A Realizability Model for a Semantical Value Restriction

Rodolphe Lepigre

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 here we extend it to handle dependent products. The main idea is to internalize observational equivalence as a new non-computable operation. A crucial property of the 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 developped proof assistants following the Curry-Howard isomorphism are Coq [1] and Agda [2]. The former is based on Coquand and Huet's calculus of constructions [3] and the latter on Martin-Löf's dependent type theory [4]. These two constructive theories provide *dependent types* (i.e. indexed families) which allow the definition of very expressive specifications[1]. ML-like languages may also be seen as proof assistants following Curry-Howard. However they lack *dependent types* 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 building a system called *PML* whose proof-terms embed a real ML-like language. Their aim is to design a proof assistant (or type system) that remains close to the ML philosophy while achieving *soundness* in two ways: the logic should be consistent and programs should be type-safe.

Type annotations. There are two different ways of writing typed λ-terms. The first one, called *Church style*, consists in annotating λ-abstractions with the type of their argument. Type abstractions and type applications are then used to express polymorphism. For instance, the term $\Lambda\alpha\lambda x^{\alpha} x$ corresponds to the polymorphic identity function. The main defect of this syntax is that the user has to write a lot of code that is not useful in terms of computation. The computational content of terms can even be made difficult to identify. The second approach, called *Curry style*, works on pure λ-terms. Hence $\lambda x\, x$ can be given any type of the form $\alpha \to \alpha$, which is implicitly generalized into the polymorphic type $\forall\alpha.\alpha \to \alpha$. Such a function can thus be applied to any input transparently. As no indications are provided, a type inference procedure is required and decidability of type-checking is lost. This is probably the reason why Coq and Agda use Church style syntax. In this paper we follow ML and consider a Curry style syntax. Our type-checking algorithm will hence be partial: it may give up or not terminate.

Evaluation strategy. In languages with the Church-Rosser property like Coq or Agda, evaluation order is irrelevant. Here, on the contrary, we consider a language with side-effects as we want to interpret classical logic using control operators. In particular we need to decide what is to happen when a constant function is applied to an argument generating side-effects. Thus we need to pick an evaluation order. *Call-by-name* postpones the computation of the argument of a function to the time of its effective use. In particular if the argument is never used, it is never computed. *Call-by-value* computes the argument before performing the application. We choose to follow most implementations of ML and use call-by-value. Due to the presence of control operators, some non-trivial work will be required in order to preserve soundness. A commonly used method is value restriction.

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 problem stems from a bad interaction between Hindley-Milner polymorphism and side-effects. It was first formulated in terms of references, which were added to ML using the following primitive polymorphic procedures.

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

As explained very well by Andrew Wright (see [5] section 2), this naive approach is doomed. For instance, the following program is accepted by the type-checker, even though it adds a boolean to an integer.

[1]For an introduction to programming with dependent types see [2].

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

To solve this problem, many researchers have built type systems compatible with references (for example [6]–[9]). However, they all introduced a complexity that contrasted with the beautiful simplicity of ML's original type system[2]. A simple and elegant solution finally arose in the nineties. Andrew Wright suggested restricting generalization in let-bindings[3] to cases where the bound term is a value [5], [10]. In slightly more expressive type systems, this restriction appears in the typing rule for the introduction of the universal quantifier. The usual rule

$$\frac{\Gamma \vdash t : A \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash t : \forall\alpha\ A}\ \forall'_i$$

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 the publication of Timothy Griffin's seminal paper [12], it has been well known that classical logic can be given a computational interpretation in terms of control operators. In 1991, Robert Harper and Mark Lillibridge found a complex program breaking the type safety of ML extended with *call/cc*[4] [13]. However, as with references, value restriction solves the inconsistency and yields a sound type system.

An equivalent way of obtaining control structures is Michel Parigot's $\lambda\mu$-calculus [14]. It replaces control operators with a new binder $\mu\alpha\, t$ capturing the current continuation in the μ-variable α. The continuation can then be restored in t using the syntax $u * \alpha$ (originally denoted $[\alpha]u$ in [14]). In terms of control operators, $\mu\alpha\, t$ can be translated to $\texttt{callcc}\,(\lambda\alpha\, t)$ and $u * \alpha$ as $\texttt{throw}\,\alpha\, u$.

In the context of the $\lambda\mu$-calculus, the soundness issue arises when evaluating $t\,(\mu\alpha\, u)$ when $\mu\alpha\, u$ has a polymorphic type. Such a situation cannot happen with value restriction. Indeed $\mu\alpha\, u$ is not a value, hence its type cannot be generalized. Without value restriction, subject reduction (i.e. type preservation) fails for the following reduction rule[5].

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

The natural way of transforming the corresponding proof-tree leads to an illegal universal quantifier introduction rule, as shown in figure 1 (typing rules can be found in figure 4).

