



# A Realizability Model for a Semantical Value Restriction

Rodolphe Lepigre

## ► To cite this version:

| Rodolphe Lepigre. A Realizability Model for a Semantical Value Restriction. 2015. hal-01107429v1

**HAL Id: hal-01107429**

**<https://inria.hal.science/hal-01107429v1>**

Preprint submitted on 21 Jan 2015 (v1), last revised 24 Mar 2016 (v3)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# A Realizability Model for a Semantical Value Restriction

Rodolphe Lepigre

Laboratoire de Mathématiques, UMR 5127 CNRS

Université Savoie Mont-Blanc

73 376 Le Bourget-du-Lac, France

rodolphe.lepigre@univ-savoie.fr

**Abstract**—Reconciling dependent product and classical logic with call-by-value evaluation is a difficult problem. It is the first step toward a classical proof system for an ML-like language. In such a system, the introduction rule for universal quantification and the elimination rule for dependent product need to be restricted: they can only be applied to values. This value restriction is acceptable for universal quantification (ML-like polymorphism) but makes dependent product unusable in practice.

In order to circumvent this limitation we introduce new typing rules and prove their consistency by constructing a realizability model in three layers (values, stacks and terms). Such a technique has already been used to account for classical ML-like polymorphism in call-by-value, and we here extend it to handle dependent product. The main idea is to internalize observational equivalence as a new non-computable operation in the calculus. A crucial property of the obtained model is that the biorthogonal of a set of values which is closed under observational equivalence does not contain more values than the original set.

## INTRODUCTION

The most actively developed proof assistants following Curry-Howard are Coq [1] and Agda [2]. These systems are both based on constructive type theories: Martin-Löf's dependent type theory [3] or Coquand and Huet's calculus of constructions [4]. ML-like languages may also be considered as proof assistants following the Curry-Howard isomorphism. However they lack *dependent types* (i.e. indexed families) and are inconsistent as a logic (terms of type  $\forall\alpha\ \alpha$  can be defined).

Christophe Raffalli, Pierre Hyvernat, Tom Hirschowitz and the author are trying to build a system called *PML* which proof-terms embed a real ML-like language. We want this language to be *sound* in two ways: the logic should be consistent and programs should be type-safe. Several choices are to be made when designing such a proof assistant or type system.

*Type annotations.* There are two different styles for denoting typed  $\lambda$ -terms in the syntax:

- The *à la Church* approach consists in adding type information on  $\lambda$ -abstractions. Thus the term  $\lambda x^A\ x$  can only be given type  $A \rightarrow A$ . In this context, the programming language includes a type abstraction and a type application. For example, one could define the identity function as  $\Lambda\alpha\ \lambda x^\alpha\ x$  so that it can be applied to any type. However, the programmer is to provide an additional type argument at each application of the function.
- The *à la Curry* approach works on pure  $\lambda$ -terms, thus a type inference mechanism is required in order to type-check terms. However, the  $\lambda$ -term  $\lambda x\ x$  can now be given any type of the form  $\alpha \rightarrow \alpha$ . Such a type can be generalized and made into the polymorphic type  $\forall\alpha.\alpha \rightarrow \alpha$  so that the identity function can be used on any input type.

Both Coq [1] and Agda [2] adopted the Church style syntax. This decision is probably due to the fact that Curry style syntax has undecidable type-checking. Here we choose to follow ML and opt for a Curry style syntax anyway. As a consequence, our system will have a partial type-checking algorithm. Although this choice might seem surprising, one should consider the fact that Coq and Agda use mechanisms handling meta-variables and implicit arguments that are also partial. If such systems are required to obtain a usable proof assistant, we might as well have an undecidable system from the start.

*Evaluation strategy.* In languages with the Church-Rosser property like Coq and Agda, the evaluation order is not relevant. One of the design choices of *PML* is to have control operators to interpret classical logic, hence the order of evaluation matters. There are mainly two alternatives when evaluating  $\lambda$ -terms:

- *Lazy evaluation* postpones the computation of the argument of a function to the time of its effective use.
- *Call-by-value* evaluation computes the argument of functions before their effective application.

Here we choose to follow most implementations of ML (SML, Caml) and select call-by-value<sup>1</sup>. In the context of Curry style call-by-value languages, some non-trivial work is required to add side-effects while remaining sound. Control operators are probably the most troublesome kind of side-effects in presence of dependent product.

The favored solution is value restriction that we will explain now in more details.

## A. A brief history of ML and value restriction

The soundness issues related to side-effects and call-by-value arose in the early seventies with the advent of ML. This

<sup>1</sup>There are some call-by-name implementations of ML like Lazy ML, but mainstream implementations are all call-by-value.

problem related to a bad interaction between Hindley-Milner polymorphism and side-effects have been extensively studied.

When a reference with a polymorphic type is created, one should not be able to assign it or dereference it using different types. As explained very well by Andrew Wright ([5] section 2), the naive approach consisting in defining the following three primitive polymorphic procedures for references is doomed.

```
val ref   : ∀α. α → α ref
val (!)   : ∀α. α ref → α
val (:=)  : ∀α. α ref → α → unit
```

For instance the following program is accepted by the type-checker, even though it performs an addition between a boolean and an integer.

```
let r = ref [] in r := [true]; 42 + (hd !r)
```

Many researchers built type systems supporting references that were both sound and conservative over Hindley-Milner [6] for purely functional programs [7]–[9] (not exhaustive). However, they all introduced a complexity that contrasted with the beautiful simplicity of ML's original type system. Annotations had to be added to the types of imperative programs, which broke the modularity of the language (see [10] section 2 and [11] section 2 for a detailed account).

It became clear around 1993 that it would not be possible to achieve the goal of conservativity in a satisfactory way. Xavier Leroy first proposed to extend the language with a call-by-name let-binding<sup>2</sup> forbidding side effects [12]. This solution left the type system unchanged, but made the syntax more complex. At last, Andrew Wright designed the most simple and elegant solution: value restriction [5], [10]. The idea was to restrict the generalization in let-bindings to cases where the bound term is a value.

In slightly more expressive type systems, this restriction appears on the typing rule for the introduction of the universal quantification. The usual rule

$$\frac{\Gamma \vdash t : A \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash t : \forall \alpha A}$$

cannot be proved safe in a system with side-effects if  $t$  is not a syntactic value.

### B. On control operators and classical logic

Since Timothy Griffin's seminal paper [13], it has been well-known that classical logic can be given a computational interpretation in terms of control operators. Here we chose to use another well-known framework supporting classical logic: Michel Parigot's  $\lambda\mu$ -calculus [14].

In 1991, Robert Harper and Mark Lillibridge found a program breaking type safety for ML extended with the control operator *call/cc*<sup>3</sup> [15].

```
val callcc : ∀α. (α cont → α) → α
val throw  : ∀αβ. α cont → α → β
```

<sup>2</sup>In ML the polymorphism mechanism is strongly linked with let-bindings (i.e. expressions of the form `let x = u in t`).

<sup>3</sup>*Call/cc* means “call with current continuation” and was first introduced in the language Scheme.

Here we chose a syntax similar to Michel Parigot's  $\lambda\mu$ -calculus [14]. In this calculus *call/cc* and *throw* are replaced with a new binder  $\mu\alpha t$  to capture the current continuation in the  $\mu$ -variable  $\alpha$ . To restore this continuation we use the syntax  $t * \alpha$  (denoted  $[\alpha]t$  in the original calculus). In terms of control operators,  $\mu\alpha t$  can be translated to *callcc*  $(\lambda\alpha u)$  and  $t * \alpha$  as *throw*  $\alpha t$ .

The soundness problem arises when evaluating  $t (\mu\alpha u)$  when  $\mu\alpha u$  has a polymorphic type. The natural way to transform the corresponding proof-tree leads to an illegal introduction rule for the universal quantifier. Figure 1 shows how subject reduction (i.e. type preservation) fails for the following reduction rule.

$$t (\mu\alpha u) \longrightarrow_{\mu} \mu\beta (t u [\_ * \alpha := t \_ * \beta])$$

The substitution  $[\_ * \alpha := t \_ * \beta]$  here replaces any term of the form  $t' * \alpha$  by  $t t' * \beta$ . Every rule used in figure 1 is part of the type system of *PML* (figure 4).

As for references, value restriction solves the inconsistency and yields a sound type system. This forbids to generalize the type of  $\mu\alpha u$  as it is not a value.

### C. The call-by-value dependent product problem

In much the same way as the introduction rule for the universal quantification, the elimination rule for the dependent product

$$\frac{\Gamma \vdash t : \Pi_{x:A} B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B[x := u]}$$

cannot be proved safe in the case where  $u$  is not a value. Now the question is: can we live with such restriction? The answer is yes on the logical side as  $A$  is logically equivalent to  $\top \Rightarrow A$ . However, this solution is not satisfactory as it consists in emulating call-by-name evaluation by guarding terms with a dummy  $\lambda$ -abstraction. The dependent product type is made very weak since it is no longer possible to write types like  $List(n + m)$  as  $n + m$  is not a value.

The problem arises with the same reduction of  $t (\mu\alpha u)$  as above when  $t$  has a type of the form  $\Pi_{x:A} B$  (see figure 2).

### D. Toward a semantical value restriction

The main contribution of this paper is a new approach to value restriction. We chose to replace the syntactic restriction on terms by a semantical restriction expressed in terms of an observational equivalence denoted  $t \equiv u$ . The introduction of the universal quantification and the elimination of the dependant product hence become the following.

$$\frac{\Gamma, u \equiv v \vdash u : A \quad \alpha \notin FV(\Gamma)}{\Gamma, u \equiv v \vdash u : \forall \alpha A}$$

$$\frac{\Gamma, u \equiv v \vdash t : \Pi_{x:A} B \quad \Gamma, u \equiv v \vdash u : A}{\Gamma, u \equiv v \vdash t u : B[x := v]}$$

We do not require  $u$  to be a value, however we require that it is equivalent to some value  $v$ . This can be understood in terms of convergence: a term either reduces to a value or it diverges

$$\begin{array}{c}
\frac{\Gamma, \alpha : \neg A \vdash t' : A}{\Gamma, \alpha : \neg A \vdash t' * \alpha : C} * \\
\vdots \\
\frac{\Gamma, \alpha : \neg A \vdash u : A \quad X \notin FV(\Gamma, \alpha : \neg A)}{\Gamma, \alpha : \neg A \vdash \mu \alpha u : \forall X A} \forall_i \\
\frac{\Gamma, \alpha : \neg A \vdash t : (\forall X A) \Rightarrow B \quad \Gamma, \alpha : \neg A \vdash \mu \alpha u : \forall X A}{\Gamma, \alpha : \neg A \vdash t (\mu \alpha u) : B} \Rightarrow_e
\end{array}$$

