System F with Coercion Constraints

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

***Abstract*— We present a second-order λ-calculus with coercion constraints that generalizes a previous extension of System F with parametric coercion abstractions [1] by allowing multiple but simultaneous type and coercion abstractions, as well as recursive coercions and equi-recursive types. This allows to present in a uniform way several type system features that had previously been studied separately: type containment, bounded and instance-bounded polymorphism, which are already encodable with parametric coercion abstraction, and ML-style subtyping constraints. Our framework allows for a clear separation of language constructs with and without computational content. We also distinguish coherent coercions that are fully erasable from potentially incoherent coercions that suspend the evaluation—and enable the encoding of GADTs.**

Technically, *type* coercions that witness subtyping relations between types are replaced by a more expressive notion of *typing* coercions that witness subsumption relations between typings, *e.g.* pairs composed of a typing environment and a type. Our calculus is equipped with a strong notion of reduction that allows reduction under abstractions—but we also introduce a form of weak reduction as reduction cannot proceed under incoherent type abstractions. Type soundness is proved by adapting the step-indexed semantics technique to strong reduction strategies, moving indices inside terms so as to control the reduction steps internally—but this is only detailed in the extended version.

I. Introduction

Type systems are syntactical languages to express properties and invariants of programs. Their objects are usually types, typing contexts, and typing derivations. These can be interpreted as mathematical objects or proofs. Typically, a typing judgment $\Gamma \vdash a : \tau$ can be interpreted as a proof that the term a is well-behaved and that its computational behavior is approximated by the type τ when the approximations of the behaviors of its free variables are given by the typing context Γ. The simply-typed λ-calculus extended with constants such as pairs or integers to model the core of a real programming language is the simplest of all type systems. It is also somewhat canonical: it just contains one type construct for each related construct of the language: arrow types $\tau \to \sigma$ to classify functions, $\tau \times \sigma$ to classify pairs, *etc.* and nothing else. Each type construct has a counter-part in terms and we may call them *computational* types.

Simply-typed λ-calculus is however too restrictive and type systems are usually extended with some form of polymorphism that allows an expression to have several types, or rather a type that stands for a whole collection of other types. Parametric polymorphism, whose System F is the reference introduces polymorphic types $\forall\alpha\,\tau$. A typing judgment $\Gamma \vdash a : \forall\alpha\,\tau$ means that the program a has also type $\tau[\alpha \leftarrow \sigma]$ (*i.e.* τ where α has been replaced by σ) for all types σ.

This operation, called type instantiation is in fact independent of the program a and can be captured as an auxiliary instantiation judgment $\forall\alpha\,\tau \leq \tau[\alpha \leftarrow \sigma]$. This means that any term that has type $\forall\alpha\,\tau$ also has type $\tau[\alpha \leftarrow \sigma]$. Type instantiation is only a very specific form of some more general concept called *type containment* introduced by Mitchell [2]. Mitchell showed that adding type containment to System F is equivalent to closing System F by η-expansion (hence the name F_η for the resulting system). Type containment allows instantiation to be performed deeper inside terms (by contrast with System F where it remains superficial), following the structure of types covariantly, or contravariantly on the left of arrow types. Type containment contains the germs of subtyping, which usually refers to a restriction of type containment that does not include type instantiation as part of the subtyping relation, but instead injects primitive subtyping relations between constants such as $\mathsf{int} \leq \mathsf{float}$ or a primitive bottom and top types. $\mathsf{F}_{<:}$ [3] is the system of reference for combining subtyping with polymorphism. Surprisingly, the languages F_η and $\mathsf{F}_{<:}$ share the same underlying concepts but have in fact quite different flavors and are incomparable (no one is strictly more general then the other). For example, $\mathsf{F}_{<:}$ has bounded quantification $\forall(\alpha \leq \tau)\,\sigma$ that allows to abstract over all type α that are a subtype of τ, a concept not present in F_η. Although quite powerful, bounded quantification seems bridled and somewhat *ad hoc* as it allows for a unique and upper bound.

The language ML^F [4] is another variant of System F that has been introduced for performing partial type inference while retaining principal types. It has similarities with both $\mathsf{F}_{<:}$ and F_η, but introduces yet another notion, instance bounded quantification—or unique lower bounds.

In [1], we introduced F_ι^p, a language of *type coercions* with the ability to *abstract* over coercions, that can express F_η type containment, $\mathsf{F}_{<:}$ upper bounded polymorphism, and ML^F instance-bounded polymorphism, uniformly. Following a general and systematic approach to coercions lead to an expressive and modular design. However, F_ι^p still comes with a severe restriction: abstract coercions must be parametric in either their domain or their codomain, so that abstract coercions are coherent, *i.e.* their types are always inhabited by concrete coercions. This limitation is disappointing from both theoretical and practical view points. In practice, F_ι^p fails to give an account of subtyping constraints that are used for type inference with subtyping in ML. While in theory, subtyping constraints and second-order polymorphism are orthogonal concepts that should be easy to combine.

Summary of our contributions: In this work, we solve this problem and present a language F_{cc} that generalizes F_ι^p (and thus subsumes F_η, $\mathsf{F}_{<:}$, and ML^F) to also model subtyping constraints. Besides, F_{cc} includes a general form of recursive coercions, from which we can recover powerful subtyping

rules between equi-recursive types. As in our previous work, the language is equipped with a strong reduction strategy, which also models reduction on open terms and provides stronger properties. We still permit a form of weak reduction *on demand* to model incoherent abstractions when needed, *e.g.* to encode GADTs.

We also generalize *type coercions* to *typing coercions* which enables a much clearer separation between computational types and erasable types that are all treated as coercions. In particular, type abstraction becomes a coercion and distributivity rules become derivable. Another side contribution of our work which is however not detailed here by lack of space but can be found in the extended version, is an adaption of step-indexed semantics to strong reduction strategies, moving indices inside terms.

Plan: The rest of the paper is organized as follows. We discuss a few important issues underlying the design of F_{cc} in §II. We present F_{cc} formally and state its properties in §III. We discuss the expressivity of F_{cc} in §V and differences with our previous work and other related works as well as future works in §VI.

By lack of space, we have omitted many technical details and the whole proof of type soundness via step-indexed terms, which can be found in the extended version of this paper [5] and in the first author's PhD dissertation [6, chap 5]. The language F_{cc} and its soundness and normalization proofs have been formalized and mechanically verified in Coq.[1]

II. The design of F_{CC}

The language F_{cc} is designed around the notion of erasable coercions. Strictly speaking erasable coercions should leave no trace in terms and not change the semantics of the underlying untyped λ-term. When coercions are explicit and kept during reduction, as in F_t^p, one should show a bisimulation property between the calculus with explicit coercions and terms of λ-calculus after erasure of all coercions. However, since coercions do not have computational content, they may also be left implicit, as is the case in F_{cc}.

Some languages also use coercions with computational content. These are necessarily explicit and cannot be erased at runtime. They are of quite a different nature, so we restrict our study to erasable coercions.

Still, erasability is subtle in the presence of coercion abstraction, because one could easily abstract over nonsensical coercions, *e.g.* that could coerce any type into any other type. By default, these situations should be detected and rejected of course. We say that coercion abstraction is *coherent* when the coercion type is inhabited and *incoherent* when it may be uninhabited. Notice that type abstraction in System F, bounded polymorphism in $\mathsf{F}_{<:}$, and instance bounded polymorphism in MLF are all coherent.

Coherent abstraction ensures that the body of the abstraction is meaningful—whenever well-typed. Hence, it makes sense to reduce the body of the abstraction before having a concrete value for the coercion— or equivalently to reduce open terms that contain coherent abstract coercions.

[1]Scripts are available at http://gallium.inria.fr/~remy/coercions/.

Conversely, incoherent abstraction must freeze the evaluation of the body until it is specialized with a concrete coercion that provides inhabitation evidence. Therefore, *abstraction* over incoherent coercions cannot be erased, even though coercions themselves carry no information and can be represented as the unit type value, as in FC—the internal language of Haskell whose coercion abstractions are (potentially) incoherent.

Choosing a weak evaluation strategy as is eventually done in all programming languages does not solve the problem, but just sweeps it under the carpet: while type-soundness will hold, static type errors will be delayed until applications and library functions that will never be applicable may still be written.

