# From Amber to Coercion Constraints

Didier Rémy
INRIA

Julien Cretin
TrustInSoft

**Abstract**

Subtyping is a common tool in the design of type systems that finds its roots in the $\eta$-expansion of arrow types and the notion of type containment obtained by closing System F by $\eta$-expansion. Although strongly related, subtyping and type containment still significantly differ from one another when put into practice. We introduce *coercion constraints* to relate and generalize subtyping and type containment as well as all variants of F-bounded quantification and instance-bounded quantification used for first-order type inference in the presence of second-order types. We obtain a type system with a clearer separation between computational and erasable parts of terms.

## 1 The different flavors of subtyping

**Subtyping in Amber** Nowadays, subtyping is a well-understood concept and tool in the design of type systems, but it has not always been so. The origin of subtyping goes back to the 60's when type conversions between base types or type classes were introduced in programming languages, but the first formalization of subtyping is by Reynolds (1980). Subtyping was introduced in the strongly typed functional language Amber by Cardelli (1984). By contrast with simple type conversions between base types, subtyping in Amber can be propagated through arrow types covariantly on the codomains of functions and contravariantly on their domains.

The language Amber had only subtyping but no parametric polymorphism: its typing rules are those of the simply typed $\lambda$-calculus extended with a subtyping rule Sub:

$$\textsc{Sub}\ \frac{\Gamma \vdash a : \tau \qquad \tau \rhd \sigma}{\Gamma \vdash a : \sigma} \qquad\qquad \textsc{Bot}\ \bot \rhd \tau \qquad\qquad \textsc{Top}\ \tau \rhd \top \qquad\qquad \textsc{Arrow}\ \frac{\tau' \rhd \tau \qquad \sigma \rhd \sigma'}{\tau \to \sigma \rhd \tau' \to \sigma'}$$

Through this paper we use the notation $\tau \triangleright \sigma$ to mean that $\tau$ is a subtype of $\sigma$ (or, rather, $\tau$ *can be coerced to* $\sigma$) for homogeneity between different forms of coercions that we will encounter. The subtyping relation is defined by the three rules, Bot, Top, and Arrow. For the sake of brevity we only consider the bottom and top types and arrow types; the language Amber also had records and subtyping between them which played a crucial role in the encoding of objects (but this is not our focus here), as well as recursive types which we introduce later on.

**Type containment** The notion of type containment was introduced simultaneously by Mitchell (1984, 1988) who considered the closure of System F by $\eta$-expansion, called $\mathsf{F}_\eta$: by definition, a closed term is in $\mathsf{F}_\eta$ if it has an $\eta$-expansion in System F. Interestingly, $\mathsf{F}_\eta$ has also a syntactic presentation that is closely related to the notion of subtyping. It extends the typing rules of System F with a subtyping rule similar to, but richer than, the one of Amber. Indeed, System F has polymorphic types, which we write $\forall(\alpha)\,\tau$, and therefore new subtyping rules are needed to describe how subtyping relates with polymorphism:

$$\begin{array}{llll}
\text{All} & \text{Trans} & \text{InstGen} & \text{Distrib-R} \\
\dfrac{\tau \triangleright \sigma}{\forall(\alpha)\,\tau \triangleright \forall(\alpha)\,\sigma} &
\dfrac{\tau \triangleright \sigma \quad \sigma \triangleright \rho}{\tau \triangleright \rho} &
\dfrac{\bar\beta \notin \mathsf{ftv}(\forall(\bar\alpha)\,\tau)}{\forall(\bar\alpha)\,\tau \triangleright \forall(\bar\beta)\,\tau[\bar\alpha \leftarrow \bar\sigma]} &
\dfrac{\alpha \notin \mathsf{ftv}(\tau)}{\forall(\alpha)\,(\tau \to \sigma) \triangleright \tau \to \forall(\alpha)\,\sigma}
\end{array}$$

The congruence rule for polymorphic types All is unsurprising. Rule Trans is uninteresting but it is needed in this presentation as it does not follow from other rules. The two interesting rules are InstGen and Distrib-R. Rule InstGen allows implicit toplevel type instantiation, as in ML: it means that a polymorphic type can be freely used at any of its instances. By contrast with ML, this rule can also be applied under arrow types at the appropriate variance. We write $\bar\alpha$ for a sequence of type variables (and similarly for types) with the understanding that $\forall(\bar\alpha)\,\tau$ stands for a sequence of quantifiers. Free type variables of $\tau$ are written $\mathsf{ftv}(\tau)$. Notice, that although InstGen may generalize type variables that are introduced during type instantiation, it cannot generalize other type variables. Therefore, $\mathsf{F}_\eta$ still need a specific term typing rule for polymorphism introduction. Rule Distrib-R allows a quantifier to be pushed down on the right of arrow types when it does not appear on the left. The original rule allowed $\alpha$ to also occur in $\tau$; then the quantifier must be pushed down on both sides of the arrow, *i.e.* the right-hand side of the coercion becomes $(\forall(\alpha)\,\tau) \to \forall(\alpha)\,\sigma$. This more general rule is however derivable from Distrib-R and the other rules.

