpha, c : \lhd\tau) \in \Gamma}{\Gamma \vdash c : \tau \rhd \alpha}$$

$$\textsc{CoerCoerLamPos}\ \frac{\Gamma, (\alpha, c : \lhd\tau) \vdash G : \sigma \rhd \sigma' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \lambda(\alpha, c : \lhd\tau)\, G : \sigma \rhd \forall(\alpha, \lhd\tau).\sigma'}$$

$$\textsc{CoerCoerAppPos}\ \frac{\Gamma \vdash G : \sigma \rhd \forall(\alpha, \lhd\tau).\sigma' \qquad \Gamma \vdash \rho \qquad \Gamma \vdash G : \tau[\alpha \leftarrow \rho] \rhd \rho}{\Gamma \vdash G\,(\rho \lhd G) : \sigma \rhd \sigma'[\alpha \leftarrow \rho]}$$

$$\textsc{CoerVarNeg}\ \frac{\Gamma \vdash ok \qquad (\alpha, c : \rhd\tau) \in \Gamma}{\Gamma \vdash c : \alpha \rhd \tau}$$

$$\textsc{CoerCoerLamNeg}\ \frac{\Gamma, (\alpha, c : \rhd\tau) \vdash G : \sigma \rhd \sigma' \qquad \Gamma \vdash \sigma}{\Gamma \vdash \lambda(\alpha, c : \rhd\tau)\, G : \sigma \rhd \forall(\alpha, \rhd\tau).\sigma'}$$

$$\textsc{CoerCoerAppNeg}\ \frac{\Gamma \vdash G : \sigma \rhd \forall(\alpha, \rhd\tau).\sigma' \qquad \Gamma \vdash \rho \qquad \Gamma \vdash G : \rho \rhd \tau[\alpha \leftarrow \rho]}{\Gamma \vdash G\,(\rho \rhd G) : \sigma \rhd \sigma'[\alpha \leftarrow \rho]}$$

$$\textsc{CoerDistCoerPosArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\tau \to \forall(\alpha, \lhd\rho).\sigma} : \forall(\alpha, \lhd\rho).(\tau \to \sigma) \rhd \tau \to \forall(\alpha, \lhd\rho).\sigma}$$

$$\textsc{CoerDistCoerNegArrow}\ \frac{\Gamma \vdash \tau \qquad \Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\tau \to \forall(\alpha, \rhd\rho).\sigma} : \forall(\alpha, \rhd\rho).(\tau \to \sigma) \rhd \tau \to \forall(\alpha, \rhd\rho).\sigma}$$

$$\textsc{CoerDistCoerPosProduct}\ \frac{\Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\forall(\alpha, \lhd\rho).(\tau * \sigma)} : \forall(\alpha, \lhd\rho).(\tau * \sigma) \rhd ((\forall(\alpha, \lhd\rho).\tau) * (\forall(\alpha, \lhd\rho).\sigma))}$$

$$\textsc{CoerDistCoerNegProduct}\ \frac{\Gamma, \alpha \vdash \rho \qquad \Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \mathrm{Dist}^{\forall(\alpha, \rhd\rho).(\tau * \sigma)} : \forall(\alpha, \rhd\rho).(\tau * \sigma) \rhd ((\forall(\alpha, \rhd\rho).\tau) * (\forall(\alpha, \rhd\rho).\sigma))}$$

Figure 21: Parametric F_ι: coercion typing rules *wrt* F_ι

$$\textsc{TypeCoerArrowPos}\ \frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \forall(\alpha, \lhd\tau).\sigma}$$

$$\textsc{TypeCoerArrowNeg}\ \frac{\Gamma, \alpha \vdash \tau \qquad \Gamma, \alpha \vdash \sigma}{\Gamma \vdash \forall(\alpha, \rhd\tau).\sigma}$$

$$\textsc{EnvCoerPos}\ \frac{\Gamma, \alpha \vdash \tau \qquad \alpha, c \notin \mathrm{dom}(\Gamma)}{\Gamma, (\alpha, c : \lhd\tau) \vdash ok}$$

$$\textsc{EnvCoerNeg}\ \frac{\Gamma, \alpha \vdash \tau \qquad \alpha, c \notin \mathrm{dom}(\Gamma)}{\Gamma, (\alpha, c : \rhd\tau) \vdash ok}$$

Figure 22: Parametric F_ι: well-formedness judgments *wrt* F_ι

RedCoerPos
$(\lambda(\alpha, c : \lhd\tau)\, M)\,(\sigma \lhd G) \rightsquigarrow_\iota M[\alpha \leftarrow \sigma][c \leftarrow G]$

RedCoerNeg
$(\lambda(\alpha, c : \rhd\tau)\, M)\,(\sigma \rhd G) \rightsquigarrow_\iota M[\alpha \leftarrow \sigma][c \leftarrow G]$

RedTransDistCoerPosArrow
$\mathrm{Dist}^{\sigma_1 \to \forall(\alpha, \lhd\sigma_2).\sigma_3} \mathbin{@} (\lambda(\alpha, c : \lhd\rho)\, \lambda(x : \tau)\ M) \rightsquigarrow_\iota \lambda(x : \tau)\, \lambda(\alpha, c : \lhd\rho)\, M$

RedTransDistCoerNegArrow
$\mathrm{Dist}^{\sigma_1 \to \forall(\alpha, \rhd\sigma_2).\sigma_3} \mathbin{@} (\lambda(\alpha, c : \rhd\rho)\, \lambda(x : \tau)\ M) \rightsquigarrow_\iota \lambda(x : \tau)\, \lambda(\alpha, c : \rhd\rho)\, M$

RedTransDistCoerPosProduct
$\mathrm{Dist}^{\forall(\alpha, \lhd\sigma_1).(\sigma_2 * \sigma_3)} \mathbin{@} (\lambda(\alpha, c : \lhd\rho)\, (M, N)) \rightsquigarrow_\iota (\lambda(\alpha, c : \lhd\rho)\, M, \lambda(\alpha, c : \lhd\rho)\, N)$

RedTransDistCoerNegProduct
$\mathrm{Dist}^{\forall(\alpha, \rhd\sigma_1).(\sigma_2 * \sigma_3)} \mathbin{@} (\lambda(\alpha, c : \rhd\rho)\, (M, N)) \rightsquigarrow_\iota (\lambda(\alpha, c : \rhd\rho)\, M, \lambda(\alpha, c : \rhd\rho)\, N)$

