Coherent Coercion Abstraction with a step-indexed strong-reduction semantics

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

Abstract

The usual notion of type coercions that witness subtyping relations between types is generalized to a more expressive notion of typing coercions that witness subsumption relations between typings, *e.g.* pairs composed of a typing environment and a type. This is more expressive and allows for a clearer separation of language constructs with and without computational content.

This is illustrated on a second-order calculus of implicit coercions that allows multiple but simultaneous type and coercion abstractions and has recursive coercions and general recursive types. The calculus is equipped with a very liberal notion of reduction. It models a wide range of type features including type containment, bounded and instance-bounded polymorphism, as well as subtyping constraints as used for ML-style type inference with subtyping.

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.

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

General Terms Design, Languages, Theory

Keywords Type, Typings, System F, F-eta, Polymorphism, Coercion, Retyping functions, Recursive coercions, Type constraints, Type containment, Subtyping, Bounded Polymorphism, Step-indexed semantics, Strong reduction, Recursive types.

1. Introduction

Type systems are syntactical languages to express properties and invariants of programs. Their objects are usually types, environments, typings, and typing derivations. These can be interpreted as mathematical objects or proofs. For instance, a typing judgment $\Gamma \vdash a : \tau$ can be interpreted as a proof that the term a never "goes wrong" and that its computational behavior is approximated by the type τ when the approximations of the behavior of its free variables are given by the typing context Γ.

Since program invariants are approximations of their exact behaviors, it is natural to consider the induced pre-order relation between such invariants. For example, subtyping was introduced to compare types as program approximations: a subtyping judgment $\tau \leq \sigma$ is a syntactical approximation of the pre-order relation between the interpretation of types τ and σ.

Used for long in object-oriented languages as a key feature, subtyping appears nowadays in many type systems, for numerical types, objects types, records types, variants types, private types, modules types, refinement types, capabilities, *etc.*

While type inference with subtyping is possible in ML using constraints [10], the interaction of subtyping and first-class polymorphism is often a source of difficulties. For example, checking subtyping constraints in some interesting variants of $\mathsf{F}_{<:}$ or checking Mitchell's notion of type containment are undecidable. Surprisingly, typing constraints in ML are more general and more regular than bounded quantification which privileges upper bounds to lower bounds, thus breaking the symmetry in the subtyping relation. Hence, whether $\mathsf{F}_{<:}$ can be extended with subtyping constraints is a natural question to ask, whose answer is not obvious.

Subtyping derivations are mathematical objects that must be manipulated by reasoning. *Type coercions* are concrete objects that can manipulate subtyping derivations by computation. Besides replacing reasoning by computing, *type coercions* can be provided explicitly, which allows to express certain forms of subtyping that cannot be checked mechanically.

Coercions are being used more and more often, as type systems are getting more sophisticated with advanced forms of type conversions that are hard to track implicitly. A typical example is the internal language F_c used in the Haskell compiler [16].

Explicit coercions may be maintained during reduction so as to preserve well-typedness. When coercions are just subtyping witnesses, they should have no computational content. That is, they should just change types of programs but not their meaning. In such cases, there is an underlying implicit calculus of coercions, where coercions are only used in subtyping derivations but do not appear in source programs—and obviously do not take part in reduction.

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, and we restrict our study to erasable coercions. Notice that the difference between explicit and implicit coercions is not always fundamental. For example, coercions for record subtyping may have no computation content in a language with a rich runtime where the representation of records carries some information about their domains, while a language with a simple runtime will have to copy records when subtyping them, as for ML modules.

Coercion abstraction In [6], we introduced a rich language of *type coercions* that allows to abstract over coercions themselves. Coercion abstraction increases expressiveness considerably. For example, it can express subtyping under subtyping assumptions.

Unfortunately, the types of abstract coercions must be restricted in some way to preserve type soundness. Indeed, using nonsensical

types for abstract coercions would amount to allowing arbitrary casts. In [6], we introduced a language called F_ι^p where abstract coercions have to be parametric in either their domain or their range. This condition ensures that abstract coercions cannot be used in between the destructor and the constructor of a redex, forming a pattern called a *wedge* which could typically block the reduction.

Despite this restriction, the language F_ι^p already models many features of type systems: type-containment found in the language F_η [9], bounded polymorphism found in $\mathsf{F}_{<:}$ [3], and instance-bounded polymorphism found in MLF [8], which can thus all be safely combined together in the same language.

Relaxing coercion abstraction Still, parametricity remains a strong restriction and some simple forms of subtyping do not fit in F_ι^p. For instance, subtyping constraints as proposed for adding subtyping to an ML-like language with type inference necessitate the use of multiple bounds for type variables, which cannot be expressed in F_ι^p. Moreover, abstract coercions between complex types must be decomposed into multiple abstract coercions between more atomic, hopefully parametric types.

Therefore, relaxing the restriction of F_ι^p seems necessary. Unfortunately, it also revealed quite challenging. The difficulties lie in the possibility for abstract coercions to appear in between redexes and block the reduction. This pattern is called a wedge in [6] and is of the form $(c\langle\lambda(x:\tau')\,M\rangle\,N)$. Some wedges such as arbitrary casting operations are really unsafe. The first challenge is to find reasonable restrictions under which only safe wedges will ever appear during reduction. Then, when coercions are kept during the reduction, as in F_ι^p, additional reduction steps must be introduced to break wedges apart and allow the reduction to proceed. However, the residual of broken wedges must themselves be expressible as (new forms of) coercions—and typable. This is the most challenging part: since coercions may introduce binders, breaking them apart introduces residual coercions with unusual scoping rules. Solving one difficulty immediately uncovers another one (§6.1).

Simultaneous coercion abstractions In the rest of this paper we present a solution to relax coercion abstraction that compromises between expressiveness and simplicity.

We relax the parametricity restriction of F_ι^p, allowing coercions whose domain and range are simultaneously structured types. We also allow multiple *coercion* abstractions to constrain multiple *type* abstractions provided they are introduced *simultaneously*.

Coherence does not come by construction as in F_ι^p. Instead, coherence proofs must be provided explicitly 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_ι^c. 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 [1] are derivable (§2).

Implicit coercions While coercions are explicit in F_ι^p and maintained during reduction, we leave them implicit in our new proposal, so as to surround the difficulties raised by wedges. Indeed, since coercions are not represented in the source, there are no wedges any longer. We thus avoid having to introduce new forms of coercions for reducing wedges and twisting the type system to type these new forms. This simplifies the presentation considerably.

Explicit coercions were a real advantage over implicit coercions in the language F_ι^p as their reduction remained relatively simple. However, while computing is usually easier than reasoning, the lesson is that when coercion reduction becomes too intricate, simple mathematical reasoning becomes preferable over too complex computation steps and typing rules.

From type coercions to typing coercions Besides the introduction of simultaneous coercion abstractions, we also generalize our approach to coercions. 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 [6], is now internalized. Typing coercions generalize type coercions.