The Rule Arrow can be easily explained in $\mathsf{F}_\eta$ by the $\eta$-expansion of arrow types. Since $\mathsf{F}_\eta$ is closed by $\eta$-expansion, whenever a term $a$ has some type $\tau \to \sigma$ in some context $\Gamma$, any type of its $\eta$-expansion $\lambda x.\, a\, x$ should also be a type of $a$. In particular, given $\tau' \triangleright \tau$ and $\sigma \triangleright \sigma'$, we may type the $\eta$-expansion by giving the parameter $x$ the type $\tau'$ and coerce the occurrence of $x$ as an argument of $a$ to the expected type $\tau$ which results in a value of type $\sigma$ that can be coerced to $\sigma'$. Hence, the term $a$ also has type $\tau' \to \sigma'$ in $\mathsf{F}_\eta$.

What distinguishes type containment from traditional uses of subtyping is the inclusion in the subtyping relation of the implicit instantiation of quantifiers, which by congruence can be applied deeply inside terms. Unfortunately, this theoretical strength turns into a weakness for using $\mathsf{F}_\eta$ in practice since the type containment relation becomes undecidable.

**Bounded quantification** While $\mathsf{F}_\eta$ extends Amber with polymorphic types and can prove subtyping relations between polymorphic types, it does not allow to make subtyping assumptions about polymorphic type variables. Bounded quantification introduced by Cardelli and Longo (1991) in the language Quest and later in the language $\mathsf{F}_{<:}$ (Cardelli et al. 1994) solves this problem. Polymorphic type variables are introduced with an upper bound $\forall(\alpha <: \tau)\, \sigma$, whose instances are types of the form $\sigma[\alpha \leftarrow \rho]$ where $\rho$ is a subtype of $\tau$. The unbounded quantification of System $\mathsf{F}$ can be recovered as the special case where the bound is $\top$. Bounded polymorphism is quite expressive and has been used to model record subtyping and object oriented features.

The most interesting rule in $\mathsf{F}_{<:}$ is subtyping between bounded quantifications:

$$\textsc{Fsub}\quad \frac{\Gamma \vdash \tau' \triangleright \tau \qquad \Gamma, \alpha \triangleright \tau' \vdash \sigma \triangleright \sigma'}{\Gamma \vdash \forall(\alpha <: \tau)\, \sigma \triangleright \forall(\alpha <: \tau')\, \sigma'}$$

Notice that subtyping holds only between two types having bounded quantification on both sides. That is, by contrast with $\mathsf{F}_\eta$, type instantiation is not part of the subtyping relation and is made explicit at the level of terms. A good reason for this choice is to ease typechecking. Still, $\mathsf{F}_{<:}$ in its full generality is undecidable (Pierce 1994), but some restrictions of the rule, for instance where the bounds are identical on both sides, are decidable. Bounded polymorphism has also been extended to $\mathsf{F}$-bounded polymorphism that allows the variable abstracted over to also appear in the bound, which is useful in object-oriented languages. The left-premise of Rule Fsub is then replaced by $\Gamma, \alpha \triangleright \tau' \vdash \alpha \triangleright \tau$.

Bounded polymorphism has been extensively used in many subsequent languages that combine polymorphism and subtyping, and in particular, to explore typechecking of objects (Abadi and Cardelli 1996).

Still, bounded polymorphism remains surprising in several ways. First, its formulation is asymmetric since type variables are introduced with an upper bound but no lower bound: dually, is there a use for lower bounds? Second, type variables have a unique bound: could the same variable have several bounds? Finally, while bounded polymorphism adds to $\mathsf{F}_\eta$ the ability to abstract over subtyping, it does not yet generalize type containment since subtyping holds only for types with the same polymorphic structure; in particular, the implicit instantiation $\forall(\alpha <: \top)\, \tau \rhd \tau[\alpha \leftarrow \sigma]$ that is allowed in $\mathsf{F}_\eta$ is not permitted in $\mathsf{F}_{<:}$.

**Instance-bounded quantification** Surprisingly, the absence of lower bounds in $\mathsf{F}_{<:}$ seemed to have not been a practical limitation and the question remained mostly theoretical for more than a decade. The need for lower bounds finally appeared for performing first-order type inference in the presence of second-order types. Predictable, efficient type inference is usually built on a notion of principal types, *i.e.* the ability to capture as a simple type (or type constraint) the set of all possible types of an expression. Lower bounds allow more expressions to have principal types.

Full type inference for System $\mathsf{F}$ is known for not having principal types because guessing types of function parameters often leads to an infinite set of solutions. However, if we restrict our ambition to not guess polymorphic types, but just propagate them, which is the goal of $\mathsf{ML^F}$ (Le Botlan and Rémy 2009), there is still a problem to solve, namely the absence of principal solutions.