Reduction rule:  $t (\mu \alpha u) \longrightarrow_{\mu} \mu \beta (t u[_* \alpha := t_* * \beta])$

$$\begin{array}{c}
\frac{\Gamma, \beta : \neg B \vdash t : (\forall X A) \Rightarrow B \quad \frac{\Gamma, \beta : \neg B \vdash t' : A}{\Gamma, \beta : \neg B \vdash t' : \forall X A} \text{Not safe}}{\Gamma, \beta : \neg B \vdash t t' : B} \Rightarrow_e \\
\frac{\Gamma, \beta : \neg B \vdash t t' * \beta : C}{\vdots} \\
\frac{\Gamma, \beta : \neg B \vdash t : (\forall X A) \Rightarrow B \quad \Gamma, \beta : \neg B \vdash u[_* \alpha := t_* * \beta] : \forall X A}{\Gamma, \beta : \neg B \vdash t u[_* \alpha := t_* * \beta] : B} \Rightarrow_e \\
\frac{\Gamma, \beta : \neg B \vdash \mu \beta (t u[_* \alpha := t_* * \beta]) : B}{\Gamma, \beta : \neg B \vdash \mu \beta (t u[_* \alpha := t_* * \beta]) : B} \mu
\end{array}$$

Fig. 1. The failure of subject reduction on the  $\forall_i$  rule.

$$\begin{array}{c}
\frac{\Gamma, \alpha : \neg A \vdash t' : A}{\Gamma, \alpha : \neg A \vdash t' * \alpha : C} * \\
\vdots \\
\frac{\Gamma, \alpha : \neg A \vdash u : A}{\Gamma, \alpha : \neg A \vdash \mu \alpha u : A} \mu \\
\frac{\Gamma, \alpha : \neg A \vdash t : \Pi_{x:A} B \quad \Gamma, \alpha : \neg A \vdash \mu \alpha u : A}{\Gamma, \alpha : \neg A \vdash t (\mu \alpha u) : B[x := \mu \alpha u]} \Pi_e
\end{array}$$

Reduction rule:  $t (\mu \alpha u) \longrightarrow_{\mu} \mu \beta (t u[_* \alpha := t_* * \beta])$

$$\begin{array}{c}
\frac{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t : \Pi_{x:A} B \quad \Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t' : A}{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t t' : B[x := t']} \Pi_e \\
\frac{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t t' : B[x := t']}{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t t' * \beta : C} \text{Not safe} \\
\vdots \\
\frac{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t : \Pi_{x:A} B \quad \Gamma, \beta : \neg B[x := \mu \alpha u] \vdash u[_* \alpha := t_* * \beta] : A}{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash t u[_* \alpha := t_* * \beta] : B[x := \mu \alpha u]} \Pi_e \\
\frac{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash \mu \beta (t u[_* \alpha := t_* * \beta]) : B[x := \mu \alpha u]}{\Gamma, \beta : \neg B[x := \mu \alpha u] \vdash \mu \beta (t u[_* \alpha := t_* * \beta]) : B[x := \mu \alpha u]} \mu
\end{array}$$

Fig. 2. The failure of subject reduction on the  $\Pi_e$  rule.

by calling a continuation (i.e. raising an exception). Although this approach looks like a trivial fix, building a model to prove soundness semantically (theorem 6) is surprisingly difficult. We do not have yet a soundness result via subject reduction (i.e. type preservation).

We consider a system with higher-order types (section II) with two basic sorts of objects:

- *propositions* which are either true or false since we are in classical logic,
- *individuals* which are untyped programs expressed in a variant of ML with sums (i.e. polymorphic variants) and products (i.e. records). The language is defined in section I.

Programs arise in propositions using two constructs:

- a restriction predicate  $A \upharpoonright t \equiv u$  relativizing proposition  $A$  with respect to the observational equivalence  $t \equiv u$ . The interpretations of  $A \upharpoonright t \equiv u$  and  $A$  coincide if the equivalence  $t \equiv u$  is true, otherwise it is empty. This can be seen as a conjunction.
- a belonging predicate  $t \in A$  expressing the fact that the term  $t$  has type  $A$ . In the semantics  $t \in A$  will be interpreted as a singleton type (up to program equivalence).

Our system is very similar to a Curry style HOL with ML programs as individuals. The construct  $t \in A$  allows to encode a limited form of dependent product.

$$\Pi_{a:A} B := \forall a (a \in A \Rightarrow B)$$

It does not allow the definition of a type which structure depends on a term (e.g. a function with a variable number of argument). Hence, our system is placed between HOL ( $F\omega$ ) and Coq (pure calculus of constructions  $CoC$ ) in a curry style copy of Barendregt's  $\lambda$ -cube (Figure ??).

Throughout this article we build a realizability model à la Krivine [16] which uses a call-by-value abstract machine. This difference induces an interpretation of formulas in three layers (values, stacks and terms like in the work of Guillaume Munch [17]) related via orthogonality (definition 9). The crucial property (theorem 5) for the soundness of semantical value restriction is that

$$\phi^{\perp\perp} \cap \Lambda_v = \phi$$

where  $\phi$  is a set of values (closed under  $(\equiv)$ ),  $\Lambda_v$  the set of all values and  $\phi^\perp$  (resp.  $\phi^{\perp\perp}$ ) the set of all stacks (resp. terms) that are *compatible* with every value in  $\phi$  (resp. stacks in  $\phi^\perp$ ). Note that this property is completely unrelated to lemma 9 in Munch's work [17].

To obtain such a property, we need to extend our programming language with a term  $\delta_{v,w}$  which reduction depends on the observational equivalence of the values  $v$  and  $w$ . The difficulty here is to define mutually inductively observational equivalence and reduction.

### E. Other works

We are not aware of any system similar to ours. The nearest system seems to be the work of Alexandre Miquel [18] where

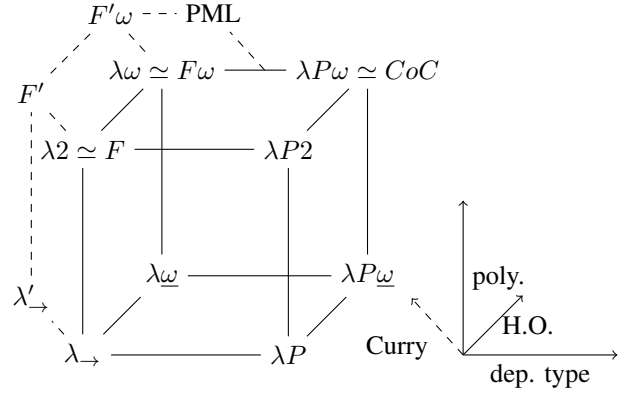


Fig. 3. PML in Barendregt's  $\lambda$ -cube ( $X'$  denotes curry style version of  $X$ )

propositions can be classical and curry style. However the rest of the language is still Church style and does not embed a full ML language.

There are quite a few works extending ML (DML, ATS, Idris) with dependent types, however all of them avoid the problem treated in this article because none of them have both classical logic and call-by-value evaluation.

## I. SYNTAX, REDUCTION AND EQUIVALENCE

The language is expressed in terms of a *Krivine Abstract Machine* [19], which is a stack-based machine. It is formed using four syntactic entities: values, terms, stacks and processes. Note that the syntactic distinction between terms and values is specific to the *call-by-value* presentation, they would be collapsed in *call-by-name*. We suppose given three distinct countable sets of variables:

- $\mathcal{V}_\lambda = \{x, y, z, \dots\}$  containing usual  $\lambda$ -variables,
- $\mathcal{V}_\mu = \{\alpha, \beta, \gamma, \dots\}$  containing  $\mu$ -variables (we will often call them stack variables) and
- $\mathcal{V}_t = \{a, b, c, \dots\}$  containing term variables which will be bound in formulas, but never in terms.

We also require a countable set  $\mathcal{L} = \{l, l_1, l_2, \dots\}$  of *labels* to name *record fields*<sup>4</sup>, and a countable set  $\mathcal{C} = \{C, C_1, C_2, \dots\}$  of *constructors*<sup>5</sup>.

**Definition 1.** *Values, terms, stacks and processes are mutually inductively defined by the following grammars. The names of the corresponding sets are displayed on the right.*

$$v, w ::= x \mid \lambda x. t \mid C[v] \mid \{l_1 = v_1; \dots l_n = v_n\} \quad (\Lambda_v)$$

$$t, u ::= a \mid v \mid t u \mid \mu \alpha. t \mid p \mid v.l \mid \text{case}_v B \mid \delta_{v,w} \quad (\Lambda)$$

$$\pi, \rho ::= \alpha \mid v.\pi \mid [t]\pi \quad (\Pi)$$

$$p, q ::= t * \pi \quad (\Lambda \times \Pi)$$

Terms and values form a variation of the  $\lambda\mu$ -calculus [14] enriched with ML-like constructs (i.e. records and variants<sup>6</sup>).

<sup>4</sup>A record is a tuple which fields are name using *labels*.

<sup>5</sup>A *Constructor* is to be thought as a kind of wrapper that can be recognised during a case analysis.

<sup>6</sup>In  $\text{case}_v B$ ,  $B$  is a list of patterns of the form  $C[x] \rightarrow t$ .

For a technical purposes that will become clear further, we extend the language with a special kind of term  $\delta_{v,w}$ . It will only be used to build the model and is not intended to be accessed directly by the user. One may note that values and processes are terms. In particular, a process of the form  $t * \alpha$  will corresponds exactly to a named term  $[\alpha]t$  in the most usual presentation of the  $\lambda\mu$ -calculus. Here we chose to embed processes into terms in order to obtain a more elegant reduction rule. It is also important to note that we enforce values in variant argument, record fields, projection and case analysis. This makes the calculus simpler<sup>7</sup>, and has no consequence for the programmer as we can define syntactic sugar to hide the restriction. We will follow the usual notational convention: application is left associative and both  $\lambda$ -abstraction and  $\mu$ -abstraction bind stronger than application<sup>8</sup>.

**Definition 2.** Given a value, term, stack or process  $\psi$  we denote  $FV_\lambda(\psi)$  (resp.  $FV_\mu(\psi)$ ,  $TV(\psi)$ ) the set of free  $\lambda$ -variables (resp. free  $\mu$ -variables, term variables) contained in  $\psi$ . We say that  $\psi$  is closed if it does not contain any free variable of any kind.

**Remark.** A stack, and hence a process, can never be closed as they always at least contain a stack variable.

#### A. Call-by-value reduction relation

Processes form the internal state of our abstract machine. They are to be thought of as a term put in some evaluation context represented using a stack. Intuitively, the stack  $\pi$  in the process  $t * \pi$  contains the functional arguments to be applied to  $t$ . Since we are in call-by-value the stack also handles the storing of functions while their arguments are being evaluated. This is why we need stack frames (i.e. stacks of the form  $[t]\pi$ ). The operational semantics of our language is given by a relation  $(\succ)$  over processes.