Figure 23: Parametric F_ι: new reduction rules *wrt* F_ι

$$
\begin{array}{rl}
(\lambda(\alpha, c : \lhd\tau)\, W)^\circ &= \lambda\alpha\, \lambda(c : \tau^\circ \rhd \alpha)\, W^\circ \\
(\lambda(\alpha, c : \rhd\tau)\, W)^\circ &= \lambda\alpha\, \lambda(c : \alpha \rhd \tau^\circ)\, W^\circ \\
(W\ (\rho \lhd G))^\circ &= W^\circ\ \rho\ \$\ G^\circ \\
(W\ (\rho \rhd G))^\circ &= W^\circ\ \rho\ \$\ G^\circ \\
(\mathrm{Dist}^{\tau \to \forall(\alpha, \lhd\rho).\sigma})^\circ &= \mathrm{Dist}^{\tau^\circ \to \forall\alpha.(\rho^\circ \rhd \alpha) \Rightarrow \sigma^\circ} \mathbin{@} (\lambda\alpha\, \mathrm{Dist}^{\tau^\circ \to (\rho^\circ \rhd \alpha) \Rightarrow \sigma^\circ} \mathbin{@} ({\diamond}^{\forall\alpha.(\rho^\circ \rhd \alpha) \Rightarrow \tau^\circ \to \sigma^\circ}\ \alpha)) \\
(\mathrm{Dist}^{\tau \to \forall(\alpha, \rhd\rho).\sigma})^\circ &= \mathrm{Dist}^{\tau^\circ \to \forall\alpha.(\alpha \rhd \rho^\circ) \Rightarrow \sigma^\circ} \mathbin{@} (\lambda\alpha\, \mathrm{Dist}^{\tau^\circ \to (\alpha \rhd \rho^\circ) \Rightarrow \sigma^\circ} \mathbin{@} ({\diamond}^{\forall\alpha.(\alpha \rhd \rho^\circ) \Rightarrow \tau^\circ \to \sigma^\circ}\ \alpha)) \\
(\mathrm{Dist}^{\forall(\alpha, \lhd\rho).(\tau * \sigma)})^\circ &= \mathrm{Dist}^{\forall\alpha.((\rho^\circ \rhd \alpha) \Rightarrow \tau^\circ * (\rho^\circ \rhd \alpha) \Rightarrow \sigma^\circ)} \mathbin{@} (\lambda\alpha\, \mathrm{Dist}^{(\rho^\circ \rhd \alpha) \Rightarrow (\tau^\circ * \sigma^\circ)} \mathbin{@} ({\diamond}^{\forall\alpha.(\rho^\circ \rhd \alpha) \Rightarrow (\tau^\circ * \sigma^\circ)}\ \alpha)) \\
(\mathrm{Dist}^{\forall(\alpha, \rhd\rho).(\tau * \sigma)})^\circ &= \mathrm{Dist}^{\forall\alpha.((\alpha \rhd \rho^\circ) \Rightarrow \tau^\circ * (\alpha \rhd \rho^\circ) \Rightarrow \sigma^\circ)} \mathbin{@} (\lambda\alpha\, \mathrm{Dist}^{(\alpha \rhd \rho^\circ) \Rightarrow (\tau^\circ * \sigma^\circ)} \mathbin{@} ({\diamond}^{\forall\alpha.(\alpha \rhd \rho^\circ) \Rightarrow (\tau^\circ * \sigma^\circ)}\ \alpha))
\end{array}
$$

$$
\begin{array}{ll}
(\forall(\alpha, \lhd\tau).\sigma)^\circ = \forall\alpha.(\tau^\circ \rhd \alpha) \Rightarrow \sigma^\circ & (\Gamma, (\alpha, c : \lhd\tau))^\circ = \Gamma^\circ, \alpha, (c : \tau^\circ \rhd \alpha) \\
(\forall(\alpha, \rhd\tau).\sigma)^\circ = \forall\alpha.(\alpha \rhd \tau^\circ) \Rightarrow \sigma^\circ & (\Gamma, (\alpha, c : \rhd\tau))^\circ = \Gamma^\circ, \alpha, (c : \alpha \rhd \tau^\circ)
\end{array}
$$

Figure 24: Traduction of F_ι^p into F_ι

of RedType and RedCoer. We also adapt the two reduction rules about coercion abstraction distributivity, namely rules RedTransDistCoerArrow and RedTransDistCoerProduct, by adding four new reduction rules.

5.3 Properties

Considering Parametric F_ι as a restriction of F_ι where coercion abstraction is always preceded by a type abstraction, it suffices to observe that Parametric F_ι is syntactically closed by reduction to deduce that normalization and subject reduction properties are both preserved.

Definition 28 (F_ι^p to F_ι traduction)**.** *We define a traduction from F_ι^p to F_ι witnessing the language inclusion on Figure 24. We write M° the traduction of M. We only give the traductions which do not simply reuse the same construct by calling recursively the traduction function on subtrees.*

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

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

Lemma 30. *The following assertions hold.*

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

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

Proof. Using Lemma 29 and Lemma 30, we build $\Gamma^\circ \vdash M^\circ : \tau^\circ$ and $M^\circ \rightsquigarrow_{\beta\iota}^+ N^\circ$. Using Lemma 9, potentially several times, in F_ι, we build $\Gamma^\circ \vdash N^\circ : \tau^\circ$. Finally, we use Lemma 29 to show $\Gamma \vdash N : \tau$. □

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

Proof. Assume we have an infinite reduction path starting from M. Then using Lemma 29 and Lemma 30, we build an infinite reduction path in F_ι. However it contradicts Corollary 14. □

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 is F_ι), so confluence is still preserved. Progress is a proof on its own, but the proof is very similar to the one in F_ι.

Corollary 33 (Confluence)**.** *Reduction in* F_ι^p *is confluent.*

Proof. There is no critical pairs in F_ι^p. Thus the reduction in F_ι^p is locally confluent. Because it is terminating, it is confluent (Newman). □

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

1. *If* τ *is of the form* $\tau \to \tau$*, then* v *is of the form* $\lambda(x : \tau)\, v$*.*
2. *If* τ *is of the form* $(\tau * \tau)$*, then* v *is of the form* (v, v)*.*
3. *If* τ *is of the form* $\forall\alpha.\tau$*, then* v *is of the form* $\lambda\alpha\, v$*.*
4. *If* τ *is of the form* $\forall(\alpha, \lhd\tau).\tau$*, then* v *is of the form* $\lambda(\alpha, c : \lhd\tau)\, v$*.*
5. *If* τ *is of the form* $\forall(\alpha, \rhd\tau).\tau$*, then* v *is of the form* $\lambda(\alpha, c : \rhd\tau)\, v$*.*
6. *If* τ *is of the form* $\top$*, then* v *is of the form* $\mathtt{Top}^\tau \mathbin{@} v$*.*

Proof. This holds with a simple induction on v. □

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

(Proof p. 41)

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

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

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

Proof. We simply use Lemma 29 and Lemma 30 to call Lemma 16. □

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

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

4. *If* τ *is* $\forall(\alpha, \rhd\sigma).\tau'$ *then* Q *is* $\lambda(\alpha, c : \rhd\sigma)\, Q'$.
5. *For all* $(\alpha, c : \rhd\sigma)$ *in* Γ, τ *is not* α.