To illustrate this, consider the function `revapp` defined as $\lambda x.\, \lambda f.\, f\, x$ of type $\forall(\alpha)\, \forall(\beta)\, \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$ and its partial application to the identity function `id` of type $\forall(\gamma)\, \gamma^2$ where $\gamma^2$ stands for $\gamma \rightarrow \gamma$. What should be the type of `revapp id`? The type $\forall(\beta)\, ((\forall(\gamma)\, \gamma^2) \rightarrow \beta) \rightarrow \beta$, say $\tau_1$, obtained by keeping the identity polymorphic? Or the type $\forall(\gamma)\, \forall(\beta)\, (\gamma^2 \rightarrow \beta) \rightarrow \beta$, say $\tau_2$, obtained by instantiating `id` to $\gamma^2$ before the application and generalizing the result of the whole application afterwards? Unfortunately, no other type of the application `revapp id` is an instance of both of these two types in System $\mathsf{F}$. (Nor in $\mathsf{F}_\eta$! but, conversely, both types have a common instance $\forall(\beta)\, (\forall(\gamma)\, (\gamma^2 \rightarrow \beta)) \rightarrow \beta$ in $\mathsf{F}_\eta$.)

This dilemma is solved in $\mathsf{ML^F}$ by introducing lower-bounded (or, rather, instance-bounded) polymorphism $\forall(\alpha :> \tau)\, \sigma$, which reads "*for all $\alpha$ that is an instance of $\tau$, $\sigma$*." The application `revapp id` can then be assigned the type $\forall(\alpha :> \forall(\gamma)\, \gamma^2)\, \forall(\beta)\, (\alpha \rightarrow \beta) \rightarrow \beta$, which happens to be principal in $\mathsf{ML^F}$. In particular, $\tau_1$ can be obtained by inlining the bound which is permitted by instantiation in $\mathsf{ML^F}$; and $\tau_2$ can be obtained by applying instantiation under the lower bound, generalizing $\gamma$ afterwards which gives $\forall(\gamma)\, \forall(\alpha :> \gamma^2)\, \forall(\beta)\, (\alpha \rightarrow \beta) \rightarrow \beta$, and finally inlining the bound.

While in this regard $\mathsf{ML^F}$ appears to be dual of $\mathsf{F}_{<:}$, it is still quite different: $\mathsf{ML^F}$ does not use contravariance of arrow types as both $\mathsf{F}_{<:}$ and $\mathsf{F}_\eta$ do; conversely, $\mathsf{ML^F}$ can freely instantiate polymorphic types, which $\mathsf{F}_{<:}$ cannot do: each of the three languages shares one feature with others but is still missing one of their key features.

$$
\begin{array}{rcll}
a, b & ::= & x \mid \lambda x.\, a \mid a\, a & \text{Terms} \\
\tau, \sigma & ::= & \alpha \mid \tau \to \tau \mid \bot \mid \top \mid \forall(\alpha : \kappa)\, \tau \mid \langle\rangle \mid \langle \tau, \tau \rangle \mid \pi_{\mathrm{i}}\, \tau & \text{Types} \\
\kappa & ::= & \star \mid \{\alpha : \kappa \mid \mathrm{P}\} \mid 1 \mid \kappa \times \kappa & \text{Kinds} \\
\mathrm{P} & ::= & (\Delta \vdash \tau) \rhd \tau \mid \exists\, \kappa \mid \top \mid \mathrm{P} \wedge \mathrm{P} & \text{Propositions} \\
\Gamma & ::= & \varnothing \mid \Gamma, (\alpha : \kappa) \mid \Gamma, (x : \tau) & \text{Environments}
\end{array}
$$

Figure 1: Syntax

**Subtyping constraints** The absence of multiple bounds in $\mathsf{F}_{<:}$ is somewhat surprising since $\mathsf{ML}$ extended with subtyping, say $\mathsf{ML}_{\leq}$, naturally comes with (and requires the use of) subtyping constrains that mix arbitrary upper and lower bounds (Odersky et al. 1999). It uses constrained type schemes of the form $\forall(\alpha|C)\, \tau$ where $C$ is a set of arbitrary but coherent subtyping constraints between simple types. Can we extend subtyping constraint to first-class polymorphic types?

## 2 The language $\mathsf{F}_{cc}$

We have seen several type systems with different combinations of subtyping features, but none of them supersedes all others. In particular, $\mathsf{F}_{\eta}$, $\mathsf{F}_{<:}$, $\mathsf{ML}^{\mathsf{F}}$, and $\mathsf{ML}_{\leq}$ are pairwise incomparable. We now describe an extension of System $\mathsf{F}$ with coercion constraints, called $\mathsf{F}_{cc}$, that combines all features together so that each of the languages above becomes a (still interesting) subset of $\mathsf{F}_{cc}$. Here, we present a small subset of the language just to carry the main ideas underlying its design. We refer the reader to (Cretin and Rémy 2014b) for technical details.

**Terms** The language $\mathsf{F}_{cc}$ is implicitly typed for reasons that will be explained later—but this happens to be an advantage for conciseness: terms are just those of the untyped $\lambda$-calculus, whose notations are reminded in Fig. 1.

Although we focus on the syntactic presentation of $\mathsf{F}_{cc}$ hereafter, we assume that its semantics is given by full $\beta$-reduction. This is to build its type system on solid ground, without taking advantage of the evaluation strategy: it prevents from sweeping errors under $\lambda$'s just because their evaluation is delayed. Full $\beta$-reduction also models reduction of open terms. A practical language based on $\mathsf{F}_{cc}$ will eventually have a call-by-name or call-by-value evaluation strategy, which being a subset of full $\beta$-reduction, will remain sound.