**Definition 3.** The relation  $(\succ) \subseteq (\Lambda \times \Pi)^2$  is defined as the smallest relation satisfying the rules exhibited bellow. We will denote  $(\succ^+)$  its transitive closure,  $(\succ^*)$  its reflexive-transitive closure and  $(\succ^k)$  its  $k$ -fold application.

$$\begin{array}{lcl}
t * u * \pi & \succ & u * [t]\pi \\
v * [t]\pi & \succ & t * v.\pi \\
\lambda x t * v.\pi & \succ & t[x:=v] * \pi \\
\mu \alpha t * \pi & \succ & t[\alpha:=\pi] * \pi \\
p * \pi & \succ & p \\
\{\dots l = v; \dots\}.l * \pi & \succ & v * \pi \\
case_{C[v]} \dots C[x] \rightarrow t \dots * \pi & \succ & t[x:=v] * \pi
\end{array}$$

The first three rules are those that handle  $\beta$ -reduction. When the abstract machine encounters an application, the function is stored in a stack-frame in order to evaluate its argument first. Once the argument have been completely computed a value faces the stack-frame containing the function. At this point the

function can be evaluated and the value is stored in the stack ready to be consumed by the function as soon as it evaluates to a  $\lambda$ -abstraction. A capture-avoiding substitution can then be performed to effectively apply the argument to the function. The next two rules handle the classical part of computation. When a  $\mu$ -abstraction is reached, the current stack (i.e. the current evaluation context) is captured and stored into the corresponding  $\mu$ -variable. Conversely, when a process is reached, the current stack is thrown away and evaluation resumes with the process. The last two rules perform projection and case analysis in the expected way. Note that for now, states of the form  $\delta_{v,w} * \pi$  are left unaffected by the reduction relation.

**Lemma 1.** The reduction relation  $(\succ)$  is compatible with substitutions of variables of any kind. More formally, if  $p$  and  $q$  are processes such that  $p \succ q$  then:

- for all  $x \in \mathcal{V}_\lambda$  and  $v \in \Lambda_v$ ,  $p[x := v] \succ q[x := v]$ ,
- for all  $\alpha \in \mathcal{V}_\mu$  and  $\pi \in \Pi$ ,  $p[\alpha := \pi] \succ q[\alpha := \pi]$ ,
- for all  $a \in \mathcal{V}_l$  and  $t \in \Lambda$ ,  $p[a := t] \succ q[a := t]$ .

Consequently, if  $\sigma$  is a substitution for variables of any kind and if  $p \succ q$  (resp.  $p \succ^* q$ ,  $p \succ^+ q$ ,  $p \succ^k q$ ) then  $p\sigma \succ q\sigma$  (resp.  $p\sigma \succ^* q\sigma$ ,  $p\sigma \succ^+ q\sigma$ ,  $p\sigma \succ^k q\sigma$ ).

*Proof:* Immediate case analysis on the reduction rules. ■

We are now going to give the vocabulary that will be used to describe some specific classes of processes. In particular we need to identify processes that are to be considered as the evidence of a successful computation, and those that are to be recognised as expressing failure.

**Definition 4.** A process  $p \in \Lambda \times \Pi$  is said to be:

- final if there is a value  $v \in \Lambda_v$  and a stack variable  $\alpha \in \mathcal{V}_\mu$  such that  $p = v * \alpha$ ,
- $\delta$ -like if there are values  $v, w \in \Lambda_v$  and a stack  $\pi \in \Pi$  such that  $p = \delta_{v,w} * \pi$ ,
- blocking if there is no  $q \in \Lambda \times \Pi$  such that  $p \succ q$ ,
- blocked if it is not final nor  $\delta$ -like, and if for every substitution  $\sigma$ ,  $p\sigma$  is blocking,
- looping if there is no blocking process  $q \in \Lambda \times \Pi$  such that  $p \succ^* q$ .

**Lemma 2.** Let  $p$  be a process and  $\sigma$  be a substitution for variables of any kind. If  $p$  is final (resp.  $\delta$ -like, blocked, looping) then  $p\sigma$  is also final (resp.  $\delta$ -like, blocked, looping).

*Proof:* Immediate by definition. ■

**Lemma 3.** A blocking process  $p \in \Lambda \times \Pi$  is either blocked, final,  $\delta$ -like, or of one of the forms:

$$x.l * \pi \quad x * v.\pi \quad case_x B * \pi \quad a * \pi$$

where  $x \in \mathcal{V}_\lambda$ ,  $l \in \mathcal{L}$ ,  $a \in \mathcal{V}_l$ ,  $v \in \Lambda_v$ ,  $\pi \in \Pi$  and  $B$  is the body of a case analysis.

*Proof:* Simple enumeration of all the possible forms of processes. This result was verified using the exhaustivity checker of OCaml's pattern matching. ■

<sup>7</sup>In particular only  $\beta$ -reduction has an effect on the content of the stack.

<sup>8</sup>This means that  $\lambda x t u v$  is to be read as  $\lambda x ((t u) v)$ .

**Remark.** A blocked state is of one of the form:

$$C[v].l * \pi \quad (\lambda x t).l * \pi \quad C[v] * w.\pi \quad \{\dots\} * v.\pi$$

$$\text{case}_{\lambda x t} B * \pi \quad \text{case}_{\{\dots\}} B * \pi$$

or of the form  $\text{case}_{C[v]} B * \pi$  where  $C$  is not matched in  $B$ , or of the form  $\{\dots\}.l * \pi$  where  $l$  is not a label of the record.

### B. Reduction of $\delta_{v,w}$ and equivalence

The idea now is to define a notion of observational equivalence over terms using a relation ( $\equiv$ ). We then extend the reduction relation with a rule reducing a state of the form  $\delta_{v,w} * \pi$  to  $v * \pi$  if  $v \not\equiv w$ , and leaving it blocked otherwise. Such reduction rule depends on what equivalence relation ( $\equiv$ ) over terms have been chosen. For instance equivalence and reduction might become interdependent, which may lead to a badly defined model.

**Definition 5.** Given a reduction relation  $R$ , we say that a process  $p \in \Lambda \times \Pi$  converges, and write  $p \Downarrow_R$ , if there is a final state  $q \in \Lambda \times \Pi$  such that  $pR^*q$  (where  $R^*$  is the reflexive-transitive closure of  $R$ ). If  $p$  does not converge we say that it diverges and write  $p \not\Downarrow_R$ . We will use the notations  $p \Downarrow_i$  and  $p \uparrow_i$  when working with indexed notation symbols like  $(\rightarrow_i)$ .

**Definition 6.** For every natural number  $i$  we define a reduction relation  $(\rightarrow_i)$  and an equivalence relation  $(\equiv_i)$  which negation will be denoted  $(\neq_i)$ .

$$(\rightarrow_i) = (\succ) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid \exists j < i, v \neq_j w\}$$

$$(\equiv_i) = \{(t, u) \mid \forall j \leq i, \forall \pi, \forall \sigma, t\sigma * \pi \Downarrow_j \Leftrightarrow u\sigma * \pi \Downarrow_j\}$$

In particular, one can easily see that  $(\rightarrow_0) = (\succ)$ . For every natural number  $i$ , the relation  $(\equiv_i)$  is indeed an equivalence relation as it can be seen as the intersection of trivial equivalence relations. Its negation can be expressed as follows.

$$(\neq_i) = \{(t, u), (u, t) \mid \exists j \leq i, \exists \pi, \exists \sigma, t\sigma * \pi \Downarrow_j \wedge u\sigma * \pi \uparrow_j\}$$

**Definition 7.** We define a reduction relation  $(\rightarrow)$  and an equivalence relation  $(\equiv)$  which negation will be denoted  $(\neq)$ .

$$(\rightarrow) = \bigcup_{i \in \mathbb{N}} (\rightarrow_i) \quad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i)$$

These relations can be expressed directly (i.e. without the need of a union or an intersection) in the following way.

$$(\equiv) = \{(t, u) \mid \forall i, \forall \pi, \forall \sigma, t\sigma * \pi \Downarrow_i \Leftrightarrow u\sigma * \pi \Downarrow_i\}$$

$$(\neq) = \{(t, u), (u, t) \mid \exists i, \exists \pi, \exists \sigma, t\sigma * \pi \Downarrow_i \wedge u\sigma * \pi \uparrow_i\}$$

$$(\rightarrow) = (\succ) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid v \neq w\}$$

**Remark.** The relations  $(\rightarrow)$  and  $(\equiv)$  are fixpoints of  $(\rightarrow_i)_{i \in \mathbb{N}}$  and  $(\equiv_i)_{i \in \mathbb{N}}$ . In particular for every natural number  $i$  we have  $(\rightarrow_i) \subseteq (\rightarrow_{i+1})$  and  $(\equiv_{i+1}) \subseteq (\equiv_i)$ .

**Theorem 1.** Let  $t$  and  $u$  be terms. If  $t \equiv u$  then for every stack  $\pi \in \Pi$  and substitution  $\sigma$  we have  $t\sigma * \pi \Downarrow \Leftrightarrow u\sigma * \pi \Downarrow$ .

*Proof:* We suppose that  $t \equiv u$  and we take  $\pi_0 \in \Pi$  and a substitution  $\sigma_0$ . By symmetry we can assume that  $t\sigma_0 * \pi_0 \Downarrow$  and show that  $u\sigma_0 * \pi_0 \Downarrow$ . By definition there is  $i_0 \in \mathbb{N}$  such that  $t\sigma_0 * \pi_0 \Downarrow_{i_0}$ . Since  $t \equiv u$  we know that for every  $i \in \mathbb{N}$ ,  $\pi \in \Pi$  and substitution  $\sigma$  we have  $t\sigma * \pi \Downarrow_i \Leftrightarrow u\sigma * \pi \Downarrow_i$ . This is true in particular for  $i = i_0$ ,  $\pi = \pi_0$  and  $\sigma = \sigma_0$ . We hence obtain  $u\sigma_0 * \pi_0 \Downarrow_{i_0}$  which give us  $u\sigma_0 * \pi_0 \Downarrow$ . ■

**Remark.** The converse implication is not true in general. If  $p$  is a process such that  $p \Downarrow$ , we cannot obtain that  $p \Downarrow_i$  for a given natural number  $i$ .

**Corollary 1.** Let  $t$  and  $u$  be terms and  $\pi$  be a stack. If  $t \equiv u$  and  $t * \pi \Downarrow$  then  $u * \pi \Downarrow$ .

*Proof:* Direct consequence of theorem 1 using  $\pi$  and an empty substitution. ■

### C. Extensionality of the language

In order to be able to work with the equivalence relation ( $\equiv$ ), we need to check that it is extensional. In other words, we need to be able to replace equals by equals at any place in a term without changing their observed behaviour. This property is summarized in the following two theorems.

**Theorem 2.** Let  $v$  and  $w$  be values,  $E$  be a term and  $x$  be a  $\lambda$ -variable. If  $v \equiv w$  then  $E[x := v] \equiv E[x := w]$ .