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

- $[]$: Conditions 2 to 4 do not apply. Conditions 1 and 5 hold trivially.
- $\lambda\alpha\, Q'$: Conditions 2 and 5 hold trivially. Other conditions do not apply.
- $Q'\ \tau'$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \forall\alpha.\rho'$ such that $\rho'[\alpha \leftarrow \tau'] = \tau$. By induction hypothesis we have Q' of the form $\lambda\alpha\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedType applies.
- $G \mathbin{@} Q'$: By induction on G.
  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, and $M.2$: These are refused by typing, because they are terms instead of coercions.
  - $\lambda\alpha\, W$, $W\ \tau$, $G \mathbin{@} W$, $\lambda(\alpha, c : \lhd\tau)\, W$, $\lambda(\alpha, c : \rhd\tau)\, W$, $W\ (\tau \lhd G)$ and $W\ (\tau \rhd G)$: These are not in ι-normal-form, since RedTransCoer applies.
  - $\diamond^\tau$: It is not in ι-normal-form, since RedTransDot applies.
  - c when $(\alpha, c : \lhd\tau)$: Conditions 1 to 4 do not apply. And 5 holds because α cannot be bounded twice in Γ and it is already present with $(\alpha, c : \lhd\tau)$.
  - c when $(\alpha, c : \rhd\tau)$: This case is rejected by induction hypothesis, since we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \alpha$ with $(\alpha, c : \rhd\tau) \in \Gamma$.
  - $G \xrightarrow{\tau} G$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \sigma \to \sigma'$. By induction hypothesis we have that Q' is empty, which contradicts the fact that we were in ι-normal-form, since RedTransArrow applies.
  - $\mathrm{Dist}^{\tau \to \forall\alpha.\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda\alpha\, \lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedTransDist applies.
  - $\mathrm{Dist}^{\tau \to \forall(\alpha, \lhd\tau).\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda(\alpha, c : \lhd\sigma)\, \lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedTransDistCoerPosArrow applies.
  - $\mathrm{Dist}^{\tau \to \forall(\alpha, \rhd\tau).\tau}$: By typing and induction hypothesis used twice, we have that Q' is $\lambda(\alpha, c : \rhd\sigma)\, \lambda(x : \rho)\ M$, which contradicts the fact that we were in ι-normal-form, since RedTransDistCoerNegArrow applies.
  - $\mathtt{Top}^\tau$, $(G * G)$, $\mathrm{Dist}^{\forall\alpha.(\tau * \tau)}$, $\mathrm{Dist}^{\forall(\alpha, \lhd\tau).(\tau * \tau)}$, and $\mathrm{Dist}^{\forall(\alpha, \rhd\tau).(\tau * \tau)}$: Conditions 1 to 4 do not apply, and condition 5 trivially holds.
- $\lambda(\alpha, c : \lhd\sigma)\, Q'$: Conditions 3 and 5 hold trivially. Other conditions do not apply.
- $\lambda(\alpha, c : \rhd\sigma)\, Q'$: Conditions 4 and 5 hold trivially. Other conditions do not apply.
- $Q'\ (\sigma' \lhd G)$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \forall(\alpha, \lhd\sigma).\tau$. By induction hypothesis we have Q' of the form $\lambda(\alpha, c : \lhd\sigma)\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedCoerPos applies.
- $Q'\ (\sigma' \rhd G)$: By typing we have $\Gamma \vdash Q'[\lambda(x : \rho)\ M] : \forall(\alpha, \rhd\sigma).\tau$. By induction hypothesis we have Q' of the form $\lambda(\alpha, c : \rhd\sigma)\, Q''$, which contradicts the fact that we were in ι-normal-form, since RedCoerNeg applies.

□

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

$$\textsc{Kernel-Fsub}\quad \frac{A, \alpha <: T \vdash S <: S'}{A \vdash \forall(\alpha <: T)\ S <: \forall(\alpha <: T)\ S'} \qquad \textsc{Full-Fsub}\quad \frac{A \vdash T' <: T \qquad A, \alpha <: T' \vdash S <: S'}{A \vdash \forall(\alpha <: T)\ S <: \forall(\alpha <: T')\ S'}$$

$$\textsc{F-Bounded}\quad \frac{A, \alpha <: T' \vdash \alpha <: T \qquad A, \alpha <: T' \vdash S <: S'}{A \vdash \forall(\alpha <: T)\ S <: \forall(\alpha <: T')\ S'}$$

Figure 25: Bounded polymorphism: variants on the subtyping rule

Proof. The proof schema is not original [Manzonetto and Tranquilli, 2010]. We show that the ι-normal-form of M β-reduces to N with $\lfloor N \rfloor$ equal to a. Since F_ι strongly normalizes, we may assume *wlog* that M is already in ι-normal-form. Because $\lfloor M \rfloor$ reduces, we can use the reduction derivation to show that it must be of the form $e[(\lambda x.a_1)\, a_2]$. By inversion of the coercion-erasure function, we show that M is of the form $E[Q[\lambda(x : \tau)\ M_1]\ M_2]$ where E is an evaluation context and Q a retyping context of arbitrary depth, such that E, M_1, and M_2 erase to e, a_1, and a_2 respectively. We show using Lemma 37 that if a ι-normal term of the form $Q[\lambda(x : \tau)\ M]$ has an arrow type, then Q is empty. Hence, M is of the form $E[(\lambda(x : \tau)\ M_1)\ M_2]$ and β-reduces to $E[M_1[x \leftarrow M_2]\ ]$ whose erasure is $e[a_1[x \leftarrow a_2]\,]$. A similar proof holds for pairs instead of arrows. □

6 Expressiveness of Parametric F_ι

Despite its by-design limitation, Parametric F_ι is already a quite interesting spot in the design space, as it subsumes in the same 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]). We show and discuss these inclusions in the rest of this section.

To show the inclusion of a typed language in F_ι, we exhibit a translation of typing judgments from the source language to typing judgments of F_ι so that the coercion erasure of the translation of a source term m is equal to the type erasure of m, which ensures that the translation is semantics preserving.

To avoid confusion between source and target, we write m or n for terms, T or S for types, and A for typing environments, in the source language. Formally, we exhibit a translation of judgments $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ that is well-defined and type and semantics preserving. That is, if $A \vdash m : T$ then $A \vdash m : T \rightsquigarrow \Gamma \vdash M : \tau$ holds for some Γ, M, and τ such that $\Gamma \vdash M : \tau$ and $\lfloor m \rfloor = \lfloor M \rfloor$. Moreover, since for each source language, reduction will be simulated in F_ι, it will necessarily terminate.

By construction, F_η is included (and simulated) in Parametric F_ι. We show that $x\mathsf{MLF}$ and $\mathsf{F}_{<:}$ are also included and simulated into Parametric F_ι. These inclusions are strict, indeed.