**Types, Kinds, and Propositions** Types, defined on Figure 1, are simple types (type variables, arrow types, top, and bottom types) extended with con-

$$\frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau}\;\textsc{TermVar}\qquad \frac{\Gamma\vdash\tau:\star\qquad \Gamma,(x:\tau)\vdash a:\sigma}{\Gamma\vdash\lambda x.\,a:\tau\to\sigma}\;\textsc{TermLam}\qquad \frac{\Gamma\vdash a:\tau\to\sigma\qquad\Gamma\vdash b:\tau}{\Gamma\vdash a\,b:\sigma}\;\textsc{TermApp}$$

$$\frac{\Gamma,\Delta\vdash a:\tau\qquad \Gamma\vdash(\Delta\vdash\tau)\rhd\sigma}{\Gamma\vdash a:\sigma}\;\textsc{TermCoer}$$

Figure 2: Term typing rules

strained polymorphic types of the form $\forall(\alpha:\kappa)\,\tau$ where $\kappa$ is a kind that restricts the set of types in which $\alpha$ may range.

The kind $\star$ is that of ordinary types, *e.g.* to form arrow types. The interesting kind is $\{\alpha:\kappa \mid \mathrm{P}\}$—the kind of types $\alpha$ of kind $\kappa$ that satisfy the proposition P.

The grammars of types and kinds also include type tuples[1] classified by tuple kinds. Namely, type tuples are constructed from the empty tuple $\langle\rangle$ of kind 1 and pairs of types $\langle\tau_1,\tau_2\rangle$ of kind $\kappa_1\times\kappa_2$ when $\tau_i$ has kinds $\kappa_i$; and type projection $\pi_{\mathrm{i}}\,\tau$ to return the $i$'s component of kind $\kappa_i$ of a tuple type of kind $\kappa_1\times\kappa_2$. Tuple types are useful for capturing multiple binders but are not technically difficult: the main technical change is to consider types equal modulo the projection of tuples and closing equality by equivalence and congruence for all syntactical constructs. Thus, we will ignore tuples (which are grayed in Figure 1) in the rest of the technical overview.

Propositions are used to restrict sets of types. The most useful proposition is the coercion proposition $(\Delta\vdash\tau)\rhd\sigma$ which states that there exists a coercion from type $\tau$ in some context extended with $\Delta$, to type $\sigma$. For now, one may just consider the particular case of *simple* coercions where $\Delta$ is empty, in which case the proposition is abbreviated as $\tau\rhd\sigma$ and just means that type $\tau$ can be coerced to type $\sigma$. The general case will be explained together with the term typing rules.

The proposition $\exists\,\kappa$ asserts that the kind $\kappa$ is coherent (intuitively, that it is inhabited, but this is relative to its typing context). It is interesting that coherence can be internalized as a proposition. The proposition $\top$ is the true proposition and the proposition $\mathrm{P}_1\wedge\mathrm{P}_2$ is the conjunction of $\mathrm{P}_1$ and $\mathrm{P}_2$.

Finally, typing environments $\Gamma$ bind type variables to kinds and program variables to types. The letter $\Delta$ is used to range over environments that only bind type variables.

**Term typing judgment ($\Gamma\vdash a:\tau$)** Typing rules, given in Figure 2 are almost as simple as in Amber: they contain the typing rules of the simply typed

[1]Not to be confused with the type of term tuples, which we do not include in this core version.

$$\frac{\Gamma, (\alpha : \kappa) \Vdash \mathrm{P} \qquad \Gamma \vdash \tau : \kappa \qquad \Gamma \vdash \mathrm{P}[\alpha \leftarrow \tau]}{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}\ \textsc{TypePack} \qquad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma \vdash \tau : \kappa}\ \textsc{TypeUnpack}$$

Figure 3: Type judgment relation (excerpt)

$\lambda$-calculus plus the coercion typing rule, which differs from the Amber Rule in two important ways. First, the coercion judgment depends on the context $\Gamma$, as in $\mathsf{F}_{<:}$, so that coercion assumptions in the context $\Gamma$ can be used to prove a coercion judgment such as $\Gamma \vdash \tau \rhd \sigma$. More importantly, coercions are of the more general form $(\Delta \vdash \tau) \rhd \sigma$ rather than just $\tau \rhd \sigma$. As we can see in Rule TermCoer, the context $\Delta$ contains additional type variable bindings that are added to the context $\Gamma$ of the premise under which the coerced term must have type $\tau$. The typical use of this generalization is in the judgment $\Gamma \vdash (\alpha : \kappa \vdash \tau) \rhd \forall(\alpha : \kappa)\, \tau$, which holds whenever $\kappa$ is coherent and $\forall(\alpha : \kappa)\, \tau$ is well-formed in $\Gamma$ (see Rule CoerGen below). As a consequence, the rule for polymorphism introduction

$$\frac{\Gamma \vdash \exists \kappa \qquad \Gamma, (\alpha : \kappa) \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha : \kappa)\, \tau}\ \textsc{TermGen}$$

is derivable by TermCoer and CoerGen. It is remarkable that polymorphism introduction (as well as all erasable features of the type system) can be handled purely as a coercion typing rule and need no counterpart in term typing rules.