*Proof:* We are going to prove the contrapositive so we suppose  $E[x := v] \not\equiv E[x := w]$  and show  $v \neq w$ . By definition there is  $i \in \mathbb{N}$ ,  $\pi \in \Pi$  and a substitution  $\sigma$  such that  $(E[x := v])\sigma * \pi \Downarrow_i$  and  $(E[x := w])\sigma * \pi \uparrow_i$  (up to symmetry). Since we can rename  $x$  in such a way that it does not appear in  $\text{dom}(\sigma)$ , we can suppose  $E\sigma[x := v\sigma] * \pi \Downarrow_i$  and  $E\sigma[x := w\sigma] * \pi \uparrow_i$ . In order to show  $v \neq w$  we need to find  $i_0 \in \mathbb{N}$ ,  $\pi_0 \in \Pi$  and a substitution  $\sigma_0$  such that  $v\sigma_0 * \pi_0 \Downarrow_{i_0}$  and  $w\sigma_0 * \pi_0 \uparrow_{i_0}$  (up to symmetry). We take  $i_0 = i$ ,  $\pi_0 = [\lambda x E\sigma]\pi$  and  $\sigma_0 = \sigma$ . These values are suitable since by definition  $v\sigma_0 * \pi_0 \Downarrow \rightarrow E\sigma[x := v\sigma] * \pi_{i_0} \Downarrow_{i_0}$  and  $w\sigma_0 * \pi_0 \Downarrow \rightarrow E\sigma[x := w\sigma] * \pi_{i_0} \uparrow_{i_0}$ . ■

**Lemma 4.** Let  $s$  be a process,  $t$  be a term,  $a$  be a term variable and  $k$  be a natural number. If  $s[a := t] \Downarrow_k$  then there is a blocking state  $p$  such that  $s \succ^* p$  and either

- $p = v * \alpha$  for some value  $v$  and a stack variable  $\alpha$ ,
- $p = a * \pi$  for some stack  $\pi$ ,
- $k > 0$  and  $p = \delta(v, w) * \pi$  for some values  $v$  and  $w$  and stack  $\pi$ , and  $v[a := t] \not\equiv_j w[a := t]$  for some  $j < k$ .

*Proof:* Let  $\sigma$  be the substitution  $[a := t]$ . If  $s$  is non-terminating, lemma 2 tells us that  $s\sigma$  is also non-terminating, which contradicts  $s\sigma \Downarrow_k$ . Consequently, there is a blocking process  $p$  such that  $s \succ^* p$  since  $(\succ) \subseteq (\rightarrow_k)$ . Using lemma 1 we get  $s\sigma \succ^* p\sigma$  from which we obtain  $p\sigma \Downarrow_k$ . The process  $p$  cannot be blocked, otherwise  $p\sigma$  would also be blocked by lemma 2, which would contradict  $p\sigma \Downarrow_k$ . Let us now suppose that  $p = \delta_{v,w} * \pi$  for some values  $v$  and  $w$  and some stack  $\pi$ . Since  $\delta_{v,w} * \pi \Downarrow_k$  there must be  $i < k$  such that  $v\sigma \not\equiv_j w\sigma$ , otherwise this would contradict  $\delta_{v,w} * \pi \Downarrow_k$ . In this case we

necessarily have  $k > 0$ , otherwise there would be no possible candidate for  $i$ . According to lemma 3 we need to rule out four more forms of terms:  $x.l * \pi$ ,  $x * v.\pi$ ,  $\text{case}_x B * \pi$  and  $b * \pi$  in the case where  $b \neq a$ . If  $p$  was of one of these forms the substitution  $\sigma$  would not be able to unblock the reduction of  $p$ , which would contradict again  $p\sigma \Downarrow_k$ . ■

**Lemma 5.** *Let  $t_1, t_2$  and  $E$  be terms and  $a$  be a term variable. For every  $k \in \mathbb{N}$ , if  $t_1 \equiv_k t_2$  then  $E[a := t_1] \equiv_k E[a := t_2]$ .*

*Proof:* Let us take  $k \in \mathbb{N}$ , suppose that  $t_1 \equiv_k t_2$  and show that  $E[a := t_1] \equiv_k E[a := t_2]$ . By symmetry we can assume that we have  $i \leq k$ ,  $\pi \in \Pi$  and a substitution  $\sigma$  such that  $(E[a := t_1])\sigma * \pi \Downarrow_i$  and show that  $(E[a := t_2])\sigma * \pi \Downarrow_i$ . As we are free to rename  $a$ , we can suppose that it does not appear in  $\text{dom}(\sigma)$ ,  $TV(\pi)$ ,  $TV(t_1)$  or  $TV(t_2)$ . In order to lighten the notations we define  $E' = E\sigma$ ,  $\sigma_1 = [a := t_1]\sigma$  and  $\sigma_2 = [a := t_2]\sigma$ . We are hence assuming  $E'\sigma_1 * \pi \Downarrow_i$  and trying to show  $E'\sigma_2 * \pi \Downarrow_i$ .

We will now build a sequence  $(E_i, \pi_i, l_i)_{i \in I}$  in such a way that  $E'\sigma_1 * \pi \rightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1$  in  $l_i$  steps for every  $i \in I$ . Furthermore, we require that  $(l_i)_{i \in I}$  is increasing and that it has a strictly increasing subsequence. Under this condition our sequence will necessarily be finite. If it was infinite the number of reduction steps that could be taken from the state  $E'\sigma_1 * \pi$  would not be bounded, which would contradict  $E'\sigma_1 * \pi \Downarrow_i$ . We now denote our finite sequence  $(E_i, \pi_i, l_i)_{i \leq n}$  with  $n \in \mathbb{N}$ . In order to show that  $(l_i)_{i \leq n}$  has a strictly increasing subsequence, we will ensure that it does not have three equal consecutive values. More formally, we will require that if  $0 < i < n$  and  $l_{i-1} = l_i$  then  $l_{i+1} > l_i$ .

To define  $(E_0, \pi_0, l_0)$  we consider the reduction of  $E' * \pi$ . Since we know that  $(E' * \pi)\sigma_1 = E'\sigma_1 * \pi \Downarrow_i$  we use lemma 4 to obtain a blocking state  $p$  such that  $E' * \pi \succ^j p$ . We can now take  $E_0 * \pi_0 = p$  and  $l_0 = j$ . By lemma 1 we have  $(E' * \pi)\sigma_1 \succ^j E_0\sigma_1 * \pi_0\sigma_1$  from which we can deduce that  $(E' * \pi)\sigma_1 \rightarrow_k^* E_0\sigma_1 * \pi_0\sigma_1$  in  $l_0 = j$  steps.

To define  $(E_{i+1}, \pi_{i+1}, l_{i+1})$  we consider the reduction of the process  $E_i\sigma_1 * \pi_i$ . By construction we know that  $E'\sigma_1 * \pi \rightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1 = (E_i\sigma_1 * \pi_i)\sigma_1$  in  $l_i$  steps. Using lemma 4 we know that  $E_i * \pi_i$  might be of three shapes.

- If  $E_i * \pi_i = v * \alpha$  for some value  $v$  and stack variable  $\alpha$  then the end of the sequence was reached with  $n = i$ .
- If  $E_i = a$  then we consider the reduction of  $E_i\sigma_1 * \pi_i$ . Since  $(E_i\sigma_1 * \pi_i)\sigma_1 \Downarrow_k$  we know from lemma 4 that there is a blocking process  $p$  such that  $E_i\sigma_1 * \pi_i \succ^j p$ . Using lemma 1 we obtain that  $E_i\sigma_1 * \pi_i\sigma_1 \succ^j p$  from which we can deduce that  $E_i\sigma_1 * \pi_i\sigma_1 \rightarrow_k^* p\sigma_1$  in  $j$  steps. We then take  $E_{i+1} * \pi_{i+1} = p$  and  $l_{i+1} = l_i + j$ .

Is it possible to have  $j = 0$ ? This can only happen when  $E_i\sigma_1 * \pi_i$  is of one of the three forms of lemma 4. It cannot be of the form  $a * \pi$  as we assumed that  $a$  does not appear in  $t_1$  or  $\sigma$ . If it is of the form  $v * \alpha$ , then we reached the end of the sequence with  $i + 1 = n$  so there is no trouble. The process  $E_i\sigma_1 * \pi_i$  may be of the form  $\delta(v, w) * \pi$ , but we will have  $l_{i+2} > l_{i+1}$ .

- If  $E_i = \delta(v, w)$  for some values  $v$  and  $w$  we know that there is  $m < k$  such that  $v\sigma_1 \not\equiv_m w\sigma_1$ . Hence  $E_i\sigma_1 * \pi_i = \delta(v\sigma_1, w\sigma_1) * \pi_i \rightarrow_k v\sigma_1 * \pi_i$  by definition. Moreover  $E_i\sigma_1 * \pi_i\sigma_1 \rightarrow_k v\sigma_1 * \pi_i\sigma_1$  by lemma 1. Since  $E'\sigma_1 * \pi \rightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1$  in  $l_i$  steps we obtain that  $E'\sigma_1 * \pi \rightarrow_k^* v\sigma_1 * \pi_i\sigma_1$  in  $l_i + 1$  steps. This also gives us  $(v\sigma_1 * \pi_i)\sigma_1 = v\sigma_1 * \pi_i\sigma_1 \Downarrow_k$ .

We now consider the reduction of the process  $v\sigma_1 * \pi_i$ . By lemma 4 there is a blocking process  $p$  such that  $v\sigma_1 * \pi_i \succ^j p$ . Using lemma 1 we obtain  $v\sigma_1 * \pi_i\sigma_1 \succ^j p\sigma_1$  from which we deduce that  $v\sigma_1 * \pi_i\sigma_1 \rightarrow_k^* p\sigma_1$  in  $j$  steps. We then take  $E_{i+1} * \pi_{i+1} = p$  and  $l_{i+1} = l_i + j + 1$ . Note that in this case we have  $l_{i+1} > l_i$ .

Intuitively  $(E_i, \pi_i, l_i)_{i \leq n}$  mimics the reduction of  $E'\sigma_1 * \pi$  while making explicit every substitution of  $a$  and every reduction of a  $\delta$ -like state.