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 $\forall(\alpha <: T)\ S$ restricts types T' that α range over to be subtypes of the bound T. The differences lie in when the subtyping judgment $A \vdash \forall(\alpha <: T)\ S <: \forall(\alpha <: T')\ S'$ holds. Different versions of the corresponding subtyping rule are given on Figure 25. In Kernel $\mathsf{F}_{<:}$, the bounds T and T' must be equal, whereas Full $\mathsf{F}_{<:}$ does only requires the bound T' to be a subtype of the bound T. Moreover, α cannot appear free in the bounds T or T' in Kernel or Full $\mathsf{F}_{<:}$, while $\mathsf{F}_{\mu<:}$ allows this form of recursion. The most general assumption, $\Gamma, \alpha <: T' \vdash \alpha <: T$, is that of $\mathsf{F}_{\mu<:}$. Perhaps surprisingly, this is a slightly more general rule [Baldan et al., 1999] than the more intuitive one $\Gamma, \alpha <: T' \vdash T' <: T$. In summary, we have Kernel $\mathsf{F}_{<:} \subset$ Full $\mathsf{F}_{<:} \subset \mathsf{F}_{\mu<:}$ where all inclusions are strict.

$$
\begin{array}{rcll}
W & ::= & \ldots \not| \; G \xrightarrow{\tau} G \not| \; \mathrm{Dist}^{\tau\to\forall\alpha.\tau} \not| \; \mathrm{Dist}^{\tau\to\forall(\alpha,\lhd\tau).\tau} \not| \; \mathrm{Dist}^{\tau\to\forall(\alpha,\rhd\tau).\tau} & \text{expressions} \\
& & \not| \; (G * G) \not| \; \mathrm{Dist}^{\forall\alpha.(\tau*\tau)} \not| \; \mathrm{Dist}^{\forall(\alpha,\lhd\tau).(\tau*\tau)} \not| \; \mathrm{Dist}^{\forall(\alpha,\rhd\tau).(\tau*\tau)} & \\
& & \not| \; \lambda(\alpha, c : \rhd\tau)\, W \not| \; W\,(\tau \rhd G) & \\
\tau & ::= & \ldots \not| \; \forall(\alpha, \rhd\tau).\tau & \text{types}
\end{array}
$$

Figure 26: $\mathsf{F}_\iota^{\mu x}$: syntax and notations

We show that the most expressive version $\mathsf{F}_{\mu<:}$ is also included into F_ι^p. The translation of typing judgments uses an auxiliary translation of subtyping judgments $A \vdash T <: S \rightsquigarrow \Gamma \vdash G : \tau \rhd \sigma$ of which the most interesting case is for bounded quantification:

$$
\frac{\begin{array}{c} A, \alpha <: T' \vdash \alpha <: T \rightsquigarrow \Gamma, (\alpha, c : \rhd\tau') \vdash G : \alpha \rhd \tau \\ A, \alpha <: T' \vdash S <: S' \rightsquigarrow \Gamma, (\alpha, c : \rhd\tau') \vdash G' : \sigma \rhd \sigma' \end{array}}{\begin{array}{c} A \vdash \forall(\alpha <: T)\; S <: \forall(\alpha <: T')\; S' \rightsquigarrow \\ \Gamma \vdash \lambda(\alpha, c : \rhd\tau')\, G' \mathbin{@} (\diamond\,(\alpha \rhd G)) : \forall(\alpha, \rhd\tau).\sigma \rhd \forall(\alpha, \rhd\tau').\sigma' \end{array}}
$$

The translation of bounded polymorphism $\forall(\alpha <: T)\; S$ is a negative coercion abstraction $\forall(\alpha, \rhd\tau).\sigma$ which encodes upper bounds. (Positive coercion abstractions $\forall(\alpha, \lhd\tau).\sigma$ encode lower bounds and are never needed in the translation of $\mathsf{F}_{\mu<:}$.)

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

It is remarkable that the coercion approach of F_ι^p naturally matches the most expressive version $\mathsf{F}_{\mu<:}$.

Additionally, F_ι^p could provide a simpler proof of type soundness for $\mathsf{F}_{\mu<:}$, as coercions are explicit.

Instance bounded polymorphism. The language $x\mathsf{MLF}$ [Rémy and Yakobowski, 2010] has been designed as the internal language of MLF. The language MLF is itself an extension of System F with *instance bounded polymorphism*, which allows to delay type instantiation and have principal types—given optional type annotations of function parameters. This is a key to type inference in MLF. However, our concern is not type inference but expressiveness, so we use $x\mathsf{MLF}$ for comparison. Moreover, by lack of space, we cannot formally present $x\mathsf{MLF}$. Instead, we identify a subset F_ι^x of F_ι^p and explains how it closely relates to $x\mathsf{MLF}$ without giving all the details of $x\mathsf{MLF}$.

We first define the language $\mathsf{F}_\iota^{\mu x}$ on Figure 26 by removing negative coercion abstraction along with its application, and everything about η-expansion (congruence and distributivity constructs about arrow and product) from Parametric F_ι. Accordingly, we also remove the negative coercion abstraction type, and typing and reduction rules for these constructs.

We then define F_ι^x as the restriction of $\mathsf{F}_\iota^{\mu x}$ where a type variable cannot appear in their instance bound, *i.e.* α is not free in τ in $\forall(\alpha, \lhd\tau).\sigma$. The language $x\mathsf{MLF}$ is equivalent to F_ι^x. The proof for the direct inclusion is similar to the one in [Manzonetto and Tranquilli, 2010]. The proof for the reverse inclusion is new but not much more difficult. In summary, we have $x\mathsf{MLF} \equiv \mathsf{F}_\iota^x \subset \mathsf{F}_\iota^{\mu x} \subset \mathsf{F}_\iota^p$. It is interesting that the natural restriction of F_ι that resembles $x\mathsf{MLF}$ allows variables to appear in their instance bounds, much as with F-bounded polymorphism, which suggest an extension to $x\mathsf{MLF}$ with recursively defined bounds. We do not know whether this extension can be easily transported back to its type inference version in MLF.

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

Summary Features of F_ι^p and its variants are summed up on Figure 27: deep type abstraction corresponds to the $\lambda\alpha\, G$ and $G\,\tau$ constructs; positive and negative coercion abstraction correspond

Feature	F_η	*x*MLF	$\mathsf{F}_{<:}$	F_ι^p
Deep type abstraction	y	y	-	y
Positive coercion abstraction	-	y	-	y
Negative coercion abstraction	-	-	y	y
Arrow congruence	y	-	y	y
Distributivity	y	-	-	y

Figure 27: Language comparison

to the $\lambda(\alpha, c : \lhd\tau)\, G$ and $G\,(\tau \lhd G)$ constructs, and $\lambda(\alpha, c : \rhd\tau)\, G$ and $G\,(\tau \rhd G)$ constructs, respectively; arrow congruence is the $G \xrightarrow{\tau} G$ construct; and distributivity is $\mathrm{Dist}^{\tau\to\forall\alpha.\tau}$.