Conversely, a strong reduction strategy better exercises the typing rules: that is, type soundness for a strong reduction strategy provides stronger guarantees. In our view, type systems should be designed to be sound for strong reduction strategies even if their reduction is eventually restricted to weak strategies for efficiency reasons. This is how programming languages based on System F or $\mathsf{F}_{<:}$ have been conceived, indeed.

Therefore, F_{cc} is equipped with strong reduction as the default and this is a key aspect of our design which could otherwise have been much simpler but also less useful.

However, we also permit abstraction over (potentially) incoherent coercions *on demand*, as this is needed to encode some form of dynamic typing, as can be found in programming languages with GADTs, for example. Indeed, GADTs allow to define parametric functions that are partial and whose body may only make sense for some but not all type instances. When accepting a value of a GADT as argument, the function may gain evidence that some type equality holds and that the value is indeed in the domain of the function. We claim that incoherent abstraction should be used exactly when needed and no more. In particular, one should not make all abstractions incoherent just because some of them must be.

From explicit to implicit coercions: Coherence is ensured in F_t^p by the parametricity restriction that limits abstraction to have a unique upper or lower bound. This also prevents abstract coercions from appearing in between the destructor and the constructor of a redex, a pattern of the form $c\langle\lambda(x : \tau')\, M\rangle\, N$ (where $c\langle\cdot\rangle$ is the application of a coercion) called a *wedge*, which could typically block the reduction of explicitly typed terms—therefore loosing the bisimulation with reduction of untyped terms. While the coherence of the abstract coercion c should make it safe to break it apart into two pieces, one attached to the argument N, the other one attached to the body M, this would require new forms of coercions, new reduction rules and quite sophisticated typing rules to keep track of the relation between the residual of wedges after they have been split apart. Even though it should be feasible in principle, this approach seemed far too complicated in the general case to be of any practical use.

Therefore our solution is to give up explicit coercions and leave them implicit. While this removes the problem of wedges at once, it also prevents us from doing a syntactic proof of type soundness. Instead, type soundness in F_{cc} is proved semantically by interpreting types as sets of terms and

coercions as proofs of inclusion between types.

Simultaneous coercion abstractions: In order to relax the parametricity restriction of F_ι^p and allow coercions whose domain and range are simultaneously structured types, while preserving coherence, we permit multiple type abstractions to be introduced simultaneously with all coercion abstractions that constraint them. Since coherence does not come by construction anymore, coherence proofs must be provided explicitly for each block of abstraction as witnesses that the types of coercions are inhabited, *i.e.* that they can be at least instantiated once in the current environment.

Grouping related abstractions allows to provide coherence proofs independently for every group of abstractions, and simultaneously for every coercion in the same group.

From type *coercions to* typing *coercions:* A type coercion $\tau \triangleright \sigma$ is a proof that all programs of type τ have also type σ in some environment Γ. Pushing the idea of coercions further, typings (the pair of an environment and a type, written $\Gamma \vdash \tau$) are themselves approximations of program behaviors, which are also naturally ordered. Thus, we may consider syntactical objects, which we call *typing coercions*, to be interpreted as proofs of inclusions between the interpretation of typings. By analogy with *type coercions* that witness a subtyping relation between types, typing coercions witness a relation between typings. This idea, which was already translucent in our previous work [1], is now internalized.

Interestingly, type generalization can be expressed as a typing coercion—but not as a type coercion: it turns a typing $\Gamma, \alpha \vdash \tau$ into the typing $\Gamma \vdash \forall \alpha\, \tau$. This allows to replace what is usually a term typing rule by a coercion typing rule, with two benefits: superficially, it allows for a clearer separation of *term* constructs that are about computation from *coercion* constructs that do not have computational content (type abstraction and instantiation, subtyping, *etc.*); more importantly, it makes type generalization automatically available anywhere a coercion can be used and, in particular, as parts of bigger coercions. An illustration of this benefit is that the distributivity rules (*e.g.* as found in F_η) are now derivable by composing type generalization, type instantiation, and η-expansion (generalization of the subtyping rule for computational types).

The advantage of using *typing* coercions is particularly striking in the fact that all erasable type system features studied in this paper can be expressed as coercions, so that computation and typing features are perfectly separated.

III. Language definition

A. The syntax and semantics of terms

The syntax of the language is given in Figure 1. Because our calculus is implicitly typed, its syntax is in essence that of the λ-calculus extended with pairs. Terms contain variables x, abstractions $\lambda x\, a$, applications $a\, b$, pairs $\langle a, b\rangle$, and projections $\pi_{\mathrm{i}}\, a$ for i in $\{1, 2\}$.

Terms also contain two new constructs $\partial\, a$ and $a \diamondsuit$ called incoherent abstraction and incoherent application, respectively. The incoherent abstraction $\partial\, a$ can be seen as a marker on the term a that freezes its evaluation, while the incoherent application $a \diamondsuit$ allows evaluation of the frozen term a to be resumed. These two constructs enforce a form of weak reduction in a calculus with strong reduction by default. They are required to model GADTs, but removing them consistently everywhere preserves all the properties of F_{cc}. Hence, one can always ignore them in a first reading of the paper.

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
x, y & & & \text{Term variables} \\
a, b & ::= & x \mid \lambda x\, a \mid a\, a \mid \langle a, a\rangle \mid \pi_{\mathrm{i}}\, a \mid \partial\, a \mid a \diamondsuit & \text{Terms} \\
\kappa & ::= & \star \mid 1 \mid \kappa \times \kappa \mid \{\alpha : \kappa \mid \mathrm{P}\} & \text{Kinds} \\
\tau, \sigma & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \forall(\alpha : \kappa)\, \tau & \text{Types} \\
 & & \mid \Pi(\alpha : \kappa)\, \tau \mid \mu\alpha\, \tau \mid \bot \mid \top & \\
 & & \mid \langle\rangle \mid \langle \tau, \tau\rangle \mid \pi_{\mathrm{i}}\, \tau & \\
\mathrm{P} & ::= & \top \mid \mathrm{P} \wedge \mathrm{P} \mid (\Sigma \vdash \tau) \triangleright \tau & \text{Propositions} \\
 & & \mid \exists \kappa \mid \forall(\alpha : \kappa)\, \mathrm{P} & \\
\Theta & ::= & \varnothing \mid \Theta, \mathrm{P} \mid \Theta, \mathrm{P}^\dagger & \text{Coinduction environments} \\
\Gamma & ::= & \varnothing \mid \Gamma, (\alpha : \kappa) \mid \Gamma, (x : \tau) & \text{Environments} \\
\chi & ::= & \mathsf{ne} \mid \mathsf{wf} & \text{Recursive tokens}
\end{array}
$$

Fig. 1. Syntax

$$
\begin{array}{rcll}
\Sigma & ::= & \varnothing \mid \Sigma, (\alpha : \kappa) & \text{Erasable environments} \\
p & ::= & x \mid p\, v \mid \pi_{\mathrm{i}}\, p \mid p \diamondsuit & \text{Prevalues} \\
v & ::= & p \mid \lambda x\, v \mid \langle v, v\rangle \mid \partial\, a & \text{Values} \\
h & ::= & \lambda x\, a \mid \langle a, a\rangle \mid \partial\, a & \text{Constructors} \\
D & ::= & []\, a \mid \pi_{\mathrm{i}}\, [] \mid [] \diamondsuit & \text{Destructors} \\
E & ::= & \lambda x\, [] \mid []\, a \mid a\, [] \mid \cancel{\partial\, []} \mid [] \diamondsuit & \text{Contexts} \\
 & & \mid \langle [], a\rangle \mid \langle a, []\rangle \mid \pi_{\mathrm{i}}\, [] &
\end{array}
$$

Fig. 2. Notations

The reduction rules are given on Figure 3. We write $a[x \leftarrow b]$ for the capture avoiding substitution of the term b for the variable x in the term a, defined as usual. Head reduction is described by the β-reduction rule RedApp, the projection rule RedProj, and RediApp for unfreezing frozen computations. Reduction can be used under any evaluation context as described by Rule RedCtx. Evaluation contexts, written E, are defined on Figure 2. Since we choose a strong reduction relation, all possible contexts are allowed—except reduction under incoherent abstractions. The notation $\cancel{\partial\, []}$ is to emphasize that $\partial\, []$ is not an evaluation context.) Notice that evaluation contexts contain a single node, since the context rule RedCtx can be applied recursively.