To end the proof we show that for every  $i \leq n$  we have  $E_i\sigma_2 * \pi_i\sigma_2 \Downarrow_k$ . For  $i = 0$  this will give us  $E'\sigma_2 * \pi \Downarrow_k$  which is the expected result. Since  $E_n * \pi_n = v * \alpha$  we have  $E_n\sigma_2 * \pi_n\sigma_2 = v\sigma_2$  from which we trivially obtain  $E_n\sigma_2 * \pi_n\sigma_2 \Downarrow_k$ . We now suppose that  $E_{i+1}\sigma_2 * \pi_{i+1}\sigma_2 \Downarrow_k$  for  $0 \leq i < n$  and show that  $E_i\sigma_2 * \pi_i\sigma_2 \Downarrow_k$ . By construction  $E_i * \pi_i$  can be of two shapes<sup>9</sup>:

- If  $E_i = a$  then  $t_1\sigma * \pi_i \rightarrow_k^* E_{i+1}\pi_{i+1}$ . Using lemma 1 we obtain  $t_1\sigma * \pi_i\sigma_2 \rightarrow_k E_{i+1}\sigma_2 * \pi_{i+1}\sigma_2$  from which we deduce  $t_1\sigma * \pi_i\sigma_2 \Downarrow_k$  by induction hypothesis. Since  $t_1 \equiv_k t_2$  we obtain  $t_2\sigma * \pi_i\sigma_2 = (E_i * \pi_i)\sigma_2 \Downarrow_k$ .
- If  $E_i = \delta(v, w)$  then  $v * \pi_i \rightarrow_k E_{i+1} * \pi_{i+1}$  and hence  $v\sigma_2 * \pi_i\sigma_2 \rightarrow_k E_{i+1}\sigma_2 * \pi_{i+1}\sigma_2$  by lemma 1. Using the induction hypothesis we obtain  $v\sigma_2 * \pi_i\sigma_2 \Downarrow_k$ . It remains to show that  $\delta(v\sigma_2, w\sigma_2) * \pi_i\sigma_2 \rightarrow_k^* v\sigma_2 * \pi_i\sigma_2$ . We need to find  $j < k$  such that  $v\sigma_2 \not\equiv_j w\sigma_2$ . By construction there is  $m < k$  such that  $v\sigma_1 \not\equiv_m w\sigma_1$ . We are going to show that  $v\sigma_2 \not\equiv_m w\sigma_2$ . By using the global induction hypothesis twice we obtain  $v\sigma_1 \equiv_m v\sigma_2$  and  $w\sigma_1 \equiv_m w\sigma_2$ . Now if  $v\sigma_2 \equiv_m w\sigma_2$  then  $v\sigma_1 \equiv_m w\sigma_1$  contradicts  $v\sigma_1 \not\equiv_m w\sigma_1$ . Hence we must have  $v\sigma_2 \not\equiv_m w\sigma_2$ . ■

**Theorem 3.** *Let  $t_1, t_2$  and  $E$  be three terms and  $a$  be a term variable. If  $t_1 \equiv t_2$  then  $E[a := t_1] \equiv E[a := t_2]$ .*

*Proof:* We suppose that  $t_1 \equiv t_2$  which means that  $t_1 \equiv_i t_2$  for every  $i \in \mathbb{N}$ . We need to show that  $E[a := t_1] \equiv E[a := t_2]$  so we take  $i_0 \in \mathbb{N}$  and show  $E[a := t_1] \equiv_{i_0} E[a := t_2]$ . By hypothesis we have  $t_1 \equiv_{i_0} t_2$  and hence we can conclude using lemma 5. ■

## II. FORMULAS AND SEMANTICS

The syntax presented in the previous section is part of a realizability machinery that will be built upon here. We aim at obtaining a semantical interpretation of the higher-order type system that will be defined shortly. Our abstract machine slightly differs from the mainstream presentation of *Krivine's*

<sup>9</sup>Only  $E_n * \pi_n$  can be of the form  $v * \alpha$ .



classical realizability which is usually call-by-name. Although call-by-value presentations have rarely been published, such developpments are well-known among a restricted community of experts. We do not claim any credit on this point. The addition of the  $\delta$  instruction and the related modifications are however due to the author.

#### A. Pole and orthogonality

As always in classical realizability, the model is parametrized by a pole, which will serve as an exchange point between the world of programs and the world of execution contexts.

**Definition 8.** A pole is a set of processes  $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$  which is closed under backward reduction<sup>10</sup>. More formally, if  $q \in \perp\!\!\!\perp$  and  $p \rightarrow q$  then  $p \in \perp\!\!\!\perp$ .

Here, for the sake of simplicity and brevity, we are only going to use the pole

$$\perp\!\!\!\perp = \{p \in \Lambda \times \Pi \mid p \Downarrow\}$$

which is clearly closed under backward reduction. Note that this particular pole is also closed under the reduction relation ( $\rightarrow$ ), even though this is not a general property. In particular  $\perp\!\!\!\perp$  contains every final processes.

The notion of *orthogonality* is central in Krivine's Realizability. In this framework a type is interpreted (or realized) by programs computing corresponding values. This interpretation is spread in a three-layered construction, even though it is fully determined by the first layer (and the choice of the pole). The first layer consists in a set of values that we will call the *raw semantics*. It gathers all the syntactic values that should be considered as having the corresponding type. As an example, if we were to consider the type of natural numbers, its raw semantics would be the set  $\{\bar{n} \mid n \in \mathbb{N}\}$  where  $\bar{n}$  is some encoding of  $n$ . The second layer, called *falsity value* is a set containing every stack that is a candidate for building a valid process using any value from the raw semantics. The notion of validity depends on the choice of the pole. Here for instance, a valid process is a normalizing one (i.e. one that reduces to a final state). The third layer, called *truth value* is a set of terms that is built by iterating the process once more. The formalism for the two levels of orthogonality are given in the following definition.

**Definition 9.** For every set  $\phi \subseteq \Lambda_v$  we define a set  $\phi^\perp \subseteq \Pi$  and a set  $\phi^{\perp\perp} \subseteq \Lambda$  as follows.

$$\begin{aligned}\phi^\perp &= \{\pi \in \Pi \mid \forall v \in \phi, v * \pi \in \perp\!\!\!\perp\} \\ \phi^{\perp\perp} &= \{t \in \Lambda \mid \forall \pi \in \phi^\perp, t * \pi \in \perp\!\!\!\perp\}\end{aligned}$$

It is important to check that the pole we chose does not yield a degenerate model. In particular we check that no term is able to face every stacks.

<sup>10</sup>It is said to be *saturated* in Krivine's original work.

**Theorem 4.** The pole  $\perp\!\!\!\perp$  is consistent, which means that for every closed term  $t$  there is a stack  $\pi$  such that  $t * \pi \notin \perp\!\!\!\perp$ .

*Proof:* Let  $t$  be a closed term and  $\alpha$  be a stack constant. If we do not have  $t * \alpha \Downarrow$  then we can directly take  $\pi = \alpha$ . Otherwise we know that  $t * \alpha \rightarrow^* v * \alpha$  for some value  $v$ . Since  $t$  is closed  $\alpha$  is the only available stack variable. We now show that  $\pi = [\lambda x \{\}\{\}]\beta$  is suitable. We denote  $\sigma$  the substitution  $[\alpha := \pi]$ . Using a trivial extension of lemma 1 to the ( $\rightarrow$ ) relation we obtain  $t * \pi = (t * \alpha)\sigma \rightarrow^* (v * \alpha)\sigma = v\sigma * \pi$ . We hence have  $t * \pi \rightarrow^* v\sigma * [\lambda x \{\}\{\}] \rightarrow^2 \{\} * \{\}.\beta \notin \perp\!\!\!\perp$ . ■

#### B. Properties of orthogonality

The construction involving  $\delta$  and ( $\equiv$ ) in the previous section is now going to gain meaning. The following theorem, which is our central result, does not hold in every realizability model. Obtaining a proof required us to internalize observational equivalence as a non-computable operation.

**Theorem 5.** If  $\Phi \subseteq \Lambda_v$  is a set of values closed under ( $\equiv$ ), then  $\Phi^{\perp\perp} \cap \Lambda_v = \Phi$ .

*Proof:* The direction  $\Phi \subseteq \Phi^{\perp\perp} \cap \Lambda_v$  is left to the reader (straight forward by definition of  $\Phi^{\perp\perp}$ ). We are going to show that  $\Phi^{\perp\perp} \cap \Lambda_v \subseteq \Phi$ , which amounts to showing that for every value  $v \in \Phi^{\perp\perp}$  we have  $v \in \Phi$ . We are going to show the contrapositive, so let us assume  $v \notin \Phi$  and show  $v \notin \Phi^{\perp\perp}$ . We need to find a stack  $\pi_0$  such that  $v * \pi_0 \notin \perp\!\!\!\perp$  and for every value  $w \in \Phi$ ,  $w * \pi_0 \in \perp\!\!\!\perp$ . We take  $\pi_0 = [\lambda x \delta(x, v)]\alpha$  and show that it is suitable. By definition of the reduction relation  $v * \pi_0$  reduces to  $\delta(v, v) * \alpha$  which is blocked ( $v \equiv v$  by reflexivity). Let us now take  $w \in \Phi$ . Again by definition,  $w * \pi_0$  reduces to  $\delta(w, v) * \alpha$ , but this time we have  $w \not\equiv v$  since  $\Phi$  was supposed to be closed under ( $\equiv$ ) and  $v \notin \Phi$ . Hence  $w * \pi_0$  reduces to  $w * \alpha \in \perp\!\!\!\perp$ . ■

We now give some properties of orthogonality that are true in general for every classical realizability model. They will be useful when proving the soundness of our type system.

**Lemma 6.** If  $\phi \subseteq \Lambda_v$  is a set of values, then  $\phi \subseteq \phi^{\perp\perp}$ .

*Proof:* Immediate following the definition of  $\phi^{\perp\perp}$ . ■

**Lemma 7.** Let  $\phi \subseteq \Lambda_v$  and  $\psi \subseteq \Lambda_v$  be two sets of values. If  $\phi \subseteq \psi$  then  $\psi^\perp \subseteq \phi^\perp$ .

*Proof:* Immediate by definition of orthogonality. ■

**Lemma 8.** Let  $S \subseteq \Pi$  and  $T \subseteq \Pi$  be two sets of stacks. If  $S \subseteq T$  then  $T^\perp \subseteq S^\perp$ .

*Proof:* Immediate by definition of orthogonality. ■

**Lemma 9.** Let  $\phi \subseteq \Lambda_v$  and  $\psi \subseteq \Lambda_v$  be two sets of values. If  $\phi \subseteq \psi$  then  $\phi^{\perp\perp} \subseteq \psi^{\perp\perp}$ .

*Proof:* Direct consequence of lemma 7 and lemma 8. ■

#### C. Formulas and their semantics

In this paper we limit ourselves to second-order logic, even though the system can easily be extended to higher-

order. For every natural number  $n$  we require a countable set  $\mathcal{V}_n = \{X_n, Y_n, Z_n, \dots\}$  of  $n$ -ary predicate variables.

**Definition 10.** *The syntax of formulas is given by the following grammar, where  $I$  denotes any finite subset of  $\mathbb{N}$ .*

$$\begin{aligned} A, B ::= & X_n(t_1, \dots, t_n) \mid A \Rightarrow B \mid \forall a A \mid \exists a A \\ & \mid \forall X_n A \mid \exists X_n A \mid \{l_i : A_i\}_{i \in I} \\ & \mid [C_i : A_i]_{i \in I} \mid t \in A \mid A \upharpoonright t \equiv u \end{aligned}$$

Terms appear in several places in formulas, in particular, they form the individuals of the logic. They can be quantified over and are used as arguments for predicate variables. Apart from the ML-like formers for sums and products (i.e. records and variants) we add a belonging predicate and a restriction operation. The belonging predicate  $t \in A$  is used to express the fact that the term  $t$  has type  $A$ . It was suggested by Alexandre Miquel during a visit to Montevideo as a way to encode the dependent product type using universal quantification and the arrow type. In this sense, it is inspired and related to Krivine's relativization of quantifiers.

$$\Pi_{x:A} B := \forall a(a \in A \Rightarrow B)$$

The restriction operator can be thought of as a kind of conjunction with no algorithmic content. The formula  $A \upharpoonright t \equiv u$  is to be interpreted in the same way as  $A$  if the equivalence  $t \equiv u$  holds, and as  $\perp$  otherwise<sup>11</sup>. In order to handle free term variables and free predicate variables in formulas we introduce valuations.