Conversely, there are type system features that can be described by typing coercions but not by type coercions. Type generalization is one example: 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.* found in F_η) are now derivable by composing type generalization, type instantiation, and η-expansion (a generalization of the subtyping rule for the arrow type).

The advantage of using *typing* coercions is particularly striking in the fact that all type system features studied in this paper can be expressed as coercions, so that computation and typing features are perfectly separated. This can already be seen in Figure 4, but more details can be found in Section §2.

Type soundness proof via step-indexed terms While moving from explicit to implicit coercions, we are simultaneously changing our approach to type soundness. Instead of the standard syntactic proof based on subject reduction and progress, which became lengthy and intractable, a semantic approach where types are interpreted as sets of terms is better suited and more concise.

Our calculus is equipped with a strong reduction semantics, for several reasons. As our language is implicitly typed, both type abstraction and coercion abstraction are implicit and cannot reasonably stop evaluation. However, even if they were explicit, as in F_ι^p, they could not stop evaluation because our assumption is that coercions are erasable and thus do not change the underlying semantics of the untyped term. While it would be possible to choose a weak reduction strategy just for value abstraction, and a programming language will probably do so, there is no reason to do so in our calculus: proving type soundness for a more liberal strong reduction strategy gives a stronger result. Indeed, weak-reduction strategies are a subset of strong reduction strategies, hence by restricting the semantics afterward, the absence of errors during the reduction will still be guaranteed.

Moreover, this better fits our intuition of a safe type system. Even if we eventually pick a weak reduction strategy, our system

should be sound for all reduction strategies. Indeed, it would be weird if the type system accepted as well-typed a program containing the addition of an integer and a boolean when appearing under an abstraction.

Since our calculus features general recursive types, we follow a step-indexed approach. Unfortunately, the step-indexed approaches to type soundness do not seem to work directly for strong reduction strategies. We propose a new approach based on step-indexing terms that pushes the indices inside terms themselves, which we believe is another interesting side contribution of our work.

Summary Our contribution is multiple facets. First, we introduce a language F_ι^c with simultaneous coercion abstractions that relaxes the parametricity restriction and is thus more expressive than F_ι^p. Second, our language F_ι^c can also model ML-style type constraints—with some differences. This shows for instance that MLF can be safely combined with ML-style type constraints—leaving type inference aside of course. Third, this includes a general form of recursive coercions, from which we can recover powerful subtyping rules between equi-recursive types. Fourth, we present our coercions as an implicit calculus with a denotational semantics soundness proof, which provides another new insight into coercions. Fifth, we generalize our coercion framework from type coercions to typing coercions, bringing a clearer separation of language constructs. Last, we adapt step-indexed semantics to strong reduction strategies by moving indices inside terms.

The rest of the paper is organized as follows. We present F_ι^c—our type system for the λ-calculus—to illustrate these ideas in §2. We introduce our variant of step-indexed denotational semantics in Section §3 and apply it to prove the soundness of our calculus in Section §4. We discuss the expressivity of F_ι^c in §5 and differences with our previous work and other related works as well as future works in Section §6.

The language F_ι^c and its soundness proof have been formalized and mechanically verified[1] in Coq [14].

2. Language definition

Since our calculus is implicitly typed, its syntax is that of the λ-calculus extended with pairs, reminded on Figure 1. Terms are variables x, abstractions $\lambda x\, a$, applications $(a\, b)$, pairs $\langle a, b\rangle$, and projections $\pi_{\mathrm{i}}\ a$ for i in $\{1, 2\}$.

The reduction rules are given on Figure 3. We write $a[x \leftarrow b]$ for the capture avoiding substitution of the term b for the variable x in the term a, defined as usual. Head reduction is described by the β-reduction rule RedApp, and the projection rule RedProj. Reduction can be used under any evaluation context as described by Rule RedCtx. Evaluation contexts, written E, are defined on Figure 2. Since we choose a strong reduction relation, all possible contexts are allowed. Notice that evaluation contexts contain a single node, since the context rule RedCtx can be applied recursively.

The terms we are interested are the sound ones, *i.e.* whose evaluation never produces an error. We write Ω for the set of *errors*. They are the subset of syntactically well-formed terms that "we don't want to see" neither in source programs nor during their evaluation. Namely, an error r is either immediate, *i.e.* the application of a pair or the projection of an abstraction, or an error occurring in an arbitrary context E. Notice that we always put parentheses around applications.

The absence of errors implies as a corollary that well-behaved terms that cannot be further reduced are values. Values are given on Figure 2. They are either constructors applied to values, or prevalues, which are either variables or destructors applied to prevalues.

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

$$
\begin{array}{rcll}
\alpha, \beta & & & \text{Type variables} \\
x, y & & & \text{Term variables} \\
\tau, \sigma, \rho & ::= & \alpha \mid \tau \to \tau \mid \tau \times \tau & \text{Types} \\
& & \mid\ \forall(\overline{\alpha}, C)\, \tau \mid \mu\alpha\, \tau \mid \bot \mid \top & \\
a, b & ::= & x \mid \lambda x\, a \mid (a\, a) \mid \langle a, a\rangle \mid \pi_1\ a \mid \pi_2\ a & \text{Terms} \\
B & ::= & (x : \tau) \mid (\overline{\alpha}, C) & \text{Bindings} \\
C & ::= & \varnothing \mid C, \tau \triangleright \tau & \text{Coercions} \\
\Gamma & ::= & \varnothing \mid \Gamma, B & \text{Environments} \\
\Theta & ::= & \varnothing \mid \Theta, (\tau \triangleright \tau) \mid \Theta, (\tau \blacktriangleright \tau) & \text{Fix coercions} \\
\mathsf{wf} & ::= & \mathsf{WF} \mid \mathsf{NE} & \text{Well-foundness}
\end{array}
$$

Figure 1. Syntax

$$
\begin{array}{rcll}
\Sigma & ::= & \varnothing \mid \Sigma, (\overline{\alpha}, C) & \text{Coercion environments} \\
p & ::= & x \mid (p\, v) \mid \pi_{\mathrm{i}}\ p & \text{Prevalues} \\
v & ::= & p \mid \lambda x\, v \mid \langle v, v\rangle & \text{Values} \\
r & ::= & E[r] \mid (\langle a, a\rangle\ a) \mid \pi_{\mathrm{i}}\ (\lambda x\, a) & \text{Errors} \\
E & ::= & \lambda x\, [] \mid ([]\ a) \mid (a\ []) \mid \langle [], a\rangle \mid \langle a, []\rangle \mid \pi_{\mathrm{i}}\ [] & \text{Contexts}
\end{array}
$$

Figure 2. Notations

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

Figure 3. Reduction relation

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

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

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

Figure 4. Term typing rules

We use types to approximate the behavior of terms and we use environments to approximate the behavior of variables. Types are given on Figure 1. They contain type variables α, arrow types $\tau \to \sigma$, product types $\tau \times \sigma$, block-abstractions $\forall(\overline{\alpha}, C)\, \rho$, recursive types $\mu\alpha\, \tau$, the top type $\top$, and the bottom type $\bot$. We write $\overline{\alpha}$ for a sequence of variables. Coercions C are possibly empty sequences of elementary coercions $\tau \triangleright \sigma$. We also sometimes write $\overline{\tau \triangleright \sigma}$ for a sequence of coercions instead of C. We also write $\tau[\alpha \leftarrow \sigma]$ for the capture avoiding substitution of the type σ for the variable α in the type τ. We write $\mathsf{fv}(\tau)$ the set of free variables of τ, defined in the obvious way. We extend free variables and substitutions to coercions and sequences of coercions pointwise. The syntax of environments is given on Figure 1. They are simply lists of binders, which are term binders $(x : \tau)$ or block binders $(\overline{\alpha}, C)$.

