Recursive polymorphic types and parametricity in an operational framework

Paul-André Melliès [*] Jérôme Vouillon [*]

Abstract

We construct a realizability model of recursive polymorphic types, starting from an untyped language of terms and contexts. An orthogonality relation $e \perp \pi$ indicates when a term e and a context π may be safely combined in the language. Types are interpreted as sets of terms closed by biorthogonality. Our main result states that recursive types are approximated by converging sequences of interval types. Our proof is based on a "type-directed" approximation technique, which departs from the "language-directed" approximation technique developed by MacQueen, Plotkin and Sethi in the ideal model. We thus keep the language elementary (a call-by-name λ-calculus) and unstratified (no typecase, no reduction labels). We also include a short account of parametricity, based on an orthogonality relation between quadruples of terms and contexts.

1 Introduction

In this article, we describe how a large variety of typing constructions: recursive types, polymorphism, subtyping, product types, intersection types, union types, as well as parametricity, may be interpreted and studied in a somewhat naïve and operational framework. We choose to set our scene slowly, starting from a brief account of recursive types in categories of domains, then shifting gradually to the operational model we have in mind.

Recursive types. Solving recursive equations between types is generally done in categories. Take for instance the equation

$$X = 1 + X \tag{1}$$

in the category $\mathbf{Set}$ of sets and functions, where $1 + X$ denotes the disjoint union of X and of the singleton $1 = \{*\}$. This equation has two canonical solutions, namely the set $\mathbb{N}$ of natural numbers, and the set $\mathbb{N} + \{\infty\}$ of natural numbers completed by infinity. By "solution", one means a set X equipped with an isomorphism $X \longleftarrow 1 + X$, alternatively seen as its inverse map $X \longrightarrow 1 + X$. Accordingly, each of the solution sets $\mathbb{N}$ and $\mathbb{N} + \{\infty\}$ comes equipped with a function s and p, called "successor" and "predecessor":

$$\mathbb{N} \xleftarrow{s} 1 + \mathbb{N} \qquad \mathbb{N} + \{\infty\} \xrightarrow{p} 1 + \mathbb{N} + \{\infty\} \tag{2}$$

and defined respectively as $s(*) = 0$ and $s(n) = n + 1$, and as $p(0) = *$, $p(n+1) = n$ and $p(\infty) = \infty$, for all $n \in \mathbb{N}$.

The two solutions (2) are canonical in the following sense. Call prefixpoint f and postfixpoint g of equation (1) any function:

$$X \xleftarrow{f} 1 + X \quad \text{and} \quad Y \xrightarrow{g} 1 + Y.$$

Canonicity says that, for any such f (resp. g), there exists a unique function h (resp. k) making the diagram commute:

$$\begin{tikzcd} X & 1 + X \arrow[l, "f"'] & \mathbb{N} + \{\infty\} \arrow[r, "p"] & 1 + \mathbb{N} + \{\infty\} \\ \mathbb{N} \arrow[u, "h"] & 1 + \mathbb{N} \arrow[l, "s"] \arrow[u, "1+h"'] & Y \arrow[u, "k"] \arrow[r, "g"] & 1 + Y \arrow[u, "1+k"'] \end{tikzcd} \tag{3}$$

In other words, s is the "initial" prefixpoint (called inductive solution), and p is the "terminal" postfixpoint (called coinductive solution) of equation (1). What we find here is nothing but the "categorification" of Knaster-Tarski's theory of fixpoints for a monotone function in a complete lattice. A categorical terminology is generally adopted: the "prefixpoints" and "postfixpoints" are called T-algebras and T-coalgebras for the functor $T : \mathbf{Set} \longrightarrow \mathbf{Set}$ defined as $T(X) = 1 + X$. And the diagrams of equation (3) express that the T-algebra s is the "initial" T-algebra, and that the T-coalgebra p is the "terminal" T-coalgebra.

Mixed variance. Equation (1) is particularly simple because the variable X occurs only in a covariant (= positive) position. More complicated recursive equations may be also considered, like

$$X = X \Rightarrow X \tag{4}$$

where the variable X appears in a covariant as well as a contravariant (= negative) position. This kind of equation may be formulated in any cartesian closed category, where $X \Rightarrow Y$ denotes the usual arrow construct.

These equations of mixed variance have (in general) no solution in the category $\mathbf{Set}$. Typically, equation (4) has only the trivial solution $X = 1$, while equation

$$X = 1 + (X \Rightarrow X) \tag{5}$$

[*]Postal address: Equipe PPS, CNRS and Université Paris VII, 2 place Jussieu, Case 7014, 75251 Paris Cedex 05, FRANCE. Email address: **mellies@pps.jussieu.fr** and **vouillon@pps.jussieu.fr**

has no solution at all, for obvious cardinality reasons.

Scott invented Domain Theory when he realized that equations of mixed variance like (4) and (5) could be solved by shifting from the category $\mathbf{Set}$ to the category $\mathbf{CPO}$ of domains (= partial orders with a least element $\perp$ and all directed limits) and continuous (= monotone and limit-preserving) functions between them. We refer the reader to [4] for a modern exposition of Domain Theory.

We only mention here that Freyd [15] derives this existence of solutions in $\mathbf{CPO}$ from a striking property of its subcategory $\mathbf{CPO}_\perp$ of strict (= $\perp$-preserving) functions, called "algebraic compactness". The property states that the initial T-algebra and the terminal T-coalgebra *coincide* for every (well-behaved) covariant endofunctor

$$T : \mathbf{CPO}_\perp \longrightarrow \mathbf{CPO}_\perp.$$

Freyd shows that the category $\mathbf{CPO}_\perp^{op} \times \mathbf{CPO}_\perp$ is also algebraically compact, and reformulates in this way any mixed variance functor on $\mathbf{CPO}_\perp$ as a covariant endofunctor T on this category. The unique canonical fixpoint of T is shown to be of the form (D, D') where D and D' are isomorphic in $\mathbf{CPO}_\perp$. This provides the solution $D \cong D'$ to the original mixed-variance equation over $\mathbf{CPO}$.

We step back to equation (1) for illustration, slightly generalized in the category $\mathbf{CPO}$ as equation:

$$X = 1_\perp + (A \times X) \qquad (6)$$

where A denotes a fixed domain. That is, the domain X is required to be isomorphic to the coalesced sum of the lifted domain $1_\perp = \{\perp, *\}$ with the cartesian product of the domains A and X. Just as in $\mathbf{Set}$, equation (6) has an inductive solution (= the domain of lists over A) and a coinductive solution (= the domain of streams over A) in the category $\mathbf{CPO}$. But in contrast to $\mathbf{Set}$, algebraic compactness implies that the two solutions coincide in $\mathbf{CPO}$.

The ideal model. We found instructive to recall briefly the categorical approach to recursive types, because it is elegant and mainstream. We shift now to a different approach to recursive types, initiated by MacQueen, Plotkin, and Sethi in the *ideal model for types* [19]. The ideal model is still domain-theoretic, but not categorical any more. This brings us one step closer to the operational framework developed in this article.

The ideal model is built in two stages. First stage: a domain $\mathbf{V}$ of "semantic expressions" is fixed, defined in [19] as the canonical solution in $\mathbf{CPO}$ of the equation:

$$\mathbf{V} = \mathbf{T} + \mathbf{N} + (\mathbf{V} \to \mathbf{V}) + (\mathbf{V} \times \mathbf{V}) + (\mathbf{V} + \mathbf{V}) + \{\mho\}_\perp \quad (7)$$

This may be read as follows: $\mathbf{V}$ is (isomorphic to) the coalesced sum of the boolean constants $\mathbf{T}$, the integers $\mathbf{N}$, the continuous functions from $\mathbf{V}$ to $\mathbf{V}$, the product of $\mathbf{V}$ with itself, the sum of $\mathbf{V}$ with itself, and a "type-error" constant $\mho$. Or alternatively: a "semantic expression" $f \in \mathbf{V}$ is either a boolean constant, an integer, a function between expressions, a pair of expressions, a left (resp. right) injection of an expression, the error constant $\mho$, or the least element $\perp$.