**Well-formedness rules ($\Gamma \Vdash \kappa$ and $\Gamma \Vdash \mathrm{P}$)** The judgments $\Gamma \Vdash \kappa$ and $\Gamma \Vdash \mathrm{P}$ state well-formedness of kinds and propositions. Besides syntactical checks, they are recursively scanning their subexpressions for all occurrences of coercion propositions $(\Delta \vdash \tau) \rhd \sigma$ to ensure that $\Delta$, $\tau$, and $\sigma$ are well-typed, as described by the following rule:

$$\frac{\Gamma \vdash \Delta \qquad \Gamma, \Delta \vdash \tau : \star \qquad \Gamma \vdash \sigma : \star}{\Gamma \Vdash (\Delta \vdash \tau) \rhd \sigma}$$

In particular, the auxiliary judgment $\Gamma \vdash \Delta$, which treats $\Delta$ as a telescope, means that each binding $\alpha : \kappa$ of $\Delta$ is well-formed in the typing context that precedes it, which also implies that the kind $\kappa$ is coherent in any such binding relatively to its typing context.

**Kinding judgment ($\Gamma \vdash \tau : \kappa$)** An excerpt of kinding rules is given in Figure 3, but most rules have been omitted. Rule TypeUnpack states that whenever a type $\tau$ is known to be of a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, it is also of the kind $\kappa$, indeed.

$$\textsc{PropRes}\ \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma \vdash \mathrm{P}[\alpha \leftarrow \tau]} \qquad \textsc{PropExi}\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \exists\, \kappa}$$

Figure 4: Proposition judgment relation (excerpt)

$$\textsc{CoerRefl}\ \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau \rhd \tau} \qquad \textsc{CoerTop}\ \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \tau \rhd \top} \qquad \textsc{CoerBot}\ \frac{\Gamma \vdash \tau : \star}{\Gamma \vdash \bot \rhd \tau} \qquad \textsc{CoerTrans}\ \frac{\Gamma, \Delta' \vdash (\Delta \vdash \tau) \rhd \tau' \qquad \Gamma \vdash (\Delta' \vdash \tau') \rhd \tau''}{\Gamma \vdash (\Delta', \Delta \vdash \tau) \rhd \tau''}$$

$$\textsc{CoerWeak}\ \frac{\Gamma \vdash (\Delta \vdash \tau) \rhd \sigma}{\Gamma \vdash \tau \rhd \sigma} \qquad \textsc{CoerArr}\ \frac{\Gamma \vdash \tau : \star \qquad \Gamma, \Delta \vdash \tau \rhd \tau' \qquad \Gamma \vdash (\Delta \vdash \sigma') \rhd \sigma}{\Gamma \vdash (\Delta \vdash \tau' \to \sigma') \rhd \tau \to \sigma}$$

$$\textsc{CoerGen}\ \frac{\Gamma \vdash \exists\, \kappa \qquad \Gamma, (\alpha : \kappa) \vdash \tau : \star}{\Gamma \vdash (\alpha : \kappa \vdash \tau) \rhd \forall(\alpha : \kappa)\, \tau} \qquad \textsc{CoerInst}\ \frac{\Gamma, (\alpha : \kappa) \vdash \tau : \star \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash \forall(\alpha : \kappa)\, \tau \rhd \tau[\alpha \leftarrow \sigma]}$$

Figure 5: Coercion judgment relation

TypePack shows that, conversely, knowing that $\tau$ has kind $\kappa$, one must still prove that $\mathrm{P}[\alpha \leftarrow \tau]$ is satisfied to be able to consider that $\tau$ has the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$.

**Proposition judgment ($\Gamma \vdash \mathrm{P}$)** The two most interesting rules for propositions are given in Figure 4. Rule PropRes means that a type of a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ satisfies the proposition P (where $\tau$ has been substituted for $\alpha$). Rule PropExi means that one must exhibit a type $\tau$ of kind $\kappa$ to ensure that the kind $\kappa$ is coherent in its typing context $\Gamma$.

**Coercion propositions** ($\Gamma \vdash (\Delta \vdash \tau) \rhd \sigma$) We have separated the rules for coercions in Figure 5 although coercions are just a particular case of propositions. Some rules are slightly obfuscated by the coercion typing context $\Delta$ that has to be added in some premises. These rules can first be read in the particular case where $\Delta$ is the empty context, so we have grayed $\Delta$ to help with this reading.

Rule CoerRefl, CoerTop, and CoerBot are obvious and can be skipped. Rule CoerTrans is also standard—the $\Delta$'s just need to be appropriately concatenated.

Rule CoerWeak implements a form of weakening: it tells that if any term of typing[2] $\Gamma, \Delta \vdash \tau$ has typing $\Gamma \vdash \sigma$, then any term of typing $\Gamma \vdash \tau$ also has typing $\Gamma \vdash \sigma$. Weakening is required as it would not be derivable from the other rules if

[2]We say that a term $a$ has typing $\Gamma \vdash \tau$ if the typing judgment $\Gamma \vdash a : \tau$ holds.

we removed it from the definition.