As in the case of the introduction rule for universal quantification, the elimination rule for dependent product

$$\frac{\Gamma \vdash t : \Pi_{x:A}\ B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ \Pi'_e$$

cannot be proved safe when u is not a value. Figure 2 shows how subject reduction fails on $t\ (\mu\alpha\ u)$ when t has a type $\Pi_{x:A}B$.

An important question is: can we live with such a restriction? The answer is yes on the logical side as A is logically equivalent to $\top \Rightarrow A$. In other words, any term can be made into a value by guarding it with a dummy λ-abstraction. However, this solution is not satisfactory as it amounts to emulating call-by-name evaluation. Moreover, value restriction makes dependent products very weak. For instance types like $List(n+m)$ are forbidden since $n+m$ is not a value.

C. Toward a semantical value restriction

The main contribution of this paper is a new approach to value restriction. The syntactic restriction on terms is replaced 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, where v denotes a value.

$$\frac{\Gamma, t \equiv v \vdash t : A \qquad \alpha \notin Fv(\Gamma)}{\Gamma, t \equiv v \vdash t : \forall\alpha\ A}$$

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

In both rules the term t does not have to be a value but must be equivalent to some value v. This can be understood in terms of convergence: a term either reduces to a value or it diverges 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. In this paper we do not prove subject reduction but this is not a problem as our model construction implies type safety (theorem 7). Furthermore our type system is consistent as a logic (theorem 8).

In this paper, we restrict ourselves to a second order type system but it can easily be extended to higher-order. Types are built from two basic sorts of objects: *propositions* (the types themselves) and *individuals* (untyped terms of the language). Terms appear in two constructs: a restriction predicate $A \restriction t \equiv u$ and a belonging predicate $t \in A$. The former relativizes proposition A with respect to the observational equivalence of t and u. This can be seen as a conjunction with no algorithmic contents. The latter expresses the fact that term t has type A. In the semantics $t \in A$ will be interpreted as a singleton type (up to program equivalence). In particular, this construct will allow us to encode a limited form of dependent product.

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

Overall, the higher-order version of our system is very similar to a Curry style HOL with ML programs as individuals. It does not allow the definition of a type whose structure depends on a term (e.g. function with a variable number of arguments). Our system is thus placed between HOL ($F\omega$) and Coq (pure calculus of constructions CoC) in a Curry style copy of Barendregt's λ-cube (Figure 3). Note that another

[2] See [10] section 2 and [11] section 2 for a detailed account.

[3] In ML the polymorphism mechanism is strongly linked with let-bindings. They are expressions of the form `let x = u in t`.

[4] *Call/cc* means "call with current continuation" and was first introduced in the Scheme language.

[5] In the rule, the substitution $[\_ * \alpha\ :=\ t\_ * \beta]$ replaces any term of the form $t' * \alpha$ by $t\ t' * \beta$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\Delta,\Gamma,\alpha:\neg\forall XA\vdash \_ : A}{\Delta,\Gamma,\alpha:\neg\forall XA\vdash \_ * \alpha : C}\,*}\quad \dfrac{\dfrac{\Gamma,\alpha:\neg\forall XA\vdash u:A}{\Gamma,\alpha:\neg\forall XA\vdash u:\forall XA}\,\forall'_i \qquad X\notin FV(\Gamma,\alpha:\neg\forall XA)}{\Gamma\vdash \mu\alpha\ u:\forall XA}\,\mu}{\Gamma\vdash t\ (\mu\alpha\ u):B}\,\Rightarrow_e}{\Gamma\vdash t:(\forall XA)\Rightarrow B}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\Gamma\vdash t:(\forall XA)\Rightarrow B}{\Delta,\Gamma,\beta:\neg B\vdash t:(\forall XA)\Rightarrow B}\,\text{Wk} \qquad \dfrac{\Delta,\Gamma,\beta:\neg B\vdash \_ : A}{\Delta,\Gamma,\beta:\neg B\vdash \_ : \forall XA}\ \textit{Not safe}}{\Delta,\Gamma,\beta:\neg B\vdash t\ \_ : B}\,\Rightarrow_e}{\Delta,\Gamma,\beta:\neg B\vdash t\ \_ * \beta : C}}{\Gamma,\beta:\neg B\vdash t\ u[\_ * \alpha := t\_ * \beta] : B}\,\mu}{\Gamma\vdash \mu\beta(t\ u[\_ * \alpha := t\_ * \beta]) : B}$$

$$\dfrac{\dfrac{\Gamma\vdash t:(\forall XA)\Rightarrow B}{\Gamma,\beta:\neg B\vdash t:(\forall XA)\Rightarrow B}\,\text{Wk} \qquad \dfrac{\vdots}{\Gamma,\beta:\neg B\vdash u[\_ * \alpha := t\_ * \beta]:\forall XA}}{\Gamma,\beta:\neg B\vdash t\ u[\_ * \alpha := t\_ * \beta] : B}\,\Rightarrow_e$$

