System F with Coercion Constraints

Julien Cretin, Didier Rémy

System F with Coercion Constraints

Julien Cretin, Didier Rémy

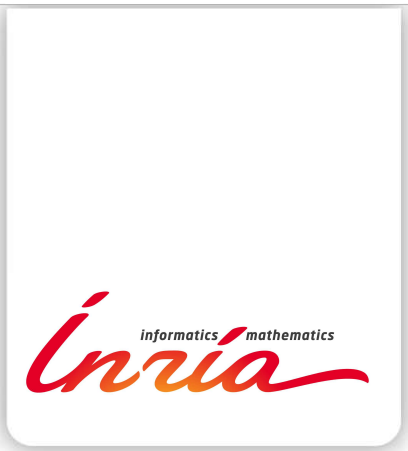
RESEARCH REPORT

N° 8456

January 2014

Project-Teams Gallium

ISSN 0249-6399 ISRN INRIA/RR--8456--FR+ENG



System F with Coercion Constraints

Julien Cretin, Didier Rémy

Project-Teams Gallium

Research Report n° 8456 — January 2014 — 33 pages

Abstract: We present a second-order λ-calculus with coercion constraints that generalizes a previous extension of System F with parametric coercion abstractions [Cretin and Rémy(2012)] 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.

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

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Système F avec des contraintes de coercion

Résumé : Nous présentons un λ-calcul de second ordre avec des contraintes de coercions qui généralisent une extension précédente du Système F avec des abstractions de coercions paramétriques [Cretin and Rémy(2012)] en permettant des abstractions multiples mais simultanées sur les types et les coercions, ainsi que des coercions récursives et des types équi-récursifs. Cela permet de présenter de façon uniforme plusieurs fonctionnalités des systèmes de types auparavant étudiées séparément: le confinement de types, le polymorphisme bornée et bornée par instance, qui sont déjà codables avec l'abstraction de coercions paramétrique, et les contraintes de sous-typage à la ML. Notre cadre permet une séparation claire entre les constructions du langage avec et sans contenu calculatoire. Nous distinguons également les coercions cohérentes qui sont entièrement effaçables des coercions incohérentes qui suspendent l'évaluation, et permettent le codage des GADTs.

Techniquement, les coercions de *type*, témoins d'une relation de sous-typage entre les types, sont remplacées par une notion plus expressive de coercions de *typages*, témoins d'une relation d'inclusion entre les typages, *e.g.* des paires composées d'un environnement de typage et d'un type. Le calcul est équipé d'une relation de réduction forte permettant la réduction sous les abstractions—mais nous introduisons également une forme de réduction faible car la réduction ne peut pas être poursuivie sous les abstractions de types incohérentes. La sûreté du typage est prouvée en adaptant une technique sémantique de types indexés aux stratégies de réduction forte, en plaçant les indices à l'intérieur des termes afin de contrôler les étapes de réduction de façon interne.

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

Contents

1 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 [Mitchell(1988)]. 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. $\mathsf{F}_{<:}$ [Cardelli et al.(1994)Cardelli, Martini, Mitchell, and Scedrov] 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 MLF [Le Botlan and Rémy(2009)] 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 [Cretin and Rémy(2012)], we introduced F^p_ι, 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 MLF instance-bounded polymorphism, uniformly. Following a general and systematic approach to coercions lead to an expressive and modular design. However, F^p_ι 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^p_ι 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 MLF) 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 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 §2. We present F_{cc} formally in §3. We introduce our variant of step-indexed denotational semantics in §4 and apply it to prove the soundness of our calculus in Section §5. We discuss the expressivity of F_{cc} in §6 and differences with our previous work and other related works as well as future works in §7.

The language F_{cc} and its soundness and normalization proofs have been formalized and mechanically verified in Coq.[1]

2 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_{ι}^{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.

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

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

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_t^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.

Recursive coercions Recursive types are another interesting new feature of F_{cc}. They are essential in practice and also very useful in theory, as they model several advanced features of programming languages, such as objects or closures. Recursive types are technically challenging however. Thus, they are often presented with some restrictions, for instance, restricting to positive recursion or to the folding-unfolding rules, which are easier to formalize. However, general recursive types are already needed in OCaml or in ML with subtyping constraints. We therefore follow a general approach to recursive types to cover these useful cases. The introduction of recursive coercions is then natural to operate on expressions with recursive types. Quite interestingly, this brings an induction principle for reasoning on recursive types from which the most general subtyping rules for recursive types [Amadio and Cardelli(1993)] are admissible (§3.3).

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 [Cretin and Rémy(2012)], 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.

3 Language definition

3.1 The syntax and semantics of terms

The syntax of the language is given in Figure 2. 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.

The reduction rules are given on Figure 4. 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

$$\textsc{WfKStar}\ \Gamma \Vdash \star \qquad \textsc{WfKOne}\ \Gamma \Vdash 1 \qquad \textsc{WfKProd}\ \frac{\Gamma \Vdash \kappa_1 \qquad \Gamma \Vdash \kappa_2}{\Gamma \Vdash \kappa_1 \times \kappa_2} \qquad \textsc{WfKRes}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, (\alpha : \kappa) \Vdash \mathrm{P}}{\Gamma \Vdash \{\alpha : \kappa \mid \mathrm{P}\}} \qquad \textsc{WfPTrue}\ \Gamma \Vdash \top$$

$$\textsc{WfPAnd}\ \frac{\Gamma \Vdash \mathrm{P}_1 \qquad \Gamma \Vdash \mathrm{P}_2}{\Gamma \Vdash \mathrm{P}_1 \wedge \mathrm{P}_2} \qquad \textsc{WfPCoer}\ \frac{\Gamma \vdash \Sigma \qquad \Gamma, \Sigma \vdash \tau : \star \qquad \Gamma \vdash \sigma : \star}{\Gamma \Vdash (\Sigma \vdash \tau) \triangleright \sigma} \qquad \textsc{WfPExi}\ \frac{\Gamma \Vdash \kappa}{\Gamma \Vdash \exists \kappa} \qquad \textsc{WfPFor}\ \frac{\Gamma, (\alpha : \kappa) \Vdash \mathrm{P}}{\Gamma \Vdash \forall (\alpha : \kappa)\, \mathrm{P}}$$

$$\textsc{WfGNil}\ \Gamma \Vdash \varnothing \qquad \textsc{WfGType}\ \frac{\Gamma \Vdash \Gamma' \qquad \Gamma, \Gamma' \Vdash \kappa}{\Gamma \Vdash \Gamma', (\alpha : \kappa)} \qquad \textsc{WfGTerm}\ \frac{\Gamma \Vdash \Gamma' \qquad \Gamma, \Gamma' \vdash \tau : \star}{\Gamma \Vdash \Gamma, (x : \tau)} \qquad \textsc{WfYNil}\ \Gamma \Vdash \varnothing \qquad \textsc{WfYConsA}\ \frac{\Gamma \Vdash \Theta \qquad \Gamma \Vdash \mathrm{P}}{\Gamma \Vdash \Theta, \mathrm{P}}$$

$$\textsc{WfYConsB}\ \frac{\Gamma \Vdash \Theta \qquad \Gamma \Vdash \mathrm{P}}{\Gamma \Vdash \Theta, \mathrm{P}^\dagger}$$

Figure 1: Well-formedness of kinds and propositions

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

Figure 2: Syntax

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 3. Since we choose a strong reduction relation, all possible contexts are allowed—except reduction under incoherent abstractions. The notation $\partial\!\!\not[]$ 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*. We write Ω for the set of errors, which is 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 3); 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 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

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

Figure 3: Notations

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

Figure 4: Reduction relation

why we prefer to state soundness as the fact that reduction never produces errors, avoiding the reference to the more fragile definition of values.

3.2 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 2. 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

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

$$\textsc{TermPair}\ \frac{(\Gamma\vdash a_i:\tau_i)^{i\in\{1,2\}}}{\Gamma\vdash\langle a_1,a_2\rangle:\tau_1\times\tau_2} \qquad \textsc{TermProj}\ \frac{\Gamma\vdash a:\tau_1\times\tau_2}{\Gamma\vdash\pi_{\mathrm{i}}\,a:\tau_i} \qquad \textsc{TermCoer}\ \frac{\Gamma,\Sigma\vdash a:\tau \qquad \Gamma\vdash(\Sigma\vdash\tau)\triangleright\sigma}{\Gamma\vdash a:\sigma}$$

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

Figure 5: Term typing rules

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 2. 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 $\mathsf{fv}(\Theta)\subseteq\mathsf{dom}(\Gamma)$.