Second stage: every type is interpreted as an *ideal* of the domain $\mathbf{V}$, that is, as a non-empty set $I \subseteq \mathbf{V}$ which is (1) downward closed, and (2) closed under directed limits. Notice that ideals are domains themselves. So, types are interpreted as domains, just as in mainstream Domain Theory. There is a major difference, though: these domains are not only domains, they are also "subdomains" of the domain $\mathbf{V}$. So, subtyping may be interpreted as set-theoretic inclusion. But this has another key consequence. Defined as the canonical solution of equation (7), the domain $\mathbf{V}$ is approximated by a sequence $(\mathbf{V}_n)_{n \in \mathbb{N}}$ of domains:

$$\mathbf{V}_0 \subseteq \mathbf{V}_1 \subseteq \cdots \subseteq \mathbf{V}_{n-1} \subseteq \mathbf{V}_n \subseteq \mathbf{V}_{n+1} \subseteq \cdots \subseteq \mathbf{V}$$

each of them image of a projection map $\pi_n : \mathbf{V} \longrightarrow \mathbf{V}_n$. Besides, every element $x \in \mathbf{V}$ is the least upper bound of the directed set $\{\pi_n(x) \mid n \in \mathbb{N}\}$ of its approximations. This "stratification" of the domain $\mathbf{V}$ enables to define a distance $d(I, J)$ between two ideals I and J as $d(I, J) = 0$ when $I = J$, and as $d(I, J) = 2^{-n}$ when $I \neq J$, for n the least number such that $\pi_n(I) \neq \pi_n(J)$. MacQueen, Plotkin and Sethi prove that the resulting metric space on ideals is Cauchy-complete; and deduce that every recursive equation

$$X = T(X, X)$$

has a *unique* solution, as long as the functor T of mixed variance is *contractive* with respect to the metric space on ideals. Remarkably, contractibility holds for a large class of functors T, including all our illustrating equations (4), (5) and (6).

Towards operational semantics. The ideal model suffers from a serious defect noted in [3] and related to the domain-theoretic definition of $\mathbf{V}$: There exist "semantic expressions" $f \in \mathbf{V}$ which are not defined in the calculus, and may distort the expected properties of types. This is illustrated by the term:

$$\begin{array}{lcll} \texttt{por} - \texttt{explode} & = & \lambda f. & \texttt{if } f(\texttt{true}, \Omega) \\ & & & \texttt{and } f(\Omega, \texttt{true}) \\ & & & \texttt{and not } f(\texttt{false}, \texttt{false}) \\ & & & \texttt{then } \mho \\ & & & \texttt{else true}. \end{array}$$

where Ω denotes the *diverging* term $\Omega = (\lambda y.y\,y)\,(\lambda y.y\,y)$ and $\mho$ denotes the "type-error" constant Ω.

What should be the type of $\texttt{por} - \texttt{explode}$? The first branch of the $\texttt{if} - \texttt{then} - \texttt{else}$ is selected only when the input f represents the "parallel-or" function (noted *por*) which returns `true` when one of its arguments is `true`, and `false` when its two arguments are `false`. Now, the function $por \in \mathbf{V}$ is a "semantic expression" which cannot be represented syntactically in the λ-calculus, or in any sequential language. So, the term $\texttt{por} - \texttt{explode}$ returns `true` for every term f of type $(\mathbf{T} \times \mathbf{T}) \to \mathbf{T}$ in any such language; and consequently should be typed $((\mathbf{T} \times \mathbf{T}) \to \mathbf{T}) \to \mathbf{T}$ there. Unfortunately, this type is not validated by the ideal model, because the term $\texttt{por} - \texttt{explode}$ interpreted in $\mathbf{V}$ returns "error" for the "semantic expression" por of type $(\mathbf{T} \times \mathbf{T}) \to \mathbf{T}$.

This example suggests to reject the mediation of Domain Theory, and to recast the ideal model directly inside operational semantics. The project is fascinating conceptually, but difficult to realize technically. As we mentioned earlier, the existence of recursive types in the ideal model is deduced from the "stratification" of the domain $\mathbf{V}$, and the existence of the projection maps $\pi_n : \mathbf{V} \to \mathbf{V}_n$. Obviously, shifting to operational semantics requires to find an operational counterpart to the "stratification" of the domain $\mathbf{V}$. How and what? This question has attracted considerable interest in the last decade, leading to a series of major advances in the field [3, 5, 8, 10, 11, 12, 20, 23]. Four solutions emerged from the period, which we recall briefly now.

1. Abadi, Pierce and Plotkin [3] do not alter the domain-theoretic definition of $\mathbf{V}$ (and thus keep its "stratification") but restrict the interpretation of types to ideals "generated" (in the order-theoretic sense) by definable elements. Strikingly, the resulting ideal model validates that the term $\texttt{por} - \texttt{explode}$ has type $((\mathbf{T} \times \mathbf{T}) \to \mathbf{T}) \to \mathbf{T}$. Other syntactic variants of the ideal model are considered, obtained in each case by restricting the interpretation to particular classes of ideals, e.g. the so-called "abstract" and "coarse" ideals, see [3] for details. In any of these variants, it is technically crucial that the projection maps $\pi_n : \mathbf{V} \to \mathbf{V}_n$ are *definable*. This requires to enrich the language (an untyped λ-calculus) with a "typecase" operator which tests whether a term e is a boolean, a natural, a pair, a sum, or a function, and then returns a different result e_i in each case:

```
cases  e   bool  :  e1
           nat   :  e2
           pair  :  e3
           sum   :  e4
           fun   :  e5
end
```

This idea has been influential and reappears in many later attacks to connect operational and denotational semantics, most notably by Smith, Mason, and Talcott [20, 23] and Birkedal and Harper [8]. We should add that the article [3] is also influential for its last section, where the three authors deliver in a visionary style the principle of an ideal model living inside operational semantics.

2. Dami [11, 12] takes up the last idea of [3] and recasts the ideal model inside operational semantics. Several variants of the λ-calculus are considered, all of them enriched with *reduction labels* inspired from Lévy [18]. These labels provide the "stratification" of the language necessary to solve recursive equations between types.

3. Chroboczek [10] recasts the ideal model in game semantics, by solving an equation similar to (7) in a category of games. The resulting game $\mathbf{G}$ is "stratified", and recursive equations are thus solved inside $\mathbf{G}$ by the same Cauchy-completeness argument as in [19]. Chroboczek observes a mismatch between his original operational semantics (a call-by-name λ-calculus), and the interpretation of this calculus in the model. He thus designs an adequate language by enriching the original language with a "located" (and in fact "stratified") notion of convergence test.

4. Appel and McAllester [5] develop a radically different approach to the problem, in which (in contrast to **1**, **2** and **3**) they do not need to enrich the original language in order to stratify it. Their language is defined using a small-step semantics. This enables them to define *intensional* types, in which an information on the number of steps to compute a value is provided. Remarkably, this extra information is sufficient to approximate the behavior of a term, and to solve recursive equations between types.

Realizability and orthogonality. These operational approaches to recursive types have in common to alter something of the original syntax of the calculus, or to alter something of the original definition of types. Here, we want to interpret recursive polymorphic types in operational semantics, but without "stratifying" the language or the convergence test (as in **1**,**2**,**3**), and without "intensionalizing" the typing (as in **4**).

This is a difficult task, which requires to design a new stratification principle in order to replace the usual "language-directed" stratification. A clarifying step is taken in a companion paper [25] where we reformulate the ideal model in a more conceptual and operational way, inspired from Krivine's *realizability* [17, 13].

In a realizability model (à la Krivine), one starts from an untyped calculus of terms and contexts, and constitutes a typed language on top of it. The cornerstone of the theory is a notion of *orthogonality* $e \perp \pi$ which indicates when a term e and a context π may be safely combined (no error at runtime). Orthogonality induces a closure operator which associates to every set U of terms the set $U^{\perp\perp}$ of terms which cannot be separated from U by a context. This set $U^{\perp\perp}$ is called the *biorthogonal* of U. Types are interpreted as sets $U = U^{\perp\perp}$ closed by biorthogonality, also called *truth values*. The formal definition appears in Section 3.

Connecting "types" and "orthogonality" is one of the nicest discoveries of "French" proof-theory. The idea emerged after intense reflection on the *reducibility candidates* method to prove strong normalization for System F. Girard reformulates these candidates as biorthogonal sets of terms, in his proof of cut elimination for linear logic [16].

The idea reappears in Parigot's proof of Strong Normalization for second order $\lambda\mu$-calculus [21]. Meanwhile, Krivine formulates a comprehensive framework based on orthogonality, in order to analyze types as *specifications* of terms. Krivine demonstrates that realizability generalizes Cohen's forcing and induces models of classical Zermelo-Fraenkel set theory [13, 17].

Parametricity. It should be said that the idea of orthogonality is not only "French": Andy Pitts discovered it independently in his remarkable work on operational equivalence [22] — see also [1]. We indicate in Section 9 (alas too briefly for lack of space) how Pitts' operational approach to parametricity may be reflected in a realizability framework.