Rule CoerArr is the usual contravariant rule for arrows when $\Delta$ is empty. Otherwise, the rule can be understood by looking at the $\eta$-expansion context $\lambda x. ([]\, x)$ for the arrow type. Placing a term with typing $\Gamma, \Delta \vdash \tau' \to \sigma'$ in the hole, we may give $\lambda x. ([]\, x)$ the typing $\Gamma \vdash \tau \to \sigma$ provided a coercion of type $\Gamma, \Delta \vdash \tau \triangleright \tau'$ is applied around $x$. The result of the application has typing $\Gamma, \Delta \vdash \sigma'$ which can in turn be coerced to $\Gamma \vdash \sigma$ if there exists a coercion of type $\Gamma \vdash (\Delta \vdash \sigma') \triangleright \sigma$. Thus, the $\eta$-expansion has typing $\Gamma \vdash \tau \to \sigma$.

Notice that CoerArr is the only $\eta$-expansion rule, since the arrow is the only computational type constructor in our core language. For each computational type constructor that we would add to the language, we would need a corresponding $\eta$-expansion coercion rule. Erasable type constructors need not have an $\eta$-expansion coercion rule, since these are derivable as their introduction and elimination rules are already coercions.

Rule CoerGen implements type generalization, but as a coercion rule. It says that $\Gamma \vdash (\alpha : \kappa \vdash \tau) \triangleright \forall(\alpha : \kappa)\, \tau$ is a valid coercion as long as kind $\kappa$ is coherent in $\Gamma$ and type $\tau$ has kind $\star$ in $\Gamma, (\alpha : \kappa)$. This coercion makes TermGen derivable as explained above.

Similarly, CoerInst is the counterpart of type instantiation coercion in $\mathsf{F}_\eta$: it says that $\Gamma \vdash \forall(\alpha : \kappa)\, \tau \triangleright \tau[\alpha \leftarrow \sigma]$ is a valid coercion as long as $\tau$ has kind $\star$ in $\Gamma, (\alpha : \kappa)$ and $\sigma$ has kind $\kappa$ in $\Gamma$. Since CoerGen is a coercion rule, we do not need the more involved version of $\mathsf{F}_\eta$ that performs generalization afterwards.

Notice that there is no counterpart to Distrib-R since it is derivable by a combination of type generalization, type instantiation, and $\eta$-expansion.

## Adding recursive types and coinduction

The full language $\mathsf{F}_{cc}$ also has recursive types. These are indeed present in the language $\mathsf{Amber}$, $\mathsf{F}_{<:}$, *etc.* as they are unavoidable in a programming language.

We thus extend the grammar of types with the production $\mu\alpha\, \tau$, with the usual restriction on well-foundedness of recursive types that we will not detail here. We write $\alpha \mapsto \tau : \mathsf{wf}$ to mean that the function $\alpha \mapsto \tau$ is well-founded and can be used to form the recursive type $\mu\alpha\, \tau$.

Since types are implicit, we chose equi-recursive types, *i.e.* the equality of recursive types is witnessed by coercions, which are implicit. We add two coercions to witness folding and unfolding of recursive types:

$$\textsc{CoerUnfold}\quad \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma, (\alpha : \star) \vdash \tau : \star}{\Gamma \vdash \mu\alpha\, \tau \triangleright \tau[\alpha \leftarrow \mu\alpha\, \tau]} \qquad \textsc{CoerFold}\quad \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma, (\alpha : \star) \vdash \tau : \star}{\Gamma \vdash \tau[\alpha \leftarrow \mu\alpha\, \tau] \triangleright \mu\alpha\, \tau}$$

To reason with recursive types we also add coinduction in propositions. In order to do so, we change the judgment $\Gamma \vdash \mathrm{P}$ to $\Gamma; \Theta \vdash \mathrm{P}$ where $\Theta$ is used to collect coinductive hypotheses. We introduce two new rules PropFix and PropVar to allow reasoning by coinduction.

$$\frac{\Gamma \Vdash \mathrm{P} \qquad \Gamma; \Theta, \mathrm{P} \vdash \mathrm{P}}{\Gamma; \Theta \vdash \mathrm{P}}\ \textsc{PropFix} \qquad \frac{\mathrm{P}^{\dagger} \in \Theta}{\Gamma; \Theta \vdash \mathrm{P}}\ \textsc{PropVar} \qquad \frac{\Gamma, \Delta; \Theta^{\dagger} \vdash \tau \rhd \tau' \qquad \Gamma; \Theta^{\dagger} \vdash (\Delta \vdash \sigma') \rhd \sigma}{\Gamma; \Theta \vdash (\Delta \vdash \tau' \to \sigma') \rhd \tau \to \sigma}\ \textsc{CoerArr}$$

Rule PropFix allows the judgment $\Gamma; \Theta \vdash \mathrm{P}$ to be proved under the additional coinductive hypothesis P (provided P is well-formed in $\Gamma$). Of course, coinductive hypotheses must be guarded before they can be used. This is implemented by tagging guarded propositions in $\Theta$ that are then ready for coinductive use with †. Hence, rule PropVar can prove P only if P appears guarded, *i.e.* as $\mathrm{P}^{\dagger}$, in the coinductive context $\Theta$.