3.3 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.	$\Gamma\vdash\Gamma$	environment coherence and well-formedness
$\Gamma\vdash\tau:\kappa$	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 and well-formedness judgment $\Gamma\vdash\Gamma'$ checks that every kind appearing in Γ' is coherent in the environment that precedes it and that every type has kind

$$\textsc{TypeEq}\ \frac{\Gamma \vdash \tau : \kappa \qquad \kappa = \kappa' \qquad \Gamma \Vdash \kappa'}{\Gamma \vdash \tau : \kappa'} \qquad \textsc{TypeVar}\ \frac{(\alpha : \kappa) \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \textsc{TypeArr}\ \frac{\Gamma \vdash \tau : \star \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \tau \to \sigma : \star}$$

$$\textsc{TypeProd}\ \frac{\Gamma \vdash \tau : \star \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \tau \times \sigma : \star} \qquad \textsc{TypeFor}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, (\alpha : \kappa) \vdash \tau : \star}{\Gamma \vdash \forall (\alpha : \kappa)\, \tau : \star} \qquad \textsc{TypePi}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, (\alpha : \kappa) \vdash \tau : \star}{\Gamma \vdash \Pi (\alpha : \kappa)\, \tau : \star}$$

$$\textsc{TypeBot}\ \Gamma \vdash \bot : \star \qquad \textsc{TypeTop}\ \Gamma \vdash \top : \star \qquad \textsc{TypeUnit}\ \Gamma \vdash \langle\rangle : 1$$

$$\textsc{TypePair}\ \frac{\Gamma \vdash \tau_1 : \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \langle \tau_1, \tau_2 \rangle : \kappa_1 \times \kappa_2} \qquad \textsc{TypeProj}\ \frac{\Gamma \vdash \tau : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_{\mathrm{i}}\, \tau : \kappa_i}$$

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

$$\textsc{TypeUnpack}\ \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma \vdash \tau : \kappa}$$

Figure 6: Type judgment relation

star. It is defined by the three rules:

$$\Gamma \vdash \varnothing \qquad \frac{\Gamma \vdash \Gamma' \qquad \Gamma, \Gamma' \vdash \kappa}{\Gamma \vdash \Gamma', (\alpha : \kappa)} \qquad \frac{\Gamma \vdash \Gamma' \qquad \Gamma, \Gamma' \vdash \tau : \star}{\Gamma \vdash \Gamma', (x : \tau)}$$

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:

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

Well-formedness rules are defined in Figure 1.

The type judgment $\Gamma \vdash \tau : \kappa$ is defined in Figure 6. 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 straightforward.

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, as stated below by the extraction lemma (Lemma 21).

Term typing rules are given on Figure 5. 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

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

$$\textsc{PropAndProj}\ \frac{\Gamma;\Theta \vdash \mathrm{P}_1 \wedge \mathrm{P}_2}{\Gamma;\Theta \vdash \mathrm{P}_i} \qquad \textsc{PropForIntro}\ \frac{\Gamma \Vdash \kappa \qquad \Gamma, (\alpha : \kappa); \Theta \vdash \mathrm{P}}{\Gamma;\Theta \vdash \forall(\alpha : \kappa)\,\mathrm{P}} \qquad \textsc{PropForElim}\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma;\Theta \vdash \forall(\alpha : \kappa)\,\mathrm{P}} \qquad \textsc{PropRes}\ \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid \mathrm{P}\}}{\Gamma;\Theta \vdash \mathrm{P}[\alpha \leftarrow \tau]}$$

$$\textsc{PropExi}\ \frac{\Gamma \vdash \tau : \kappa}{\Gamma;\Theta \vdash \exists\,\kappa} \qquad \textsc{PropFix}\ \frac{\Gamma \Vdash \mathrm{P}}{\Gamma;\Theta, \mathrm{P} \vdash \mathrm{P}}$$

Figure 7: Proposition judgment relation

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 7 and rules specific to coercion propositions in Figure 8. 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 3.3).

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: the fact that any term that admits the typing $\Gamma, \Sigma \vdash \tau$ also admits the typing $\Gamma \vdash \sigma$ does not imply that the judgment $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \sigma$ is derivable; therefore, the coercion typing judgment is semantically incomplete.) One could expect this 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 $\lambda x\,((\square\, x))$. Placing a term with typing $\Gamma, \Sigma \vdash \tau' \to \sigma'$ in the hole, we may give $\lambda x\,((\square\, x))$ 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 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 [Amadio and Cardelli(1993)] are admissible using PropFix (we write $\tau_1 \triangleleft\triangleright \tau_2$ for $\tau_1 \triangleright \tau_2 \wedge \tau_2 \triangleright \tau_1$):

CoerPeriod
$$\frac{\alpha \mapsto \sigma : \mathsf{wf} \qquad \Gamma; \Theta \vdash (\tau_i \triangleleft\triangleright \sigma[\alpha \leftarrow \tau_i])^{i \in \{1,2\}}}{\Gamma; \Theta \vdash \tau_1 \triangleright \tau_2}$$

CoerEtaMu
$$\frac{\Gamma, (\alpha, \beta, \alpha \triangleright \beta); \Theta \vdash \tau \triangleright \sigma}{\Gamma; \Theta \vdash \mu\alpha\,\tau \triangleright \mu\beta\,\sigma}$$

Interestingly, the proof for CoerPeriod requires reinforcement of the coinduction hypothesis since we need $\tau_1 \triangleleft\triangleright \tau_2$ and not just $\tau_1 \triangleright \tau_2$ 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).

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

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

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

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

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

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

Figure 8: Coercion judgment relation

$$\textsc{RecVar}\ \alpha \mapsto \alpha : \mathsf{ne} \qquad \textsc{RecArr}\ \frac{(\alpha \mapsto \tau_i : \mathsf{ne})^{i\in\{1,2\}}}{\alpha \mapsto \tau_1 \to \sigma_2 : \mathsf{wf}} \qquad \textsc{RecProd}\ \frac{(\alpha \mapsto \tau_i : \mathsf{ne})^{i\in\{1,2\}}}{\alpha \mapsto \tau_1 \times \sigma_2 : \mathsf{wf}} \qquad \textsc{RecFor}\ \frac{\alpha \notin \mathsf{fv}(\kappa) \qquad \alpha \mapsto \tau : \chi}{\alpha \mapsto \forall(\beta:\kappa)\,\tau : \chi}$$

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

Figure 9: Well-foundedness judgment relation

4 Semantics

A term is sound if none of its reductions lead to an error. To avoid the negation, it is easier to reason with valid terms defined as the complement of Ω, *i.e.* terms that are not errors, which we write $\mho$. Hence, a term is sound if all its reduction paths lead to valid terms. Since this construction appears repeatedly, we define the *expansion* of a set of terms R, which we write $(\rightsquigarrow^\star\mathsf{R})$, the set of terms a such that any reduction path starting with a leads to a term in R. The set $\mathcal{S}$ of sound terms is the expansion $(\rightsquigarrow^\star\mho)$ of valid terms.

Head normal forms Δ are terms whose root node is a constructor, *i.e.* abstractions, pairs, and incoherent abstractions, while neutral terms ∇ are variables, applications, projections, and incoherent application. Notice that Δ and ∇ are complement of one another, *i.e.* terms are the disjoint union of Δ and ∇.

Progress is a way to double-check the definition of the semantics, by defining values syntactically and checking that semantic values (irreducible valid terms) are syntactic values (and neutral values are prevalues):

Lemma 1 (Progress)**.** *If $a \in \mho$ and $a \not\rightsquigarrow$, then a is of the form v.*

$$\begin{array}{ll} \lfloor x^k \rfloor_j = x^{kj} & \lfloor \langle e_1, e_2 \rangle^k \rfloor_j = \langle \lfloor e_1 \rfloor_j, \lfloor e_2 \rfloor_j \rangle^{kj} \\ \lfloor \lambda^k x\, e \rfloor_j = \lambda^{kj} x\, \lfloor e \rfloor_j & \lfloor {\pi_{\mathrm{i}}}^k\, e \rfloor_j = {\pi_{\mathrm{i}}}^{kj}\, \lfloor e \rfloor_j \\ \lfloor (e\, f)^k \rfloor_j = (\lfloor e \rfloor_j\, \lfloor f \rfloor_j)^{kj} & \lfloor \partial^k\, e \rfloor_j = \partial^{kj}\, \lfloor e \rfloor_j \\ & \lfloor e \diamondsuit^k \rfloor_j = \lfloor e \rfloor_j \diamondsuit^{kj} \end{array}$$

Figure 10: Lower function

The converse is also true, *i.e.* values do not contain errors. However, this won't remain true when we restrict the strategy, *e.g.* to call-by-value. In this case, redefining the grammar of values, progress will still hold, but some grammatically well-formed values may contain "inaccessible" errors, such as errors occurring under an abstraction.

Type soundness states that well-typed terms are sound. We prove this by interpreting syntactic types as semantic types which are themselves sets of terms. However, since we allow general recursive types the evaluation of terms may not terminate. This is not a problem, since type soundness is not about termination, but ruling out unsound terms, which if they reach an error do so in a finite number of steps.

The idea of step-indexed techniques is to stop the reduction after a certain number of steps, as if some initially available fuel (the number of allowed reduction steps) has all been consumed. Since errors are necessarily reached after a finite number of steps, we may always detect errors with some finite but arbitrary large number of reduction steps.