Notice that *x*MLF and $\mathsf{F}_{<:}$ features are disjoint, while every other pair has a non-empty intersection. The expressiveness of F_η, *x*MLF, and $\mathsf{F}_{<:}$ can be compared by checking which feature is present in one language and not the other.

7 Towards a more general setting?

Even though F_ι^p subsumes *x*MLF, F_η, and $\mathsf{F}_{<:}$, the restrictions of F_ι to either parametric polymorphism or weak evaluation strategies are not quite satisfactory. First, they are incomparable. Can we find a more general setting that contains them both? Additionally, there are concrete examples, *e.g.* in FC_2, where coercion variables are used between function types, which cannot currently be written in either restriction. We also prefer a language defined with strong reduction because it helps understanding the language and reinforces our belief that the language is well-defined and not ad-hoc.

An interesting but not easy way of solving this problem is to add projectors in the language to decompose coercions. This is the solution adopted in FC_2, but in a simpler setting than F_ι. In the rest of the section we explain this idea and discuss difficult problems that it raises in the general setting of F_ι, both for erasability and soundness.

Decomposing coercions The following wedge $(c \mathbin{@} (\lambda(x : \tau)\ M))\ N$ is stuck in F_ι. The typing constraints imply that c has a coercion type of the form $\tau \to \sigma \rhd \tau' \to \sigma'$. Hence, c could for instance be instantiated by a concrete coercion $G_1 \to G_2$. Then, the coercion could be decomposed into G_1 and G_2 and reduction could pursue. Hence, the solution is to add coercion projectors, $\mathtt{Left}\, G$ and $\mathtt{Right}\, G$, that stand for the domain and codomain parts of G, as if G was equivalent to $\mathtt{Left}\, G \to \mathtt{Right}\, G$. Accordingly, coercion projectors are typed as follows:

$$\frac{\Gamma \vdash G : \tau \to \sigma \rhd \tau' \to \sigma'}{\Gamma \vdash \mathtt{Left}\, G : \tau' \rhd \tau}\ \textsc{CoerLeft} \qquad \frac{\Gamma \vdash G : \tau \to \sigma \rhd \tau' \to \sigma'}{\Gamma \vdash \mathtt{Right}\, G : \sigma \rhd \sigma'}\ \textsc{CoerRight}$$

We also add a reduction rule that mimics RedArrow on the left-hand side of an application:

$$(G \mathbin{@} (\lambda(x : \tau)\ M))\ N \rightsquigarrow_\iota (\lambda(x : \tau')\ (\mathtt{Right}\, G) \mathbin{@} M[x \leftarrow (\mathtt{Left}\, G) \mathbin{@} x])\ N \quad \textsc{RedPushArrow}$$

The type τ' must be computed from G: it is the argument type of the codomain of G. In other words, it is the type τ' such that G has the coercion type $\tau \to \sigma \rhd \tau' \to \sigma'$. Hence, reduction is non-local. This is a minor technical problem as it can always be awkwardly solved by annotating all coercions with their coercion types.

Introducing binding coercions However, this new reduction rule is not enough to recover erasability. The term $(c \mathbin{@} (\lambda\alpha\, \lambda(x:\tau)\ M))\ N$, which may be typed in a suitable typing environment because we can abstract on coercions of any type, is still stuck. Intuitively, the retyping context $\lambda\alpha\, []$ (but in general one of arbitrary depth), should be folded around c so that Rule RedPushArrow applies, as it retypes an arrow into another arrow. To avoid retyping contexts of arbitrary depth in the reduction rule, terms may be considered up to a structural equivalence relation $M \approx N$ that contains some of the ι-reduction rules encoding the explicit substitution of a term into a retyping context. Omitting type annotations, we would have:

EquivNil	EquivCons
$\diamond \mathbin{@} M \approx M$	$P[G] \mathbin{@} M \approx P[G \mathbin{@} M]$

However, EquivCons does not preserve scopes, because we have $\lambda\alpha\, M \approx (\lambda\alpha \diamond) \mathbin{@} M$ and α may appear in M. This can be solved by making coercions bind. The new coercion judgment becomes $\Gamma \vdash G : (\Delta, \tau) \rhd \sigma$ and reads "G is a coercion retyping a term of type τ in Γ, Δ, into a term of type σ in Γ". This can be seen in the following typing rule:

$$\frac{\Gamma \vdash G : (\Delta, \tau) \rhd \sigma \qquad \Gamma, \Delta \vdash M : \tau}{\Gamma \vdash G \mathbin{@} M : \sigma}$$

We now have the following judgment: $\Gamma \vdash \lambda\alpha \diamond^{\tau} : (\alpha, \tau) \rhd \forall\alpha.\tau$. Then, EquivCons preserves both scoping and typing.

Structural equivalence allows to see all wedging configurations in the form $(G@(\lambda(x:\tau)\ M))\ N$ so that RedPushArrow always applies to them. Then, the backward simulation trivially holds. However, the backward simulation in not much a guarantee in the absence of type soundness, confluence, or termination.

Loosing confluence? Unfortunately, the introduction of binding coercions forces us to revisit the typing rules for coercion projectors, but the fix is not obvious at all. Moreover, coercion projectors also introduce problems with confluence. Most problems in the combination of binding coercions and coercion projectors can already be seen on System F terms: for instance $(\lambda\alpha\, \lambda(x:\tau)\ M)\, \sigma\ N$ can be reduced with either RedType (as usual) or RedPushArrow as it is structurally equivalent to $(((\lambda\alpha \diamond)\, \sigma) \mathbin{@} (\lambda(x:\tau)\ M))\ N$. To have confluence, we should reduce coercions and not only terms, which means we must add the following evaluation context: $[] \mathbin{@} M$. But this is not obvious either, as reduction inside this context should do substitution inside M.

Consistency of coercions Assume that we have a system where we can abstract on any coercions without restrictions and where bisimulation holds. Then, reduction cannot be strongly normalizing! Indeed, we could abstract over two coercion variables of coercion types $\tau \rhd \tau \to \tau$ and $\tau \to \tau \rhd \tau$ and then write a term whose erasure is any given term of the λ-calculus—including ω defined as $(\lambda x.x\,x)\,(\lambda x.x\,x)$. The bisimulation implies that the source term whose erasure is ω does not terminate. This is actually unsurprising if we allow abstraction on coercions of any type without further restriction and allow reduction under coercion abstraction, because some coercion types are uninhabited. FC_2 uses a consistency condition to prevent abstraction on arbitrary coercions and to ensure progress. This condition requests that domain and codomain of a coercion always have the same head form. This is too restrictive however, as we cannot write any of $F_{<:}$ or xMLF coercion abstractions under this restriction. To be more flexible, a weaker notion of consistency could treat all coercions together and request that all typing environments are consistent in the sense that one can find at least one concrete instantiation for all its type variables and coercion variables.