The terms we are interested in are the sound ones, *i.e.* whose evaluation never produces an *error*. Errors are the subset of syntactically well-formed terms that "we don't want to see" neither in source programs nor during their evaluation: an error is either immediate or occurring in an arbitrary context E (Figure 2); immediate errors are potential redexes $D[h]$ (the application of a destructor D to a constructor h) that are not valid redexes (the left-hand side of a head-reduction rule). Conversely, values are the irreducible terms that we expect as results of evaluation: they are either constructors applied to values or prevalues which are themselves either variables

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

$$\textsc{TermApp}\ \frac{\Gamma\vdash a:\tau\to\sigma \qquad \Gamma\vdash b:\tau}{\Gamma\vdash a\,b:\sigma} \qquad \textsc{TermPair}\ \frac{(\Gamma\vdash a_i:\tau_i)^{i\in\{1,2\}}}{\Gamma\vdash\langle a_1,a_2\rangle:\tau_1\times\tau_2}$$

$$\textsc{TermProj}\ \frac{\Gamma\vdash a:\tau_1\times\tau_2}{\Gamma\vdash\pi_{\mathrm{i}}\,a:\tau_i} \qquad \textsc{TermCoer}\ \frac{\Gamma,\Sigma\vdash a:\tau \qquad \Gamma\vdash(\Sigma\vdash\tau)\triangleright\sigma}{\Gamma\vdash a:\sigma}$$

$$\textsc{TermBlock}\ \frac{\Gamma\Vdash\kappa \qquad \Gamma,(\alpha:\kappa)\vdash a:\tau}{\Gamma\vdash\partial\,a:\Pi(\alpha:\kappa)\,\tau} \qquad \textsc{TermUnblock}\ \frac{\Gamma\vdash\sigma:\kappa \qquad \Gamma\vdash a:\Pi(\alpha:\kappa)\,\tau}{\Gamma\vdash a\diamondsuit:\tau[\alpha\leftarrow\sigma]}$$

Fig. 4. Term typing rules

$$\textsc{RedCtx}\ \frac{a\rightsquigarrow b}{E[a]\rightsquigarrow E[b]}$$

$$(\lambda x\,a)\,b\rightsquigarrow a[x\leftarrow b] \quad \text{(RedApp)}$$
$$\pi_{\mathrm{i}}\,\langle a_1,a_2\rangle\rightsquigarrow a_i \quad \text{(RedProj)}$$
$$(\partial\,a)\diamondsuit\rightsquigarrow a \quad \text{(RedIApp)}$$

Fig. 3. Reduction relation

or destructors applied to prevalues. Notice that the definition of errors is independent of the reduction strategy while the definition of values is not. This is why we prefer to state soundness as the fact that reduction never produces errors, avoiding the reference to the more fragile definition of values.

B. Types, kinds, propositions, and coercions

We use types to approximate the behavior of terms, but types are themselves classified by kinds. So let us present kinds first. Although we do not have type functions, we need to manipulate tuples of types because several type variables and coercion constraints sometimes need to be introduced altogether. For sake of simplicity and a slight increase in flexibility, we mix types, type sequences, and constrained types into the same syntactical class of types which are then classified by kinds. Kinds are written κ. The star kind $\star$ classifies sets of terms, as usual. The unit kind 1 and the product kind $\kappa\times\kappa$ are used to classify the unit object and pairs of types, which combined together, may encode type sequences: for example, a type variable of kind $\kappa_1\times\kappa_2$ may stand for a pair of variables of kinds κ_1 and κ_2. The constrained kind $\{\alpha:\kappa\mid \mathrm{P}\}$ restricts the set κ to the elements α satisfying the proposition P. For instance, $\{\alpha:\star\mid\alpha\triangleright\tau\}$ is the set of types σ that can be coerced to (*e.g.* are a subtype of) τ—assuming that α is not free in τ.

Instead of having only proofs of inclusion between sets of terms, which we call coercions, we define a general notion of propositions, written P. Propositions contain the true proposition $\top$, conjunctions $\mathrm{P}\wedge\mathrm{P}$, coercions $(\Sigma\vdash\tau)\triangleright\tau$, coherence propositions $\exists\,\kappa$, and polymorphic propositions $\forall(\alpha:\kappa)\,\mathrm{P}$. The proposition $(\Sigma\vdash\tau)\triangleright\sigma$ in a context Γ means the existence of a coercion from the typing $\Gamma,\Sigma\vdash\tau$ to the typing $\Gamma\vdash\sigma$. When Σ is $\varnothing$, we write $\tau\triangleright\sigma$ for $(\varnothing\vdash\tau)\triangleright\sigma$ and recover the usual notation for type coercions. For example, $\alpha\triangleright\tau$ means that α can be coerced to τ (*e.g.* α is a subtype of τ). The coherence proposition of the constrained kind $\{\alpha:\kappa\mid\mathrm{P}\}$, namely $\exists\,\{\alpha:\kappa\mid\mathrm{P}\}$, gives the usual existential proposition, because coherence corresponds to inhabitation and a type τ is in the constrained kind $\{\alpha:\kappa\mid\mathrm{P}\}$ if it is in κ and satisfies P.

Types are described on Figure 1. They are written τ or σ. They contain type variables α, arrow types $\tau\to\sigma$, product types $\tau\times\sigma$, coherent polymorphic types $\forall(\alpha:\kappa)\,\tau$, incoherent polymorphic types $\Pi(\alpha:\kappa)\,\tau$, recursive types $\mu\alpha\,\tau$, the top type $\top$, and the bottom type $\bot$.

Types also contain the unit object $\langle\rangle$, pairs of types $\langle\tau,\tau\rangle$, and projections $\pi_{\mathrm{i}}\,\tau$ to construct and project type sequences. We define the projections of pairs $\pi_{\mathrm{i}}\,\langle\tau_1,\tau_2\rangle$ to be equal to the corresponding components τ_i. Type equality is then closed by reflexivity, symmetry, transitivity, and congruence for all syntactical constructs. This defines equality judgments on types ($\tau_1=\tau_2$), kinds ($\kappa_1=\kappa_2$), and propositions ($\mathrm{P}_1=\mathrm{P}_2$), which are used in typing rules below. Notice that equality is never applied implicitly.

We use environments to approximate the behavior of variables. The syntax of environments, written Γ, is described on Figure 1. Environments are lists of type binders $(\alpha:\kappa)$ and term binders $(x:\tau)$. We write Σ for environments that do not contain term bindings. We also use lists of propositions, written Θ, called coinduction environments, to keep track of which propositions can be used coinductively.

Let t be a type, a kind, a proposition, a typing environment, a sequence, or a set of such objects. We write $\mathsf{fv}(t)$ the set of free variables of t, defined in the obvious way, and $t[\alpha\leftarrow\tau]$ for the capture avoiding substitution of τ for the variable α in t. All objects t are taken up to α-conversion of their bound variables.

We assume that environments are well-scoped. That is, $\Gamma,(x:\tau)$ can only be formed when $x\notin\mathsf{dom}(\Gamma)$ and $\mathsf{fv}(\tau)\subseteq\mathsf{dom}(\Gamma)$; and $\Gamma,(\alpha:\kappa)$ can only be formed when $\alpha\notin\mathsf{dom}(\Gamma)$ and $\mathsf{fv}(\kappa)\subseteq\mathsf{dom}(\Gamma)$. Similarly, $\Gamma;\Theta$ requires $\mathsf{fv}(\Theta)\subseteq\mathsf{dom}(\Gamma)$.

C. Typing judgments

Types, kinds, and propositions are recursively defined and so are their typing judgments. We actually have the following judgments all recursively defined:

$\Gamma\vdash a:\tau$	term	$\Gamma\vdash\kappa$	kind coherence
$\Gamma;\Theta\vdash\mathrm{P}$	prop.	$\Gamma\vdash\Sigma$	environment coherence
$\Gamma\vdash\tau:\kappa$	type	$\Gamma\Vdash\kappa$	kind well-formedness
		$\Gamma\Vdash\mathrm{P}$	prop. well-formedness

We assume that judgments are always well-scoped: free variables of objects appearing on the right of the turnstile must be bound in the typing environment Γ.

Auxiliary judgments: The main two judgments are for terms and coercions. Others are auxiliary judgments and we describe them first.