Type-directed stratification. What about recursive types? We are looking for an operational counterpart to "algebraic compactness" in Domain Theory. This should ensure (for instance) that the type of lists of booleans and the type of streams of booleans coincide in the model. Take the set U of boolean lists $(e_1, ..., e_n)$ in which each term e_i is either `true` or `false`. Any such list is easily encoded in a λ-calculus with pairs. Now, take the term:

$$e_\infty = Y\,(\lambda x.(\texttt{true}, x))$$

in which $Y = \lambda f.(\lambda x.f\,xx)(\lambda x.f\,xx)$ is the Kleene fixpoint. The term e_∞ implements the infinite stream of `true`, thus is element of the truth value V of boolean streams. But e_∞ is not an element of U. It is not difficult to see however that e_∞ is an element of $U^{\perp\perp}$. Indeed, every context π which combines safely with all the boolean lists, combines safely with all the boolean streams, including e_∞. We conclude from this and $U \subseteq V$ that $V = U^{\perp\perp}$.

The equality $U^{\perp\perp} = V$ captures the essence of coincidence, and we shall prove it for every recursive type (Theorem 5). Note that the equality generally fails when orthogonality amounts to *termination* ($e \perp \pi$ iff e combined to π converges) instead of *safety*. Indeed, there may exist a context π (think of a length function) which terminates on every list and loops on every stream.

The framework described in [25] is technically enlightening, but still based on a "language-directed" stratification technique, which we reject here. We develop instead a "type-directed" stratification technique, in which every (possibly infinite) type τ is approximated by finite trees called *interval types*. Each interval type K is interpreted in the model as a triple (U, V, ϕ_K) where $U \subseteq V$ are truth values, and ϕ_K is a conversion term sending every term $e \in V$ to a term $\phi_K\, e \in U$. These "type-directed" ϕ_K replace the "language-directed" projections π_n of the ideal model. The resulting "type-directed" picture is closer to Domain Theory, in which the solution of a recursive equation $X = T(X, X)$ is computed as limit of a categorical diagram defined by the type T.

Related works. As noted earlier, the literature on types is huge, even if one restricts one's attention to recursive types, subtyping, or polymorphism. We did our best to give a comprehensive panorama of the field in the introduction, but it is obviously too brief, and far from exhaustive, for lack of space. The interested reader will find complementary information in the companion paper [25].

Outline. In the remainder of the paper, we introduce a call-by-name calculus (Section 2) for which we formulate an orthogonality relation between terms and stacks, in the style of Krivine (Section 3). This defines a truth value as a set of terms orthogonal to a set of stacks. Then, we introduce our syntax of types and of interval types (Section 4). We interpret types as truth values in two stages: first, we interpret *inductively* every interval type as a pair of truth values, with a conversion term between them (Section 5); then, we interpret types by *approximating* them with interval types (Section 6). We sketch how to treat intersection and union types by moving to a nondeterministic language (Section 7). We prove soundness of our interpretation for a typing system with universal and existential types, and subtyping (Section 8). Finally, we give a brief account of parametricity (Section 9) and conclude (Section 10).

2 A simple call-by-name calculus

2.1 The terms

We start from an untyped λ-calculus with pairs and conditional branch, defined by the syntax below:

e	$::=$	x	variable
	$\mid$	$\lambda x.e$	abstraction
	$\mid$	$e\,e$	application
	$\mid$	(e, e)	pair
	$\mid$	$\texttt{fst}(e)$	first projection
	$\mid$	$\texttt{snd}(e)$	second projection
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional branch
	$\mid$	`true`	constant true
	$\mid$	`false`	constant false

2.2 The operational semantics

We choose to apply a call-by-name evaluation strategy between terms, which we describe using a small-step semantics. This is only a matter of choice: all the constructions in this paper work also if one starts from a *call-by-value* λ-calculus. The definition goes in two steps. First, we introduce a class of *evaluation contexts*, indicating where a symbolic transformation may be applied in a term. Then, we specify five rewriting rules, formulated as an interaction between a term and its evaluation context.

Evaluation contexts are finite lists defined by the grammar:

$$\begin{array}{rcll} \mathrm{E} & ::= & \mathtt{nil} & \text{head context} \\ & \mid & e \cdot \mathrm{E} & \text{application} \\ & \mid & \mathtt{fst} \cdot \mathrm{E} & \text{first projection} \\ & \mid & \mathtt{snd} \cdot \mathrm{E} & \text{second projection} \\ & \mid & (\mathtt{if}\ e, e) \cdot \mathrm{E} & \text{conditional branch} \end{array}$$

Every term e and evaluation context E may be combined to generate a term denoted $\langle e \mid \mathrm{E} \rangle$ and defined as follows:

$$\begin{array}{lcl} \langle e \mid \mathtt{nil} \rangle & = & e \\ \langle e \mid e' \cdot \mathrm{E} \rangle & = & \langle e\, e' \mid \mathrm{E} \rangle \\ \langle e \mid \mathtt{fst} \cdot \mathrm{E} \rangle & = & \langle \mathtt{fst}(e) \mid \mathrm{E} \rangle \\ \langle e \mid \mathtt{snd} \cdot \mathrm{E} \rangle & = & \langle \mathtt{snd}(e) \mid \mathrm{E} \rangle \\ \langle e \mid (\mathtt{if}\ e_1, e_2) \cdot \mathrm{E} \rangle & = & \langle \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathrm{E} \rangle \end{array}$$

The reduction relation $\rightarrow$ is defined as the smallest relation between terms containing any instance of five rewriting rules: the usual β-rule:

$$\langle \lambda x.e \mid e' \cdot \mathrm{E} \rangle \rightarrow \langle e[x := e'] \mid \mathrm{E} \rangle$$

two rules for the products:

$$\begin{array}{lcl} \langle (e_1, e_2) \mid \mathtt{fst} \cdot \mathrm{E} \rangle & \rightarrow & \langle e_1 \mid \mathrm{E} \rangle \\ \langle (e_1, e_2) \mid \mathtt{snd} \cdot \mathrm{E} \rangle & \rightarrow & \langle e_2 \mid \mathrm{E} \rangle \end{array}$$

and two rules for the conditional:

$$\begin{array}{lcl} \langle \mathtt{true} \mid (\mathtt{if}\ e_1, e_2) \cdot \mathrm{E} \rangle & \rightarrow & \langle e_1 \mid \mathrm{E} \rangle \\ \langle \mathtt{false} \mid (\mathtt{if}\ e_1, e_2) \cdot \mathrm{E} \rangle & \rightarrow & \langle e_2 \mid \mathrm{E} \rangle \end{array}$$

where e, e', e_1, e_2 denote terms and E denotes an evaluation context. Observe that the resulting reduction $\rightarrow$ is *deterministic* in the sense that:

$$\forall e, e_1, e_2, \qquad e \rightarrow e_1 \text{ and } e \rightarrow e_2 \quad \Rightarrow \quad e_1 = e_2.$$

3 Realizability

3.1 The safe terms

We write $\rightarrow^*$ for the reflexive and transitive closure of the relation $\rightarrow$, and say that:

- a term e *reduces* to a term e' when $e \rightarrow^* e'$,
- a term e *loops* when there exists an infinite sequence of reductions:
$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots$$
- a term is *safe* when it loops, or when it reduces to one of the two boolean constants `true` or `false`,
- a term is *unsafe* when it is not safe.

An example of safe term is Ω, defined as:

$$\Omega = (\lambda x.x\, x)\, (\lambda x.x\, x).$$

An example of unsafe term is $\mho$, defined as:

$$\mho = (\mathtt{true})\, (\mathtt{true}).$$

3.2 The stacks

The terms of the language will be tested by evaluation contexts E, as well as by two "constant" contexts Ω and $\mho$ which we add here for convenience. We call these testing contexts *stacks*, and note them π as in [17, 13].

$$\begin{array}{rcll} \pi & ::= & \mathrm{E} & \text{evaluation context} \\ & \mid & \Omega & \text{safe} \\ & \mid & \mho & \text{unsafe} \end{array}$$

A stack π is called *strict* when it is an evaluation context, *safe* when $\pi = \Omega$ and *unsafe* when $\pi = \mho$.

We extend the definition of $\langle - \mid \mathrm{E} \rangle$ to stacks in the expected way. Thus for every term e:

$$\langle e \mid \Omega \rangle \overset{\text{def}}{=} \Omega, \qquad\qquad \langle e \mid \mho \rangle \overset{\text{def}}{=} \mho.$$

Similarly, we extend to stacks the constructors defined for evaluation contexts in Section 2.2. This is simply done by applying the convention below, for any terms e, e_1, e_2:

$$e \cdot \Omega = \mathtt{fst} \cdot \Omega = \mathtt{snd} \cdot \Omega = (\mathtt{if}\ e_1, e_2) \cdot \Omega \overset{\text{def}}{=} \Omega,$$
$$e \cdot \mho = \mathtt{fst} \cdot \mho = \mathtt{snd} \cdot \mho = (\mathtt{if}\ e_1, e_2) \cdot \mho \overset{\text{def}}{=} \mho.$$

3.3 Orthogonality

The *orthogonality* relation $\perp$ between terms and stacks is defined as follows:

$$e \perp \pi \iff \text{the term } \langle e \mid \pi \rangle \text{ is safe.}$$

Thus, a term e and a stack π are orthogonal when combining them induces a term $\langle e \mid \pi \rangle$ which loops, or reduces to one of the boolean constants `true` or `false`.