We write $a : \Gamma \vdash \tau$ instead of the usual notation $\Gamma \vdash a : \tau$ to mean that the term a has the typing $\Gamma \vdash \tau$, *i.e.* its behavior is approximated by τ whenever its free variables are in the approximations described by Γ. This can also be read syntactically in the usual way as "term a can be typed with τ in environment Γ". The

judgment $a : \Gamma \vdash \tau$ implies that τ is well-formed under Γ which implies that Γ is also well-formed, as stated below by the extraction lemma (Lemma 19). Typing rules for term derivations are given on Figure 4. Observe that these are exactly the typing rules of the simply-typed λ-calculus, except for the new rule TERMCOER. This rule says that if a term a admits the typing $\Gamma, \Sigma \vdash \tau$ and there exists a coercion from the typing $\Gamma, \Sigma \vdash \tau$ to the typing $\Gamma \vdash \sigma$, which we write as $\Gamma \vdash \Sigma\tau \triangleright \sigma$, then the term a also admits the typing $\Gamma \vdash \sigma$.

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.

The general coercion typing judgment $\Gamma; \Theta \vdash \Sigma\tau \triangleright \sigma$ is defined on Figure 5. The coercion environment Θ contains additional coercions hypotheses used coinductively. When Θ is empty, we just write $\Gamma \vdash \Sigma\tau \triangleright \sigma$. We first explain typing rules ignoring the environment Θ, as it only matters for coercion fixpoints. We explain the role of Θ last, together with Rule COERFIX.

Intuitively, the judgment $\Gamma \vdash \Sigma\tau \triangleright \sigma$ 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 $\Gamma; \varnothing \vdash \Sigma\tau \triangleright \sigma$ is derivable; therefore, the coercion typing judgment is incomplete, unsurprisingly.) One could expect the right premise of TERMCOER to be written $(\Gamma, \Sigma \vdash \tau) \triangleright (\Gamma \vdash \sigma)$, or more generally of the form $(\Gamma_1 \vdash \tau_1) \triangleright (\Gamma_2 \vdash \tau_2)$. However, we only use this judgment when the environment on the left-hand side is an extension of the environment on the right-hand side.[2] Thus, we maintain this invariant in the syntax and just write $\Gamma \vdash \Sigma\tau \triangleright \sigma$. Moreover, when Σ is empty, we write $\Gamma \vdash C$ for the conjunction of judgments $\Gamma \vdash \tau \triangleright \sigma$ for all coercions $\tau \triangleright \sigma$ in C. Notice that $\Gamma; \Theta \vdash \Sigma\tau \triangleright \sigma$ implies that τ is well-formed under Γ, Σ and σ is well-formed under Γ, which again implies that Γ and Γ, Σ are well-formed as stated by the extraction lemma (Lemma 19). In particular, Θ is ignored by well-formedness judgments.

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 $\Gamma, \Sigma_2, \Sigma_1 \vdash \tau_1$ ignoring Θ. 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.

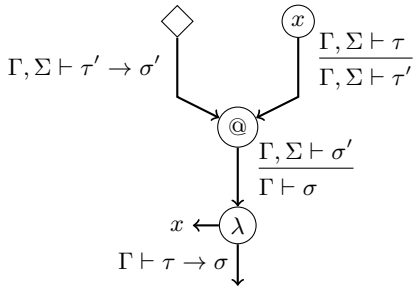
The Rule COERWEAK implements a form of weakening. It tells that if any term of typing $\Gamma, \Sigma \vdash \tau$ can be seen as $\Gamma \vdash \sigma$, then any term of typing $\Gamma \vdash \tau$ can also be seen as $\Gamma \vdash \sigma$. Since weakening holds for term judgments, we can do the following 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 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 not derivable from other rules—nor admissible (if we removed it from the definition). Notice that this is the only rule that removes binders. The two next 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 $\Gamma \vdash \top$.

[2] The general case would be useful if we treated weakening explicitly as a coercion, but weakening is much simpler implicitly.

The next two rules, COERETAPROD and COERETAARR, 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 since 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 of the arrow type which is $\lambda x\,([]\,x)$. If we place in the hole a term with typing $\Gamma, \Sigma \vdash \tau' \to \sigma'$, we may give $\lambda x\,([]\,x)$ the typing $\Gamma \vdash \tau \to \sigma$ provided a coercion of type $\Gamma, \Sigma \vdash \tau \triangleright \tau'$ is applied around x. The result of the application has typing $\Gamma, \Sigma \vdash \sigma'$ which can in turn be coerced to $\Gamma \vdash \sigma$ if there exists a coercion of type $\Gamma \vdash \Sigma\sigma' \triangleright \sigma$. 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 argument a needs not bind variables, since the variable a could not use them anyway.



The next three rules, COERVAR, COERGEN, and COERINST, implement the main feature of the language, namely simultaneous coercion abstractions. Rule COERVAR for coercion assumptions reads the coercion type from the context. Rule COERGEN combines several type and coercion abstractions. It uses an auxiliary binding judgment, described below, to check that the binding is well-formed and thus coherent. Rule COERINST performs simultaneous instantiation of types and coercions with a substitution $[\overline{\alpha \leftarrow \sigma}]$: the premises check well-formedness of the types $\overline{\sigma}$ and well-typedness of the instantiation of the coercions with $[\overline{\alpha \leftarrow \sigma}]$; the conclusion returns the instantiation of the schema with $[\overline{\alpha \leftarrow \sigma}]$. Notice that COERGEN is the only rule that extends the environment. This rule can be presented as a typing coercion but could not have been written as a type coercion.

The next two rules COERFOLD and COERUNFOLD are the usual folding and unfolding of recursive types, which give the equivalence between $\mu\alpha\,\tau$ and $\tau[\alpha \leftarrow \mu\alpha\,\tau]$.