Premises of the rule CoerArr use only subterms of the arrow type, so the rule acts as a guard for all coinductive assumptions, therefore all propositions of $\Theta$ are tagged in its premises. This is only the case of expansion rules for computational type constructors, which have a counterpart in terms. Currently, CoerArr is the only such rule. All other rules just transport $\Theta$ unchanged from the premise to the conclusion.

Reasoning by induction makes the following standard rules for equi-recursive types derivable (we write $\tau \lhd\!\rhd \sigma$ for a pair of simple coercions $\tau \rhd \sigma$ and $\sigma \rhd \tau$):

$$\frac{\alpha \mapsto \sigma : \mathsf{wf} \qquad \Gamma; \Theta \vdash (\tau_i \lhd\!\rhd \sigma[\alpha \leftarrow \tau_i])^{i \in \{1,2\}}}{\Gamma; \Theta \vdash \tau_1 \rhd \tau_2}\ \textsc{CoerPeriod} \qquad \frac{\Gamma, (\alpha, \beta, \alpha \rhd \beta); \Theta \vdash \tau \rhd \sigma}{\Gamma; \Theta \vdash \mu\alpha\, \tau \rhd \mu\beta\, \sigma}\ \textsc{CoerEtaMu}$$

Interestingly, the proof for CoerPeriod requires reinforcement of the coinduction hypothesis since we need $\tau_1 \lhd\!\rhd \tau_2$ and not just $\tau_1 \rhd \tau_2$ in the coinduction hypothesis.

## 3 Strength and weaknesses of $\mathsf{F}_{cc}$

**Soundness** The type system of $\mathsf{F}_{cc}$ is sound for the full $\beta$-reduction semantics. Type soundness is not proved syntactically for reasons explained next, but semantically, by interpreting types as sets of terms. As a consequence of the presence of general recursive types (*i.e.* we do not restrict to positive recursion), we use a step-indexed technique. Unfortunately, the usual technique of Appel and McAllester (2001) does not apply to a full $\beta$-reduction setting. We propose a new technique where indexes that are traditionally outside terms are placed directly on terms and are transformed during reduction. See (Cretin and Rémy 2014a) for details.

**Termination** As a sanity check, the reduction of well-typed programs always terminates in the absence of recursive types and coinduction.

**(Lack of a proof of) Subject reduction** The reason not to do a syntactic proof is that we do not know how to prove subject reduction for $\mathsf{F}_{cc}$. The problem is that the type system is either too expressive or too weak: it allows to type programs that are indeed safe, but with involved non local coercion constraints that cannot be easily traced during reduction.

Doing a syntactic proof would amount to having an explicitly typed version of the language and reduction rules for explicitly typed terms that preserve well-typedness; moreover, reduction of explicitly typed terms must be in bisimulation with the reduction of implicitly typed ones. The main obstacle is that, in the general case, abstract coercions may appear in the middle of a redex. Explaining why this is difficult in the general case is a bit tricky, as solving one issue immediately raises another one—see Cretin and Rémy (2012) for details. Quite interestingly, this configuration can never occur when we restrict to abstract coercions that are parametric in either their domain or their codomain, *i.e.* coercions of the form $\alpha \triangleright \tau$ or $\tau \triangleright \alpha$. Remarkably, these two subcases coincide with bounded quantification and instance-bounded quantification. Under this restriction, we can design an explicitly typed language that enjoys subject reduction (Cretin and Rémy 2012).

**(Lack of a good) Surface language** Since the type system is implicit, it is undecidable, of course. What is a good surface language for $\mathsf{F}_{cc}$ is still an open question. It is always possible to be fully explicit by introducing term syntax for describing typing derivations in source terms, hence turning type inference into an easy checking process. However, this would not only contain type annotations but also full coercion bodies (information which is typically inferred in languages such as $\mathsf{F}_{<:}$) and of coercion coherence proofs (information which is usually obtained by construction). Programming at this level of detail would be too cumbersome for the programmer. Notice however, that $\mathsf{F}_{\eta}$ already suffers from a similar problem, since its coercion relation is undecidable.

This issue can be addressed in two directions. Remaining within $\mathsf{F}_{cc}$, we may apply partial type inference techniques and hope that sufficient type and coercion information can be reconstructed. It would also be interesting to look for subsystems of $\mathsf{F}_{cc}$ that compromise expressiveness for a smaller amount of annotations. The restriction to coercions that are parametric in either their domain or their codomain is one such solution. Are there other sweet spots?

**Expressiveness** As announced earlier, $\mathsf{F}_{cc}$ contains $\mathsf{F}_{\eta}$, $\mathsf{F}_{<:}$, $\mathsf{ML}^{\mathsf{F}}$, and $\mathsf{ML}_{\leq}$ as sublanguages. This confirms that all their features are compatible and can safely be combined together.

Writing $\forall(\alpha \mid \mathrm{P})\,\tau$ for $\forall(\alpha : \{\alpha : \star \mid \mathrm{P}\})\,\tau$, bounded quantification and instance-bounded quantification can be encoded in $\mathsf{F}_{cc}$ as $\forall(\alpha \mid \alpha \triangleright \tau)\,\sigma$ and $\forall(\alpha \mid \tau \triangleright \alpha)\,\sigma$. It is then not difficult to see that the typing rules of $\mathsf{F}_{<:}$ (including F-Bounded quantification) and of $\mathsf{ML}^{\mathsf{F}}$ are derivable.