Some readers will find it unexpected to see terms like $\lambda x.e$ counted among our unsafe terms. This seems to contradict the accepted notion of *value* in functional programming. Well, not really. The idea is that a term like $\lambda x.e$ is unsafe until it receives an argument e' and induces a safe term $(\lambda x.e)\, e'$. We write this $\lambda x.e \perp e' \cdot \mathtt{nil}$, and note that observing boolean constants (and only them) is enough to characterize types semantically.

Note finally that the stack Ω is orthogonal to every term, and that the stack $\mho$ is orthogonal to no term.

3.4 Truth values

A term e is orthogonal to a set of stacks Π when:

$$\forall \pi \in \Pi, e \perp \pi.$$

In that way, every set of stacks Π defines a set of terms $\Pi^\perp$, called the *orthogonal* of Π:

$$\Pi^\perp = \{e \mid \forall \pi \in \Pi : e \perp \pi\}.$$

Conversely, every set of terms Λ defines a set of stacks $\Lambda^{\perp}$, consisting of all the stacks orthogonal to Λ:

$$\Lambda^{\perp} = \{\pi \mid \forall e \in \Lambda : e \perp \pi\}.$$

Taking twice the orthogonal of a set of terms Λ induces a set of terms $\Lambda^{\perp\perp}$ called the *biorthogonal* of Λ. This operation $(\Lambda \mapsto \Lambda^{\perp\perp})$ defines a closure operator in the lattice of sets of terms, ordered by inclusion. In particular, $\Lambda \subseteq \Lambda^{\perp\perp}$.

A *truth value* U is a set of terms closed by biorthogonality, that is, satisfying $U = U^{\perp\perp}$. Note that the orthogonal $\Pi^{\perp}$ to a given set of stacks Π is always a truth value, and that truth values are closed under (arbitrary) intersection.

3.5 Two constructions on truth values

Suppose that U and V are two truth values. We define the truth values $U \Rightarrow V$ and $U \times V$ as follows.

The arrow construction. The truth value $U \Rightarrow V$ is defined as the set of terms orthogonal to the stacks $e \cdot \pi$ where $e \in U$ and π is a stack orthogonal to V.

Lemma 1 *For every truth values U, V and term e, the following facts are equivalent :*

1. $e \in U \Rightarrow V$;
2. $\forall e' \in U,\ e\,e' \in V$.

The product construction. The truth value $U \times V$ is defined as the set of terms orthogonal to the stacks $\texttt{fst} \cdot \pi$ where π is a stack orthogonal to U, and $\texttt{snd} \cdot \pi$ where π is a stack orthogonal to V.

Lemma 2 *A term is element of $U \times V$ iff the term loops, or reduces to a pair (e, e') where $e \in U$ and $e' \in V$.*

4 Types

4.1 Syntax of types

Types are defined in two steps. First, finite patterns (called *type patterns*) are defined inductively. Then, these patterns are assembled coinductively into possibly infinite trees (called *types*). This two-step construction rules out ill-defined types, such as $\tau = \forall\alpha.\tau$, in that case because $\forall\alpha.\tau$ is not a pattern. Indeed, any occurrence of a type in a pattern is below a *constructor* $\rightarrow$ or $\times$.

We assume given a set of type variables α and a single type constant $Bool$. Given a set of types τ, we define *type patterns* t inductively by the grammar below.

t	$::=$	$Bool$	boolean type
	$\mid$	$\tau \times \tau$	pair type
	$\mid$	$\tau \rightarrow \tau$	function type
	$\mid$	α	type variable
	$\mid$	$\top$	top type
	$\mid$	$\forall\alpha.t$	universal quantification
	$\mid$	$\perp$	bottom type
	$\mid$	$\exists\alpha.t$	existential quantification

The different type constructions are standard. See Sections 5 and 6 for a precise description of their meaning.

We write $t(\tau_1, \ldots, \tau_k)$ when the pattern t has leaves τ_1, $\ldots$, τ_k, where each τ_i occurs linearly in t. The finite patterns t are assembled coinductively as follows:

$$\tau \quad ::= \quad t(\tau_1, \ldots, \tau_k) \quad \text{coinductively.}$$

By coinduction, every type τ is of the form $t(\tau_1, \ldots, \tau_k)$. So, we can reason inductively on the structure of type patterns, then coinductively on the structure of types. This turns out to be very convenient. Besides, all the constructions $\rightarrow$, $\times$, $\ldots$, on type patterns define constructions on types in the obvious way. This enables to write types like $\tau_1 \rightarrow \tau_2$, $\tau_1 \times \tau_2$ or $\forall\alpha.\tau$.

Types are considered modulo renaming of their bound variables. This does not contradict the coinductive definition of types on the alphabet of patterns since, in fact, α-conversion is only a handy presentation of de Bruijn indices. Note also that we don't assume types to be *regular*: types may have an infinite number of distinct subtrees.

Remark: the sum types are not treated for lack of space only. They are very easily integrated in the framework by extending the language of terms with three operators $\texttt{inl}(e)$, $\texttt{inr}(e)$, $\texttt{caseof}(e, e_1, e_2)$, and the language of evaluation contexts with one operator $(\texttt{case}\ e_1, e_2) \cdot \mathrm{E}$, with the following equation:

$$\langle e \mid (\texttt{case}\ e_1, e_2) \cdot \mathrm{E}\rangle \ = \ \langle \texttt{caseof}(e, e_1, e_2) \mid \mathrm{E}\rangle$$

and the two additional rewriting rules:

$$\begin{array}{lcl} \langle \texttt{inl}(e) \mid (\texttt{case}\ e_1, e_2) \cdot \mathrm{E}\rangle & \rightarrow & \langle e_1\, e \mid \mathrm{E}\rangle \\ \langle \texttt{inr}(e) \mid (\texttt{case}\ e_1, e_2) \cdot \mathrm{E}\rangle & \rightarrow & \langle e_2\, e \mid \mathrm{E}\rangle \end{array}$$

4.2 Syntax of interval types

In contrast to types, which may be infinite, *interval types* are finite trees, defined inductively by the grammar below.

K	$::=$	$Bool$	boolean type
	$\mid$	$K \times K$	pair type
	$\mid$	$K \rightarrow K$	function type
	$\mid$	α	type variable
	$\mid$	$\top$	top type
	$\mid$	$\forall\alpha.K$	universal quantification
	$\mid$	$\perp$	bottom type
	$\mid$	$\exists\alpha.K$	existential quantification
	$\mid$	$[\perp, \top]$	interval

In Section 6, we will use these interval types to "approximate" types, in order to interpret them. Accordingly, the type constructions are the same as for types. The only novelty is the interval type $[\perp, \top]$, which will be interpreted in the next section (Section 5) as the largest possible "interval", bounded by the smallest and largest nonempty truth values.

4.3 Types approximated by interval types

We say that an interval type K approximates a type τ, which we write as $K \sqsubseteq \tau$, when the type τ may be obtained syntactically by replacing every leaf labelled $[\bot, \top]$ in K by a type. For instance:

$$\forall\alpha.\forall\beta.[\bot,\top] \Rightarrow [\bot,\top] \;\sqsubseteq\; \forall\alpha.\forall\beta.\alpha \Rightarrow (\alpha \Rightarrow \beta).$$

5 Interpretation of interval types

5.1 Adjunction

Let ϕ be a term and ψ be a function on stacks. One says that ψ is the adjoint of ϕ when, for every term e and stack π:

$$\phi\, e \perp \pi \iff e \perp \psi\, \pi.$$

Note that the adjoint ψ is characterized by ϕ modulo observational equivalence, in the sense that if ψ' is another adjoint of ϕ, then, for every stack π:

$$\{\psi\,\pi\}^{\perp} = \{\psi'\,\pi\}^{\perp}.$$

This enables to use the notation ϕ^* for the adjoint ψ.

5.2 Semantic intervals

A *semantic interval* is a triple (U, V, ϕ) consisting of two nonempty truth values U and V satisfying $U \subseteq V$, and a term $\phi \in V \Rightarrow U$ having an adjoint ϕ^*. We generally note semantic intervals as follows:

$$U \xleftarrow{\phi} V.$$

The term ϕ is called the conversion of the semantic interval. Recall from Section 3 that $\phi \in V \Rightarrow U$ means that:

$$\forall e \in V,\ \phi\, e \in U.$$

Lemma 3 *The function ϕ^* sends every stack $\pi \in U^{\perp}$ to a stack $\phi^*\pi \in V^{\perp}$.*

Remark: The adjoint $\phi^\star$ is here to take full advantage of the duality between terms and stacks, saying that every existential type on *terms* is at the same time a universal type on *stacks*. Or similarly, that every union type on terms (see Section 7) is at the same time an intersection type on stacks. This dual perspective is crucial, we believe, to interpret existential and union types in the presence of recursive types (without any recourse to a "language-based" stratification).