The kind judgment $\Gamma\vdash\kappa$ states that the kind κ is coherent relative to the environment κ. This judgment is actually

equivalent to the proposition judgment $\Gamma \vdash \exists\,\kappa$ that will be explained below. The environment coherence judgment $\Gamma \vdash \Sigma$ checks that every kind appearing in Σ is coherent in the environment that precedes it. It is defined by the two rules:

$$\Gamma \vdash \varnothing \qquad \frac{\Gamma \vdash \Sigma \qquad \Gamma, \Sigma \vdash \kappa}{\Gamma \vdash \Sigma, (\alpha : \kappa)}$$

Kind and proposition well-formedness are recursively scanning their subexpressions for all occurrences of coercion propositions $(\Sigma \vdash \tau) \triangleright \sigma$ to ensure that Σ, τ, and σ are well-typed, as described by the following rule:

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

See the extended version for the complete description of well-formedness rules.

The type judgment $\Gamma \vdash \tau : \kappa$ is defined in Figure 5, but we only present the most interesting rules. Rule TYPEPACK is used to turn a type τ of kind κ satisfying a proposition P into a type of the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$. Conversely, TYPEUNPACK turns back a type of the constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$ into one of kind κ, unconditionally. TYPEMU allows to build the recursive type $\mu\alpha\,\tau$, which can be formed whenever τ is productive as stated by the judgment $\alpha \mapsto \tau : \mathsf{wf}$. Other rules are omitted by lack of space.

Term typing rules: Following the tradition, we write $\Gamma \vdash a : \tau$ to mean that in environment Γ the term a has type τ. However, we would also like to write this $a : \Gamma \vdash \tau$ too and say that the term a has the typing $\Gamma \vdash \tau$, that is, a is approximated by the type τ whenever its free variables are in the approximations described by Γ. We will keep the standard notation to avoid confusion, but we will read the judgment as above when helpful. The judgment $\Gamma \vdash a : \tau$ implies that τ has kind $\star$ under Γ whenever Γ is well-formed.

Term typing rules are given on Figure 4. Observe that the first five rules are exactly the typing rules of the simply-typed λ-calculus.

The last two rules are for incoherent abstraction and application (they could be skip at first): Rule TERMBLOCK says that the program $\partial\, a$ whose evaluation is frozen may be typed with the incoherent polymorphic type $\Pi(\alpha : \kappa)\,\tau$ if a can be typed with τ in an extended context that assigns a well-formed kind κ to α. Notice that $\Gamma \Vdash \kappa$, as opposed to $\Gamma \vdash \kappa$, does not imply that the kind is coherent, but well-formed. Rule TERMUNBLCK is the counterpart of TERMBLOCK. If we have a term a of an incoherent polymorphic type $\Pi(\alpha : \kappa)\,\tau$, *i.e.* whose evaluation has been frozen and a type σ of kind κ, we know that the kind κ is inhabited by σ. Therefore, we may safely unfreeze a and give it the type $\tau[\alpha \leftarrow \sigma]$.

Rule TERMCOER is at the heart of our approach which delegates most of the logic of typing to the existence of appropriate typing coercions. The rule reads as follows: if a term a admits the typing $\Gamma, \Sigma \vdash \tau$ and there exists a coercion from τ to σ introducing Σ under Γ, which we write $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$, then the term a also admits the typing $\Gamma \vdash \sigma$. The presence of Σ allows the coercion to manipulate the typing context as well as the type, which is the reason for our generalization from type coercions to typing coercions. When Σ is $\varnothing$, the rule looks more familiar and resembles the usual subtyping rule: if a term a has type τ under Γ and there exists a coercion from the type τ to the type σ under Γ (which is written $\Gamma \vdash \tau \triangleright \sigma$), then the term a has also type σ under Γ.

This factorization of all rules but those of the simply-typed λ-calculus under one unique rule, namely TERMCOER, emphasizes that coercions are only decorations for terms. Rule TERMCOER annotates the term a to change its typing without changing its computational content, as the resulting term is a itself. This is only made possible by using *typing* coercions instead of *type* coercions.

Propositions typing judgment: The judgment $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ is in fact an abbreviation for $\Gamma; \varnothing \vdash (\Sigma \vdash \tau) \triangleright \sigma$, which is itself a special case of the more general judgment $\Gamma; \Theta \vdash \mathrm{P}$ when Θ is $\varnothing$ and P is $(\Sigma \vdash \tau) \triangleright \sigma$. Indeed, $(\Sigma \vdash \tau) \triangleright \sigma$ a particular proposition P stating the existence of a typing coercion from τ to σ introducing Σ. The proposition environment Θ contains additional hypotheses that can be used coinductively when proving that a coercion holds.

The proposition judgment is split into two figures, with rules for general propositions in Figure 6 and rules specific to coercion propositions in Figure 7. We first explain typing rules for general propositions.

Rule PROPEQ allows the use of type equality. Rule PROPVAR allows the use of a coinductive hypothesis P in Θ. This is written $\mathrm{P}^\dagger \in \Theta$ because propositions that are guarded are marked † in Θ and only those are safe to use coinductively.

In particular, rule PROPFIX which we do not usually find in type systems allows to prove a proposition by coinduction: if P is true assuming P in the unguarded coinduction environment, then P is true without this additional hypothesis. Coinductive propositions are introduced as unguarded so that they cannot be used directly, which would be ill-founded. Only some of the coercion rules (described below) allow coinduction to be guarded. The usual rules about recursive types that can be found in other type systems are derivable from this general rule (see Section III-C).

Rules PROPTRUE, PROPANDPAIR, and PROPANDPROJ are uninteresting. Rule PROPFORINTRO and PROPFORELIM are unsurprising. Rule PROPEXI allows to embed the coherence of a kind κ, *i.e.* the existence of a type inhabitant of kind κ as the proposition $\exists\,\kappa$. Rule PROPRES allows to extract a proposition from a type τ of a constrained kind $\{\alpha : \kappa \mid \mathrm{P}\}$, replacing the variable α of kind κ by the witness τ of kind κ.

Coercions: We now explain typing rules for coercion propositions. We may ignore the environment Θ in most cases, as it is just unused or transferred to the premises unchanged, except for the three η-expansion rules that mark the environment as guarded $\Theta^\dagger$ in their premises, therefore allowing coinductive uses of propositions Θ via Rule PROPFIX. These are the rules that decompose computational types that have a counterpart in terms, namely COERARR, COERPROD, and COERPI.

We now explain coercion rules ignoring Θ. Intuitively, the judgment $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ implies that any term that admits the typing $\Gamma, \Sigma \vdash \tau$ also admits the typing $\Gamma \vdash \sigma$. (The converse is not true as the coercion typing judgment is semantically incomplete.) One could expect this

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

Fig. 5. Type judgment relation (excerpt)

$$\textsc{PropEq}\ \frac{\Gamma; \Theta \vdash \mathrm{P} \qquad \mathrm{P} = \mathrm{P}' \qquad \Gamma \Vdash \mathrm{P}'}{\Gamma; \Theta \vdash \mathrm{P}'} \qquad \textsc{PropVar}\ \frac{\mathrm{P}^{\dagger} \in \Theta}{\Gamma; \Theta \vdash \mathrm{P}} \qquad \textsc{PropTrue}\ \Gamma; \Theta \vdash \top \qquad \textsc{PropAndPair}\ \frac{(\Gamma; \Theta \vdash \mathrm{P}_i)^{i \in \{1,2\}}}{\Gamma; \Theta \vdash \mathrm{P}_1 \wedge \mathrm{P}_2} \qquad \textsc{PropAndProj}\ \frac{\Gamma; \Theta \vdash \mathrm{P}_1 \wedge \mathrm{P}_2}{\Gamma; \Theta \vdash \mathrm{P}_i}$$

$$\textsc{PropForIntro}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, (\alpha : \kappa); \Theta \vdash \mathrm{P}}{\Gamma; \Theta \vdash \forall(\alpha : \kappa)\,\mathrm{P}} \qquad \textsc{PropForElim}\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma; \Theta \vdash \forall(\alpha : \kappa)\,\mathrm{P}} \qquad \textsc{PropRes}\ \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma; \Theta \vdash \mathrm{P}[\alpha \leftarrow \tau]} \qquad \textsc{PropExi}\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma; \Theta \vdash \exists\,\kappa} \qquad \textsc{PropFix}\ \frac{\Gamma \Vdash \mathrm{P}}{\Gamma; \Theta, \mathrm{P} \vdash \mathrm{P}}$$

Fig. 6. Proposition judgment relation

judgment to be of the form $(\Gamma, \Sigma \vdash \tau) \triangleright (\Gamma \vdash \sigma)$, or even $(\Gamma_1 \vdash \tau_1) \triangleright (\Gamma_2 \vdash \tau_2)$. However, in our notation, Σ describes environment actions under Γ in a compositional manner and eventually permits to go from Γ_1 to Γ_2.