However, there is a difficulty applying this technique with strong reduction strategies, which we solve by including the fuel inside terms, called indexed terms, and block the reduction internally when terms do not have enough fuel, rather than control the number of reduction steps externally.

4.1 The indexed calculus

Terms of the indexed calculus are terms of the λ-calculus where each node is annotated with a natural number called the index (or fuel) of this node. They are written with letter f or e and formally defined on Figure 11: indexed terms are variables x^k, abstractions $\lambda^k x\, e$, applications $(e\, f)^k$, pairs $\langle e, f \rangle^k$, projections ${\pi_{\mathrm{i}}}^k\, e$, incoherent abstractions $\partial^k\, e$, and incoherent applications $e \diamondsuit^k$. We consistently label one hole contexts and errors. We write E^k for one-hole context with index k.

Intuitively, indices indicate the maximum number of reduction steps allowed under the given node. Since redexes usually involve several nodes, we must take the minimum of indices of the redex nodes. We use an auxiliary lowering function on indexed terms to keep track of such constraints by lowering the indices in a subterm. It is written $\lfloor e \rfloor_j$ and defined on Figure 10. We use concatenation of indices to denote the minimum of their values. This is not ambiguous since we never use multiplication of indices. Lowering simply changes all indices in the term e with their minimum with j.

The capture avoiding substitution $e[x \leftarrow f]$ of term f for variable x in the term e replaces in e all free occurrences x^j of x by $\lfloor f \rfloor_j$. The definition is generalized in the obvious way to simultaneous substitutions. We use letter γ to range over substitutions. The lowering of substituted occurrences is necessary to make substitution commute with the lowering function:

Lemma 2. $\lfloor e[x \leftarrow f] \rfloor_k = \lfloor e \rfloor_k [x \leftarrow f] = \lfloor e \rfloor_k [x \leftarrow \lfloor f \rfloor_k]$

In particular, renaming commutes with the lowering function.

$$\begin{aligned} e, f &::= x^k \mid \lambda^k x\, e \mid (e\, e)^k \mid \langle e, e\rangle^k \mid \pi_{\mathrm{i}}{}^k\, e \mid \partial^k\, e \mid e \diamondsuit^k \\ E^k &::= \lambda^k x\, [] \mid ([]\, e)^k \mid (e\, [])^k \mid \langle [], e\rangle^k \mid \langle e, []\rangle^k \mid \pi_{\mathrm{i}}{}^k\, [] \mid \cancel{\partial^k\, []} \mid [] \diamondsuit^k \end{aligned}$$

Figure 11: Syntax of indexed terms

$$\textsc{FRedCtx}\ \frac{e \rightsquigarrow f}{E^{k+1}[e] \rightsquigarrow E^k[f]} \qquad \textsc{FRedApp}\ ((\lambda^{j+1} x\, e)\, f)^{k+1} \rightsquigarrow \lfloor e[x \leftarrow f] \rfloor_{kj} \qquad \textsc{FRedProj}\ \pi_{\mathrm{i}}{}^{k+1}\, \langle e_1, e_2\rangle^{j+1} \rightsquigarrow \lfloor e_i \rfloor_{kj}$$

$$\textsc{FRedIApp}\ (\partial^{j+1}\, e) \diamondsuit^{k+1} \rightsquigarrow \lfloor e \rfloor_{kj}$$

Figure 12: Indexed calculus reduction relation

The reduction rules of the indexed calculus mimic those of the λ-calculus, but with some index manipulation, as described on Figure 12. Reduction can only proceed when the index on the nodes involved in the reduction are strictly positive; the indices are lowered after reduction by the minimum of the involved indices decremented by one. As a corollary, reduction cannot occur at or under a node with a null index. This applies both to head reduction rules (FRedApp, FRedProj, and FRedIApp) and to reduction in an evaluation context (Rule FRedCtx). That is, a head reduction can only be applied along a path of the form $E_1^{k_1}[\dots E_p^{k_p}[e]]$ when indices k_i's are all strictly positive; they are all decremented after the reduction.

For example, here is a decorated reduction of apply (the λ-term $\lambda x\, \lambda y\, (x\, y)$) applied to two terms a and b:

$$\begin{aligned} & (((\lambda^{k_3+1} x\, \lambda^{j_1} y\, (x^{j_3}\, y^{j_4})^{j_2})\, a)^{k_2+1}\, b)^{k_1+1} \\ \rightsquigarrow\quad & ((\lambda^{j_1 k_2 k_3} y\, (\lfloor a \rfloor_{j_3 k_2 k_3}\, y^{j_4 k_2 k_3})^{j_2 k_2 k_3})\, b)^{k_1} \end{aligned}$$

Since the reduction happens under the external application, it must have some fuel $k_1 + 1$, which is decreased by one in the result. Then, for the redex to fire, the application must have some fuel $k_2 + 1$ as well as the abstraction $k_3 + 1$, which are both decreased by 1 and combined as $k_2 k_3$ to lower the result of the reduction. Before that, the term a, which has been substituted for x^{j_3} has been lowered to j_3 in the result. The important feature is that b has not been lowered, which is an important difference with what would happen with the traditional step-indexed approach when indices are outside terms.

Strong normalization By design, the indexed calculus is strongly normalizing, *i.e.* all reduction paths of all terms are finite. In particular, they are bounded by the index of their root node.

4.2 Bisimulation

To show that reduction between undecorated terms and decorated terms coincides, we define $\lfloor e \rfloor$ the erasure of an indexed term e obtained by dropping all indices. We lift this function to sets of terms: $\lfloor \mathsf{R} \rfloor$ is the set $\{\lfloor e \rfloor \mid e \in \mathsf{R}\}$. By construction, dropping is stronger than lowering, *i.e.* dropping after lowering is the same as dropping, or in math, $\lfloor \lfloor e \rfloor_j \rfloor = \lfloor e \rfloor$. As for lowering, dropping commutes with substitution: $\lfloor e[x \leftarrow f] \rfloor = \lfloor e \rfloor [x \leftarrow \lfloor f \rfloor]$.

We overload the notations Ω and ∇ for the sets of errors and neutral terms for indexed terms. This overloading is not a problem since it is always clear from context which version of terms we mean. Moreover, the definitions coincide with $\lfloor \Omega \rfloor$, and $\lfloor \nabla \rfloor$, so it could also be seen as leaving the dropping implicit.

$$\mathsf{P}\,x^k = \mathsf{P}\,k \qquad \begin{aligned} \mathsf{P}\,(\lambda^k x\,e) = \mathsf{P}\,({\pi_{\mathrm{i}}}^k\,e) &= \mathsf{P}\,k \wedge \mathsf{P}\,e \\ \mathsf{P}\,(\partial^k\,e) = \mathsf{P}\,(e\,\diamondsuit^k) &= \mathsf{P}\,k \wedge \mathsf{P}\,e \\ \mathsf{P}\,(e_1\,e_2)^k = \mathsf{P}\,\langle e_1, e_2\rangle^k &= \mathsf{P}\,k \wedge \mathsf{P}\,e_1 \wedge \mathsf{P}\,e_2 \end{aligned}$$

Figure 13: Lifting integer predicates to indexed terms

$$\begin{aligned} x^j \star x^k &= j \star k \\ \lambda^j x\,e \star \lambda^k x\,f &= {\pi_{\mathrm{i}}}^j\,e \star {\pi_{\mathrm{i}}}^k\,f = j \star k \wedge e \star f \\ \partial^j\,e \star \partial^k\,f &= e\,\diamondsuit^j \star f\,\diamondsuit^k = j \star k \wedge e \star f \\ \left.\begin{array}{r}(e_1\,e_2)^j \star (f_1\,f_2)^k \\ \langle e_1, e_2\rangle^j \star \langle f_1, f_2\rangle^k\end{array}\right\} &= j \star k \wedge e_1 \star f_1 \wedge e_2 \star f_2 \end{aligned}$$

Figure 14: Lifting of a binary predicate $\star$ on indices to terms

We also overload the notation $\mathcal{S}$ for the set of sound indexed terms. Although it is defined as for λ-terms as $(\rightsquigarrow^\star \mathcal{U})$, the meaning is different since the reduction is now bridled by indices.

The calculus on indexed terms is just an instrumentation of the λ-calculus that behaves the same up to the consumption of all the fuel. Formally, we show that they can simulate one another, up to some condition on the indices.

Indexed terms can be simulated by λ-terms. That is, if an indexed term can reduce, then the same reduction step can be performed after dropping indices.

Lemma 3 (forward simulation)**.** *If $e \rightsquigarrow f$, then $\lfloor e \rfloor \rightsquigarrow \lfloor f \rfloor$.*

In order to make the other direction concise, we lift predicates on integers to predicates on indexed terms by requiring the predicate to hold for all indices occurring in the term. For instance, $e \leq k$ means that the indices in e are smaller or equal to k. This is formally defined on Figure 13.