5.3 Interpretation of interval types

We call *semantic environment* ρ any function from type variables to truth values. To any such environment ρ and interval type K we define a semantic interval $[\![K]\!]_\rho$ by structural induction on K. So, all along the section, we suppose given two interval types K and K' interpreted as:

$$[\![K]\!]_\rho = U \xleftarrow{\phi} V, \qquad [\![K']\!]_\rho = U' \xleftarrow{\psi} V'.$$

One needs to prove for each construction that the interpretation defines a semantic interval — which is not really difficult.

Arrow type: $[\![K \to K']\!]_\rho = V \Rightarrow U' \xleftarrow{\phi \Rightarrow \psi} U \Rightarrow V'$ where $\phi \Rightarrow \psi = \lambda x.(\psi \circ x \circ \phi) = \lambda x.\lambda y.\psi\,(x\,(\phi\, y))$.

Product type: $[\![K \times K']\!]_\rho = U \times U' \xleftarrow{\phi \times \psi} V \times V'$ where $\phi \times \psi = \lambda x.(\phi\,\mathtt{fst}(x), \psi\,\mathtt{snd}(x))$.

Boolean type: $[\![Bool]\!]_\rho = W \xleftarrow{\lambda x.x} W$ where W is the biorthogonal of the set $\{\mathtt{true}, \mathtt{false}\}$.

Bottom: $[\![\bot]\!]_\rho = W \xleftarrow{\lambda x.x} W$ where W is the smallest nonempty truth value, alternatively the set of looping terms, or the biorthogonal of the singleton $\{\Omega\}$.

Top: $[\![\top]\!]_\rho = W \xleftarrow{\lambda x.x} W$ where W is the largest truth value, that is, the set of all terms, or alternatively, the set of all terms orthogonal to the stack Ω.

Type variable: $[\![\alpha]\!]_\rho = W \xleftarrow{\lambda x.x} W$ where W is the truth value associated to the type variable α by the environment ρ.

Universal type: $[\![\forall\alpha.K]\!]_\rho = U \xleftarrow{\phi} V$ where U (resp. V) is the intersection of all truth values U_T (resp. V_T) such that $[\![K]\!]_{\rho,(\alpha\mapsto T)} = U_T \xleftarrow{\phi_T} V_T$ for T ranging over truth values. As usual, $[\![K]\!]_{\rho,(\alpha\mapsto T)}$ denotes the interpretation of K in the environment ρ in which the type variable α is assigned to T. The term ϕ is defined by showing that $\phi_T = \phi_{T'}$ for any truth values T and T', and then taking $\phi = \phi_T$ for any truth value T.

Existential type: $[\![\exists\alpha.K]\!]_\rho = U \xleftarrow{\phi} V$ where U (resp. V) is the biorthogonal of the union of all truth values U_T (resp. V_T) such that $[\![K]\!]_{\rho,(\alpha\mapsto T)} = U_T \xleftarrow{\phi_T} V_T$ for T ranging over truth values. The term ϕ is defined by showing that $\phi_T = \phi_{T'}$ for any truth values T and T', and then taking $\phi = \phi_T$ for any truth value T.

Interval type: $[\![[\bot,\top]]\!]_\rho = U \xleftarrow{\Omega} V$ where U (resp. V) is the smallest (resp. largest) nonempty truth value. Note that the term Ω transports every term $e \in V$ to the looping term $\Omega\, e \in U$.

6 Interpretation of types

Here comes the crux of the paper: we show that every (possibly infinite) type τ generates a converging sequence of interval types, the limit of which defines the interpretation of τ in the model. The proof is based on a simulation lemma (lemma 4) showing that, under some appropriate conditions, the conversion terms ϕ_K associated to our semantic intervals behave like η-conversions or reduction labels in the λ-calculus [18].

6.1 Term expansion

A term e' obtained from a term e by inserting conversion terms ϕ_K is called an expansion of e. We write this $e \rightsquigarrow e'$. This may be formalized by structural induction on the term:

$$e \rightsquigarrow e \qquad \frac{e \rightsquigarrow e'}{e \rightsquigarrow \phi_K\, e'} \qquad \frac{e \rightsquigarrow e'}{\lambda x.e \rightsquigarrow \lambda x.e'}$$

$$\frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2'} \qquad \frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2'}{(e_1, e_2) \rightsquigarrow (e_1', e_2')}$$

$$\frac{e \rightsquigarrow e'}{\mathtt{fst}(e) \rightsquigarrow \mathtt{fst}(e')} \qquad \frac{e \rightsquigarrow e'}{\mathtt{snd}(e) \rightsquigarrow \mathtt{snd}(e')}$$

$$\frac{e_1 \rightsquigarrow e_1' \qquad e_2 \rightsquigarrow e_2' \qquad e_3 \rightsquigarrow e_3'}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightsquigarrow \mathtt{if}\ e_1'\ \mathtt{then}\ e_2'\ \mathtt{else}\ e_3'}$$

where ϕ_K indexed by the interval type K denotes the conversion of any semantic interval $[\![K]\!]_\rho$, for ρ an arbitrary semantic environment.

The depth of an interval type K is defined by structural induction:

$$\begin{array}{lcl}
\|K \times K'\| \;=\; \|K \to K'\| & = & 1 + \min(\|K\|, \|K'\|) \\
\|\alpha\| \;=\; \|Bool\| & = & +\infty \\
\|\bot\| \;=\; \|\top\| & = & +\infty \\
\|\forall\alpha.K\| \;=\; \|\exists\alpha.K\| & = & \|K\| \\
\|[\bot, \top]\| & = & 0
\end{array}$$

We speak of an *expansion of depth* k when all the ϕ_K introduced by expansion are indexed by interval types K of depth greater or equal to k. In that case, we write $e \rightsquigarrow_k e'$.

6.2 The simulation lemma

Lemma 4 (simulation) *For every pair of terms (e, e') such that $e \to^* e'$, there exists an integer p such that whenever $e \rightsquigarrow_{p+k} f$ for a safe term f and integer k, there exists a term f' satisfying $f \to^* f'$ and $e' \rightsquigarrow_k f'$.*

6.3 Interpretation of types

Let ρ denote a semantic environment. Every type τ defines a set of approximating interval types $K \sqsubseteq \tau$, each of them interpreted as a semantic interval

$$[\![K]\!]_\rho = U_K \xleftarrow{\phi_K} V_K.$$

We define $U_\infty \overset{\text{def}}{=} \bigcup_{K \sqsubseteq \tau} U_K$ and $V_\infty \overset{\text{def}}{=} \bigcap_{K \sqsubseteq \tau} V_K$. Obviously, $U_\infty \subseteq V_\infty$ because $U_K \subseteq V_K$ for every $K \sqsubseteq \tau$. We deduce that V_∞ coincides with the biorthogonal of U_∞ (theorem 5) from the property:

$$\forall e \in V_\infty, \forall \pi \in U_\infty^\bot,\ e \perp \pi.$$

Theorem 5 (coincidence) $V_\infty = U_\infty^{\bot\bot}$.

We therefore interpret the type τ in the environment ρ as $[\![\tau]\!]_\rho = V_\infty = U_\infty^{\bot\bot}$. Note that the definition ensures substitution properties like $[\![\tau]\!]_{\rho + (\alpha \mapsto [\![\tau']\!]_\rho)} = [\![\tau[\alpha := \tau']]\!]_\rho$ for every types τ and τ', and type variable α.