The coercion typing rules can be understood under the light of Rule TermCoer. The first two rules, CoerRefl and CoerTrans, close the coercion relation by reflexivity and transitivity. To understand CoerTrans let's take a term a with typing $\Gamma, \Sigma_2, \Sigma_1 \vdash \tau_1$. Applying Rule TermCoer with the second premise of Rule TermTrans ensures that the term a admits the typing $\Gamma, \Sigma_2 \vdash \tau_2$. Applying Rule TermCoer again with the first premise of Rule TermTrans, ensures that a admits the typing $\Gamma \vdash \tau_3$ as if we have applied Rule TermCoer to the original typing of a with the conclusion of Rule CoerTrans.

The Rule CoerWeak implements a form of weakening. It tells that if any term of typing $\Gamma, \Sigma \vdash \tau$ can be seen as $\Gamma \vdash \sigma$, then any term of typing $\Gamma \vdash \tau$ can also be seen as $\Gamma \vdash \sigma$. Since weakening holds for term judgments, we can do the following reasoning to justify this rule. Assume that the premise $\Gamma, \Sigma \vdash \tau$ holds; we argue that the conclusion should also hold. Indeed, a term that admits the typing $\Gamma \vdash \tau$ also have typing $\Gamma, \Sigma \vdash \tau$ by weakening; therefore, by the premise of Rule CoerWeak, it must also have typing $\Gamma \vdash \sigma$. However, this reasoning is mathematical and based on our interpretation of coercions: Rule CoerWeak is required as it is would not be derivable from other rules—not even admissible—if we removed it from the definition. Notice that this is the only rule that removes binders.

The rules CoerBot and CoerTop close the coercion relation with extrema. For any typing $\Gamma \vdash \tau$, there is a smaller typing, namely $\Gamma \vdash \bot$, and a bigger typing, namely $\Gamma \vdash \top$.

The rules CoerProd, CoerArr, and CoerPi close the coercion relation by η-expansion, which is the main feature of subtyping. Here, η-expansion is generalized to typing coercions instead of type coercions. The η-expansion rules describe how the coercion relation goes under computational type constructors, *i.e.* those of the simply-typed λ-calculus. Interestingly, the η-expansion rules for erasable type constructors can be derived as their introduction and elimination rules are already coercions.

Intuitively, η-expansion rules can be understood by decorating the η-expansion context with coercions at their respective type constructor. These coercions are erasable because the η-expansion of a term has the same computational behavior as the term itself.

For example, consider the η-expansion context for the arrow type $\lambda x\,([]\,x)$. Placing a term with typing $\Gamma, \Sigma \vdash \tau' \to \sigma'$ in the hole, we may give $\lambda x\,([]\,x)$ the typing $\Gamma \vdash \tau \to \sigma$ provided a coercion of type $\Gamma, \Sigma \vdash \tau \triangleright \tau'$ is applied around x. The result of the application has typing $\Gamma, \Sigma \vdash \sigma'$ which can in turn be coerced to $\Gamma \vdash \sigma$ if there exists a coercion of type $\Gamma \vdash (\Sigma \vdash \sigma') \triangleright \sigma$. Thus, the η-expansion has typing $\Gamma \vdash \tau \to \sigma$. While the coercion applied to the result of the application may bind variables Σ for the hole (and the argument), the coercion applied to the variable x needs not bind variables, since the variable x could not use them anyway.

Rules CoerGen and CoerInst, implement the main feature of the language, namely simultaneous coherent coercion abstractions. Intuitively, Rule CoerGen combines several type and coercion abstractions. This is however transparent in rule CoerGen since the simultaneous abstractions are grouped in the kind κ. Hence, this rule looks like a standard generalization rule. The only key here is the left premise that requires the coercion to be coherent. Rule CoerInst is the counter part of CoerGen: it instantiates the abstraction by a type of the right kind. Notice that CoerGen is the only rule using typing coercions in a crucial way and that could not be presented as a coercion if we just had type coercions.

Rule CoerPi is an η-expansion rule and should be understood by typing the η-expansion of the incoherent polymorphic type $\partial\,([]\,\diamondsuit)$, inserting a coercion around the incoherent application. Placing a term with typing $\Gamma, \Sigma \vdash \Pi(\alpha' : \kappa')\,\tau'$ in the hole, we may first apply weakening to get a typing of the form $\Gamma, (\alpha : \kappa), \Sigma \vdash \Pi(\alpha' : \kappa')\,\tau'$. By instantiation (Rule TermInst), we get a typing $\Gamma, (\alpha : \kappa), \Sigma \vdash \tau'[\alpha' \leftarrow \sigma']$ provided $\Gamma, (\alpha : \kappa), \Sigma \vdash \sigma' : \kappa'$. Applying a coercion $(\Sigma \vdash \tau'[\alpha' \leftarrow \sigma']) \triangleright \tau$ (Rule TermCoer), we obtain the typing $\Gamma, (\alpha : \kappa) \vdash \tau$, which we may generalized (Rule TermGen) to obtain the typing $\Gamma \vdash \Pi(\alpha : \kappa)\,\tau$ of $\partial\,([]\,\diamondsuit)$. Notice that, as Rule CoerGen, we do not require coherence for the kind κ, just its well-formedness. However, we require the coherence of the type environment extension Σ under Γ. This is a very important premise because we do not want the incoherence

$$\textsc{RecVar}\ \frac{}{\alpha \mapsto \alpha : \mathsf{ne}} \qquad \textsc{RecArr}\ \frac{(\alpha \mapsto \tau_i : \mathsf{ne})^{i\in\{1,2\}}}{\alpha \mapsto \tau_1 \to \sigma_2 : \mathsf{wf}} \qquad \textsc{RecProd}\ \frac{(\alpha \mapsto \tau_i : \mathsf{ne})^{i\in\{1,2\}}}{\alpha \mapsto \tau_1 \times \sigma_2 : \mathsf{wf}}$$

$$\textsc{RecFor}\ \frac{\alpha \notin \mathsf{fv}(\kappa) \qquad \alpha \mapsto \tau : \chi}{\alpha \mapsto \forall(\beta : \kappa)\,\tau : \chi} \qquad \textsc{RecPi}\ \frac{\alpha \notin \mathsf{fv}(\kappa) \qquad \alpha \mapsto \tau : \mathsf{ne}}{\alpha \mapsto \Pi(\beta : \kappa)\,\tau : \mathsf{wf}}$$

$$\textsc{RecMu}\ \frac{\beta \mapsto \tau : \mathsf{wf} \qquad \alpha \mapsto \tau : \chi}{\alpha \mapsto \mu\beta\,\tau : \chi} \qquad \textsc{RecWf}\ \frac{\alpha \notin \mathsf{fv}(\tau)}{\alpha \mapsto \tau : \mathsf{wf}} \qquad \textsc{RecNe}\ \frac{\alpha \mapsto \tau : \mathsf{wf}}{\alpha \mapsto \tau : \mathsf{ne}}$$

Fig. 8. Well-foundedness judgment relation

of κ to leak in Σ and thus under the coercion, because η-expansions are coercions and thus erasable.

Rules CoerFold and CoerUnfold are the usual folding and unfolding of recursive types, which give the equivalence between $\mu\alpha\,\tau$ and $\tau[\alpha \leftarrow \mu\alpha\,\tau]$. Interestingly, the usual rules for reasoning on recursive types [7] are admissible using PropFix (we write $\tau_1 \lhd\rhd \tau_2$ for $\tau_1 \rhd \tau_2 \wedge \tau_2 \rhd \tau_1$):

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

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.

Finally, the well-foundedness judgment, written $\alpha \mapsto \tau : \chi$, means that $\alpha \mapsto \tau$ is well-founded when χ is wf or non-expansive when χ is ne. The rules are unsurprising. The most interesting rules are RecArr, RecProd, and RecPi which ensure well-foundedness provided the components are themselves non-expansive. Conversely, rules RecFor and RecMu just transfer both well-foundedness and non-expansiveness from their components. For polymorphic types the abstract variable should not appear in its bound to ensure well-foundedness or non-expansiveness. Of course, recursive types can only be well-formed if they are well founded (left premise of RecMu).