The notation is genealized to bindings, writing $(\overline{\alpha} \mid \mathrm{P})$ for $(\alpha : \{\alpha : \star^n \mid \mathrm{P}\})$ where $n$ is the size of $\bar{\alpha}$ and, by abuse of notation, let us write $\alpha_i$ for $\pi_{\mathrm{i}}\,\alpha$ when $\alpha_i$ is the $i$' component of a sequence $\bar{\alpha}$. Then, a constrained typing judgment $\Gamma \vdash e : \tau \mid C$ in $\mathsf{ML}_{\leq}$ can be seen as the $\mathsf{F}_{cc}$ judgment $(\bar{\alpha} \mid C), \Gamma \vdash e : \tau$ where $\overline{\alpha}$ are free variables of $\Gamma$, $C$, and $\tau$.

We have only presented a core subset of $\mathsf{F}_{cc}$. The full language (Cretin and Rémy 2014a) also contains products. Sum types are could be also be easily added. Existential types can be emulated by their CPS encoding. The language of propositions contains polymorphic propositions $\forall(\alpha : \kappa)\,\mathrm{P}$, and could also be enriched with other propositions.

**Incoherent coercions** In the subset of $\mathsf{F}_{cc}$ described above, coercions are always coherent relative to the typing context in which they are used. More precisely, when an expression $a$ has type $\forall(\alpha : \kappa)\,\tau$ in $\Gamma$, it is always the case that $\Gamma \vdash \exists\,\kappa$. This is necessary because type abstraction does not have any counterpart in terms and, in particular, does not block the evaluation of $a$ which may proceed immediately. Without coherence, one could abstract over the absurd kind constraint $\top \triangleright \bot$ and be able to type any program.

However, there are situations in which abstraction over incoherent coercions would be useful. First, the coercion may be coherent only for *some* instances of the typing context. This is typically the case in the presence of GADTs. Second, coherence at the abstraction point is often harder to prove than at instantiation points where types have been specialized. For these reasons, we have also extended $\mathsf{F}_{cc}$ with incoherent coercion abstractions. Of course, this abstraction must now block the evaluation and therefore have a counterpart in terms. Interestingly, this allows to model GADTs as incoherent coercions. Incoherent abstractions are indeed used in $\mathsf{FC}$ (Weirich et al. 2011), the intermediate language of Haskell. See (Cretin and Rémy 2014a) for further details.

## Conclusions

We have given a tour of coercion constraints and shown how they can be used to explain several type systems that had been designed separated for different purposes, but all around some variations on the notion of subtyping. There are an amazingly large number of interesting works on subtyping, many of which actually initiated by Cardelli, and we could unfortunately just select a few citations among all the relevant ones. As for coercions, the idea is not at all new. There are in fact several notions of coercions and many works on type systems have already used coercions as a tool or studied them on their own. So, many more references could have been included here as well. We refer the reader to (Cretin and Rémy 2014a) for a more thorough treatment of related works.

Coercion constraints can factor many type features of programming languages. This allows sharing an important part of their meta-theoretical studies, such as type soundness. It also opens the door to new combinations of features. Besides, it helps separate the computational and erasable parts of type systems. We thus believe that they are an interesting framework for designing and studying type systems.

However, $\mathsf{F}_{cc}$ still lacks a good surface language for the programmer as well an explicitly-typed calculus to be used as an internal language in a compiler. While partial type inference techniques could be used for the surface language, finding an explicit version of coercion constraints that enjoys subject reduction without sacrificing expressiveness seems much more challenging.

## References

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996. ISBN 0387947752.

A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems.*, 23(5), Sept. 2001.

L. Cardelli. A semantics of multiple inheritance. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc. ISBN 3-540-13346-1. URL `http://dl.acm.org/citation.cfm?id=1096.1098`.

L. Cardelli and G. Longo. A semantic basis for quest. *J. Funct. Program.*, 1(4): 417–458, 1991.

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.

J. Cretin and D. Rémy. On the power of coercion abstraction. In *Proceedings of the annual symposium on Principles Of Programming Languages*, 2012.

J. Cretin and D. Rémy. System F with Coercion Constraints. Rapport de recherche RR-8456, INRIA, Jan. 2014a. URL `http://hal.inria.fr/hal-00934408`.

J. Cretin and D. Rémy. System F with Coercion Constraints. In *Logic In Computer Science (LICS)*, July 2014b. To appear.

D. Le Botlan and D. Rémy. Recasting MLF. *Information and Computation*, 207 (6), 2009.

J. C. Mitchell. Type inference and type containment. In *Proc. Of the International Symposium on Semantics of Data Types*, pages 257–277, New York, NY, USA, 1984. Springer-Verlag New York, Inc. ISBN 3-540-13346-1. URL `http://dl.acm.org/citation.cfm?id=1096.1106`.

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

M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

B. C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

J. C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 211–258, Berlin, 1980. Springer-Verlag.

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, 2011.