7 Intersection and Union

We indicate briefly how we interpret union and intersection types in the presence of recursive types, see also [24]. The first step is to define two constructs $\wedge$ (intersection) and $\vee$ (union) on truth values U, V, just in the expected way:

$$U \wedge V = U \cap V, \qquad U \vee V = (U \cup V)^{\bot\bot}.$$

Our proof technique in Sections 5 and 6 requires to define the conversions $\phi \wedge \psi$ and $\phi \vee \psi$ associated to the constructs $\wedge$ and $\vee$ on interval types. We believe that this not possible in the operational model based on the call-by-name λ-calculus defined in Section 2. But this may be achieved by enriching the language with an "error-avoiding" nondeterministic choice operator $\|$, with the additional rules:

$$\langle e_1 \| e_2 \mid \mathrm{E}\rangle \to \langle e_1 \mid \mathrm{E}\rangle \qquad \langle e_1 \| e_2 \mid \mathrm{E}\rangle \to \langle e_2 \mid \mathrm{E}\rangle$$

with E an evaluation context in the sense of Section 2.2. The conversions $\phi \wedge \psi$ and $\phi \vee \psi$ may then be defined as the term $\phi \| \psi$. The existence of an adjoint for the term $\phi \| \psi$ requires to extend our class of stacks with an operator $\|$, building stacks $\pi_1 \| \pi_2$ with the obvious action on terms:

$$\langle e \mid \pi_1 \| \pi_2\rangle \quad = \quad \langle e \mid \pi_1\rangle \;\|\; \langle e \mid \pi_2\rangle$$

We clarify now the orthogonality relation $\perp$ associated to this non-deterministic calculus. A term is called *safe* when it may loop, or may reduce to `true` or `false`. A term e is *orthogonal* to a stack π when $\langle e \mid \pi\rangle$ is safe. For instance, the term $e = \mathtt{true} \;\|\; \lambda x.x$ is orthogonal to the stack $\pi = \mho \;\|\; (\mathtt{true} \cdot \mathtt{nil})$ because the term $\langle e \mid \pi\rangle$ reduces to the constant `true`. It is not difficult to see then that the adjoint of $\phi \| \psi$ is the function which associates to every stack π the stack $\phi^*\pi \| \psi^*\pi$, where ϕ^* (resp. ψ^*) denotes the adjoint of the term ϕ (resp. ψ).

8 Typing judgement

In order to demonstrate the power of our semantic analysis, we deliver a series of typing rules (figure 1) for the untyped calculus of Section 2, and prove that these rules are *sound*. We take the usual notions of *typing environment* Γ

$$\textsc{Var-Access}\ \frac{\Gamma(x)=\tau}{\Gamma\vdash x:\tau}\qquad \textsc{App}\ \frac{\Gamma\vdash e_1:\tau_2\to\tau_1\quad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash e_1\,e_2:\tau_1}\qquad \textsc{Abs}\ \frac{\Gamma,x:\tau_2\vdash e:\tau_1}{\Gamma\vdash\lambda x.e:\tau_2\to\tau_1}$$

$$\textsc{Pair}\ \frac{\Gamma\vdash e_1:\tau_1\qquad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash(e_1,e_2):\tau_1\times\tau_2}\qquad \textsc{Fst}\ \frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{fst}(e):\tau_1}\qquad \textsc{Snd}\ \frac{\Gamma\vdash e:\tau_1\times\tau_2}{\Gamma\vdash \mathtt{snd}(e):\tau_2}$$

$$\textsc{Constant true}\ \ \Gamma\vdash\mathtt{true}:\mathit{Bool}\qquad \textsc{Constant false}\ \ \Gamma\vdash\mathtt{false}:\mathit{Bool}$$

$$\textsc{Conditional}\ \frac{\Gamma\vdash e_1:\mathit{Bool}\quad \Gamma\vdash e_2:\tau\qquad \Gamma\vdash e_3:\tau}{\Gamma\vdash\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3:\tau}\qquad \textsc{Fixpoint}\ \frac{\Gamma\vdash e:\tau\to\tau}{\Gamma\vdash Y\,e:\tau}$$

$$\textsc{All-Intro}\ \frac{\Gamma,\alpha\vdash e:\tau}{\Gamma\vdash e:\forall\alpha.\tau}\qquad \textsc{All-Elim}\ \frac{\Gamma\vdash e:\forall\alpha.\tau}{\Gamma\vdash e:\tau[\tau'/\alpha]}$$

$$\textsc{Exists-Intro}\ \frac{\Gamma\vdash e:\tau[\tau'/\alpha]}{\Gamma\vdash e:\exists\alpha.\tau}\qquad \textsc{Exists-Elim}\ \frac{\Gamma\vdash e:\exists\alpha.\tau'\quad \Gamma,\alpha,x:\tau'\vdash\langle x\mid \mathrm{E}\rangle:\tau\quad \alpha\notin\mathrm{FV}(\tau)\qquad x\notin\mathrm{FV}(\mathrm{E})}{\Gamma\vdash\langle e\mid\mathrm{E}\rangle:\tau}\qquad \textsc{Sub}\ \frac{\Gamma\vdash e:\tau'\qquad \tau'<:\tau}{\Gamma\vdash e:\tau}$$

Figure 1. Typing rules

and of *typing judgement* $\Gamma\vdash e:\tau$, and refer to [25] for formal definitions. Our typing rules are also canonical, except for the elimination rule of the existential, which is inspired from [14]. We write $\Gamma(x)$ for the type of the variable x in the environment Γ, and FV for the set of free variables of a type or an evaluation context.

The typing system depends on a subtyping relation $<:$ between types, which appears in the Sub rule. The only hypothesis that we make on the relation $<:$ is that for every two types τ,τ' such that $\tau<:\tau'$, and for every semantic environment ρ, we have the inclusion $[\![\tau]\!]_\rho\subseteq[\![\tau']\!]_\rho$. This is sufficient to establish that the typing system is sound:

Theorem 6 (Soundness) $\vdash e:\tau$ *implies* $e\in[\![\tau]\!]$.

This establishes that every closed term of type *Bool* is safe.

9 Parametricity and typed realizability

We explain briefly how realizability may be adapted to account for logical relations and parametricity. Following [2, 6, 9, 7, 22] among a few others, we would like to interpret types as *saturated relations*. A saturated relation $(\sim_1,r,\sim_2)$ over the set of terms Λ consists of a binary relation $r\subseteq\Lambda^2$ and two partial equivalence relations (pers) $\sim_1$ and $\sim_2$ over Λ, such that:

$$\forall(a,b,c,d)\in\Lambda^4,\quad a\sim_1 b \text{ and } b\;r\;c \text{ and } c\sim_2 d\;\Rightarrow\; a\;r\;d.$$

We find useful to express every saturated relation $(\sim_1,r,\sim_2)$ as the following set of quadruples:

$$R=\{(a,b,c,d)\in\Lambda^4, a\sim_1 b,\ b\;r\;c \text{ and } c\sim_2 d\}.$$

We define an orthogonality relation $\perp^{\mathsf{sat}}$ between quadruples of terms and stacks of our deterministic language of Section 2, by writing $(e_1,e_2,e_3,e_4)\perp^{\mathsf{sat}}(\pi_1,\pi_2,\pi_3,\pi_4)$ precisely when

$$\begin{gathered}\langle e_1\mid\pi_1\rangle=_{\text{safe}}\langle e_1\mid\pi_2\rangle=_{\text{safe}}\langle e_2\mid\pi_1\rangle=_{\text{safe}}\langle e_2\mid\pi_2\rangle\\ \langle e_3\mid\pi_3\rangle=_{\text{safe}}\langle e_3\mid\pi_4\rangle=_{\text{safe}}\langle e_4\mid\pi_3\rangle=_{\text{safe}}\langle e_4\mid\pi_4\rangle\\ \langle e_2\mid\pi_2\rangle=_{\text{safe}}\langle e_3\mid\pi_3\rangle\end{gathered}$$

where $e=_{\text{safe}}e'$ means that (1) the terms e and e' are safe and (2a) either e and e' reduce to the same boolean constant, or (2b) both e and e' loop. A key observation follows:

Lemma 7 (Saturation) *Every biorthogonal set of quadruples of terms is a saturated relation R.*

It is then easy to construct a realizability model of recursive types based on *biorthogonal relations* instead of truth values. The operators $\Rightarrow$ and $\times$ are adapted to biorthogonal relations. That is, $R\Rightarrow S$ denotes the set of quadruples orthogonal to every quadruple $(e_1\cdot\pi_1,e_2\cdot\pi_2,e_3\cdot\pi_3,e_4\cdot\pi_4)$ where $(e_1,e_2,e_3,e_4)\in R$ and $(\pi_1,\pi_2,\pi_3,\pi_4)\in S^\perp$. Similarly for the product $R\times S$ of two biorthogonal relations.

Then, one interprets recursive types by approximating them by interval types, in the lines of Sections 5 and 6. This defines a biorthogonal relation $[\![\tau]\!]_\rho$ for every type τ and semantic environment ρ from type variables to biorthogonal relations. Now, suppose that the subtyping relation $<:$ verifies that for every two types $\tau<:\tau'$, and for every semantic environment ρ, we have the inclusion $[\![\tau]\!]_\rho\subseteq[\![\tau']\!]_\rho$. We prove that:

Theorem 8 (Soundness) $\vdash e:\tau$ *implies* $(e,e,e,e)\in[\![\tau]\!]$.