**Definition 11.** *A valuation is a finite map  $\rho$  ranging over variables (of both kinds) such that:*

- if  $a \in \text{dom}(\rho)$  then  $\rho(a) \in \Lambda$ ,
- if  $X_n \in \text{dom}(\rho)$  then  $\rho(X_n) \in \Lambda^n \rightarrow \mathcal{P}(\Lambda_v / \equiv)$ .

Given a formula  $A$  we denote  $FV(A)$  the set of its free variables. Given a valuation  $\rho$  such that  $\text{dom}(\rho) \subset FV(A)$  we write  $A[\rho]$  the closed formula built by applying  $\rho$  to  $A$ . In the semantics we interpret closed formulas as sets of values closed under the equivalence relation ( $\equiv$ ).

**Definition 12.** *Given a formula  $A$  and a valuation  $\rho$  such that  $A[\rho]$  is closed, we define the raw semantics  $\llbracket A \rrbracket_\rho \subseteq \Lambda_v / \equiv$  of  $A$  under the valuation  $\rho$  as follows.*

$$\begin{aligned} \llbracket X_n(t_1, \dots, t_n) \rrbracket_\rho &= \rho(X_n)(t_1\rho, \dots, t_n\rho) \\ \llbracket A \Rightarrow B \rrbracket_\rho &= \{\lambda x t \mid \forall v \in \llbracket A \rrbracket_\rho, t[x := v] \in \llbracket B \rrbracket_\rho^{\perp\perp}\} \\ \llbracket \forall a A \rrbracket_\rho &= \bigcap_{t \in \Lambda} \llbracket A[a := t] \rrbracket_\rho \\ \llbracket \exists a A \rrbracket_\rho &= \bigcup_{t \in \Lambda} \llbracket A[a := t] \rrbracket_\rho \\ \llbracket \forall X_n A \rrbracket_\rho &= \bigcap_{P \in \Lambda^n \rightarrow \mathcal{P}(\Lambda_v / \equiv)} \llbracket A[X_n := P] \rrbracket_\rho \\ \llbracket \exists X_n A \rrbracket_\rho &= \bigcup_{P \in \Lambda^n \rightarrow \mathcal{P}(\Lambda_v / \equiv)} \llbracket A[X_n := P] \rrbracket_\rho \\ \llbracket \{l_i : A_i\}_{i \in I} \rrbracket_\rho &= \bigcap_{i \in I} \{v \in \Lambda_v \mid v.l_i \in \llbracket A_i \rrbracket_\rho^{\perp\perp}\} \\ \llbracket [C_i : A_i]_{i \in I} \rrbracket_\rho &= \bigcup_{i \in I} \{C_i[v] \mid v \in \llbracket A_i \rrbracket_\rho\} \end{aligned}$$

<sup>11</sup>Here,  $\perp$  and  $\top$  can be obtain using the usual second-order encoding:  $\perp = \forall X_0 X_0$  and  $\top = \exists X_0 X_0$ .

$$\begin{aligned} \llbracket t \in A \rrbracket_\rho &= \{v \in \llbracket A \rrbracket_\rho \mid t\rho \equiv v\} \\ \llbracket A \upharpoonright t \equiv u \rrbracket_\rho &= \llbracket A \rrbracket_\rho \text{ if } t\rho \equiv u\rho \\ &\quad \emptyset \text{ otherwise} \end{aligned}$$

In the model, programs will realize closed formulas in two different ways according to their syntactic class. The interpretation of values will be given in terms of raw semantics, and the interpretation of terms in general will be given in terms of truth values.

**Definition 13.** *Let  $A$  be a formula and  $\rho$  a valuation such that  $A[\rho]$  is closed. We say that:*

- $v \in \Lambda_v$  realizes  $A[\rho]$  if  $v \in \llbracket A \rrbracket_\rho$ ,
- $t \in \Lambda$  realizes  $A[\rho]$  if  $t \in \llbracket A \rrbracket_\rho^{\perp\perp}$ .

#### D. Contexts and typing rules

Before giving the typing rule of our system we need to define contexts and judgements. As explained in the introduction, several typing rules require a value restriction in our context. This is reflected in typing rule by the presence of two forms of judgements.

**Definition 14.** *A context is a set of type declarations for  $\lambda$ -variables and  $\mu$ -variables. In our case it also contains term equalities and inequalities. Contexts are built using the following grammar.*

$$\Gamma, \Delta ::= \bullet \mid \Gamma, x : A \mid \Gamma, \alpha : \neg A \mid \Gamma, t \equiv u \mid \Gamma, t \not\equiv u$$

**Definition 15.** *There are two forms of typing judgements:*

- $\Gamma \vdash_v v : A$  meaning that the value  $v$  has type  $A$  in the context  $\Gamma$  and
- $\Gamma \vdash t : A$  meaning that the term  $t$  has type  $A$  in the context  $\Gamma$ .

The typing rules of the system are given in Fig. 4. Note that we here limit ourselves to a second-order type system, but it can be easily extended to higher-order. Although most of the rules are fairly usual, our type system differs in several ways. For instance the last height rules are related the the extensionality of the calculus. One can note the value restriction in several places: both universal quantification introduction rules and the introduction of the belonging predicate. If fact, some value restriction is also hidden in the rules of the elimination of the existential quantifiers and the elimination rule for the restriction connective. These rules are presented under their leftern-side variation, and only values can appear on the left of the sequent. It is not surprising that elimination of an existential quantifier requires value restriction as it is the dual of the introduction rule of a universal quantifier.

Here the most important and interesting difference with existing type systems is the presence of  $v \uparrow$  and  $v \downarrow$ . These two rules allow one to go from one kind of sequent to the other when working on values. Going from  $\Gamma \vdash_v v : A$  to  $\Gamma \vdash v : A$  is straight-forward. Going the other direction is the main motivation for our model as this will allow use to lift the value restriction expressed in the syntax to a restriction expressed in terms of equivalence. As an example, the rule

$$\begin{array}{c}
\frac{}{\Gamma, x : A \vdash_v x : A} Ax \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash_v \lambda x t : A \Rightarrow B} \Rightarrow_i \quad \frac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B} \Rightarrow_e \\
\\
\frac{\Gamma, \alpha : \neg A \vdash t : A}{\Gamma \vdash \mu \alpha t : A} \mu \quad \frac{\Gamma, \alpha : \neg A \vdash t : A}{\Gamma, \alpha : \neg A \vdash t * \alpha : B} * \\
\\
\frac{\Gamma \vdash_v v : A \quad a \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall a A} \forall_i \quad \frac{\Gamma \vdash t : \forall a A}{\Gamma \vdash t : A[a := u]} \forall_e \\
\\
\frac{\Gamma \vdash t : A[a := u]}{\Gamma \vdash t : \exists a A} \exists_i \quad \frac{\Gamma, y : A \vdash t : B}{\Gamma, y : \exists a A \vdash t : B} \exists_e \\
\\
\frac{\Gamma \vdash_v v : A \quad X_n \notin FV(\Gamma)}{\Gamma \vdash_v v : \forall X_n A} \forall_I \quad \frac{\Gamma \vdash t : \forall X_n A}{\Gamma \vdash t : A[X_n := P]} \forall_E \\
\\
\frac{\Gamma \vdash t : A[X_n := P]}{\Gamma \vdash t : \exists X_n A} \exists_I \quad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : \exists X_n A \vdash t : B} \exists_E \\
\\
\frac{\{\Gamma \vdash_v v_i : A_i\}_{i=1}^n}{\Gamma \vdash_v \{l_i = v_i\}_{i=1}^n : \{l_i : A_i\}_{i=1}^n} \times_i \quad \frac{\Gamma \vdash_v v : \{l_i : A_i\}_{i=1}^n}{\Gamma \vdash v.l_i : A_i} \times_e \\
\\
\frac{\Gamma \vdash_v v : A_i}{\Gamma \vdash_v C_i[v] : [C_i : A_i]_{i=1}^n} +_i \quad \frac{\Gamma \vdash_v v : [C_i : A_i]_{i=1}^n \quad \{\Gamma, x : A_i, C_i[x] \equiv v \vdash t_i : B\}_{i=1}^n}{\Gamma \vdash case_v [C_i[x] \rightarrow t_i]_{i=1}^n : B} +_e \\
\\
\frac{\Gamma \vdash_v v : A}{\Gamma \vdash_v v : v \in A} \in_i \quad \frac{\Gamma \vdash t : u \in A}{\Gamma \vdash t : A} \in_e \quad \frac{\Gamma, u_1 \equiv u_2 \vdash t : A}{\Gamma, u_1 \equiv u_2 \vdash t : A \upharpoonright u_1 \equiv u_2} \upharpoonright_i \\
\\
\frac{\Gamma, x : A, u_1 \equiv u_2 \vdash t : B}{\Gamma, x : A \upharpoonright u_1 \equiv u_2 \vdash t : B} \upharpoonright_e \quad \frac{\Gamma \vdash_v v : A}{\Gamma \vdash v : A} v\uparrow \quad \frac{\Gamma \vdash v : A}{\Gamma \vdash_v v : A} v\downarrow \\
\\
\frac{\Gamma, w_1 \equiv w_2 \vdash_v v[x := w_1] : A}{\Gamma, w_1 \equiv w_2 \vdash_v v[x := w_2] : A} \equiv_{v,v,l} \quad \frac{\Gamma, t_1 \equiv t_2 \vdash_v v[a := t_1] : A}{\Gamma, t_1 \equiv t_2 \vdash_v v[a := t_2] : A} \equiv_{v,t,l} \\
\\
\frac{\Gamma, w_1 \equiv w_2 \vdash_v v : A[x := w_1]}{\Gamma, w_1 \equiv w_2 \vdash_v v : A[x := w_2]} \equiv_{v,v,r} \quad \frac{\Gamma, t_1 \equiv t_2 \vdash_v v : A[a := t_1]}{\Gamma, t_1 \equiv t_2 \vdash_v v : A[a := t_2]} \equiv_{v,t,r} \\
\\
\frac{\Gamma, w_1 \equiv w_2 \vdash t[x := w_1] : A}{\Gamma, w_1 \equiv w_2 \vdash t[x := w_2] : A} \equiv_{t,v,l} \quad \frac{\Gamma, t_1 \equiv t_2 \vdash t[a := t_1] : A}{\Gamma, t_1 \equiv t_2 \vdash t[a := t_2] : A} \equiv_{t,t,l} \\
\\
\frac{\Gamma, w_1 \equiv w_2 \vdash t : A[x := w_1]}{\Gamma, w_1 \equiv w_2 \vdash t : A[x := w_2]} \equiv_{t,v,r} \quad \frac{\Gamma, t_1 \equiv t_2 \vdash t : A[a := t_1]}{\Gamma, t_1 \equiv t_2 \vdash t : A[a := t_2]} \equiv_{t,t,r}
\end{array}$$

Fig. 4. Second-order type system.

$$\frac{\Gamma, t \equiv v \vdash t : A \quad a \notin FV(\Gamma)}{\Gamma, t \equiv v \vdash t : \forall a A} \forall_{i,\equiv}$$

can be derived as follows.

$$\begin{array}{c}
\frac{\Gamma, t \equiv v \vdash t : A}{\Gamma, t \equiv v \vdash v : A} \equiv_{t,t,l} \\
\frac{\Gamma, t \equiv v \vdash v : A \quad v\downarrow \quad a \notin FV(\Gamma)}{\Gamma, t \equiv v \vdash_v v : \forall a A} \forall_i \\
\frac{\Gamma, t \equiv v \vdash_v v : \forall a A}{\Gamma, t \equiv v \vdash v : \forall a A} v\uparrow \\
\frac{\Gamma, t \equiv v \vdash v : \forall a A}{\Gamma, t \equiv v \vdash t : \forall a A} \equiv_{t,t,l}
\end{array}$$

The value restriction can be removed on every other rule in a similar way.