IV. Properties

System F_{cc} is sound: erroneous terms never appear during the evaluation of well-typed programs. Moreover, in the absence of recursive types and coinduction, all reduction paths terminate. Finally, in the absence of incoherent abstraction, the reduction is confluent.

However, we loose confluence in the presence of incoherent abstraction, since a partially evaluated term a may be substituted under an incoherent abstraction, after which further reductions won't be permitted inside a. This is a well-known problem when mixing strong and weak reduction strategies with also a well-known solution [8]: confluence can easily be restored by using explicit substitutions to hold the substitution at the entry of an incoherent abstraction until the abstraction is applied and evaluation may be resumed. A variant of this solution (in the spirit of [9]) is to add in the language an incoherent weakening construct to cancel the freezing effect of the incoherent abstraction: reduction could still be performed in the weakened part of the incoherent abstraction. This avoids explicit substitutions, but complicates reduction contexts that have to look under incoherent abstractions for occurrences of incoherent weakenings.

Subject reduction is also lost in F_{cc}—although the language is sound. This is just the consequence of our decision to move from an explicit calculus of coercions to implicit coercions. The type system is just not rich enough to track after reduction the invariants that can be expressed by coercions on source programs, typically when coercions are used in wedges.

The language F_{cc} has been formalized in Coq and its properties have been mechanically verified[2].

V. Expressivity

The language F_{cc} is more expressive than F_{ι}^{p}: apart from the change of presentation, moving from type coercions to typing coercions and from explicit coercions to implicit coercions, the only significant change is for type and coercion abstraction: the new construct of F_{cc} which by design generalizes the two forms of coercion abstraction in F_{ι}^{p}. Indeed, we can choose $\bot$ (*resp.* $\top$) to witness coercions that are parametric in their domain (*resp.* range). Therefore the languages $\mathsf{F}_{<:}$, ML^{F}, and F_{η} which are subsumed by F_{ι}^{p} can also be seen as sublanguages of F_{cc}.

A. Encoding subtyping constraints

We claim that languages with ML-like subtyping constraints [10] can be simulated in F_{cc}. With subtyping constrains, term judgments may be written $A \vdash e : \tau \mid C$ where A is the environment, e the expression, τ its type, and C is a sequence of subtyping constraints, *e.g.* as in [11].

To ease the embedding of subtyping constraints in F_{cc}, we slightly abuse of notations. First, we see let-bindings as the usual syntactic sugar for redexes. We write $\overline{\alpha}$ for a sequence of variables $\alpha_1 \ldots \alpha_n$ where n is left implicit. Given a sequence of variables $\overline{\alpha}$, we see α_i as $\alpha.i$, the i'th projection of α. We write $(\overline{\alpha} \mid \mathrm{P})$ as a shorthand for the type binding $(\alpha : \{\alpha : \star^n \mid \mathrm{P}\})$ where n is the size of $\overline{\alpha}$. Finally, we see constraint type schemes $\forall\overline{\alpha}.C \Rightarrow \tau$ as the coherent polymorphic type $\forall(\overline{\alpha} \mid C)\,\tau$.

A term judgment $A \vdash e : \tau \mid C$ can then be seen as the F_{cc} judgment $(\bar{\alpha} \mid C), A \vdash e : \tau$ where $\overline{\alpha}$ are free variables of A, C, and τ. Notice that the environment in the translation of judgments is always of the form $(\overline{\alpha} \mid C), A$ composed of a single abstraction block $(\overline{\alpha} \mid C)$ followed by term bindings A.

Type systems with subtyping constraints use two notions, solvability and consistency, that coincide in ML. Solvability means that one can find a ground instance for type variables that solves the constraints. Consistency means that transitive and congruence closure of the set of constraints does not contain inconsistencies.

We claim that solvability of a set of constraints C implies the consistency of the translation of C, since it amounts to

[2] See http://gallium.inria.fr/~remy/coercions/ for details.

$$\textsc{CoerRefl}\ \frac{\Gamma \vdash \tau : \star}{\Gamma;\Theta \vdash \tau \triangleright \tau} \qquad \textsc{CoerTrans}\ \frac{\Gamma,\Sigma';\Theta \vdash (\Sigma \vdash \tau) \triangleright \tau' \qquad \Gamma;\Theta \vdash (\Sigma' \vdash \tau') \triangleright \tau''}{\Gamma;\Theta \vdash (\Sigma',\Sigma \vdash \tau) \triangleright \tau''}$$

$$\textsc{CoerWeak}\ \frac{\Gamma;\Theta \vdash (\Sigma \vdash \tau) \triangleright \sigma}{\Gamma;\Theta \vdash \tau \triangleright \sigma} \qquad \textsc{CoerTop}\ \frac{\Gamma \vdash \tau : \star}{\Gamma;\Theta \vdash \tau \triangleright \top}$$

$$\textsc{CoerBot}\ \frac{\Gamma \vdash \tau : \star}{\Gamma;\Theta \vdash \bot \triangleright \tau} \qquad \textsc{CoerArr}\ \frac{\Gamma,\Sigma;\Theta^\dagger \vdash \tau \triangleright \tau' \qquad \Gamma;\Theta^\dagger \vdash (\Sigma \vdash \sigma') \triangleright \sigma}{\Gamma;\Theta \vdash (\Sigma \vdash \tau' \to \sigma') \triangleright \tau \to \sigma}$$

$$\textsc{CoerProd}\ \frac{(\Gamma;\Theta^\dagger \vdash (\Sigma \vdash \tau_i) \triangleright \sigma_i)^{i\in\{1,2\}}}{\Gamma;\Theta \vdash (\Sigma \vdash \tau_1 \times \tau_2) \triangleright \sigma_1 \times \sigma_2}$$

$$\textsc{CoerPi}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma \vdash \Sigma \qquad \Gamma,(\alpha:\kappa),\Sigma \vdash \sigma' : \kappa' \qquad \Gamma,(\alpha:\kappa);\Theta^\dagger \vdash (\Sigma \vdash \tau'[\alpha' \leftarrow \sigma']) \triangleright \tau}{\Gamma;\Theta \vdash (\Sigma \vdash \Pi(\alpha':\kappa')\,\tau') \triangleright \Pi(\alpha:\kappa)\,\tau} \qquad \textsc{CoerGen}\ \frac{\Gamma \vdash \kappa \qquad \Gamma,(\alpha:\kappa) \vdash \tau : \star}{\Gamma;\Theta \vdash (\alpha:\kappa \vdash \tau) \triangleright \forall(\alpha:\kappa)\,\tau}$$

$$\textsc{CoerInst}\ \frac{\Gamma,(\alpha:\kappa) \vdash \tau : \star \qquad \Gamma \vdash \sigma : \kappa}{\Gamma;\Theta \vdash \forall(\alpha:\kappa)\,\tau \triangleright \tau[\alpha \leftarrow \sigma]} \qquad \textsc{CoerUnfold}\ \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma,(\alpha:\star) \vdash \tau : \star}{\Gamma;\Theta \vdash \mu\alpha\,\tau \triangleright \tau[\alpha \leftarrow \mu\alpha\,\tau]} \qquad \textsc{CoerFold}\ \frac{\alpha \mapsto \tau : \mathsf{wf} \qquad \Gamma,(\alpha:\star) \vdash \tau : \star}{\Gamma;\Theta \vdash \tau[\alpha \leftarrow \mu\alpha\,\tau] \triangleright \mu\alpha\,\tau}$$

Fig. 7. Coercion judgment relation

exhibit type witnesses such that the constraints hold. These witnesses lie in a syntax with simple types and recursive types. Moreover, since solvability is equivalent to consistency, we conclude that consistency is equivalent to coherence.