Consider two closed terms e_1 and e_2 typed as $\vdash e_1:\tau$ and $\vdash e_2:\tau$ in our typing system. We say that the terms e_1,e_2 are parametrically equivalent (noted $e_1\Delta^\tau e_2$) when $(e_1,e_1,e_2,e_2)\in[\![\tau]\!]$. And that they are contextually equivalent (noted $e_1\sim^\tau_{\mathsf{ctx}}e_2$) when $e\,e_1=_{\text{safe}}e\,e_2$ for every closed term e typed as $\vdash e:\tau\to\mathit{Bool}$ in our typing system. We prove that for every type τ:

Lemma 9 (in untyped realizability) $\Delta^\tau\subseteq\sim^\tau_{\mathsf{ctx}}$.

This indicates that there are more parametricity tests in the untyped realizability universe, than in the typed syntax. To obtain equality of Δ^τ and $\sim^\tau_{\mathsf{ctx}}$, we thus need to shift to a *typed* setting, in which only typed terms and stacks (à la

Church) are considered. The orthogonality relation $e \perp \pi$ holds when the term $\langle e \mid \pi \rangle$ is *well-typed* and safe. The definition of Δ^τ is immediately adapted to this typed setting. We establish in this way the key property stated by Pitts for his PolyPCF (theorem 4.15 [22]), reformulated in our polymorphic typing system with subtyping and recursive types.

Theorem 10 (in typed realizability) $\Delta^\tau = \sim^\tau_{\mathsf{ctx}}$.

10 Conclusion and future works

We have shown how recursive polymorphic types, as well as subtyping, may be interpreted operationally without altering the original syntax of the λ-calculus. We have also indicated how parametricity may be integrated smoothly in the framework. It will be interesting to see in future work how the methodology scales up to languages with effects, and to process calculi.

References

[1] M. Abadi. top-top-closed relations and admissibility. *Mathematical Structures in Computer Science*, 10(3):313–320, 2000.

[2] M. Abadi, P.-L. Curien, and G. Plotkin. Formal parametric polymorphism. *Theoretic Computer Science*, 121(1 and 2):9–58, 1993.

[3] M. Abadi, B. Pierce, and G. Plotkin. Faithful ideal models for recursive polymorphic types. *International Journal of Foundations of Computer Science*, 2(1):1–21, Mar. 1991. Summary in Fourth Annual Symposium on Logic in Computer Science, June, 1989.

[4] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[5] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[6] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. In G. Huet, editor, *Logical Foundations of Functional Programming*, pages 315–330. Addison-Wesley, Reading, MA, 1990.

[7] R. Bellucci, M. Abadi, and P.-L. Curien. A model for formal parametric polymorphism: A per interpretation for system r. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications: Proc. of the 2nd International Conference on Typed Lambda Calculi and Applications*, pages 32–46. Springer, Berlin, Heidelberg, 1995.

[8] L. Birkedal and R. Harper. Constructing interpretations of recursives types in an operational setting. *Information and Computation*, 155:3–63, 1999.

[9] K. Bruce and J. C. Mitchell. Per models of subtyping, recursive types and higher-order polymorphism. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–327, New York, NY, USA, 1992. ACM Press.

[10] J. Chroboczek. Games semantics and subtyping. PhD thesis and LFCS report ECS-LFCS-03-432, University of Edinburgh, Great Britain, 2003.

[11] L. Dami. Labelled reductions, runtime errors and operational subsumption. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 782–793. Springer, 1997.

[12] L. Dami. Operational subsumption, an ideal model of subtyping. In A. D. Gordon, A. M. Pitts, and C. Talcott, editors, *Second Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.

[13] V. Danos and J.-L. Krivine. Disjunctive tautologies and synchronisation schemes. In *Computer Science Logic'00*, volume 1862 of *Lecture Notes in Computer Science*, pages 292–301. Springer, 2000.

[14] J. Dunfield and F. Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Proc. 6th International Conference on Foundations of Software Science and Computation Structures (FOSSACS'03)*, Lecture Notes in Computer Science. Springer–Verlag, 2003.

[15] P. J. Freyd. Algebraically complete categories. In A. Carboni, M. C. Pedicchio, and G. Rosolini, editors, *Proceedings of the 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156. Springer–Verlag, 1991.

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[17] J.-L. Krivine. Typed lambda-calculus in classical zermelo-fraenkel set theory. *Archive of Mathematical Logic*, 40(3):189–205, 2001.

[18] J.-J. Lévy. An algebraic interpretation of the lambda beta K-calculus; and an application of a labelled lambda-calculus. *Theoretical Computer Science*, 2(1):97–114, June 1976.

[19] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[20] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

[21] M. Parigot. Strong normalization for second order classical natural deduction. In *8th Annual IEEE Symposium on Logic in Computer Science*, pages 39–46, Montreal, Canada, June 1993. IEEE Computer Society Press.

[22] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in computer Science*, 10:321–359, 2000.

[23] S. F. Smith. The coverage of operational semantics. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 307–346. Cambridge University Press, 1998.

[24] J. Vouillon. Subtyping union types. In J. Marcinkowski and A. Tarlecki, editors, *18th International Workshop CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 415–429. Springer-Verlag, September 2004.

[25] J. Vouillon and P.-A. Melliès. Semantic types: A fresh look at the ideal model for types. In *Proceedings of the 31th ACM Conference on Principles of Programming Languages*, pages 52–63, Venezia, Italia, Jan. 2004. ACM Press.

Appendix

Proofs of section 5

Arrow type. We prove that

$$V \Rightarrow U' \overset{\phi \Rightarrow \psi}{\longleftarrow} U \Rightarrow V'$$

defines a semantic interval.
Conversion : We prove first that

$$(\phi \Rightarrow \psi) \in (U \Rightarrow V') \Rightarrow (V \Rightarrow U'). \qquad (8)$$

Suppose that $e \in U \Rightarrow V'$. We want to prove that $(\phi \Rightarrow \psi)\, e \in V \Rightarrow U'$. It is enough to prove that $(\phi \Rightarrow \psi)\, e$ is orthogonal to every stack $\pi' = e' \cdot \pi$ where $e' \in V$ and π is orthogonal to U'. By hypothesis, U' is nonempty; thus, π is not the unsafe stack $\mho$. On the other hand, the case when π is the safe stack Ω is immediate. There remains to treat the case when π is an evaluation context E. In that case,

$$\begin{array}{rcl} \langle (\phi \Rightarrow \psi)\, e \mid \pi' \rangle & = & \langle \lambda x.\lambda y.\psi\,(x\,(\phi\,y)) \mid e \cdot e' \cdot \mathrm{E} \rangle \\ & \rightarrow & \langle \lambda y.\psi\,(e\,(\phi\,y)) \mid e' \cdot \mathrm{E} \rangle \\ & \rightarrow & \langle \psi\,(e\,(\phi\,e')) \mid \mathrm{E} \rangle \end{array}$$

There remains to show that the term $\psi\,(e\,(\phi\,e'))$ is element of U'. But this is a consequence of lemma 1, and the joint facts that $\phi \in V \Rightarrow U$, that $e \in U \Rightarrow V'$ and that $\psi \in V' \Rightarrow U'$. We conclude that $(\phi \Rightarrow \psi)\, e \perp \pi'$, and thus property (8).
Adjoint : We define $(\phi \Rightarrow \psi)^*$ and prove that this is indeed the adjoint of $(\phi \Rightarrow \psi)$.

- $(\phi \Rightarrow \psi)^* \pi = \pi$ when π is the safe stack Ω. Indeed, in that case, both $(\phi \Rightarrow \psi)\, e \perp \pi$ and $e \perp \pi$, for every term e.
- $(\phi \Rightarrow \psi)^* \pi = \phi\, e \cdot \psi^* \mathrm{E}$ when $\pi = e \cdot \mathrm{E}$ for some term e and evaluation context E. Indeed, $(\phi \Rightarrow \psi)\, e' \perp e \cdot \mathrm{E}$ iff $\psi\,(e'\,(\phi\, e)) \perp \mathrm{E}$ iff $e'\,(\phi\, e) \perp \psi^* \mathrm{E}$ iff $e' \perp \phi\, e \cdot \psi^* \mathrm{E}$, for e' an arbitrary term.
- $(\phi \Rightarrow \psi)^* \pi = \mho$ otherwise. Indeed, in that case, the term $\langle (\phi \Rightarrow \psi)\, e \mid \pi \rangle$ is easily shown to be unsafe.