Another track? The search for a more general setting for coercion abstraction seems a Pandora's box where any start of a solutions raises new harder problems. Although more investigation is still worthwhile, perhaps another less ambitious solution would be to search for a general form

of coercion abstractions including coercion projectors but in a restriction of F_ι, *i.e.* not containing all of F_η. This is sustained by the simpler definition of coercion projectors in FC_2 thanks to the absence of distributivity and deep instantiation.

8 Related work

Many extensions of type systems can be reexplained using coercions: inheritance, bounded quantification, static contracts, subtyping. We focus on works that make the connection with coercions explicit. There is also a regain of interest for non-erasable coercions. These are only loosely related works that we will not discuss here.

Although we have already widely discussed F_η, $\mathsf{F}_{<:}$, and $x\mathsf{MLF}$, parts of F_ι are closely related to the work of Manzonetto and Tranquilli [Manzonetto and Tranquilli, 2010] who proposed the first encoding of $x\mathsf{MLF}$ in a calculus of coercions, which is very close to F^x_ι, but introduced for the main purpose of proving the termination of $x\mathsf{MLF}$. They exhibit a type and semantics preserving encoding of $x\mathsf{MLF}$ into (their version of) F^x_ι and show a simulation of computation between their F^x_ι and System F. Unfortunately, subject reduction does not hold in their system, as well as some other properties that depend on it. Our version of F^x_ι can be seen as a fix to their definition. Hence, there are many resemblances between their development of F^x_ι and our development of F_ι—but the typing rules differ. We omitted the proof of inclusion from $x\mathsf{MLF}$ into F^x_ι by lack of space, but also because it resembles theirs. In fact, their translation of $x\mathsf{MLF}$ into F^x_ι is itself inspired by the translation of MLF into System F by Leijen and Löh [Leijen and Löh, 2005, Leijen, 2007]. However, Manzonetto and Tranquilli restrict their study to the termination of $x\mathsf{MLF}$ without any interest in F_η and $\mathsf{F}_{<:}$, while our main interest in F_ι is first a general treatment of coercions, and as a side result a possible enhancement of $x\mathsf{MLF}$.

Although F_ι subsumes core $\mathsf{F}_{<:}$, we have not included records in F_ι, which are the main application of $\mathsf{F}_{<:}$. In fact, our formalization includes tuples, and therefore models tuple inclusion, but we have omitted them in this presentation by lack of space. 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 that shows that the semantics of the translation does not actually depend on the places where coercions are inserted.

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

Coercions introduced in FC_2 [Weirich et al., 2011], the internal language of Haskell, are interesting because they cannot be expressed in F_ι and thus justify the search for a more general setting. Although FC_2 uses a weak evaluation strategy, it inserts abstract coercions at the toplevel and introduces coercion projectors to regain erasability. However, coercions in FC_2 do not have distributivity nor deep instantiation of quantifiers and are thus *structural*, which allows a simple criteria to be used for consistency checking. This could be inspiration for finding reasonable restrictions to our general setting—if we cannot solve it otherwise.

9 Conclusions

We have explored extensions of System F with general erasable coercions and abstraction over them. We have proposed a typed calculus F_ι that is confluent, terminating, and sound. However, abstraction over coercions must be restricted to ensure erasability. Two solutions are proposed: Weak F_ι prevents evaluation under coercion abstraction while Parametric F_ι prevents coercion variables to appear in the middle of redexes. Parametric F_ι unifies F_η, $\mathsf{F}_{<:}$, and $x\mathsf{MLF}$ and suggests other combinations of features.

Still, Parametric F_ι only permits a limited use of coercion abstraction. We have sketched a more general setting with coercion projectors to release the restriction on coercion abstraction. However, more difficult problems arise and we leave the study of this general case and of suitable restrictions for future work.

Several features of programming languages have also been left out of F_ι. Although products are not included in our presentation, by lack of space, we have already verified that they can be added in the obvious way. Hence, labeled products should work as well. We do not expect difficulties with tagged unions either or iso-recursive types, *e.g.* following Crary. We don't foresee any difficulties for adding fix points to the source language either.

However, special care is needed for existential types, as they already raise a problem in System F where they cannot have an erasing semantics with a strong evaluation strategy. This is the reason why we left them out of F_ι.

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

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

References

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

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

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

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

L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1-2):4–56, 1994. ISSN 0890-5401. URL `http://dx.doi.org/10.1006/inco.1994.1013`.

A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 213–224, Sept. 2008. URL `http://gallium.inria.fr/~fpottier/publis/chargueraud-pottier-capabilities.ps.gz`.

K. Crary. Typed compilation of inclusive subtyping. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP)*, pages 68–81, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. URL `http://doi.acm.org/10.1145/351240.351247`.

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

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

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

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

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

F. Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, Jan. 2011. Supplementary material.

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

N. Swamy, M. Hicks, and G. M. Bierman. A theory of typed coercions and its applications. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 329–340, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596598. URL `http://doi.acm.org/10.1145/1596550.1596598`.

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

A Delayed Proofs

Proof of Prop 11

This is a standard proof using Lemma 10. By induction on M. Using RedContextBeta and RedContextIota, we can only consider cases where subterms are values.

- x, $\lambda(x:\tau)\,v$, $\lambda\alpha\,v$, $\lambda(c:\tau\rhd\tau)\,v$, and (v,v): These are values.
- $\diamond^\tau$, $G \xrightarrow{\tau} G$, $\mathrm{Dist}^{\tau\to\forall\alpha.\sigma}$, $\mathrm{Dist}^{\tau\to(\rho\rhd\rho')\Rightarrow\sigma}$, $(G * G)$, $\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)}$, $\mathrm{Dist}^{(\rho\rhd\rho')\Rightarrow(\tau*\sigma)}$, $\mathtt{Top}^\tau$, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\ v$: Using Lemma 10 on v_1.
  - p: It is a value.
  - $\lambda(x:\tau)\,v$: RedTerm applies.