Indexed terms can simulate λ-terms, provided they have enough fuel. This means that if an indexed term has strictly positive indices and can be reduced after dropping its indices, then the same reduction step can be performed on the indexed term.

Lemma 4 (backward simulation)**.** *If $e > 0$ and $\lfloor e \rfloor \rightsquigarrow a'$, then there exists e' such that $e \rightsquigarrow e'$ and $\lfloor e' \rfloor = a'$.*

4.3 Semantic types

To define semantic types concisely, it is convenient to have a few helper operations on sets of indexed terms. We first lift binary properties on indices to indexed terms. This is done by asking the two terms to share the same skeleton (they drop to the same λ-term) and the indices of corresponding nodes to be related by the property on indices. A formal definition is given in Figure 14.

The *interior* of a set R is the set $\mathsf{R}{\downarrow}$ containing all terms smaller than a term in R, *i.e.* $\{f \mid \exists e \in \mathsf{R}, f \leq e\}$. The *contraction* of a set R is the set $(\mathsf{R}{\rightsquigarrow})$ of all terms obtained by one-step reduction of a term in R, *i.e.* $\{f \mid \exists e \in \mathsf{R}, e \rightsquigarrow f\}$.

A *pretype* is a set of sound terms that contains both its interior and its contraction. We write $\mathbb{P}$ the set of pretypes.

Definition 5 (Pretypes)**.** $\mathbb{P} \stackrel{\text{def}}{=} \{\mathsf{R} \subseteq \mathcal{S} \mid \mathsf{R}{\downarrow} \cup (\mathsf{R}{\rightsquigarrow}) \subseteq \mathsf{R}\}$

Notice that the empty set and $\mathcal{S}$ are pretypes. Pretypes only contain sound terms since types will be pretypes and types will be sets of sound terms. The closure of pretypes by interior is just technical. The main property of pretypes is to be closed by reduction. Types are pretypes that are also closed by a form of expansion. As a first approximation, sound terms that reduce to a term in a type R should also be in R. However, a type R should still not contain unsound terms even if these reduce to some term in R. Moreover, the meaning of a set of terms R is in essence determined by its set of head normal forms, which we call the kernel of R. We use concatenation for intersection of sets of terms. Hence, the kernel of R is $\Delta\mathsf{R}$. A type R need not contain every head normal form that reduces to some term in R. Consider for example the term e_0 equal to $\lambda x\, x$ and one of its expansion is the term e_1 equal to $\lambda x\, ((\lambda y\, x)\, (x\, x))$. The sets $\{e_0\}$ and $\{e_0, e_1\}$ have quite different meanings. Notice that by definition, the kernel is an idempotent operation: $\Delta(\Delta\mathsf{R}) = \Delta\mathsf{R}$.

The expansion-closure of a set of terms R, written $\diamondsuit\,\mathsf{R}$, is the set $(\rightsquigarrow^\star(\nabla\mho \uplus \Delta\mathsf{R}))$, which contains terms of which every reduction path leads to either a valid neutral term or a head normal form of R. By definition, the expansion closure is monotonic: if $\mathsf{R} \subseteq \mathsf{S}$, then $\diamondsuit\,\mathsf{R} \subseteq \diamondsuit\,\mathsf{S}$; it is also idempotent: $\diamondsuit\,(\diamondsuit\,\mathsf{R}) = \diamondsuit\,\mathsf{R}$.

Finally, semantic types are pretypes that are stable by expansion closure:

Definition 6 (Semantic types)**.** $\mathbb{T} \stackrel{\text{def}}{=} \{\mathsf{R} \in \mathbb{P} \mid \diamondsuit\,\mathsf{R} \subseteq \mathsf{R}\}$.

The kernel of a type is a pretype—but not a type. Conversely, the expansion-closure of a pretype is a type. Actually expansion-closure and kernel are almost inverses of one another: if R is a type, then $\diamondsuit\,(\Delta\mathsf{R}) = \mathsf{R}$.

The smallest type, called the bottom type and written $\hat{\bot}$, is equal to $\diamondsuit\,\{\}$, that is $(\rightsquigarrow^\star(\nabla\mho))$. The largest type, $\hat{\top}$, called the top type is the set $\mathcal{S}$ of sound terms.

4.4 Simple types

We can now define the semantics of functions and products as semantic type operators.

Definition 7 (Arrow, product, and incoherent operators)**.**
$\mathsf{R} \hat{\to} \mathsf{S} \stackrel{\text{def}}{=} \diamondsuit\,\{\lambda^k x\, e \in \mathcal{S} \mid k > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x \leftarrow f] \rfloor_{k-1} \in \mathsf{S}\}$
$\mathsf{R} \hat{\times} \mathsf{S} \stackrel{\text{def}}{=} \diamondsuit\,\{\langle e, e' \rangle^k \in \mathcal{S} \mid k > 0 \Rightarrow \lfloor e \rfloor_{k-1} \in \mathsf{R} \wedge \lfloor e' \rfloor_{k-1} \in \mathsf{S}\}$
$\hat{\Pi}_{\mathsf{I}}\mathsf{F} \stackrel{\text{def}}{=} \diamondsuit\,\{\partial^k\, e \mid k > 0 \Rightarrow \forall i \in \mathsf{I}\ \lfloor e \rfloor_{k-1} \in \mathsf{F}_i\}$

The arrow, product, and incoherent operators preserve types.

Lemma 8. *If* R *and* S *are types, then so are* $\mathsf{R} \hat{\to} \mathsf{S}$ *and* $\mathsf{R} \hat{\times} \mathsf{S}$.

Lemma 9. *If* F_i *is type for all* $i \in \mathsf{I}$*, then so is* $\hat{\Pi}_{\mathsf{I}}\mathsf{F}$.

The proof uses the following easy properties on indices:

- $\lfloor \lfloor e \rfloor_j \rfloor_k = \lfloor e \rfloor_{kj}$
- If $k' \leq k$ and $e' \leq e$, then $\lfloor e' \rfloor_{k'} \leq \lfloor e \rfloor_k$.
- If $e' \leq e$ and $f' \leq f$, then $e'[x \leftarrow f'] \leq e[x \leftarrow f]$.

And this less easy one:

Lemma 10. *If* $e \rightsquigarrow f$ *holds, then* $\lfloor e \rfloor_{k+1} \rightsquigarrow f'$ *and* $\lfloor f \rfloor_k \leq f'$ *for some* f'.

Proof. We only detail the proof for the arrow operator, which uses indexed terms in a crucial way. The proof for the product operator is similar, but easier. Since the arrow operator is defined by expansion-closure, it is a type if its kernel is a pretype. Its kernel contains only sound terms by definition. So it remains to show that the definition contains its interior and contraction.

Let $\lambda^j x\, e' \leq \lambda^k x\, e$ (**1**), $\lambda^k x\, e \in \mathcal{S}$ (**2**), and $k > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x \leftarrow f] \rfloor_{k-1} \in \mathsf{S}$ (**3**), and show that $\lambda^j x\, e' \in \mathcal{S}$ (**4**) and $j > 0 \Rightarrow \forall f, \lfloor f \rfloor_{j-1} \in \mathsf{R} \Rightarrow \lfloor e'[x \leftarrow f] \rfloor_{j-1} \in \mathsf{S}$ (**5**). The first assertion (4) comes easily with (1) and (2) since $\mathcal{S}$ contains its interior. To show (5), let $j > 0$ and $\lfloor f \rfloor_{j-1} \in \mathsf{R}$ (**6**) and show $\lfloor e'[x \leftarrow f] \rfloor_{j-1} \in \mathsf{S}$ (**7**). By (1) we have $j \leq k$, so $k > 0$. We also have $\lfloor \lfloor f \rfloor_{j-1} \rfloor_{k-1} = \lfloor f \rfloor_{j-1}$ which is in R by (6). So from (3) we have $\lfloor e[x \leftarrow \lfloor f \rfloor_{j-1}] \rfloor_{k-1} \in \mathsf{S}$. Since S is a type, it contains its interior so $\lfloor e'[x \leftarrow \lfloor f \rfloor_{j-1}] \rfloor_{j-1} \in \mathsf{S}$. Since the substitution and the lowering function commute, we conclude (7).