### E. Adequacy

We are now going to prove the soundness of our type system by showing that it is compatible with our realizability model in some sense. This property is specified by the following theorem which is traditionally called adequacy lemma.

**Definition 16.** Let  $\Gamma$  be a context and  $\rho$  be a valuation such that  $\Gamma[\rho]$  is closed. We say that the substitution  $\sigma$  realizes the closed context  $\Gamma[\rho]$  if:

- for every  $x : A$  in  $\Gamma$  we have  $\sigma(x) \in \llbracket A \rrbracket_\rho$ ,
- for every  $\alpha : \neg A$  in  $\Gamma$  we have  $\sigma(\alpha) \in \llbracket A \rrbracket_\rho^\perp$ ,
- for every  $t \equiv u$  in  $\Gamma$  we have  $t\sigma \equiv u\sigma$  and
- for every  $t \neq u$  in  $\Gamma$  we have  $t\sigma \neq u\sigma$ .

**Theorem 6.** (Adequacy lemma) *Let  $\Gamma$  be a context,  $A$  be a formula and  $\rho$  be a valuation such that  $\Gamma[\rho]$  and  $A[\rho]$  are closed. Let  $\sigma$  be a substitution realizing  $\Gamma[\rho]$ .*

- If  $\Gamma \vdash v : A$  then  $v\sigma \in \llbracket A \rrbracket_\rho$ .
- if  $\Gamma \vdash t : A$  then  $t\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$ .

*Proof:* We proceed by induction on the derivation of the judgement  $\Gamma \vdash v : A$  (resp.  $\Gamma \vdash t : A$ ) and we reason by case on the last rule used.

(Ax) By hypothesis  $\sigma$  realizes  $(\Gamma, x : A)[\rho]$  from which we get  $x\sigma = \sigma(x) \in \llbracket A \rrbracket_\rho$ .

( $\Rightarrow_i$ ) We have to show  $\lambda x t\sigma \in \llbracket A \Rightarrow B \rrbracket_\rho$ . By definition of  $\llbracket A \Rightarrow B \rrbracket_\rho$  this means that we need to take  $v \in \llbracket A \rrbracket_\rho$  and show  $t\sigma[x:=v] \in \llbracket B \rrbracket_\rho^{\perp\perp}$ . Since  $\sigma[x:=v]$  realizes  $(\Gamma, x : A)[\rho]$  we can conclude using the induction hypothesis.

( $\Rightarrow_e$ ) We need to show  $t\sigma u\sigma \in \llbracket B \rrbracket_\rho^{\perp\perp}$ , hence we take  $\pi \in \llbracket B \rrbracket_\rho^{\perp}$  and show  $t\sigma u\sigma * \pi \in \perp$ . As  $\perp$  is closed under backward reduction it is enough to show  $u\sigma * [t\sigma]\pi \in \perp$ . By induction hypothesis  $u\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$  so it remains to show  $[t\sigma]\pi \in \llbracket A \rrbracket_\rho^{\perp}$ . To do so we take  $v \in \llbracket A \rrbracket_\rho$  and show  $v * [t\sigma]\pi \in \perp$ . Here we can again take a reduction step and show  $t\sigma * v.\pi \in \perp$ . By induction hypothesis we have  $t\sigma \in \llbracket A \Rightarrow B \rrbracket_\rho^{\perp\perp}$ , hence it is enough to show  $v.\pi \in \llbracket A \Rightarrow B \rrbracket_\rho^{\perp}$ . We now take a value  $\lambda x t_x \in \llbracket A \Rightarrow B \rrbracket_\rho$  and show that  $\lambda x t_x * v.\pi \in \perp$ . We then apply again a reduction step and show  $t_x[x:=v] * \pi \in \perp$ . Since  $\pi \in \llbracket B \rrbracket_\rho^{\perp}$  we only need to show  $t_x[x:=v] \in \llbracket B \rrbracket_\rho^{\perp\perp}$  which is true by definition of  $\llbracket A \Rightarrow B \rrbracket_\rho$ .

( $\mu$ ) We need to show that  $\mu\alpha t\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$  hence we take  $\pi \in \llbracket A \rrbracket_\rho^{\perp}$  and show  $\mu\alpha t\sigma * \pi \in \perp$ . Since  $(\rightarrow)$  is closed under backward reduction showing  $t\sigma[\alpha := \pi] * \pi \in \perp$  is enough. As  $\sigma[\alpha := \pi]$  realizes  $(\Gamma, \alpha : \neg A)[\rho]$  we conclude by induction hypothesis.

(\*) We need to show  $t\sigma * \alpha\sigma \in \llbracket B \rrbracket_\rho^{\perp\perp}$  so we take  $\pi \in \llbracket B \rrbracket_\rho^{\perp}$  and show  $(t\sigma * \alpha\sigma) * \pi \in \perp$ . Since we can take a reduction step it is enough to show  $t\sigma * \alpha\sigma \in \perp$ . By induction hypothesis  $t\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$  hence it is enough to show  $\alpha\sigma = \sigma(\alpha) \in \llbracket A \rrbracket_\rho^{\perp}$  which is true by hypothesis.

( $\forall_i$ ) We need to show that  $v\sigma \in \llbracket \forall a A \rrbracket_\rho = \bigcap_{t \in \Lambda} \llbracket A[a:=t] \rrbracket_\rho$  so we take  $t \in \Lambda$  and show  $v\sigma \in \llbracket A[a:=t] \rrbracket_\rho = \llbracket A \rrbracket_{\rho[a:=t]}$ . This is true by induction hypothesis since  $a \notin FV(\Gamma)$  and hence  $v\sigma$  realizes  $\Gamma[\rho[a:=t]]$ .

( $\forall_e$ ) We need to show  $t\sigma \in \llbracket A[a:=u] \rrbracket_\rho^{\perp\perp}$  for some  $u \in \Lambda$ . By induction hypothesis we know  $t\sigma \in \llbracket \forall a A \rrbracket_\rho^{\perp\perp}$  hence we only need to show that  $\llbracket \forall a A \rrbracket_\rho^{\perp\perp} \subseteq \llbracket A \rrbracket_{\rho[a:=u]}^{\perp\perp}$ . Since  $\llbracket \forall a A \rrbracket_\rho \subseteq \llbracket A \rrbracket_{\rho[a:=u]}$  we can conclude using lemma 9.

( $\exists_i$ ) The proof for this rule is similar to the one for ( $\forall_e$ ). We need to show that  $\llbracket A[a:=u] \rrbracket_\rho^{\perp\perp} \subseteq \llbracket \exists a A \rrbracket_\rho^{\perp\perp}$ . By definition  $\llbracket A[a:=u] \rrbracket_\rho \subseteq \llbracket \exists a A \rrbracket_\rho$  hence we use lemma 9.

( $\exists_e$ ) We need to show that  $t\sigma \in \llbracket B \rrbracket_\rho^{\perp\perp}$ . To be able to use the induction hypothesis  $\sigma$  must realizes  $(\Gamma, y : A)[\rho]$ . By hypothesis  $\sigma$  realizes  $(\Gamma, y : \exists a A)[\rho]$  hence it remains to show  $\sigma(y) \in \llbracket A \rrbracket_\rho$ . Since  $\sigma(y) \in \llbracket \exists a A \rrbracket_\rho = \bigcup_{u \in \Lambda} \llbracket A[a:=u] \rrbracket_\rho$  we can conclude directly.

( $\forall_I, \forall_E, \exists_I, \exists_E$ ) Similar to ( $\forall_i$ ), ( $\forall_e$ ), ( $\exists_i$ ), ( $\exists_e$ ).

( $\times_i$ ) We need to show that  $\{l_i = v_i\sigma\}_{i=1}^n \in \llbracket \{l_i : A_i\}_{i=1}^n \rrbracket_\rho$  so we take  $0 \leq i \leq n$  and show  $\{\dots l_i = v_i\sigma \dots\}.l_i \in \llbracket A_i \rrbracket_\rho^{\perp\perp}$ . We take  $\pi \in \llbracket A_i \rrbracket_\rho^{\perp}$  and show  $\{\dots l_i = v_i\sigma \dots\}.l_i * \pi \in \perp$ . We then take one step of reduction and show  $v_i\sigma * \pi \in \perp$ . It remains to show that  $v_i\sigma \in \llbracket A_i \rrbracket_\rho^{\perp\perp}$ . By induction hypothesis  $v_i\sigma \in \llbracket A_i \rrbracket_\rho$  hence we can conclude using lemma 6.

( $\times_e$ ) We need to show that  $v\sigma.l_i \in \llbracket A_i \rrbracket_\rho^{\perp\perp}$  for  $0 \leq i \leq n$ . By induction hypothesis  $v\sigma \in \llbracket \{l_i : A_i\}_{i=1}^n \rrbracket_\rho$  and hence we obtain  $v\sigma.l_i \in \llbracket A_i \rrbracket_\rho^{\perp\perp}$  by definition of  $\llbracket \{l_i : A_i\}_{i=1}^n \rrbracket_\rho$ .

( $+$ ) We need to show  $C_i[v\sigma] \in \llbracket [C_i \text{ of } A_i]_{i=0}^n \rrbracket_\rho$  for some  $0 \leq i \leq n$ . By induction hypothesis  $v\sigma \in \llbracket A_{i_0} \rrbracket_\rho$  hence we can conclude by definition of  $\llbracket [C_i \text{ of } A_i]_{i=0}^n \rrbracket_\rho$ .