- $v\,\tau$: Using Lemma 10 on v.
  - p: It is a value.
  - $\lambda\alpha\, v$: RedType applies.
- $v\,\$\,G$: Using Lemma 10 on v.
  - p: It is a value.
  - $\lambda(c : \tau \rhd \tau)\, v$: RedCoer applies.
- $v.1$: Using Lemma 10 on v.
  - p: It is a value.
  - (v, v): RedFirst applies.
- $v.2$: Using Lemma 10 on v.
  - p: It is a value.
  - (v, v): RedSecond applies.
- $G \mathbin{@} v$: By induction on G.
  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, $M.2$: These expressions are rejected by typing because they are terms and not coercions.
  - c, $\mathtt{Top}^\tau$: These are values.
  - $\diamond^\tau$: RedDot applies.
  - $\lambda\alpha\, W$, $W\,\tau$, $\lambda(c : \tau \rhd \tau)\, W$, $W\,\$\,G$, and $G \mathbin{@} W$: RedTransCoer applies.
  - $G \xrightarrow{\tau} G$: Using Lemma 10 on v.
    - p: It is a value.
    - $\lambda(x : \tau)\, v$: RedTransArrow applies.
  - $\mathrm{Dist}^{\tau \to \forall\alpha.\tau}$: Using Lemma 10 on v.
    - p: It is a value.
    - $\lambda\alpha\, v$: Using Lemma 10 on v.
      - p: It is a value.
      - $\lambda(x : \tau)\, v$: RedTransDist applies.
  - $\mathrm{Dist}^{\tau \to (\tau \rhd \tau) \Rightarrow \tau}$: Using Lemma 10 on v.
    - p: It is a value.
    - $\lambda(c : \tau \rhd \tau)\, v$: Using Lemma 10 on v.
      - p: It is a value.
      - $\lambda(x : \tau)\, v$: RedTransDistCoerArrow applies.
  - $(G * G)$: Using Lemma 10 on v.
    - p: It is a value.
    - (v, v): RedTransProduct applies.
  - $\mathrm{Dist}^{\forall\alpha.(\tau * \tau)}$: Using Lemma 10 on v.
    - p: It is a value.
    - $\lambda\alpha\, v$: Using Lemma 10 on v.
      - p: It is a value.
      - (v, v): RedTransDistTypeProduct applies.
  - $\mathrm{Dist}^{(\tau \rhd \tau) \Rightarrow (\tau * \tau)}$: Using Lemma 10 on v.

- p: It is a value.
- $\lambda(c : \tau \rhd \tau)\, v$: Using Lemma 10 on v.
  - p: It is a value.
  - (v, v): RedTransDistCoerProduct applies.

■

Proof of Lem 13

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

1. By induction on $M \rightsquigarrow_\beta N$.
   - RedContextBeta: By case on the context. Evaluation contexts are reified on System F contexts, and because we are in strong reduction, all contexts are evaluation contexts.
   - RedTerm, RedFirst, and RedSecond: These rules were already in System F and are reified on themselves.
2. By induction on $M \rightsquigarrow_\iota N$.
   - RedContextIota: Same argument as for β-reduction.
   - RedType: This was already a rule of System F and is reified on itself.
   - RedTransArrow: We have
   $$\begin{aligned}
   &(\lambda(y : \lceil\tau'\rceil \to \lceil\sigma\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ (y\ (\lceil G_1\rceil\ x)))\ (\lambda(x : \lceil\tau'\rceil)\ \lceil M\rceil)\\
   \rightsquigarrow_\beta\ &\lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ ((\lambda(x : \lceil\tau'\rceil)\ \lceil M\rceil)\ (\lceil G_1\rceil\ x))\\
   \rightsquigarrow_\beta\ &\lambda(x : \lceil\tau\rceil)\ \lceil G_2\rceil\ M[x \leftarrow \lceil G_1\rceil\ x]
   \end{aligned}$$
   - RedTransDist: We have
   $$\begin{aligned}
   &(\lambda(y : \forall\alpha.\lceil\tau\rceil \to \lceil\sigma\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lambda\alpha\, y\, \alpha\ x)\ (\lambda\alpha\, \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\\
   \rightsquigarrow_\beta\ &\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\, (\lambda\alpha\, \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\, \alpha\ x\\
   \rightsquigarrow_\iota\ &\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\, (\lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x\\
   \rightsquigarrow_\beta\ &\lambda(x : \lceil\tau\rceil)\ \lambda\alpha\, \lceil M\rceil
   \end{aligned}$$
   - RedTransDistCoerArrow: We have
   $$\begin{aligned}
   &(\lambda(y : (\lceil\rho\rceil \to \lceil\rho'\rceil) \to \lceil\tau\rceil \to \lceil\sigma\rceil)\ \lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ y\ x_c\ x)\\
   &(\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\\
   \rightsquigarrow_\beta\ &\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x_c\ x\\
   \rightsquigarrow_\beta\ &\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ (\lambda(x : \lceil\tau\rceil)\ \lceil M\rceil)\ x\\
   \rightsquigarrow_\beta\ &\lambda(x : \tau)\ \lambda(x_c : \lceil\rho\rceil \to \lceil\rho'\rceil)\ \lceil M\rceil
   \end{aligned}$$
   - RedTransProduct: We have
   $$\begin{aligned}
   &(\lambda(y : (\lceil\tau\rceil * \lceil\sigma\rceil))\ (\lceil G_1\rceil\ y.\mathtt{1}, \lceil G_2\rceil\ y.\mathtt{2}))\ (\lceil M\rceil, \lceil N\rceil)\\
   \rightsquigarrow_\beta\ &(\lceil G_1\rceil\ (\lceil M\rceil, \lceil N\rceil).\mathtt{1}, \lceil G_2\rceil\ (\lceil M\rceil, \lceil N\rceil).\mathtt{2})\\
   \rightsquigarrow_\beta\ \rightsquigarrow_\beta\ &(\lceil G_1\rceil\ \lceil M\rceil, \lceil G_2\rceil\ \lceil N\rceil)
   \end{aligned}$$
   - RedTransDistTypeProduct: We have
   $$\begin{aligned}
   &(\lambda(y : \forall\alpha.(\tau * \sigma))\ (\lambda\alpha\, (y\, \alpha).\mathtt{1}, \lambda\alpha\, (y\, \alpha).\mathtt{2}))\ (\lambda\alpha\, (\lceil M\rceil, \lceil N\rceil))\\
   \rightsquigarrow_\beta\ &(\lambda\alpha\, ((\lambda\alpha\, (\lceil M\rceil, \lceil N\rceil))\, \alpha).\mathtt{1}, \lambda\alpha\, ((\lambda\alpha\, (\lceil M\rceil, \lceil N\rceil))\, \alpha).\mathtt{2})\\
   \rightsquigarrow_\iota\ \rightsquigarrow_\iota\ &(\lambda\alpha\, (\lceil M\rceil, \lceil N\rceil).\mathtt{1}, \lambda\alpha\, (\lceil M\rceil, \lceil N\rceil).\mathtt{2})\\
   \rightsquigarrow_\beta\ \rightsquigarrow_\beta\ &(\lambda\alpha\, \lceil M\rceil, \lambda\alpha\, \lceil N\rceil)
   \end{aligned}$$

- RedTransDistCoerProduct: We have

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

- RedTransDot: We have $(\lambda(x : \lceil\tau\rceil)\ x)\ \lceil M\rceil \rightsquigarrow_\beta \lceil M\rceil$
- RedTransCoer: We have $(\lambda(x : \lceil\tau\rceil)\ P[\lceil G\rceil\ x])\ \lceil M\rceil \rightsquigarrow_\beta P[\lceil G\rceil\ \lceil M\rceil]$
- RedCoer: This reifies on a simple RedTerm.