Let $\lambda^k x\, e \rightsquigarrow e_1$ (**8**), $\lambda^k x\, e \in \mathcal{S}$ (**9**), and $k > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x \leftarrow f] \rfloor_{k-1} \in \mathsf{S}$ (**10**). By inversion of the reduction relation we have $k = k' + 1$ and $e_1 = \lambda^{k'} x\, e'$ for some k' and e' such that $e \rightsquigarrow e'$ (**11**). We now have to show that $\lambda^{k'} x\, e' \in \mathcal{S}$ (**12**) and $k' > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k'-1} \in \mathsf{R} \Rightarrow \lfloor e'[x \leftarrow f] \rfloor_{k'-1} \in \mathsf{S}$ (**13**). We show (12) with (8) and (9) since $\mathcal{S}$ contains its contraction. To show (13), let $k' > 0$ and $\lfloor f \rfloor_{k'-1} \in \mathsf{R}$ (**14**) and show $\lfloor e'[x \leftarrow f] \rfloor_{k'-1} \in \mathsf{S}$ (**15**). We have $\lfloor \lfloor f \rfloor_{k'-1} \rfloor_{k-1} = \lfloor f \rfloor_{k'-1}$ which is in R by (14). So from (10) we have $\lfloor e[x \leftarrow \lfloor f \rfloor_{k'-1}] \rfloor_{k-1} \in \mathsf{S}$. Since S is a type, it contains its contraction and interior so $\lfloor e'[x \leftarrow \lfloor f \rfloor_{k'-1}] \rfloor_{k'-1} \in \mathsf{S}$ by Lemma 10. Since the substitution and the lowering function commute, we conclude (15). □

4.5 Intersection types

The intersection $\bigcap_{i \in \mathsf{I}} \mathsf{R}_i$ of a nonempty family of types $(\mathsf{R}_i)^{i \in I}$ is a type. (As a particular case, the bottom type $\hat{\bot}$ is also the intersection of all types.)

4.6 Recursive types

This section follows the usual description of recursive types using approximations as done in [Appel and McAllester(200
This addition of recursive types is the main reason for using a step-indexed semantics. However, while they require the need for step-indexed semantics, they do not raise any difficulty once the semantics has been correctly set up.

The k-approximation of a set R, written $\langle \mathsf{R} \rangle_k$ is the subset $\{e \in \mathsf{R} \mid e < k\}$ of element of R that are smaller than k

The following properties of approximations immediately follow from the definition. $\langle \mathsf{R} \rangle_0$ is the empty set; a sequence of approximations is the approximation by the minimum of the sequence: $\langle \langle \mathsf{R} \rangle_j \rangle_k = \langle \mathsf{R} \rangle_{jk}$; Two sets of terms that are equal at all approximations are equal: If $\langle \mathsf{R} \rangle_k = \langle \mathsf{S} \rangle_k$ holds for all k, then $\mathsf{R} = \mathsf{S}$.

Definition 11 (Well-foundness)**.** *A function* F *on sets of terms is well-founded (resp. non-expansive) if for any set of terms* R*, the approximations of* $\mathsf{F}\,\mathsf{R}$ *and* $\mathsf{F}\,\langle \mathsf{R} \rangle_k$ *are equal at rank* $k+1$ *(resp.* k*),* i.e. $\langle \mathsf{F}\,\mathsf{R} \rangle_{k+1} = \langle \mathsf{F}\,\langle \mathsf{R} \rangle_k \rangle_{k+1}$ *(resp.* $\langle \mathsf{F}\,\mathsf{R} \rangle_k = \langle \mathsf{F}\,\langle \mathsf{R} \rangle_k \rangle_k$*)*

Intuitively, well-foundedness (resp. non-expansiveness) ensures that F builds terms smaller than $k+1$ (resp. k) by only looking at terms smaller than k in its argument.

The iteration of a well-founded function F does not look at its argument for terms of small indices: $\langle \mathsf{F}^k\,\mathsf{R} \rangle_k$ is independent of R; in particular, it is equal to $\langle \mathsf{F}^k \bot \rangle_k$. Therefore, $\langle \mathsf{F}^j\,\mathsf{R} \rangle_{kj}$ and $\langle \mathsf{F}^k\,\mathsf{R} \rangle_{kj}$ are equal.

Definition 12 (Recursive operator)**.** *Given a well-founded function* F *on sets of terms, we define* $\hat{\mu}\,\mathsf{F}$ *as the set of terms* $\bigcup_{k \geq 0} \langle \mathsf{F}^k \bot \rangle_k$*.*

The recursive operator preserves semantic types:

Lemma 13. *If* F *is well-founded and maps semantic types to semantic types, then* $\hat{\mu}\,\mathsf{F}$ *is a semantic type.*

Moreover, recursive types can be unfolded or folded as expected: if F is well-founded, then $\hat{\mu}\,\mathsf{F} = \mathsf{F}\,(\hat{\mu}\,\mathsf{F})$. This is proved by showing that $\langle \hat{\mu}\,\mathsf{F}\rangle_k$ is equal to both $\langle \mathsf{F}^k\,\bot\rangle_k$ and $\langle \mathsf{F}\,(\hat{\mu}\,\mathsf{F})\rangle_k$ for every k.

The following Lemma, which although in a different settings, is stated exactly as with traditional step-indexed semantics [Appel and McAllester(2001)]:

Lemma 14. *We have the following properties:*

- *Every well-founded function is non-expansive.*
- $\mathsf{X} \mapsto \mathsf{X}$ *is non-expansive.*
- $\mathsf{X} \mapsto \mathsf{R}$ *where* X *is unused in* R *(*R *is constant) is well-founded.*
- *The composition of non-expansive functors is non-expansive.*
- *The composition of a non-expansive functor with a well-founded functor (in either order) is well-founded.*
- *If* F *and* G *are non-expansive, then* $\mathsf{X} \mapsto \mathsf{F}\,\mathsf{X} \mathbin{\hat{\to}} \mathsf{G}\,\mathsf{X}$ *and* $\mathsf{X} \mapsto \mathsf{F}\,\mathsf{X} \mathbin{\hat{\times}} \mathsf{G}\,\mathsf{X}$ *are well-founded.*
- *If* $(\mathsf{F}_i)^{i\in I}$ *is a family of non-expansive functors, then* $\mathsf{X} \mapsto \hat{\Pi}_{i\in I}(\mathsf{F}_i\,\mathsf{X})$ *is well-founded.*
- *If* $(\mathsf{F}_i)^{i\in I}$ *is a family of non-expansive (resp. well-founded) functors, then* $\mathsf{X} \mapsto \bigcap_{i\in I}(\mathsf{F}_i\,\mathsf{X})$ *is non-expansive (resp. well-founded).*
- *If* $\mathsf{X} \mapsto \mathsf{F}\,\mathsf{X}\,\mathsf{Y}$ *is non-expansive (resp. well-founded) for every* Y *and* $\mathsf{F}\,\mathsf{X}$ *is well founded for every* X*, then* $\mathsf{X} \mapsto \hat{\mu}\,(\mathsf{F}\,\mathsf{X})$ *is non-expansive (resp. well-founded).*

Just for illustration $\mathsf{X} \mapsto \mathsf{X} \mathbin{\hat{\to}} \mathcal{S}$ is well-founded since $\mathsf{X} \mapsto \mathsf{X}$ is non-expansive and $\mathsf{X} \mapsto \mathcal{S}$ is constant, thus well-founded, and therefore non-expansive.

4.7 Semantic judgment

A binding is a pair $(x : \mathsf{R})$ of a variable and a semantic type. A context is a set of bindings $(x : \mathsf{R})$, defining a finite mapping from term variables to types. We say that a substitution γ is *compatible* with a context G and we write $\gamma : \mathsf{G}$ if $\mathsf{dom}(\gamma)$ and $\mathsf{dom}(\mathsf{G})$ coincide and for all $(x : \mathsf{R})$ in G, the term $\gamma\,x$ is in R.

We define the semantic judgment $\mathsf{G} \models \mathsf{S}$ as the set of terms e such that $\gamma\,e$ is in S for any substitution γ "compatible" with G.

Definition 15 (Semantic judgment)**.**

$$\begin{array}{rcl} \gamma : \mathsf{G} & \stackrel{\mathit{def}}{=} & \forall (x : \mathsf{R}) \in \mathsf{G}, \gamma\,x \in \mathsf{R} \\ \mathsf{G} \models \mathsf{S} & \stackrel{\mathit{def}}{=} & \{e \mid \forall \gamma : \mathsf{G}, \gamma\,e \in \mathsf{S}\} \end{array}$$

We may now present the semantic typing rules for the simply-typed λ-calculus.

Lemma 16 (Variable)**.** *If* R *is a type and* $(x : \mathsf{R})$ *is in* G*, then* x^k *is in* $\mathsf{G} \models \mathsf{R}$*.*

Proof. Let γ be compatible with G (**1**). We show that $\gamma\, x^k$ is in R. Since $(x : \mathsf{R})$ is in G, we have $\gamma\, x$ in R by (1), Being a type, R is closed by lowering. Hence, $\lfloor \gamma\, x \rfloor_k$ is also in R. By definition of substitution, this is equal to $\gamma\, x^k$, which is thus also in R. □

Lemma 17 (Abstraction)**.** *If* R *and* S *are types and* e *is in* $\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S}$*, then* $\lambda^k x\, e$ *is in* $\mathsf{G} \models \mathsf{R} \hat{\to} \mathsf{S}$*.*

Proof. Let γ be compatible with G (**1**). We show that $\gamma\, (\lambda^k x\, e)$ is in $\mathsf{R} \hat{\to} \mathsf{S}$ (**2**). Assume $\gamma\, (\lambda^k x\, e) \rightsquigarrow^\star e_1$. Then e_1 is necessarily of the form $\lambda^j x\, e'$ where $\gamma\, e \rightsquigarrow^\star e'$.