We may now reread the previous rules with the additional environment Θ—without changing the explanation, except for rules COERVAR and the two η-expansion rules, which we explain together with Rule COERFIX. This rule provides an induction principle for building recursive coercions. Namely, it allows to build a coercion by coinduction by adding the goal as an hypothesis in Θ. One may first read the rule ignoring coercions C, which are there to reinforce the coinduction hypothesis: that is, we may also use C coinductively to prove the conclusion $\tau \triangleright \sigma$, provided we may also prove C coinductively. Of course, coinduction would be ill-founded if we could use the coinductive hypotheses immediately. Therefore, hypotheses are first introduced as locked coercions of the form $\tau \blacktriangleright \sigma$, as opposed to a free coercion $\tau \triangleright \sigma$. $C^{\blacktriangleright}$ stands for C where every free coercion $\tau_1 \triangleright \tau_2$ has been changed into the locked coercion

$$\textsc{WFVar}\ \frac{}{\alpha \mapsto \alpha : \mathsf{NE}} \qquad \textsc{WFCst}\ \frac{\alpha \notin \mathsf{fv}(\tau)}{\alpha \mapsto \tau : \mathsf{WF}} \qquad \textsc{WFSub}\ \frac{\alpha \mapsto \tau : \mathsf{WF}}{\alpha \mapsto \tau : \mathsf{NE}}$$

$$\textsc{WFArr}\ \frac{\alpha \mapsto \tau : \mathsf{NE} \qquad \alpha \mapsto \sigma : \mathsf{NE}}{\alpha \mapsto \tau \to \sigma : \mathsf{WF}} \qquad \textsc{WFProd}\ \frac{(\alpha \mapsto \tau_i : \mathsf{NE})^{i \in \{1,2\}}}{\alpha \mapsto \tau_1 \times \tau_2 : \mathsf{WF}}$$

$$\textsc{WFMulti}\ \frac{\alpha \notin \mathsf{fv}(C) \qquad \alpha \mapsto \rho : \mathsf{wf}}{\alpha \mapsto \forall(\overline{\alpha}, C)\, \rho : \mathsf{wf}} \qquad \textsc{WFMu}\ \frac{\beta \mapsto \tau : \mathsf{WF} \qquad \alpha \mapsto \tau : \mathsf{wf}}{\alpha \mapsto \mu\beta\, \tau : \mathsf{wf}}$$

Figure 6. Well-foundness judgment relation

$\tau_1 \blacktriangleright \tau_2$. Locked coercions are inaccessible through Rule CoerVar which only searches for free coercions in Γ or Θ. However, locked coercions are freed once we use an η-expansion rule which are productive since computational. This is realized by changing Θ into $\Theta^{\triangleright}$ into the premises of rules CoerEtaArrow and CoerEtaProd which unlocks all coercions of the form $\tau_1 \blacktriangleright \tau_2$ in Θ into free coercions $\tau_1 \triangleright \tau_2$, making them available for coinductive reasoning via Rule CoerVar.

Interestingly, the usual rules for reasoning on recursive types [1] are derivable using CoerFix ($\tau_1 \lhd\rhd \tau_2$ means $\tau_1 \triangleright \tau_2, \tau_2 \triangleright \tau_1$).

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

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

The judgment $\Gamma \vdash \tau$ defines when type τ is well-formed under the environment Γ. We also write $\Gamma \vdash \overline{\tau}$ as an abbreviation for the conjunction $\Gamma \vdash \tau$ for all types τ in the sequence $\overline{\tau}$. The rules for this type judgment, given on Figure 7, are as usual. A type variable is well-formed if it is bound in its environment (Rule TypeVar). Top and Bottom types are unconditionally well-formed (rules TypeBot and TypeTop). Product and arrow types are well-formed if both components are well-formed under the same environment (rules TypeProd and TypeArr). A polymorphic type is well-formed if its binders are erasable and well-formed under the same environment and its body is well-formed under the extended environment. A recursive type is well-formed if it is well-founded and its body is well-formed under the extended environment.

The most important rule is BindTypeCoer, which defines when an erasable binder is well-formed and also checks for coherence. Judgments for well-formedness of bindings are written $\Gamma \vdash B$. Term binding is defined as usual, the variable has to be disjoint from the domain of the environment and its type has to be well-formed under this environment. The simultaneous binding of type and coercion variables is more involved: first, bound type variables $\overline{\alpha}$ must be disjoint from the domain of the current environment; then, both the domain and the range of the coercion type must be well-formed under the current environment extended with type variables $\overline{\alpha}$; finally, the last two premises ensure the coherence of the coercions $\overline{\tau_1 \triangleright \tau_2}$ requiring that there exists a substitution $[\overline{\alpha \leftarrow \sigma}]$ that makes the coercions $\overline{\tau_1 \triangleright \tau_2}$ well-formed under Γ.

Notice that the last two premises of Rule BindTypeCoer are the same as the premises of Rule CoerInst, since checking coherence is just checking that the erasable binder can be instantiated in the current environment. The substitution $[\overline{\alpha \leftarrow \sigma}]$ is called the witness for the coherence of this binder. It is not explicitly given, since coercions are implicit. An explicit calculus would provide them explicitly, so should a surface language, since inferring witnesses is certainly undecidable in the general case.

We write $\vdash \Gamma$ for the well-formedness of environments, defined on Figure 7. Well-formed environments are the empty one (Rule EnvEmpty) or the concatenation of a well-formed environment with a well-formed binding in this environment (Rule EnvBind).

Finally, the well-foundedness judgment, written $\alpha \mapsto \tau : \mathsf{wf}$, means that $\alpha \mapsto \tau$ is well-founded if wf is WF or non-expansive if wf is NE. Rule WFVar tells that the identity functor is non-expansive. Rule WFCst tells that constant functors are well-founded. Rule WFSub tells that well-founded functors are non-expansive. Rules WFArr and WFProd tell that arrow and product types are well-founded if there components are non-expansive. Rules WFMulti and WFMu tell that polymorphic and recursive types are well-founded (resp. non-expansive) if there body is well-founded (resp. non-expansive). Additionally for polymorphic types, the abstract coercions should not mention the recursive type variable, and for recursive types, it has to be well-founded as well.

The dependencies between these judgments is as follows: well-foundedness do not have dependencies; type, coercion, binding, and environment well-formedness are mutually defined and depend on well-foundedness; and term judgment depends on the preceding.