The two interesting typing judgments for subtyping constraints are for let-binding and subsumption. These are as follows and are derivable in F_{cc}:

$$\frac{\overline{\alpha} \vdash \overline{\sigma} \qquad \overline{\alpha} \vdash C'[\overline{\beta} \leftarrow \overline{\sigma}] \qquad (\overline{\alpha} \mid C),\Gamma,(x : \forall(\overline{\beta},C')\tau) \vdash b : \rho \qquad (\overline{\alpha},\overline{\beta} \mid C'),\Gamma \vdash a : \tau}{(\overline{\alpha} \mid C),\Gamma \vdash (\lambda x\, b)\, a : \rho}$$

$$\frac{(\overline{\alpha} \mid C),\Gamma \vdash a : \tau \qquad (\overline{\alpha} \mid C') \vdash C \wedge \tau \triangleright \sigma}{(\overline{\alpha} \mid C'),\Gamma \vdash a : \sigma}$$

However, there remain two differences with the way subtyping constraints are usually handled. A judgment $A \vdash e : \tau \mid C$ is valid when C is *consistent* while our corresponding judgment $(\mathsf{fv}(A,C,\tau) \mid C), A \vdash e : \tau$ is valid when C is *solvable*, *i.e.* $\vdash (\mathsf{fv}(C) \mid C)$, which must exhibit a substitution θ of domain $\mathsf{fv}(C)$ such that $\varnothing \vdash C\theta$. While consistency and solvability coincide in ML with subtyping constraints, this need not be the case. Consistency is a semantic property while solvability is a syntactic property. Using consistency instead of solvability, we only have to verify a property of the constraints without having to exhibit a concrete solution. Consistency is more flexible than solvability. In practice, it can also be checked more modularly.

We already have some flexibility to reason about coherence in F_{cc} using propositions and assumptions in the typing context. However, constraint entailment differs in both systems. In particular, we cannot express the decomposition of typing constrains, *e.g.* deduce the consistency of $\sigma \triangleright \sigma'$ from the consistency of $\tau \to \sigma \triangleright \tau \to \sigma'$, as is the case with subtyping constraints.

The reason is that subtyping constraints are syntactic and taken in a closed-world view: subtyping relations that cannot be expressed syntactically do not hold, which can be used to reinforce constraint entailment. Our approach in F_{cc} is semantic and syntactic coercions must be interpreted in the semantics. Since our semantics has more types and coercions than the syntax allows to build, some reasoning principles that would be true from a purely syntactic point of view will not hold in our semantics and thus cannot be added in the syntax. We are bound to an open-world view. Still, it would be interesting to see how our approach could be extended to allow a form of closed world view and express some negative information.

B. Encoding GADTs

Incoherent polymorphism is necessary for features that contain some form of dynamic typing, such as GADTs. It may also be a simplification for the programmer that does not have to provide the witness type that proves the coherence—but at his own risk of delaying type errors.

In this subsection we show how GADTs can be encoded with incoherent polymorphism and also how coherence and incoherent polymorphism can be interestingly mixed.

Incoherent polymorphism permits type abstraction for any well-formed kind: inhabited kinds, potentially inhabited kinds, and empty kinds. In the polymorphic type $\Pi(\alpha : \kappa)\,\tau$, the coherence of kind κ may depend over some type variable β of the type environment. Depending on how β is instantiated, the kind κ may or may not be inhabited.

Before we give a concrete example, let us first introduce existential types by their CPS encoding. Because we have two notions of polymorphism, coherent and incoherent, we also have two notions of existential types: we write $\exists(\alpha : \kappa)\,\tau$ for coherent existential types and $\Sigma(\alpha : \kappa)\,\tau$ for incoherent existential types[3] defined as follows.

$$\begin{aligned} \text{coherent:} \quad & \exists(\alpha:\kappa)\,\tau \stackrel{\mathsf{def}}{=} \forall\beta\,(\forall(\alpha:\kappa)\,(\tau \to \beta)) \to \beta \\ \text{incoherent:} \quad & \Sigma(\alpha:\kappa)\,\tau \stackrel{\mathsf{def}}{=} \forall\beta\,(\Pi(\alpha:\kappa)\,(\tau \to \beta)) \to \beta \end{aligned}$$

We define the pack and unpack term syntactic sugar for the coherent existential, and ipack and iunpack for their incoherent version. Notice that the body of the iunpack sugar is hidden under an incoherent type abstraction, and as such is allowed to be unsound because it cannot be reduced.

$$\begin{aligned} \mathsf{pack}\, a &\stackrel{\mathsf{def}}{=} \lambda x\, x\, a & \mathsf{unpack}\, a\, \mathsf{as}\, x\, \mathsf{in}\, b &\stackrel{\mathsf{def}}{=} a\,(\lambda x\, b) \\ \mathsf{ipack}\, a &\stackrel{\mathsf{def}}{=} \lambda x\, x \diamondsuit a & \mathsf{iunpack}\, a\, \mathsf{as}\, x\, \mathsf{in}\, b &\stackrel{\mathsf{def}}{=} a\,(\partial\, \lambda x\, b) \end{aligned}$$

[3] Σ here is a binder and has of course no connection with Σ used as typing environments.

Let's assume we have type-level functions and sum types. We can now define the following GADT, named Term, and with kind $\star \to \star$ (where $\tau \lhd\rhd \sigma$ stands for $\tau \rhd \sigma \wedge \sigma \rhd \tau$ as above):

$$\begin{array}{rcl} \mathsf{Term}\,\alpha & \stackrel{\mathsf{def}}{=} & \Sigma(\beta : \star \times \star \mid \alpha \lhd\rhd (\pi_1\,\beta \to \pi_2\,\beta))\,\alpha \\ & + & \exists\beta\,\mathsf{Term}\,(\beta \to \alpha) \times \mathsf{Term}\,\beta \end{array}$$

This GADT is the sum of an incoherent existential type and a coherent one. The incoherent existential type requires α to be an arrow type and stores a term of such type; it also names $\pi_1\,\beta$ the argument type and $\pi_2\,\beta$ its return type. The coherent existential type adds no constraint on α but stores a pair such that its first component applied to its second component is of type α; it names β the intermediate type. The Term GADT contains two constructors: one for the left-hand side of the sum injecting functions and one for the right-hand side of the sum freezing function applications. We can define its two constructors in the following manner:

$$\begin{array}{rcl} \mathsf{Lam}\,x & \stackrel{\mathsf{def}}{=} & \mathsf{inl}\,(\mathsf{ipack}\,x) \\ & : & \forall\alpha\,\forall\beta\,(\alpha \to \beta) \to \mathsf{Term}\,(\alpha \to \beta) \\ \mathsf{App}\,y\,x & \stackrel{\mathsf{def}}{=} & \mathsf{inr}\,(\mathsf{pack}\,\langle y, x\rangle) \\ & : & \forall\alpha\,\forall\beta\,\mathsf{Term}\,(\alpha \to \beta) \to \mathsf{Term}\,\alpha \to \mathsf{Term}\,\beta \end{array}$$

We can now define a recursive eval function taking a term of type $\mathsf{Term}\,\alpha$ and returning a term of type α for all type variable α. Said otherwise, eval has type $\forall\alpha\,(\mathsf{Term}\,\alpha \to \alpha)$. When the argument is on the left-hand side of the sum, eval simply unpacks it and returns it. When the argument is on the right-hand side of the sum, eval fist unpacks it as a pair and applies the evaluation of the first component to the evaluation of the second component. We thus use the incoherent version of unpacking on the left-hand side and the coherent version on the right-hand side.

$$\begin{array}{l} \mathsf{eval}\,x = \mathsf{case}\ x\ \mathsf{of} \\ \{\ \mathsf{inl}\,x_1 \mapsto \mathsf{iunpack}\,x_1\,\mathsf{as}\,y\,\mathsf{in}\,y \\ \mid\ \mathsf{inr}\,x_2 \mapsto \mathsf{unpack}\,x_2\,\mathsf{as}\,y\,\mathsf{in}\,(\mathsf{eval}\,(\pi_1\,y))\,(\mathsf{eval}\,(\pi_2\,y))\} \end{array}$$

Let's now suppose that we call eval with a term of type $\mathsf{Term}\,(\tau \times \sigma)$. This term is necessarily from the right-hand side of the sum because $\tau \times \sigma$ cannot be equivalent to an arrow type by consistency. However, in the first branch, in the body of the inconsistent unpack, we have access to the proposition $\tau \times \sigma \lhd\rhd \pi_1\,\beta \to \pi_2\,\beta$ which is inconsistent. This sort of inconsistency in some branches of case expressions is frequent with GADTs. Notice however, that we can reduce the second branch because we used a coherent existential type since there is a witness for β for any instantiation of α.

VI. Discussion

We first compare F_{cc} with our prior work and other related works; we then discuss language extensions and future works.

A. Comparison with F_ι^p

The closest work is of course our previous work on F_ι^p of which F_{cc} is an extension. The main improvement in F_{cc} is the ability to abstract other arbitrary constrains, but as a serious drawback one has to provide coercion witnesses to ensure the coherence.