We first show that $\lambda^j x\, e' \in \mathcal{S}$ (**3**). Since γ is compatible with G, $\gamma, x \mapsto x$ is compatible with $\mathsf{G}, (x : \mathsf{R})$ as variables are in all types. Since e is in $(\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S})$, we have $(\gamma, x \mapsto x)\, e$, *i.e.* $\gamma\, e$ in S. Since S is closed by reduction, we have e' in S and a fortiori in $\mathcal{S}$. This implies (3).

Assume $j > 0$ and $\lfloor f \rfloor_{j-1} \in \mathsf{R}$. Let γ' be $\gamma, x \mapsto \lfloor f \rfloor_{j-1}$. By construction $\gamma' : \mathsf{G}, (x : \mathsf{R})$. Since e is in $(\mathsf{G}, (x : \mathsf{R}) \models \mathsf{S})$, we have $\gamma'\, e$ in S and, since S is closed by reduction, $e'[x \leftarrow \lfloor f \rfloor_{j-1}]$ is also in S. By decreasing index we have $\lfloor e'[x \leftarrow \lfloor f \rfloor_{j-1}] \rfloor_{j-1} \in \mathsf{S}$, from which by Lemma 2 becomes $\lfloor e'[x \leftarrow f] \rfloor_{j-1} \in \mathsf{S}$. This ends the proof of (2). □

Lemma 18 (Application)**.** *If* R *and* S *are types,* e *is in* $\mathsf{G} \models \mathsf{R} \hat{\to} \mathsf{S}$*, and* f *is in* $\mathsf{G} \models \mathsf{R}$*, then* $(e\, f)^k$ *is in* $\mathsf{G} \models \mathsf{S}$ *for any* k*.*

Proof. Let γ be compatible with G. We show that $\gamma\, (e\, f)^k \in \mathsf{S}$. By hypotheses we have $\gamma\, e \in \mathsf{R} \hat{\to} \mathsf{S}$ and $\gamma\, f \in \mathsf{R}$. We prove the more general result that for all k, e, and f, if $e \in \mathsf{R} \hat{\to} \mathsf{S}$ and $f \in \mathsf{R}$ hold, then $(e\, f)^k \in \mathsf{S}$ also holds. This is proved is by induction over the strong normalization of e and f using the closure expansion of S.

The term $(e\, f)^k$ is neutral. It is also valid since e and f are sound and, by construction of $\mathsf{R} \hat{\to} \mathsf{S}$, e is an abstraction when in normal form. If $(e\, f)^k$ reduces by a context rule, we use our induction hypothesis. Otherwise, e must be of the form $\lambda^{j+1} x\, e'$ for some j and e' and k be of the form $k' + 1$ and the reduction is $(e\, f)^k \rightsquigarrow \lfloor e'[x \leftarrow f] \rfloor_{jk'}$. It remains to show $\lfloor e'[x \leftarrow f] \rfloor_{jk'} \in \mathsf{S}$ (**1**). We have $\lfloor f \rfloor_j \in \mathsf{R}$ by stability under decreasing index. So, we have $\lfloor e'[x \leftarrow f] \rfloor_j \in \mathsf{S}$ by definition of the arrow operator. Then (1) follows by stability under decreasing index. □

Lemma 19 (Pairs)**.** *Let* R_1 *and* R_2 *be types. If* e_i *is in* $\mathsf{G} \models \mathsf{R}_i$*, then* $\langle e_1, e_2 \rangle^k$ *is in* $\mathsf{G} \models \mathsf{R}_1 \hat{\times} \mathsf{R}_2$*. If* e *in* $\mathsf{G} \models \mathsf{R}_1 \hat{\times} \mathsf{R}_2$*, then* ${\pi_i}^k\, e$ *is in* $\mathsf{G} \models \mathsf{R}_i$*.*

Lemma 20 (Incoherent)**.** *Let* F_i *be types for all* $i \in \mathsf{I}$*. If* e *is in* $\mathsf{G} \models \mathsf{F}_i$ *for all* $i \in \mathsf{I}$*, then* $\partial^k\, e$ *is in* $\mathsf{G} \models \hat{\Pi}_{\mathsf{I}} \mathsf{F}$*. If* e *in* $\mathsf{G} \models \hat{\Pi}_{\mathsf{I}} \mathsf{F}$*, then* $e \diamondsuit^k$ *is in* $\mathsf{G} \models \mathsf{F}_i$ *for all* $i \in \mathsf{I}$*.*

Note that when S is a type and R is a type for all $(x : \mathsf{R}) \in \mathsf{G}$, then $\mathsf{G} \models \mathsf{S}$ is a pretype.

5 Soundness

In order to show the soundness property we need the extraction lemma (Lemma 21), which uses the usual weakening and substitution lemmas.

Lemma 21 (Extraction)**.** *If* $\varnothing \Vdash \Gamma$ *holds, then the following properties hold:*

- *If* $\Gamma \vdash \kappa$ *holds, then* $\Gamma \Vdash \kappa$ *holds.*
- *If* $\Gamma \vdash \tau : \kappa$ *holds then* $\Gamma \Vdash \kappa$ *holds.*
- *If* $\Gamma; \Theta \vdash \mathrm{P}$ *and* $\Gamma \Vdash \Theta$ *hold then* $\Gamma \Vdash \mathrm{P}$ *holds.*
- *If* $\Gamma \vdash a : \tau$ *holds then* $\Gamma \vdash \tau : \star$ *holds.*

The soundness proof is not direct. We will translate the F_{cc} type system from the λ-calculus to a temporary type system on the indexed calculus. We will prove soundness for the indexed calculus type system and migrate the result to the λ-calculus type system. The relation between both type systems will be that if a λ-term is well-typed then all indexed terms that drop to this λ-term are well-typed too. And reciprocally, if an indexed term is well-typed, then its dropped λ-term is well-typed too. Both directions preserve the typing (the pair of the environment and type). Notice that only the term judgment needs to be changed since it is the only one talking about terms.

Syntactically, the indexed term judgment $e : \Gamma \vdash \tau$ contains the exact same rules as those of the λ-term judgment. However index annotations now appear on the term node we are typing. This annotation has no constraint, which gives us that if a term is typed with annotations it can be typed without and reciprocally if a term is typed without annotations it can be typed with any annotations.

Lemma 22. *The following assertions hold:*

- *If* $e : \Gamma \vdash \tau$ *holds, then* $\lfloor e \rfloor : \Gamma \vdash \tau$ *holds.*
- *If* $a : \Gamma \vdash \tau$ *holds, then* $e : \Gamma \vdash \tau$ *holds for all* e *such that* $\lfloor e \rfloor = a$.

To state and prove the soundness of the indexed type system we interpret (syntactic) kinds, types, propositions, and typing environments.

We interpret kinds as sets of mathematical objects, which are either sets of terms, the unit object, or pairs of objects. The kind star is interpreted as the set of sets of terms. The unit kind is interpreted as the singleton containing the unit object. The product kind is interpreted as the cartesian product of the interpretation of its components. The constrained kind is interpretated as the subset of the interpretation of its kind satisfying its proposition.

Definition 23 (Kind interpretation)**.** *The interpretation of a kind* κ *under* η*, written* $|\kappa|_\eta$ *is the mathematical object recursively defined as:*

- $|\star|_\eta = \text{sets of terms}$
- $|1|_\eta = \{\langle\rangle\}$
- $|\kappa_1 \times \kappa_2|_\eta = \{\langle x_1, x_2\rangle \mid x_1 \in |\kappa_1|_\eta \wedge x_2 \in |\kappa_2|_\eta\}$
- $|\{\alpha : \kappa \mid \mathrm{P}\}|_\eta = \{x \in |\kappa|_\eta \mid \forall k \ |\mathrm{P}|^k_{\eta, \alpha \mapsto x}\}$

The interpretation of a syntactic type is a semantic type, but it is parametrized over a mapping from type variables to semantic types written η. The interpretation of a type variable is its value in the mapping. If it is not present in the mapping, the unit semantic type is returned. The interpretation of arrow, product, and incoherent polymorphic types simply use the arrow, product, and incoherent operators defined in §4.4. The interpretation of the coherent polymorphic type $\forall(\alpha : \kappa)\,\tau$ under η is the intersection of all interpretations of τ under $\eta, \alpha \mapsto x$ where x ranges in the interpretation of κ under η. The interpretation of the recursive type $\mu\alpha\,\tau$ under η is the infinite iteration of the functor mapping X to the interpretation of τ under the extension of η mapping α to X—which corresponds to the infinite unfolding of the recursive type. Top and bottom are mapped to their semantic equivalent. Finally, the unit type, pairs of types, and projections of types, are interpretated to their mathematical equivalent.