3. Semantics

A term is sound if none of its reduction 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 and pairs, while neutral terms ∇ are variables, applications, and projections. 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.*

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.

$$\textsc{CoerRefl}\ \frac{\Gamma \vdash \tau}{\Gamma;\Theta \vdash \tau \triangleright \tau} \qquad \textsc{CoerTrans}\ \frac{\Gamma;\Theta \vdash \Sigma_2\tau_2 \triangleright \tau_3 \qquad \Gamma,\Sigma_2;\Theta \vdash \Sigma_1\tau_1 \triangleright \tau_2}{\Gamma;\Theta \vdash (\Sigma_2,\Sigma_1)\tau_1 \triangleright \tau_3}$$

$$\textsc{CoerWeak}\ \frac{\Gamma;\Theta \vdash \Sigma\tau \triangleright \sigma \qquad \Gamma \vdash \tau}{\Gamma;\Theta \vdash \tau \triangleright \sigma} \qquad \textsc{CoerBot}\ \frac{\Gamma \vdash \tau}{\Gamma;\Theta \vdash \bot \triangleright \tau} \qquad \textsc{CoerTop}\ \frac{\Gamma \vdash \tau}{\Gamma;\Theta \vdash \tau \triangleright \top}$$

$$\textsc{CoerEtaArr}\ \frac{\Gamma \vdash \tau' \qquad \Gamma,\Sigma;\Theta^{\triangleright} \vdash \tau' \triangleright \tau \qquad \Gamma;\Theta^{\triangleright} \vdash \Sigma\sigma \triangleright \sigma'}{\Gamma;\Theta \vdash \Sigma(\tau \to \sigma) \triangleright \tau' \to \sigma'} \qquad \textsc{CoerEtaProd}\ \frac{(\Gamma;\Theta^{\triangleright} \vdash \Sigma\tau_i \triangleright \tau_i')^{i\in\{1,2\}}}{\Gamma;\Theta \vdash \Sigma(\tau_1 \times \tau_2) \triangleright \tau_1' \times \tau_2'}$$

$$\textsc{CoerVar}\ \frac{\tau \triangleright \sigma \in \Gamma;\Theta \qquad \Gamma \vdash \tau \qquad \Gamma \vdash \sigma}{\Gamma;\Theta \vdash \tau \triangleright \sigma}$$

$$\textsc{CoerGen}\ \frac{\Gamma \vdash \forall(\overline{\alpha}, C)\,\rho}{\Gamma;\Theta \vdash (\overline{\alpha}, C)\rho \triangleright \forall(\overline{\alpha}, C)\,\rho} \qquad \textsc{CoerInst}\ \frac{\Gamma \vdash \forall(\overline{\alpha}, C)\,\rho \qquad \Gamma \vdash \overline{\sigma} \qquad \Gamma \vdash C[\overline{\alpha \leftarrow \sigma}]}{\Gamma;\Theta \vdash \forall(\overline{\alpha}, C)\,\rho \triangleright \rho[\overline{\alpha \leftarrow \sigma}]}$$

$$\textsc{CoerFold}\ \frac{\Gamma \vdash \mu\alpha\,\tau}{\Gamma;\Theta \vdash \tau[\alpha \leftarrow \mu\alpha\,\tau] \triangleright \mu\alpha\,\tau}$$

$$\textsc{CoerUnfold}\ \frac{\Gamma \vdash \mu\alpha\,\tau}{\Gamma;\Theta \vdash \mu\alpha\,\tau \triangleright \tau[\alpha \leftarrow \mu\alpha\,\tau]} \qquad \textsc{CoerFix}\ \frac{\Gamma;\Theta, C^{\blacktriangleright}, (\tau \blacktriangleright \sigma) \vdash C, (\tau \triangleright \sigma)}{\Gamma;\Theta \vdash \tau \triangleright \sigma}$$

Figure 5. Coercion typing rules

$$\textsc{TypeVar}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{TypeArr}\ \frac{\Gamma \vdash \tau,\sigma}{\Gamma \vdash \tau \to \sigma} \qquad \textsc{TypeProd}\ \frac{\Gamma \vdash \tau_1,\tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \textsc{TypeBot}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot} \qquad \textsc{TypeTop}\ \frac{\vdash \Gamma}{\Gamma \vdash \top}$$

$$\textsc{TypeMu}\ \frac{\alpha \mapsto \tau : \mathsf{WF} \qquad \Gamma,\alpha \vdash \tau}{\Gamma \vdash \mu\alpha\,\tau} \qquad \textsc{TypeMulti}\ \frac{\Gamma \vdash (\overline{\alpha}, C) \qquad \Gamma,(\overline{\alpha}, C) \vdash \rho}{\Gamma \vdash \forall(\overline{\alpha}, C)\,\rho}$$

$$\textsc{EnvEmpty}\ \vdash \varnothing \qquad \textsc{EnvBind}\ \frac{\vdash \Gamma \qquad \Gamma \vdash B}{\vdash \Gamma, B} \qquad \textsc{BindTerm}\ \frac{x \notin \mathsf{dom}(\Gamma) \qquad \Gamma \vdash \tau}{\Gamma \vdash (x : \tau)}$$

$$\textsc{BindTypeCoer}\ \frac{\overline{\alpha} \notin \mathsf{dom}(\Gamma) \qquad \Gamma,\overline{\alpha} \vdash \overline{\tau_1},\overline{\tau_2} \qquad \Gamma \vdash \overline{\sigma} \qquad \Gamma \vdash \overline{\tau_1 \triangleright \tau_2}[\overline{\alpha \leftarrow \sigma}]}{\Gamma \vdash (\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})}$$

Figure 7. Well-formedness judgments for types, environments, and bindings

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

Figure 8. Lower function

$$\begin{array}{rcl} e, f & ::= & x^k \mid \lambda^k x\, e \mid (e\, e)^k \mid \langle e, e \rangle^k \mid {\pi_{\mathrm{i}}}^k\, e \\ E^k & ::= & \lambda^k x\, [] \mid ([]\, e)^k \mid (e\, [])^k \mid \langle [], e \rangle^k \mid \langle e, [] \rangle^k \mid {\pi_{\mathrm{i}}}^k\, [] \\ s & ::= & E^k[s] \mid (\langle e, e \rangle^k\, e)^k \mid {\pi_{\mathrm{i}}}^k\, (\lambda^k x\, e) \end{array}$$

Figure 9. Syntax of indexed terms

3.1 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 9: indexed terms are variables x^k, abstractions $\lambda^k x\, e$, applications $(e\, f)^k$, pairs $\langle e, f \rangle^k$, and projections ${\pi_{\mathrm{i}}}^k\, e$. We consistently label one hole contexts and errors. We write E^k for one-hole context with index k and s for indexed errors.

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 8. 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.

The reduction rules of the indexed calculus mimic those of the λ-calculus, but with some index manipulation, as described on Figure 10. 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 and FREDPROJ) and to reduction in an evaluation context (Rule FREDCTX). That is, a head reduction can only be applied along a path of the form $E_1^{k_1}[\ldots E_p^{k_p}[e]]$ 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:

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

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.

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

Figure 10. Indexed calculus reduction relation

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

Figure 11. Lifting integer predicates to indexed terms

3.2 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.

We also overload the notation $\mathcal{S}$ for the set of sound indexed terms. Although it is defined as for λ-terms as $(\rightsquigarrow^\star \mho)$, 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 11.