■

Proof of Lem 23

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

- x: This is refused by typing since we only have type variables in the environment.
- $\lambda(x : \tau)\ M$, $\lambda\alpha\, v$, $\lambda(c : \tau \rhd \tau)\, u$, and (v, v): These are values.
- $\diamond^\tau$, $G \xrightarrow{\tau} G$, $\mathrm{Dist}^{\tau\to\forall\alpha.\sigma}$, $(G * G)$, $\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)}$, $\mathrm{Dist}^{(\rho\rhd\rho')\Rightarrow(\tau*\sigma)}$, $\mathtt{Top}^\tau$, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\ v$: Using Lemma 22 on v_1, RedTerm applies.
- $v\,\tau$: Using Lemma 22 on v, RedType applies.
- $v\,\$\,G$: Using Lemma 22 on v, RedCoer applies.
- $v.1$: Using Lemma 22 on v, RedFirst applies.
- $v.2$: Using Lemma 22 on v, RedSecond applies.
- $G \mathbin{@} v$: By induction on G.
  - x, $\lambda(x : \tau)\ M$, $M\ M$, (M, M), $M.1$, $M.2$: These expressions are rejected by typing because they are terms and not coercions.
  - c: This is refused by typing since we only have type variables in the environment.
  - $\mathtt{Top}^\tau$: It is a value.
  - $\diamond^\tau$: RedDot applies.
  - $\lambda\alpha\, W$, $W\,\tau$, $\lambda(c : \tau \rhd \tau)\, W$, $W\,\$\,G$, and $G \mathbin{@} W$: RedTransCoer applies.
  - $G \xrightarrow{\tau} G$: Using Lemma 22 on v, RedTransArrow applies.
  - $\mathrm{Dist}^{\tau\to\forall\alpha.\tau}$: Using Lemma 22 consecutively twice on v, RedTransDist applies.
  - $(G * G)$: Using Lemma 22 on v, RedTransProduct applies.
  - $\mathrm{Dist}^{\forall\alpha.(\tau*\tau)}$: Using Lemma 22 consecutively twice on v, RedTransDistTypeProduct applies.
  - $\mathrm{Dist}^{(\tau\rhd\tau)\Rightarrow(\tau*\tau)}$: Using Lemma 22 consecutively twice on v, RedTransDistCoerProduct applies.

■

Proof of Prop 35

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

- x, $\lambda(x:\tau)\,v$, $\lambda\alpha\,v$, $\lambda(\alpha, c:\lhd\tau)\,v$, $\lambda(\alpha, c:\rhd\tau)\,v$, and (v, v): These are values.
- $\diamond^\tau$, $G \xrightarrow{\tau} G$, $\mathrm{Dist}^{\tau\to\forall\alpha.\sigma}$, $\mathrm{Dist}^{\tau\to\forall(\alpha,\lhd\rho).\sigma}$, $\mathrm{Dist}^{\tau\to\forall(\alpha,\rhd\rho).\sigma}$, $(G{*}G)$, $\mathrm{Dist}^{\forall\alpha.(\tau*\sigma)}$, $\mathrm{Dist}^{\forall(\alpha,\lhd\rho).(\tau*\sigma)}$, $\mathrm{Dist}^{\forall(\alpha,\rhd\rho).(\tau*\sigma)}$, $\mathtt{Top}^\tau$, and c: These expressions are rejected by typing because they are coercions and not terms.
- $v_1\ v$: Using Lemma 34 on v_1.
  - p: It is a value.
  - $\lambda(x:\tau)\,v$: RedTerm applies.
- $v\,\tau$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda\alpha\,v$: RedType applies.
- $v\,(\tau \lhd G)$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda(\alpha, c:\lhd\tau)\,v$: RedCoerPos applies.
- $v\,(\tau \rhd G)$: Using Lemma 34 on v.
  - p: It is a value.
  - $\lambda(\alpha, c:\rhd\tau)\,v$: RedCoerNeg applies.
- $v.\mathtt{1}$: Using Lemma 34 on v.
  - p: It is a value.
  - (v, v): RedFirst applies.
- $v.\mathtt{2}$: Using Lemma 34 on v.
  - p: It is a value.
  - (v, v): RedSecond applies.
- $G \mathbin{@} v$: Using Lemma 34 on G.
  - x, $\lambda(x:\tau)\ M$, $M\ M$, (M, M), $M.\mathtt{1}$, $M.\mathtt{2}$: These expressions are rejected by typing because they are terms and not coercions.
  - c, $\mathtt{Top}^\tau$: These are values.
  - $\diamond^\tau$: RedDot applies.
  - $\lambda\alpha\,W$, $W\,\tau$, $\lambda(\alpha, c:\lhd\tau)\,W$, $\lambda(\alpha, c:\rhd\tau)\,W$, $W\ \$\ G$, and $G \mathbin{@} W$: RedTransCoer applies.
  - $G \xrightarrow{\tau} G$: Using Lemma 34 on v.
    - p: It is a value.
    - $\lambda(x:\tau)\,v$: RedTransArrow applies.
  - $\mathrm{Dist}^{\tau\to\forall\alpha.\tau}$: Using Lemma 34 on v.
    - p: It is a value.
    - $\lambda\alpha\,v$: Using Lemma 34 on v.

· p: It is a value.
· $\lambda(x : \tau)\ v$: RedTransDist applies.

– $\mathrm{Dist}^{\tau \to \forall(\alpha, \lhd\tau).\tau}$: Using Lemma 34 on v.
  * p: It is a value.
  * $\lambda(\alpha, c : \lhd\tau)\, v$: Using Lemma 34 on v.
    · p: It is a value.
    · $\lambda(x : \tau)\ v$: RedTransDistCoerPosArrow applies.

– $\mathrm{Dist}^{\tau \to \forall(\alpha, \rhd\tau).\tau}$: Using Lemma 34 on v.
  * p: It is a value.
  * $\lambda(\alpha, c : \rhd\tau)\, v$: Using Lemma 34 on v.
    · p: It is a value.
    · $\lambda(x : \tau)\ v$: RedTransDistCoerNegArrow applies.

– $(G * G)$: Using Lemma 34 on v.
  * p: It is a value.
  * (v, v): RedTransProduct applies.

– $\mathrm{Dist}^{\forall\alpha.(\tau * \tau)}$: Using Lemma 34 on v.
  * p: It is a value.
  * $\lambda\alpha\, v$: Using Lemma 34 on v.
    · p: It is a value.
    · (v, v): RedTransDistTypeProduct applies.

– $\mathrm{Dist}^{\forall(\alpha, \lhd\tau).(\tau * \tau)}$: Using Lemma 34 on v.
  * p: It is a value.
  * $\lambda(\alpha, c : \lhd\tau)\, v$: Using Lemma 34 on v.
    · p: It is a value.
    · (v, v): RedTransDistCoerPosProduct applies.

– $\mathrm{Dist}^{\forall(\alpha, \rhd\tau).(\tau * \tau)}$: Using Lemma 34 on v.
  * p: It is a value.
  * $\lambda(\alpha, c : \rhd\tau)\, v$: Using Lemma 34 on v.
    · p: It is a value.
    · (v, v): RedTransDistCoerNegProduct applies.

■

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

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

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