Definition 24 (Type interpretation)**.** *The interpretation of a type* τ *under* η*, written* $|\tau|_\eta$ *is the set of terms recursively defined as:*

- $|\alpha|_\eta = \eta(\alpha)$
- $|\tau \to \sigma|_\eta = |\tau|_\eta \hat{\to} |\sigma|_\eta$
- $|\tau \times \sigma|_\eta = |\tau|_\eta \hat{\times} |\sigma|_\eta$
- $|\Pi(\alpha : \kappa)\, \tau|_\eta = \hat{\Pi}_{|\kappa|_\eta}(x \mapsto |\tau|_{\eta,\alpha\mapsto x})$
- $|\forall(\alpha : \kappa)\, \tau|_\eta = \bigcap_{|\kappa|_\eta}(x \mapsto |\tau|_{\eta,\alpha\mapsto x})$
- $|\mu\alpha\, \tau|_\eta = \hat{\mu}\,(\mathsf{X} \mapsto |\tau|_{\eta,\alpha\mapsto \mathsf{X}})$
- $|\bot|_\eta = \hat{\bot}$
- $|\top|_\eta = \hat{\top}$
- $|\langle\rangle|_\eta = \langle\rangle$
- $|\langle\tau, \sigma\rangle|_\eta = \langle |\tau|_\eta , |\sigma|_\eta \rangle$
- $|\pi_{\mathrm{i}}\, \tau|_\eta = \pi_{\mathrm{i}}\, |\tau|_\eta$

We define the interpretation of environments, written $|\Gamma|_{\gamma\eta}$, as the pair of a term substitution (a mapping from term variables to indexed terms) and a semantic type mapping (from type variables to semantic types). The interpretation is parametrized by initial mappings γ and η. The empty environment is interpreted by the singleton set containing the initial mappings. The interpretation of an environment Γ extended with a term binding $(x : \tau)$ extends the term mappings in the interpretation of Γ with x bound to a term in the interpretation of τ under the associated type mapping. The interpretation of an environment Γ extended with a type binding $(\alpha : \kappa)$ extends the type mapping with an element of the interpretation of κ.

We define the interpretation of proposition indexed by a type mapping η and an index k. The true and conjonction proposition are interpretated to their mathematical analogs. The coercion proposition is interpretated to the inclusion of $\forall\Sigma\, \tau$ in σ for terms smaller than k. The coherence proposition is interpretated as the inhabitation of the kind. And the polymorphic proposition is interpretated as quantified assertion.

Definition 25 (Proposition interpretation)**.** *The interpretation of a proposition* P *under* η *with* k*, written* $|\mathrm{P}|^k_\eta$ *is the mathematical assertion recursively defined as:*

- $|\top|^k_\eta = \mathrm{True}$
- $|\mathrm{P}_1 \wedge \mathrm{P}_2|^k_\eta = |\mathrm{P}_1|^k_\eta \wedge |\mathrm{P}_2|^k_\eta$
- $|(\Sigma \vdash \tau) \triangleright \sigma|^k_\eta = \forall e < k\ (\forall \eta' \in |\Sigma|_\eta\ \ e \in |\tau|_{\eta'}) \Rightarrow e \in |\tau|_\eta$
- $|\exists\, \kappa|^k_\eta = \exists x \in |\kappa|_\eta$
- $|\forall(\alpha : \kappa)\, \mathrm{P}|^k_\eta = \forall x \in |\kappa|_\eta\ \ |\mathrm{P}|^k_{\eta,\alpha\mapsto x}$

The coinduction environment interpretation $|\Theta|^k_\eta$ is a conjonction of mathematical assertions for each proposition: when the proposition is not guarded, it holds for all indices smaller than k; when it is guarded, it also holds for k and is thus accessible.

Definition 26 (Coinduction environment semantics)**.** *We define* $|\Theta|^k_\eta$ *as follows:*

- $|\varnothing|^k_\eta = \text{True}$
- $|\Theta, \mathrm{P}|^k_\eta = |\Theta|^k_\eta \wedge \forall j < k \ |\mathrm{P}|^j_\eta$
- $\left|\Theta, \mathrm{P}^\dagger\right|^k_\eta = |\Theta|^k_\eta \wedge \forall j \leq k \ |\mathrm{P}|^j_\eta$

We write $|\Gamma|$ to stand for $|\Gamma|_{\varnothing\varnothing}$. And we write $|\Gamma|_\eta$ the second projection of $|\Gamma|_{\mathsf{G}\eta}$, which does not depend on G, since the type mapping does not depend on the term mapping.

Definition 27 (Environment semantics)**.** *We define* $|\Gamma|_{\mathsf{G}\eta}$ *as follows:*

- $|\varnothing|_{\mathsf{G}\eta} = \{\mathsf{G}\eta\}$
- $|\Gamma, (x : \tau)|_{\mathsf{G}\eta} = \{(\mathsf{G}', (x : |\tau|_{\eta'})\eta' \mid \mathsf{G}'\eta' \in |\Gamma|_{\mathsf{G}\eta}\}$
- $|\Gamma, (\alpha : \kappa)|_{\mathsf{G}\eta} = \{\mathsf{G}'(\eta', \alpha \mapsto x) \mid \mathsf{G}'\eta' \in |\Gamma|_{\mathsf{G}\eta} \wedge x \in |\kappa|_{\eta'})\}$

We have the following lemmas. If τ is non-expansive (resp. well-founded) with respect to α, then its interpretation as a functor is also non-expansive (resp. well-founded). If a concatenated environment is well-formed then the interpretation of the second one under the first is nonempty. If a kind is coherent, then it is inhabited. If a type is well-kinded, then its interpretations under all type mappings in the interpretation of its environment is in the interpretation of its kind. If a proposition P is proved under Θ, then for all mappings and indices the interpretation of P holds if the interpretation of Θ does too. The interpretation of a coherent type environment is not empty and only contains semantic types. Finally, a well-typed term is in the semantic judgment of the interpretation of its typing.

Lemma 28. *The following assertions hold.*

- *If* $\alpha \mapsto \tau : \mathsf{ne}$ *holds, then* $(\mathsf{X} \mapsto |\tau|_{\eta,\alpha\mapsto\mathsf{X}})$ *is non-expansive.*
- *If* $\alpha \mapsto \tau : \mathsf{wf}$ *holds, then* $(\mathsf{X} \mapsto |\tau|_{\eta,\alpha\mapsto\mathsf{X}})$ *is well-founded.*
- *If* $\Gamma \vdash \kappa$ *holds, then* $\forall \eta \in |\Gamma| \ |\kappa|_\eta \neq \varnothing$ *holds.*
- *If* $\Gamma \vdash \tau : \kappa$ *holds, then* $\forall \eta \in |\Gamma| \ |\tau|_\eta \in |\kappa|_\eta$ *holds.*
- *If* $\Gamma; \Theta \vdash \mathrm{P}$ *holds, then* $\forall \eta \in |\Gamma| \ \forall k \ |\Theta|^k_\eta \Rightarrow |\mathrm{P}|^k_\eta$ *holds.*
- *If* $\Gamma \vdash \Gamma'$ *holds, then* $\forall \eta \in |\Gamma| \ |\Gamma'|_\eta \neq \varnothing$ *holds.*
- *If* $\Gamma \vdash \Gamma'$ *holds, then* $\forall \eta \in |\Gamma| \ \forall (x : \mathsf{R}) \in |\Gamma'|_{\varnothing\eta} \ \mathsf{R} \in \mathbb{T}$ *holds.*
- *If* $\Gamma \vdash e : \tau$ *holds, then* $\forall \mathsf{G}\eta \in |\Gamma| \ e \in \mathsf{G} \models |\tau|_\eta$ *holds.*

The filling of λ-term a at rank k is the indexed-term obtained by annotating each node of a with index k (Figure 15). By construction, we have $\lfloor \lceil a \rceil^k \rfloor = a$.

Theorem 29 (Soundness)**.** *If* $\varnothing \vdash \Gamma$ *and* $\Gamma \vdash a : \tau$ *hold, then* a *is sound.*

$$\begin{array}{ll} \lceil x \rceil^k = x^k & \lceil \langle a_1, a_2 \rangle \rceil^k = \langle \lceil a_1 \rceil^k, \lceil a_2 \rceil^k \rangle^k \\ \lceil \lambda x\, a \rceil^k = \lambda^k x\, \lceil a \rceil^k & \lceil \pi_{\mathrm{i}}\, a \rceil^k = \pi_{\mathrm{i}}{}^k\, \lceil a \rceil^k \\ \lceil (a\, b) \rceil^k = (\lceil a \rceil^k\, \lceil b \rceil^k)^k & \lceil \partial\, a \rceil^k = \partial^k\, \lceil a \rceil^k \\ & \lceil a \diamondsuit \rceil^k = \lceil a \rceil^k\, \diamondsuit^k \end{array}$$

Figure 15: Fill function

Termination in the absence of recursive types Although evaluation may not terminate because of the presence of recursive types, it remains interesting to show that recursive types are the only source of non-termination. We already know this in System F. We show that coercions do not themselves introduce non-termination, as long as all types remain non-recursive. The proof is based on reducibility candidates as for System F and does not raise any difficulties. We thus omit the details.