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 $\lfloor e' \rfloor = a'$.*

3.3 Semantic types

In order 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 on the same λ-term) and the indices of corresponding nodes to be related by the property on indices. A formal definition is given on Figure 12.

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.

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

Figure 12. Lifting of a binary predicate $\star$ on indices to terms

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

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)**.** $\mathbb{T} \stackrel{\textit{def}}{=} \{\mathsf{R} \in \mathbb{P} \mid \diamondsuit\,\mathsf{R} \subseteq \mathsf{R}\}$.

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 invert 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.

3.4 Simple types

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

Definition 7 (Arrow and product operators)**.**

$$\mathsf{R} \hat{\to} \mathsf{S} \stackrel{\textit{def}}{=} \diamondsuit\,\{\lambda^k x\, e \in \mathcal{S} \mid k > 0 \Rightarrow \forall f, \lfloor f \rfloor_{k-1} \in \mathsf{R} \Rightarrow \lfloor e[x \leftarrow f] \rfloor_{k-1} \in \mathsf{S}\}$$

$$\mathsf{R} \hat{\times} \mathsf{S} \stackrel{\textit{def}}{=} \diamondsuit\,\{\langle e, e' \rangle^k \in \mathcal{S} \mid k > 0 \Rightarrow \lfloor e \rfloor_{k-1} \in \mathsf{R} \wedge \lfloor e' \rfloor_{k-1} \in \mathsf{S}\}$$

The arrow and product operators preserve types.

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

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 9. *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 9. Since the substitution and the lowering function commute, we conclude (15). □

3.5 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.)

3.6 Recursive types

This section follows the usual description of recursive types using approximations as done in [2]. 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 10 (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 11 (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 12. *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 [2]:

Lemma 13. *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 (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.

3.7 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 14 (Semantic judgment)**.**

$$\gamma : \mathsf{G} \quad \stackrel{\mathit{def}}{=} \quad \forall (x : \mathsf{R}) \in \mathsf{G}, \gamma\, x \in \mathsf{R}$$
$$\mathsf{G} \models \mathsf{S} \quad \stackrel{\mathit{def}}{=} \quad \{e \mid \forall \gamma : \mathsf{G}, \gamma\, e \in \mathsf{S}\}$$

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

Lemma 15 (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 16 (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* $\mathsf{G} \models \mathsf{R} \mathbin{\hat{\to}} \mathsf{S}$.

Proof. Let γ be compatible with G (**1**). We show that $\gamma\,(\lambda^k x\, e)$ is in $\mathsf{R} \mathbin{\hat{\to}} \mathsf{S}$ (**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 17 (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 18 (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* $\mathsf{G} \models \mathsf{R}_i$*.*

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.

4. Soundness

In order to show the soundness property we need the extraction lemma (Lemma 19), which uses the usual weakening and substitution lemmas.

Lemma 19 (Extraction)**.** *The following properties hold:*

- *If* $\Gamma \vdash \tau$ *holds, then* $\vdash \Gamma$ *holds.*
- *If* $\Gamma; \Theta \vdash \Sigma\tau \triangleright \sigma$ *holds, then* $\Gamma, \Sigma \vdash \tau$ *and* $\Gamma \vdash \sigma$ *hold.*
- *If* $a : \Gamma \vdash \tau$ *holds, then* $\Gamma \vdash \tau$ *holds.*

The soundness proof is not direct. We will translate F_ι^c 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 on 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 20. *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) types and typing environments. 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 top semantic type is returned. The interpretation of arrow and product types simply use the arrow and product operators defined in §3.4. The interpretation of the polymorphic type $\forall(\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})\rho$ under η is the intersection of all interpretations of ρ under η' when η' ranges in $\mathcal{I}_\eta(\overline{\alpha}, \overline{\tau_1}, \overline{\tau_2})$, *i.e.* all extensions of η mapping $\overline{\alpha}$ to $\overline{\mathsf{R}}$ that validate the inclusions induced by the coercions $\overline{\tau_1 \triangleright \tau_2}$. The interpretation of the recursive type $\mu\alpha\, \tau$ under η is the infinite iteration of the functor mapping X under the extension of η mapping α to X—which corresponds to the infinite unfolding of the recursive type. Finally, top and bottom are mapped to their semantic equivalent.

Definition 21 (Type interpretation)**.** *The interpretation of a type* τ *under* η*, written* $|\tau|_\eta$ *is the set of terms recursively defined as:*

- $|\alpha|_\eta = \eta(\alpha)$
- $|\tau \to \sigma|_\eta = |\tau|_\eta \hat{\to} |\sigma|_\eta$ *and* $|\tau \times \sigma|_\eta = |\tau|_\eta \hat{\times} |\sigma|_\eta$
- $|\forall(\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})\rho|_\eta = \bigcap_{\eta' \in \mathcal{I}_\eta(\overline{\alpha}, \overline{\tau_1}, \overline{\tau_2})} |\rho|_{\eta'}$
- $|\mu\alpha\, \tau|_\eta = \hat{\mu}\,(X \mapsto |\tau|_{\eta, \alpha \mapsto X})$
- $|\bot|_\eta = \hat{\bot}$ *and* $|\top|_\eta = \hat{\top}$

where $\mathcal{I}_\eta(\overline{\alpha}, \overline{\tau_1}, \overline{\tau_2})$ *is defined as*
$\{\eta' \mid \exists \overline{R \in \mathbb{T}}, \eta' = \eta, \overline{\alpha \mapsto R} \wedge \overline{|\tau_1|_{\eta'} \subseteq |\tau_2|_{\eta'}}\}$

We define the interpretation of environments 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 an erasable binding $(\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})$ extends the type mapping η' according to $\mathcal{I}_{\eta'}(\overline{\alpha}, \overline{\tau_1}, \overline{\tau_2})$.

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

Definition 22 (Environment semantic)**.** *We define* $|\Gamma|_{\gamma\eta}$ *as follows:*

- $|\varnothing|_{\gamma\eta} = \{\gamma\eta\}$
- $|\Gamma, (x : \tau)|_{\gamma\eta} = \{(\gamma', x \mapsto e)\eta' \mid \gamma'\eta' \in |\Gamma|_{\gamma\eta} \wedge e \in |\tau|_{\eta'}\}$
- $|\Gamma, (\overline{\alpha}, \overline{\tau_1 \triangleright \tau_2})|_{\gamma\eta} = \{\gamma'\eta'' \mid \gamma'\eta' \in |\Gamma|_{\gamma\eta} \wedge \eta'' \in \mathcal{I}_{\eta'}(\overline{\alpha}, \overline{\tau_1}, \overline{\tau_2})\}$

Lemma 23 (Semantic weakening)**.** *If* $\eta' \in |\Gamma|_\eta$ *and* $\mathsf{dom}(\Gamma)$ *is disjoint from* $\mathsf{fv}(\tau)$*, then* $|\tau|_{\eta'} = |\tau|_\eta$ *holds.*