Product type. We prove that

$$[\![K \times K']\!] = U \times U' \overset{\phi \times \psi}{\longleftarrow} V \times V'$$

defines a semantic interval.
Conversion : We prove first that

$$(\phi \times \psi) \in (V \times V') \Rightarrow (U \times U'). \qquad (9)$$

Suppose that $e \in V \times V'$. We want to prove that $(\phi \times \psi)\, e \in U \times U'$. This means proving that $(\phi \times \psi)\, e$ is (1) orthogonal to every stack $\texttt{fst} \cdot \pi$ where π is orthogonal to V, and also (2) orthogonal to every stack $\texttt{snd} \cdot \pi'$ where π' is orthogonal to V'. We only show point (1), since point (2) is proved in a similar fashion. By hypothesis, V is is nonempty; thus, π is the unsafe stack $\mho$. On the other hand, the case when π is the safe stack Ω is immediate. There remains to treat the case when π is an evaluation context E. In that case,

$$\begin{array}{lr} \langle (\phi \times \psi)\, e \mid \texttt{fst} \cdot \mathrm{E} \rangle & = \\ \langle \lambda x.(\phi\, \texttt{fst}(x), \psi\, \texttt{snd}(x)) \mid e \cdot \texttt{fst} \cdot \mathrm{E} \rangle & \rightarrow \\ \langle (\phi\, \texttt{fst}(e), \psi\, \texttt{snd}(e)) \mid \texttt{fst} \cdot \mathrm{E} \rangle & \rightarrow \\ \langle \phi\, \texttt{fst}(e) \mid \mathrm{E} \rangle & \end{array}$$

Now, $\langle \phi\, \texttt{fst}(e) \mid \mathrm{E} \rangle$ is safe iff $\phi\, \texttt{fst}(e) \perp \mathrm{E}$ iff $\texttt{fst}(e) \perp \phi^* \mathrm{E}$ iff $e \perp \texttt{fst} \cdot \phi^* \mathrm{E}$. Now, the stack $\phi^* \mathrm{E}$ is orthogonal to U, and e is element of $V \times V'$. From this and definition of $V \times V'$ follows that e is orthogonal to $\texttt{fst} \cdot \phi^* \mathrm{E}$. We conclude that $\langle \phi\, \texttt{fst}(e) \mid \mathrm{E} \rangle$ is safe, and thus that $(\phi \times \psi)\, e \perp \texttt{fst} \cdot \mathrm{E}$. As we said, point (2) is established similarly. We conclude that $(\phi \times \psi)\, e \in U \times U'$.
Adjoint : We define $(\phi \times \psi)^*$ and prove that this is indeed the adjoint of $(\phi \times \psi)$.

- $(\phi \times \psi)^* \pi = \pi$ when π is the safe stack Ω. Indeed, in that case, both $(\phi \times \psi)\, e \perp \pi$ and $e \perp \pi$, for every term e.
- $(\phi \times \psi)^* \pi = \texttt{fst} \cdot \phi^* \mathrm{E}$ when $\pi = \texttt{fst} \cdot \mathrm{E}$ for some evaluation context E. Indeed, $(\phi \times \psi)\, e' \perp \texttt{fst} \cdot \mathrm{E}$ iff $\phi\, \texttt{fst}(e) \perp \mathrm{E}$ iff $e \perp \texttt{fst} \cdot \phi^* \mathrm{E}$, for every term e.
- similarly, $(\phi \times \psi)^* \pi = \texttt{snd} \cdot \phi^* \mathrm{E}$ when $\pi = \texttt{snd} \cdot \mathrm{E}$ for some evaluation context E.
- $(\phi \times \psi)^* \pi = \mho$ otherwise. Indeed, in that case, the term $\langle (\phi \times \psi)\, e \mid \pi \rangle$ is easily shown to be unsafe.

Boolean type, bottom, top, type variable. In each case, the term $\lambda x.x$ is element of $W \Rightarrow W$ and has the identity function on stacks as adjoint $(\lambda x.x)^*$.

Universal type. We prove that

$$[\![\forall \alpha.K]\!]_\rho = U \overset{\phi}{\longleftarrow} V$$

defines a semantic interval. By definition, the term ϕ is equal to the term ϕ_T for any truth value T. Consequently, the term ϕ has an adjoint $\phi^* = \phi_T$.
Conversion : There remains to prove that

$$\phi \in V \Rightarrow U.$$

Suppose that e is a term in V, and that π is a stack orthogonal to U. Suppose that T is a truth value. Then, the term e is element of V_T. Thus, the term $\phi\, e$ is element of U_T. This is true for every truth value T. Thus, the term $\phi\, e$ is element of V, for every term $e \in V$. We conclude that $\phi \in V \Rightarrow U$.

Existential type. We prove that

$$[\![\forall\alpha.K]\!]_\rho = U \xleftarrow{\phi} V$$

defines a semantic interval. By definition, the term ϕ is equal to the term ϕ_T for any truth value T. Consequently, the term ϕ has an adjoint $\phi^* = \phi_T$.
Conversion: There remains to prove that

$$\phi \in V \Rightarrow U.$$

Suppose that e is a term in V, and that π is a stack orthogonal to U. Suppose that T is a truth value. Then, the stack π is orthogonal to U_T. Thus, the stack $\phi^*\pi$ is orthogonal to V_T. This is true for every truth value T. Thus, the stack $\phi^*\,\pi$ is orthogonal to V. In particular, $e \perp \phi^*\,\pi$. It follows immediately that $\phi\, e \perp \pi$. This is true for every term e element of V and for every stack π orthogonal to U. We conclude that $\phi \in V \Rightarrow U$.

Interval type. The definition works because the term $\Omega\, e$ is element of U for every term e element of V (that is: for every term). Besides, the term Ω has an adjoint Ω^*, which transports every stack π to the safe stack Ω.

Proofs of Section 6.2 and Section 6.3 (sketched)

Lemma 4 (simulation) *For every pair of terms (e, e') such that $e \to^* e'$, there exists an integer p such that whenever $e \rightsquigarrow_{p+k} f$ for a safe term f and integer k, there exists a term f' satisfying:*

$$f \to^* f' \text{ and } e' \rightsquigarrow_k f'.$$

Proof: (sketch) The general idea is that expanding a term e to a term f with a conversion ϕ_K either induces an error in f, or behaves just like an η-expansion. Typically,

$$\langle \phi_{K_2 \Rightarrow K_1}\,(e) \mid e' \cdot \mathrm{E}\rangle \to^* \langle \phi_{K_1}\,(e\,(\phi_{K_2}\,e')) \mid \mathrm{E}\rangle.$$

One important point is that no conversion Ω associated to the interval type $[\perp, \top]$ applies inside the reduction $f \to^* f'$. This would break the simulation. For that reason, we require an expansion depth $e \rightsquigarrow_{p+k} f$ larger than the product p of the length of the shortest reduction $e \to^* e'$, and of the maximum length of an evaluation context involved in the reduction. The *length* of an evaluation context is the number of application nodes $e \cdot \mathrm{E}$ in its definition. ■

Theorem 5 (coincidence) $V_\infty = U_\infty^{\perp\perp}$.

Proof: (sketch) We deduce that $U_\infty \subseteq V_\infty$, from the fact that $U_K \subseteq V_K$ for every interval type $K \sqsubseteq \tau$. The truth value V_∞ which contains U_∞ contains also its biorthogonal $U_\infty^{\perp\perp}$. This establishes that $U_\infty^{\perp\perp} \subseteq V_\infty$. We prove the converse inclusion $V_\infty \subseteq U_\infty^{\perp\perp}$, which may be reformulated as the statement below:

$$\forall e \in V_\infty, \forall \pi \in U_\infty^{\perp},\; e \perp \pi.$$

The orthogonal of U_∞ is given by an intersection:

$$U_\infty^{\perp} = \bigcap_{K \sqsubseteq \tau} U_K^{\perp}$$

So, every stack $\pi \in U_\infty^{\perp}$ is orthogonal to U_K, and every element $e \in V_\infty$ is transported to a term $\phi_K\, e \in U_K$, for $K \sqsubseteq \tau$. We conclude that

$$\forall K \sqsubseteq \tau,\; \phi_K\, e \perp \pi. \qquad (10)$$

We claim that this implies that $e \perp \pi$. This is immediate when π is one of the stacks Ω or $\mho$. We proceed by contradiction when the stack $\pi = \mathrm{E}$ is strict (= an evaluation context). Suppose that the term $\langle e \mid \mathrm{E}\rangle$ is unsafe. This means that $\langle e \mid \mathrm{E}\rangle \to e'$ to a term e' which cannot be further reduced by $\to$, but which is neither the constant `true` nor the constant `false`. By lemma 4, we may choose an integer p such that, for every K of depth $p + k$, the term $f = \langle \phi_K\, e \mid \mathrm{E}\rangle$ reduces to a term f' such that $e' \rightsquigarrow_k f'$. It is not difficult to see that, if k is chosen larger than the length of any evaluation context in e', then the term f' is just as unsafe as e'. We conclude that $\langle \phi_K\, e \mid \mathrm{E}\rangle$ is unsafe, or equivalently that $\phi_K\, e$ is not orthogonal to E, and thus reach a contradiction with (10). We conclude that $e \perp \pi$. ■