Coherence is sufficient for type soundness, but in an explicit language of coercions it does not suffice for subject reduction, which also requires that the language has a rich *syntactic* representation to keep track during the reduction of invariants expressed by coercions. Our approach in F_{cc} is to avoid the need for decomposing abstract coercions into smaller ones by presenting an implicit version of the language. This also avoids introducing new coercion constructs in the language and their associated typing rules—which we failed to prove to be sound by syntactic means in an explicit language of coercions.

Therefore, moving from F_ι^p to F_{cc} has a cost—the lost an explicit calculus of coercions with subject reduction. Of course, one can still introduce explicit syntax for coercion typing rules in the source so as to ease type checking, but terms with explicit coercions will not have reduction rules in F_{cc}.

An interesting question is whether there are interesting languages between F_ι^p and F_{cc} that would still have a (relatively simple) calculus of explicit coercions. If we restrict to certain forms of coercions, instead of general coercions, the question of coherence may be much simpler. For example, one could just consider equality coercions as in the language FC.

In F_{cc}, we simultaneously abstract over a group of type variables and coercions that constrain those variables. The choice of grouping must be such that the group is coherence for all possible instantiation of variables in the context.

We have also explored a syntactically more atomic version of F_{cc} where type and coercion abstraction are separate constructs as in F_ι [1]. Namely, the usual type abstraction $\forall\alpha\,\tau$ and coercion abstraction $(\tau_1 \rhd \tau_2) \Rightarrow \sigma$—a term of type σ under the hypothesis that $\tau_1 \rhd \tau_2$ holds. For instance, this would permit to write a function of type $\forall\alpha\,(\alpha \to (\tau_1 \rhd \tau_2) \Rightarrow \sigma)$ and apply it to a type parameter, then to a value of type α, and finally to a coercion. However, this additional flexibility is negligible, these are just η-expansion variants of terms in F_{cc}. Moreover, related type abstractions and coercions should still be checked *simultaneously*. That is, even if the arguments are passed separately, the typing derivation must maintain a notion of grouping underneath so as to check for coherence. The solution in F_{cc} seems a better compromise between simplicity and expressiveness.

B. Comparison with other works

To the best of our knowledge there is no previous work considering typing coercions. However, the use of type coercions to study features of type system is not at all new. Coercions have also been used in the context of subtyping, but without the notion of abstraction over coercions.

The heavy use of coercions in FC, the core language of GHC, was one of our initial motivations for studying coercions in a general setting. In FC, only toplevel coercion axioms coming from type families and newtypes are checked for consistency. Local coercion abstractions are not. This is safe because all coercion abstractions in FC freeze the evaluation. This simplifies the meta-theory but at some significant cost, since the evaluation must be delayed to never reduce in a potentially inconsistent context. Our inconsistent coercions

largely coincide to—and was inspired by coercions in FC. In return what F_{cc} offers in addition is the ability to choose between coherent and incoherent coercion abstractions so that coherent coercions could be expressed as such and thus not freeze the evaluation and still bring more static guarantees to the user. While F_{cc} treats coercions in the general case, FC considers only a very specific case of equality constraints—with additional restrictions—so that *e.g.* coherence of toplevel coercions axioms can be checked automatically.

Coercions have also been used to eliminate function call overhead from datatype constructors in [12]: the folding and unfolding of datatype definitions are done using erasable coercions, thus with no run-time effect or hidden cost while preserving the semantics.

Recursive coercions have also been used to provide coercion iterators over recursive structures [13]. However, the motivations are quite different and coercions are only used as a tool to compile bounded quantification away into intersection types.

C. Extensions and variations

Higher-order types: We introduced F_{cc} as an extension of System F, thus restricting ourselves to second-order polymorphism. We have verified that our approach extends with higher-order types as in F_{ω}.

Intersection types: It should also be possible to add intersection types. (Our semantics already has them.). Following the work of Wells [14] on *branching types*, it would then be interesting to have intersection types as *branching typings*.

Existential types: We haven't included existential types in F_{cc} and just used their standard CPS encoding into universal types. Adding primitive existential types would also be interesting but not immediate. This is not so surprising as the combination of existential types with strong reduction strategies is known to raise difficulties. A solution we have started to investigate is to use a reduction strategy equivalent to strong reduction but where only terms starting with a constructor are substituted. This relates to existing calculi with explicit substitutions and generalizes call-by-need calculi to strong reduction.

Side effects: We have studied a calculus of coercions in an ideal theoretical setting, but we do not foresee any problem in applying this to a real programming language with impure features such as side effects. We are not bound to a strong reduction strategy, but on the opposite have all the freedom to choose weak reduction strategies for term abstractions. In the presence of side-effects, we would have a form of value-restriction, allowing type and coercion abstractions only on value forms. We do not expect this to raise new problems with coercions—nor do we expect them to disappear!

From implicit to explicit non-reducible coercions: When coercions are left implicit they must be inferred—as well as coercion witnesses, which is obviously undecidable in the general case. (Typechecking in F_{η} or in the most expressive variant of $\mathsf{F}_{<:}$ are already undecidable.) In practice, the user should provide both of them explicitly—or at least provide sufficient information so that they can be inferred. Hence, a surface language would probably have explicit coercions—just for typing purposes—and coercions should be dropped after typechecking. Indeed, we do not describe how coercions and, in particular, wedges can be reduced. In this setting, our soundness result still applies—reduction will not introduce erroneous programs—but it does not imply subject reduction: it may happen that after reduction there is no way to redecorate the residual program with explicit coercions to make it well-typed. We believe that this is the price to pay for the generality offered by our approach.

Conclusion

Generalizing the notion of type coercions to typing coercions, we have proposed a type system where the distinction between the computation and the typing aspects of terms have been completely separated. It subsumes many features of existing type systems including subtyping, bounded quantification, instance bounded quantification, and subtyping constraints.

The soundness of our calculus has been proved using the step-indexed semantics technique which we have adapted to work for calculi with strong reduction strategies.

As for coercions, several research directions remain to be explored. Hopefully, new type system features such dependent types could still be added. A surface language with explicit coercions annotations is a prerequisite for decidable type checking. Variations on constraints allowing closed-world views as well as restrictions to recover subject reduction are worth further investigation.

References

[1] J. Cretin and D. Rémy, "On the power of coercion abstraction," in *Proceedings of the annual symposium on Principles Of Programming Languages*, 2012.

[2] J. C. Mitchell, "Polymorphic type inference and containment," *Information and Computation*, vol. 2/3, no. 76, 1988.

[3] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov, "An extension of system f with subtyping," *Information and Computation*, vol. 109, no. 1/2, pp. 4–56, 1994.

[4] D. Le Botlan and D. Rémy, "Recasting MLF," *Information and Computation*, vol. 207, no. 6, 2009.

[5] J. Cretin and D. Rémy, "System f with coercion constraints," Jan 2014, avaliable at http://gallium.inria.fr/ remy/coercions.

[6] J. Cretin, "Erasable coercions: a unified approach to type systems," Ph.D. dissertation, Université Paris Diderot, Paris 7, 2014, to appear.

[7] R. Amadio and L. Cardelli, "Subtyping recursive types," *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, vol. 15, no. 4, pp. 575–631, 1993.

[8] J.-J. Lévy and L. Maranget, "Explicit substitutions and programming languages," in *Foundations of Software Technology and Theoretical Computer Science*, ser. LNCS, 1999, vol. 1738, pp. 181–200.

[9] T. Blanc, J.-J. Lévy, and L. Maranget, "Sharing in the weak lambda-calculus," in *Processes, Terms and Cycles: Steps on the Road to Infinity*, ser. LNCS, 2005, vol. 3838, pp. 70–87.

[10] M. Odersky, M. Sulzmann, and M. Wehr, "Type inference with constrained types," *Theory and Practice of Object Systems*, vol. 5, no. 1, pp. 35–55, 1999.

[11] F. Pottier, "Simplifying subtyping constraints," in *Proceedings of the International Conference on Functional Programming*, 1996.

[12] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng, "Typed compilation of recursive datatypes," in *Workshop on Types in Language Design and Implementation (TLDI)*, 2003.

[13] K. Crary, "Typed compilation of inclusive subtyping," in *Proceedings of the International Conference on Functional Programming*, 2000.

[14] J. B. Wells and C. Haack, "Branching types," in *Proc. of the European Symposium On Programming Languages and Systems*, 2002.