( $+$ ) We need to show  $\text{case}_{v\sigma} [C_i[x] \rightarrow t_i\sigma]_{i=0}^n \in \llbracket B \rrbracket_\rho^{\perp\perp}$ . By induction hypothesis  $v\sigma \in \llbracket [C_i \text{ of } A_i]_{i=0}^n \rrbracket_\rho$  which means that there is  $0 \leq i \leq n$  and  $w \in \llbracket A_i \rrbracket_\rho$  such that  $v\sigma = C_i[w]$ . We take  $\pi \in \llbracket B \rrbracket_\rho^{\perp}$  and show  $\text{case}_{C_i[w]} [C_i[x] \rightarrow t_i\sigma]_{i=0}^n * \pi \in \perp$ . We then take a reduction step and show  $t_i\sigma[x:=w] * \pi \in \perp$ . It remains to show that  $t_i\sigma[x:=w] \in \llbracket B \rrbracket_\rho^{\perp\perp}$ . In order to conclude using the induction we need to show that  $\sigma[x:=w]$  realizes  $(\Gamma, x : A_i, C_i[x] \equiv v)[\rho]$ . This is true since  $\sigma$  realizes  $\Gamma[\rho]$ ,  $w \in \llbracket A_i \rrbracket_\rho$  and  $C_i[w] \equiv v\sigma$  by reflexivity.

( $\in_i$ ) We need to show  $v\sigma \in \llbracket v\sigma \in A \rrbracket_\rho$ . By definition we need to show that  $t\sigma \in \llbracket A \rrbracket_\rho$  which is true by induction hypothesis. We also need to show that  $v\sigma \equiv v\sigma$  which is true since  $(\equiv)$  is an equivalence relation.

( $\in_e$ ) We need to show  $t\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$ . By induction hypothesis  $t\sigma \in \llbracket u \in A \rrbracket_\rho^{\perp\perp}$  hence we can conclude using lemma 9 since  $\llbracket u \in A \rrbracket_\rho \subseteq \llbracket A \rrbracket_\rho$ .

( $\vdash_i$ ) We need to show  $t\sigma \in \llbracket A \vdash u_1 \equiv u_2 \rrbracket_\rho^{\perp\perp}$ . By hypothesis  $u_1\rho\sigma \equiv u_2\rho\sigma$  hence  $\llbracket A \vdash u_1 \equiv u_2 \rrbracket_\rho = \llbracket A \rrbracket_\rho$ . By lemma 6 it is enough to show  $t\sigma \in \llbracket A \rrbracket_\rho^{\perp\perp}$  which is exactly the induction hypothesis.

( $\vdash_e$ ) In order to be able to use the induction hypothesis we need to show that  $\sigma$  realizes  $(\Gamma, x : A, u_1 \equiv u_2)[\rho]$ . By hypothesis we know that  $\sigma$  realizes  $\sigma \vdash (\Gamma, x : A \vdash u_1 \equiv u_2)[\rho]$ . In particular  $\sigma(x) \in \llbracket A \vdash u_1\sigma \equiv u_2\sigma \rrbracket_\rho \neq \emptyset$  hence we can deduce that  $\sigma(x) \in \llbracket A \rrbracket_\rho$  and that  $u_1\rho\sigma \equiv u_2\rho\sigma$ .

( $v\uparrow$ ) Direct consequence of lemma 6.

( $v\downarrow$ ) Direct consequence of theorem 5.

( $\equiv_{v,v,l}$ ) We need to show that  $v\sigma[x:=w_1\rho\sigma] \in \llbracket A \rrbracket_\rho$ . By hypothesis we know that  $w_1\rho\sigma \equiv w_2\rho\sigma$  from which we can deduce  $v\sigma[x:=w_1\rho\sigma] \equiv v\sigma[x:=w_2\rho\sigma]$  by extensionality (theorem 2). Since  $\llbracket A \rrbracket_\rho$  is closed under  $(\equiv)$  it is enough to show  $v\sigma[x:=w_2\rho\sigma] \in \llbracket A \rrbracket_\rho$  which is exactly the induction hypothesis.

( $\equiv_{v,t,l}, \equiv_{v,v,r}, \equiv_{v,t,r}, \equiv_{t,v,l}, \equiv_{t,t,l}, \equiv_{t,v,r}, \equiv_{t,t,r}$ ) The proof for these rules is similar to the one for  $(\equiv_{v,v,l})$  and relies on extensionality (theorem 2 and theorem 3). ■

### III. FURTHER DEVELOPMENTS

The model presented in the previous sections is intended to be used as the basis of the implementation of *PML2*.

Christophe Raffalli already investigated a system with similar ideas called *PML* [20]. Here we build up from this experience to build clean theoretical basis for the new version of the language.

A lot of work remains on the side of implementation, which is now our first priority. A working prototype is required in order to be able to try several variations of the system and assess its usability in practice. The core of the system will consist in three independent components solving three undecidable problems:

- a type-checker,
- a termination checker and
- a procedure for deciding program equivalence.

#### A. Termination checking and recursive types

We plan to extend *PML* with recursive programs reusing Pierre Hyvernats's termination checker based on size change termination [21] that has been used for the first implementation of *PML*.

We will also extend the type system with inductive types [22], [23]. They can be added to the system in the form of fixpoints  $\mu X_n A$  and  $\nu X_n A$ .

#### B. Deciding program equivalence with a pseudo-algorithm

The following rules are necessary to use program equivalence. If they are not present, the truth of an equivalences cannot be used.

$$\frac{\Gamma, u_1 \equiv u_2 \vdash t : A \quad \Gamma \vdash_{Eq} u_1 \equiv u_2}{\Gamma \vdash t : A}$$

$$\frac{\Gamma, u_1 \equiv u_2 \vdash v : A \quad \Gamma \vdash_{Eq} u_1 \equiv u_2}{\Gamma \vdash v : A}$$

A partial algorithm will be required to check program equivalence. This is not difficult to implement using classical techniques like Knuth-Bendix algorithm. However, to preserve the adequation lemma, the validity of a few equational axioms need to be proved:

- $(\lambda x t)v \equiv t[x:=v]$
- $t(\mu\alpha u) \equiv \mu\beta (t u[_* \alpha := t_* \beta])$
- $(\mu\alpha u) v \equiv \mu\beta (u[_* \alpha := v_* \beta] v)$
- Reduction rules for product and sums, maybe some  $\eta$  rules, ...

#### C. Toward mathematics

We also need to assess the expressivity of *PML2* when it comes to formalizing mathematics. This will probably requires more extensions like: subtyping [24], dependent pair type (a.k.a.  $\Sigma$ -type) or extensional quotient type

#### ACKNOWLEDGMENTS

I would like to thank Christophe Raffalli for our many interesting discussions about this work. He contributed greatly to most of the ideas presented here. I would also like to thank Alexandre Miquel for his suggestions that helped making the type system more elegant. In particular he suggested to encode the dependent product type using relativised quantification

à la Krivine. Thank you also to Pierre Hyvernats and Tom Hirschowitz for their very helpful comments about first drafts of this article.

#### REFERENCES

- [1] The Coq development team, *The Coq proof assistant reference manual*, LogiCal Project, 2004, version 8.0. [Online]. Available: <http://coq.inria.fr>
- [2] U. Norell, "Dependently typed programming in agda," in *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.
- [3] P. Martin-Löf, "Constructive mathematics and computer programming," in *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*, ser. Studies in Logic and the Foundations of Mathematics, L. J. Cohen, J. o, H. Pfeiffer, and K.-P. Podewski, Eds. North-Holland, 1982, vol. 104, pp. 153–175. [Online]. Available: [http://dx.doi.org/10.1016/S0049-237X\(09\)70189-2](http://dx.doi.org/10.1016/S0049-237X(09)70189-2)
- [4] T. Coquand and G. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988. [Online]. Available: [http://dx.doi.org/10.1016/0890-5401\(88\)90005-3](http://dx.doi.org/10.1016/0890-5401(88)90005-3)
- [5] A. K. Wright, "Simple imperative polymorphism," in *LISP and Symbolic Computation*, 1995, pp. 343–356.
- [6] R. Hindley, "The principal type-scheme of an object in combinatory logic," *Trans. Amer. Math. Soc.*, vol. 146, pp. 29–60, Dec. 1969.
- [7] M. Tofte, "Type inference for polymorphic references," *Inf. Comput.*, vol. 89, no. 1, pp. 1–34, Sep. 1990. [Online]. Available: [http://dx.doi.org/10.1016/0890-5401\(90\)90018-D](http://dx.doi.org/10.1016/0890-5401(90)90018-D)
- [8] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82. New York, NY, USA: ACM, 1982, pp. 207–212. [Online]. Available: <http://doi.acm.org/10.1145/582153.582176>
- [9] X. Leroy and P. Weis, "Polymorphic type inference and assignment," in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '91. New York, NY, USA: ACM, 1991, pp. 291–302. [Online]. Available: <http://doi.acm.org/10.1145/99583.99622>
- [10] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, Nov. 1994. [Online]. Available: <http://dx.doi.org/10.1006/inco.1994.1093>
- [11] J. Garrigue, "Relaxing the value restriction," in *Functional and Logic Programming*, ser. Lecture Notes in Computer Science, Y. Kameyama and P. Stuckey, Eds. Springer Berlin Heidelberg, 2004, vol. 2998, pp. 196–213.
- [12] X. Leroy, "Polymorphism by name for references and continuations," in *20th symposium Principles of Programming Languages*. ACM Press, 1993, pp. 220–231.
- [13] T. G. Griffin, "A formulæ-as-types notion of control," in *In Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1990, pp. 47–58.
- [14] M. Parigot, " $\lambda\mu$ -calculus: An algorithmic interpretation of classical natural deduction," in *Lecture Notes in Computer Science*, 1992, vol. 624, pp. 190–201.
- [15] R. Harper and M. Lillibridge, "ML with callcc is unsound," Jul. 1991.
- [16] J.-L. Krivine, "Realizability in classical logic," in *Interactive models of computation and program behaviour*, ser. Panoramas et synthèses. Société Mathématique de France, 2009, vol. 27, pp. 197–229.
- [17] G. Munch-Maccagnoni, "Focalisation and classical realisability."
- [18] A. Miquel, "Le calcul des constructions implicites : Syntaxe et sémantique," Ph.D. dissertation, Université Paris VII, 2001.
- [19] J.-L. Krivine, "A call-by-name lambda-calculus machine," vol. 20, pp. 199–207, 2007. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-00154508>
- [20] Christophe Raffalli, *The PML programming language*, LAMA - Université Savoie Mont-Blanc, 2012. [Online]. Available: <http://lama.univ-savoie.fr/tracpml/>
- [21] P. Hyvernats, "The size-change termination principle for constructor based languages," vol. 10, Feb. 2014.
- [22] C. Raffalli, "L'arithmétique fonctionnelle du second ordre avec points fixes," Ph.D. dissertation, Université Paris VII, 1994.
- [23] N. P. Mendler, "Recursive types and type constraints in second-order lambda calculus," in *LICS*, 1987, pp. 30–36.
- [24] C. Raffalli, "Type checking in system  $f^\eta$ ," 1998.