Lemma 24 (Semantic substitution)**.** $|\tau|_{\eta'} = |\tau[\overline{\alpha \leftarrow \sigma}]|_{\eta, \eta''}$ *where* $\eta' = \eta, \overline{\alpha \mapsto |\sigma|_\eta}, \eta''$*.*

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 type is well-formed, then its interpretations under all type mappings in the interpretation of its environment are semantic types. If a coercion is well-formed, then the intersection of its domain type is included in the interpretation of its codomain type. The intersection is taken according to the environment extension Σ and hypotheses in Θ are taken at the right level. Finally, the application of a substitution γ to a term of type τ in an environment Γ is in the interpretation of τ under η for all $\gamma\eta$ in the interpretation of Γ.

$$\lceil x \rceil^k = x^k \qquad \lceil \langle a_1, a_2 \rangle \rceil^k = \langle \lceil a_1 \rceil^k, \lceil a_2 \rceil^k \rangle^k$$
$$\lceil \lambda x\, a \rceil^k = \lambda^k x\, \lceil a \rceil^k \qquad \lceil \pi_{\mathrm{i}}\, a \rceil^k = \pi_{\mathrm{i}}{}^k\, \lceil a \rceil^k$$
$$\lceil (a\, b) \rceil^k = (\lceil a \rceil^k\, \lceil b \rceil^k)^k$$

Figure 13. Fill function

Lemma 25. *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* $\vdash \Gamma, \Gamma'$ *holds, then* $\forall \gamma\eta \in |\Gamma|\,,\ |\Gamma'|_{\gamma\eta}$ *is not empty.*
- *If* $\Gamma \vdash \tau$ *holds, then* $\forall \gamma\eta \in |\Gamma|\,, |\tau|_\eta \in \mathbb{T}$ *holds.*
- *If* $\Gamma; \Theta \ \vdash\ \Sigma\tau \triangleright \sigma$ *holds, then* $\forall \gamma\eta \in |\Gamma|\,, \forall k, |\Theta_0|_\eta^k \wedge |\Theta_1|_\eta^{k-1} \Rightarrow \forall e < k, (\forall \eta' \in |\Sigma|_\eta\,, e \in |\tau|_{\eta'}) \Rightarrow e \in |\sigma|_\eta$ *holds where* $\Theta_0 = \{\tau \triangleright \sigma \in \Theta\}$ *and* $\Theta_1 = \{\tau \blacktriangleright \sigma \in \Theta\}$.
- *If* $e : \Gamma \vdash \tau$ *holds, then* $\forall \gamma\eta \in |\Gamma|\,, \gamma\, e \in |\tau|_\eta$ *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 13). By construction, we have $\lfloor \lceil a \rceil^k \rfloor = a$.

Theorem 26 (Soundness)**.** *If* $a : \Gamma \vdash \tau$ *holds, then* $a \in \mathcal{S}$.

Proof. Let $a \rightsquigarrow^\star b$, we have to show $b \in \mathcal{U}$. Let k be the length of the reduction and e be $\lceil a \rceil^k$. We have $e : \Gamma \vdash \tau$ by Lemma 20. By Lemma 19 we have $\vdash \Gamma$ and $\Gamma \vdash \tau$. By Lemma 25 we have $\mathsf{id}\eta \in |\Gamma|$ for some η, $|\tau|_\eta \in \mathbb{T}$, and $\mathsf{id}\, e \in |\tau|_\eta$. By definition of $\mathbb{T}$ we have $|\tau|_\eta \subseteq \mathcal{S}$. From which we deduce e is in $\mathcal{S}$ and thus also in $\mathcal{U}$. Thus a is also in $\mathcal{U}$. □

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 27 (Termination)**.** *If* $a : \Gamma \vdash \tau$ *holds in the sublanguage without recursive types, 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.

4.1 Coq development

We have a Coq development of the soundness proof for F_ι^c. This Coq development also contains a proof of equivalence between 3 versions of the typing rules: a first version with minimum redundancy, a second version with enough redundancy to make extraction holds, and a last version even more redundant 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 three homogeneous environments (for types, coercions, and terms) instead of a heterogeneous dependent one. The development can be found online.[3]

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

5. Expressivity

The language F_ι^c 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_ι^c 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 $\mathsf{F}_{<:}$, MLF, and F_η which are subsumed by F_ι^p can be seen as sublanguages of F_ι^c too.

We show that languages with ML-like subtyping constraints [10] can be simulated in F_ι^c. Term judgments are usually written $A \vdash e : \tau \mid C$ where A is the environment, e the expression, and τ its type, and C is a sequence of constraints, *e.g.* in [11]. We may simulate these term judgments as $e : (\mathsf{fv}(A, C, \tau), C), A \vdash \tau$. Types and expressions are usually like ours: we unfold the let-definitions of e and we write $\forall(\overline{\alpha}, C)\tau$ instead of $\forall\overline{\alpha}.\tau \mid C$. Notice that the environment is of the form $(\overline{\alpha}, C), A$ of one multiple abstraction block $(\overline{\alpha}, C)$ followed by term bindings A.

We claim that solvability implies the existence of coercions, 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 witnesses existence.

The two interesting typing judgments we usually find are the let-binding and the subsumption rule, which are derivable in F_ι^c:

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

$$\frac{a : (\overline{\alpha}, C), \Gamma \vdash \tau \qquad (\overline{\alpha}, C') \vdash C, \tau \triangleright \sigma\ (\mathbf{1})}{a : (\overline{\alpha}, C'), \Gamma \vdash \sigma}$$

However, there are two main differences with the way subtyping constraints are handled in ML. With subtyping constraints, a judgment $A \vdash e : \tau \mid C$ is valid when C is *consistent*; our corresponding judgment $(\mathsf{fv}(A, C, \tau), C), A \vdash e : \tau$ is valid when C is *solvable*, *i.e.* $\vdash (\mathsf{fv}(C), C)$, which must exhibit a substitution θ of domain $\mathsf{fv}(C)$ such that $\varnothing \vdash C\theta$. While consistency and solvability coincide in ML, this need not be the case. Consistency is a semantic property while solvability is a syntactic property. Using consistency instead of solvability, we have only 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. It would be interesting if we could also abstract from solvability and not have to always produce explicit witnesses. Generalizing from sequences of elementary constraints to some richer language of constraints with standard logic connectives is also worth studying.

Subtyping constraints in ML are syntactic and take 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_ι^c is semantic and syntactic coercions must be interpreted in the semantics. Since our semantics has many 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. Discussion

We first compare F^c_ι with our prior work F^p_ι and other related works and discuss extensions of the language and future works.