Theorem 30 (Termination)**.** *If $\varnothing \vdash \Gamma$ and $\Gamma \vdash a : \tau$ hold in the sublanguage without recursive types and coinduction, then a strongly normalizes.*

Subject reduction While by definition, there is subject reduction on semantic types (as they are closed by reduction), we do not have subject reduction syntactically. This just means that the type system is too rough an approximation to still capture the invariant of programs after they have been reduced.

Confluence Because we have mixed strong and block reduction, we have lost confluence. This is a known problem with a known fix. One solution is to extend the language with a restricted form of explicit subsitutions so that substitution may be held at the frontier of block terms until these terms would be unblock. We have not done the instrumentation to avoid complicating the language. Besides, if we remove incoherent abstraction and application, then the language is the λ-calculus and confluence holds again.

Coq development We have a Coq development of the soundness proof for F_{cc}. This Coq development also contains a proof of equivalence between 2 versions of the typing rules: a version with minimum redundancy, and a more redundant version used to prove soundness. This version is necessary for the induction hypothesis to hold even for extracted judgments and not only the direct premises.

The development differs from the paper by using de-Bruijn indices and using two homogeneous environments (one for types and one for terms) instead of a heterogeneous dependent one. The development can be found online.[2]

6 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}.

[2] `http://gallium.inria.fr/~remy/coercions/`

6.1 Encoding subtyping constraints

We claim that languages with ML-like subtyping constraints [Odersky et al.(1999)Odersky, Sulzmann, and Wehr] 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 [Pottier(1996)].

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 $\alpha_1 \dots \alpha_n$ 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 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.

6.2 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.

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

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

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

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 $\tau \diamond\!\!\triangleright \sigma$ stands for $\tau \triangleright \sigma \wedge \sigma \triangleright \tau$ as above):

$$\begin{aligned} \mathsf{Term}\,\alpha \stackrel{\text{def}}{=}\ & \Sigma(\beta : \star \times \star \mid \alpha \diamond\!\!\triangleright (\pi_1\,\beta \to \pi_2\,\beta))\,\alpha \\ +\ & \exists\beta\,\mathsf{Term}\,(\beta \to \alpha) \times \mathsf{Term}\,\beta \end{aligned}$$

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:

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

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

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 $\tau \times \sigma \mathrel{\triangleleft\!\triangleright} \pi_1\,\beta \to \pi_2\,\beta$ 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.

7 Discussion

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

7.1 Comparison with F^p_ι

The closest work is of course our previous work on F^p_ι 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^p_ι 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^p_ι 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 [Cretin and Rémy(2012)]. Namely, the usual type

abstraction $\forall\alpha\,\tau$ and coercion abstraction $(\tau_1 \triangleright \tau_2) \Rightarrow \sigma$—a term of type σ under the hypothesis that $\tau_1 \triangleright \tau_2$ holds. For instance, this would permit to write a function of type $\forall\alpha\,(\alpha \to (\tau_1 \triangleright \tau_2) \Rightarrow \sigma)$ 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.

7.2 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.

Subtyping have been popularized by object-oriented languages, even though inheritance is somehow better modeled by matching [Bruce et al.(1997)Bruce, Petersen, and Fiech] or row polymorphism [Rémy and Vouillon(1998)]. In our view subtyping and polymorphism are both treated as coercions. Moreover, row polymorphism is just polymorphism with a richer structure of types and matching is a hidden form of row polymorphism.

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.

All language features without computational content are treated as coercions in F_{cc}. However, we have kept weakening implicit. Explicit weakening would exercise our general approach to typing inclusions which we have only used in a restricted form. Interestingly, explicit weakening has already been used in combination with explicit substitution [David and Guillaume(2001)]. Moreover, a new form of reduction is introduced to break wedges creating a particular substitution with the information about the weakening occurring in the wedge: $(\langle k\rangle\lambda t\,u) \to [0/u, k]t$. $\langle k\rangle$ is a constructor to lift a term by k de Bruijn indices and $[j/u, k]$ is the explicit substitution constructor: the j de Bruijn index is substituted by $\langle j\rangle u$, indices smaller than j are not modified, and those greater than j are incremented by $k-1$.

Coercions have also been used to eliminate function call overhead from datatype constructors in [Vanderwaart et al.(2003)Vanderwaart, Dreyer, Petersen, Crary, Harper, and Cheng]: 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 [Crary(2000)]. However, the motivations are quite different and coercions are only used as a tool to compile bounded quantification away into intersection types.

7.3 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. The changes are not significant because we already have kinds and a (very simple) form of reduction on types. However, the formalization in a proof assistant is significantly heavier. The internal language FC of GHC already has higher-order kinds, but not full β-reduction at the level of types.

Intersection types It should also be possible to add intersection types. (Our semantics already has them.). Following the work of Wells [Wells and Haack(2002)] on *branching types*, it would then be interesting to have intersection types as *branching typings*, which would be trees of typings where leaves are usual typings and nodes are chunks of typing environments.

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. The natural interpretation of existential types is the infinite union of the interpretations when the hidden part varies over all possible witnesses. The problem is already present and easier to explain with union types: the union of two semantic types is not obviously a type, and more precisely closed by expansion, as is the case for intersection. A term e in $\diamondsuit(\mathsf{R} \cup \mathsf{S})$ could a priori reduce to both e_1 in $\Delta\mathsf{R} \setminus \Delta\mathsf{S}$ and e_2 in $\Delta\mathsf{S} \setminus \Delta\mathsf{R}$ and not be in $\mathsf{R} \cup \mathsf{S}$.

In the current setting, where the underlying language is the λ-calculus, it seems that e should be in $\mathsf{R} \cup \mathsf{S}$ by a complex standardization argument [Riba(2007)] in the absence of indices and may not be applicable in the case of indices—or force us to have a more involved definition of indexing compatible with standardization, namely so that semantic types are closed by a form of standardization. In any case, this argument cannot apply anymore if we add non-determinism such as random choice to the calculus. In this case, existential types must be reduced to a head normal form before unpacking, which is exactly what the CPS encoding of existential types enforces! 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.

Alternate indexing Nevertheless, this raises the question of whether our definition of indexing is the right one. There is a lot of room for variations in the definition of indexing, since they are only a mean of abruptly stopping the reduction as long as other indexing of the same term will always allow to proceed further. However, in this process we have lost some interesting properties of the underlying λ-calculus such as confluence and standardization. Finding alternative—but probably more complex—indexing that would preserve those properties may still be worth exploring.

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!

Other application of step-indexed semantics. Our semantics is a form of unary logical relation, and we expect no difficulties when considering binary relations. Step-indexing is also used in the presence of side effects to break the recursion in the store. Checking how indexed terms would work in the presence of a store remains to be done.

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.

We have adapted the step-indexed semantics to work for calculi with strong reduction strategies and used it to prove the soundness of our language in a general setting. The step-indexed terms have been introduced just for our needs, but it would be worth exploring this approach further.

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

[Amadio and Cardelli(1993)] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 15(4):575–631, 1993.

[Appel and McAllester(2001)] 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.

[Bruce et al.(1997)Bruce, Petersen, and Fiech] K. B. Bruce, L. Petersen, and A. Fiech. Subtyping is not a good "match" for object-oriented languages. In *ECOOP*, pages 104–127, 1997.

[Cardelli et al.(1994)Cardelli, Martini, Mitchell, and Scedrov] 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.

[Crary(2000)] K. Crary. Typed compilation of inclusive subtyping. In *Proceedings of the International Conference on Functional Programming*, 2000.

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

[David and Guillaume(2001)] R. David and B. Guillaume. A lambda-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1), Feb. 2001.

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

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

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

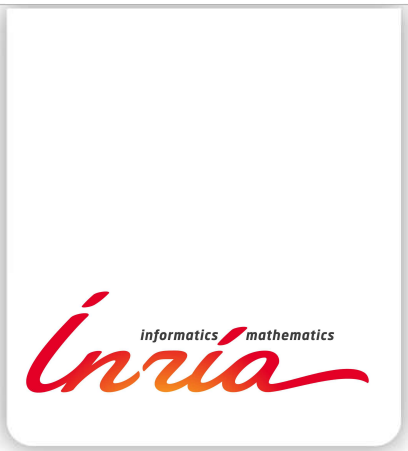
[Pottier(1996)] F. Pottier. Simplifying subtyping constraints. In *Proceedings of the International Conference on Functional Programming*, 1996.

[Rémy and Vouillon(1998)] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Riba(2007)] C. Riba. On the stability by union of reducibility candidates. In H. Seidl, editor, *FoSSaCS*, volume 4423 of *Lecture Notes in Computer Science*. Springer, 2007.

[Vanderwaart et al.(2003)Vanderwaart, Dreyer, Petersen, Crary, Harper, and Cheng] 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.

[Wells and Haack(2002)] J. B. Wells and C. Haack. Branching types. In *Proc. of the European Symposium On Programming Languages and Systems*, 2002.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399