6.1 Comparison with F^p_ι and other approaches

In previous work, we introduced a coercion language F^p_ι with type abstraction and restricted coercion abstraction. The main novelty of F^p_ι is to allow abstraction over coercions. However, this comes in a restricted form that requires abstract coercions to be parametric in either their domain or their range. This amounts to have the following two forms of bounded quantification: $\forall(\alpha \rhd \tau)\,\sigma$ and $\forall(\alpha \lhd \tau)\,\sigma$. The former means for all type variable α smaller than τ and correspond to the bounded quantification written $\forall(\alpha <: \tau)\,\sigma$ in $\mathsf{F}_{<:}$; the latter $\forall(\alpha \lhd \tau)\,\sigma$ means for all type variable α that is an instance of τ and correspond to the instance bounded quantification of MLF.

The restriction to parametric coercions in F^p_ι ensures that coercions cannot appear in wedges, which gives F^p_ι a type-erasing semantics, *i.e.* coercions only contribute to typing but not to the reduction per se. In other words, they are carried on during the reduction, but reduction proceeds as if coercions had been erased prior to reduction.

Relaxing the parametricity restriction of F^p_ι would allow to abstract over wedges, as explained in the introduction. Then to preserve the type-erasing semantics, we would have to introduce a new reduction rule to break this wedge apart into a term of the form $G_2\langle M[x \leftarrow G_1\langle N\rangle]\rangle$, where the two coercions G_1 and G_2 are built from c. The intuition is that c should eventually be instantiated by a coercion expecting an argument of type $\tau' \to \sigma'$ and applying G_1 of type $\tau \rhd \tau'$ to its argument follow by an application of G_2 of type $\sigma' \rhd \sigma$ to the result.

Even a simplification of this problem raises difficulties. Consider the following simpler problem in our implicit calculus F^c_ι. Assume that we have the following inclusion between semantic types $\mathsf{R}' \to \mathsf{S}' \subseteq \mathsf{R} \to \mathsf{S}$. There we can easily prove the inclusion $\mathsf{S}' \subseteq \mathsf{S}$. However, it is unclear whether $\mathsf{R} \subseteq \mathsf{R}'$ always holds.

Coherence is sufficient for type soundness, but in an explicit language of coercions it does not suffice for subject reduction, which also requires that G_1 and G_2 have a *syntactic* representation.

Our approach in F^c_ι is to we 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 proved to be sound by syntactic means in an explicit language of coercions.

Besides, we abstract over related types and coercions simultaneously. That is, we abstract over a set of types along with a set of coercions—and also require an example of type instantiation that is used to prove coherence of the abstraction.

Actually, we also explored a syntactically more atomic version of F^c_ι where type and coercion abstraction are separate constructs as in F_ι [6]. Namely, the usual type abstraction $\forall\alpha\,\tau$ and coercion abstraction $(\tau_1 \rhd \tau_2) \Rightarrow \sigma$—a term of type σ under the hypothesis that $\tau_1 \rhd \tau_2$ holds. The idea is write for instance a function of type $\forall\alpha\,(\alpha \to (\tau_1 \rhd \tau_2) \Rightarrow \sigma)$ and apply it to a type parameter, then a value of type α then a coercion. However, this additional flexibility is negligible, these are just η-expansion variants of terms in F^c_ι. Moreover, related type abstractions and coercions must 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. Moreover, it would be much harder to find an explicit version of the calculus well-suited for typechecking. The solution in F^c_ι seems a better compromise between simplicity and expressiveness.

6.2 Related work

To the best of our knowledge there is no previous work considering coercions as a inclusion of typings. However, the use of coercions to study features of type system is not at all new.

Subtyping have been popularized by object-oriented languages, even though inheritance is somehow better modeled by matching [4] or row polymorphism [12]. In our view subtyping and polymorphism are both treated as coercions.

The heavy use of coercions in FC, the core language of GHC, was one of our initial motivations for studying coercions. 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 reduction does not proceed under coercion abstraction in FC, and therefore, the code relying on incoherent assumptions will never be reached. This simplifies the meta-theory but at some significant cost, since coercions are not erasable in FC: They must delay the evaluation order to prevent this above inconsistencies. This makes our two works incomparable. However, if we extend our language with existential types and higher-order types, as discussed in §6.3 then the FC terms without dead codes due to incoherent assumptions should be included in this extended language.

All language features without computational content are treated as coercions in F^c_ι. 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 [7]. 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 $k-1$.

In [15] coercions are used to eliminate function call overhead from datatype constructors. The folding and unfolding of datatype definitions are done using erasable coercions, thus with no run-time effect or hidden cost and preserving the semantics.

Recursive coercions have also been used to provide coercion iterators over recursive structures [5]. However, the motivations are quite different and coercions are only a tool to compile bounded quantification away into intersection types.

6.3 Extensions and variations

Higher-order types We introduced F^c_ι as an extension of System F, thus restricting ourselves to second-order polymorphism. We believe that our approach can be transferred to a calculus with higher-order types such as F_ω. The change is however significant, since semantic types would have to be categorized by their kind. We would also interpret kinds as set of semantic types. A type τ would be of kind κ if its interpretation $|\tau|$ is in the interpretation of its kind $|\kappa|$. The internal language FC of GHC already has higher-order kinds.

Intersection types It should also be possible to add intersection types. Our semantics already has intersection types. However, following the work of Wells [17] on *branching types*, it would 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^c_ι. One possibility is using the CPS encoding of existential types into universal types.

Adding primitive existential types is 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 priory 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 [13] 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 semantics 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!

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. Still, we do not foresee any problem in applying this to a real programming language, in particular one with side effects. As we have explained in the introduction, 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.

Soundness in the presence of side-effects also require a form of value-restriction: type and thus coercions abstractions should only be allowed on values. We do not expect this to raise new problems with coercions—nor 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 coercions When coercions are left implicit they must be inferred—as well as coercion witnesses, which is obviously undecidable in the general case. In practice, the user should provide both 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. Hence coercions should be dropped after typechecking. Indeed, we do not describe how coercions—and in particular wedges—could be reduced. Thus, 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 so that it is well-typed. We believe that this is the price to pay for the generality that our approach offers.

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 type 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, many research directions remain to be explored. Hopefully, new type system features such as higher-order and dependent types could still be added. A surface language with explicit coercions or coercions annotations is a prerequisite for decidable type checking. Variations on constraints, moving from concrete coercion objects to proofs of coercibility in a logic, replacing solvability by consistency, or allowing closed-world views, are all worth further investigation.

References

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

[2] 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.

[3] P. Baldan, G. Ghelli, and A. Raffaetà. Basic theory of F-bounded quantification. *Information and Computation*, 153:173–237, 1999.

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

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

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

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

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

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

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

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

[12] 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.

[13] 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.

[14] T. C. D. Team. *The Coq Proof Assistant, Reference Manual, Version 8.4.* INRIA, 2012-2013.

[15] 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.

[16] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative type abstraction and type-level computation. POPL'11, 2011.

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