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

$$\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\Delta,\Gamma,\alpha:\neg A\vdash \_ : A}{\Delta,\Gamma,\alpha:\neg A\vdash \_ * \alpha : C}\,*}\quad \dfrac{\Gamma,\alpha:\neg A\vdash u:A}{\Gamma\vdash \mu\alpha\ u:A}\,\mu}{\Gamma\vdash t\ (\mu\alpha\ u):B[x:=\mu\alpha\ u]}\,\Pi'_e}{\Gamma\vdash t:\Pi_{x:A}B}$$

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\Delta,\Gamma\vdash t:\Pi_{x:A}B}{\Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t:\Pi_{x:A}B}\,\text{Wk} \qquad \Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash \_ : A}{\Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t\ \_ : B[x\ :=\ \_]}\,\Pi'_e}\ \textit{Not safe}}{\Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t\ \_ : B[x:=\mu\alpha\ u]}\,*}{\Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t\ \_ * \beta : C}}{\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t\ u[\_ * \alpha := t\_ * \beta] : B[x:=\mu\alpha\ u]}\,\mu}{\Gamma\vdash \mu\beta(t\ u[\_ * \alpha := t\_ * \beta]) : B[x:=\mu\alpha\ u]}$$

$$\dfrac{\dfrac{\Gamma\vdash t:\Pi_{x:A}B}{\Delta,\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t:\Pi_{x:A}B}\,\text{Wk} \qquad \dfrac{\vdots}{\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash u[\_ * \alpha := t\_ * \beta] : A}\,\Pi'_e}{\Gamma,\beta:\neg B[x:=\mu\alpha\ u]\vdash t\ u[\_ * \alpha := t\_ * \beta] : B[x:=\mu\alpha\ u]}$$

Fig. 2. The failure of subject reduction on the Π'_e rule.

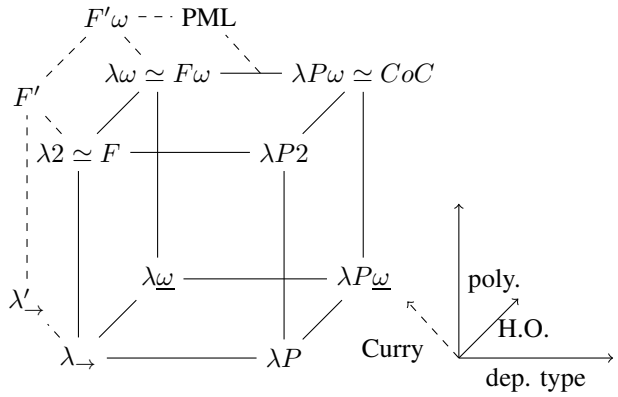

Fig. 3. PML in Barendregt's λ-cube (X' denotes Curry style version of X)

dimension should also be added as our system has support for classical logic.

Throughout this article we build a realizability model à la Krivine [15] based on a call-by-value abstract machine. As a consequence, formulas are interpreted using three layers (values, stacks and terms as in [16]) related via orthogonality (definition 9). The crucial property for the soundness of semantical value restriction (theorem 4) is that

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

for every set of values ϕ (closed under $(\equiv)$). Λ_v denotes 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 ϕ (resp. stacks in $\phi^{\perp}$). Note that this property seems completely unrelated to lemma 9 in Munch's work [16]. To obtain a model satisfying this property, we need to extend our programming language with a term $\delta_{v,w}$ whose reduction depends on the observational equivalence of two values v and w.

D. Related works and similar systems

We are not aware of any system similar to ours. The nearest system seems to be in the work of Alexandre Miquel [17], where propositions can be classical and Curry style. However the rest of the language remains Church style and does not embed a full ML-like language.

There are several works extending ML with dependent types (DML, ATS, Idris). However they do not have to deal with the problem presented here as none of them is both classical and call-by-value.

The PVS system [18] is similar to ours as it is based on classical higher-order logic. However this tool does not seem to be a programming language, but rather a specification language coupled with proof checking and model checking utilities. It is nonetheless worth mentioning that the undecidability of PVS's type system is handled by generating proof obligations. Our system will take a different approach and use a non-backtracking type-checking and type-inference algorithm.

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 distrinction 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 coutable sets of variables:

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

We also require a countable set $\mathcal{L} = \{l, l_1, l_2...\}$ of *labels* to name *record* fields[6], and a countable set $\mathcal{C} = \{C, C_1, C_2...\}$ of *constructors*[7].

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.*

$$\begin{aligned}
v, w &::= x \mid \lambda x\, t \mid C[v] \mid \{l_1 = v_1;\ ...\ l_n = v_n;\} && (\Lambda_v)\\
t, u &::= a \mid v \mid t\ u \mid \mu\alpha\, t \mid p \mid v.l \mid case_v\ B \mid \delta_{v,w} && (\Lambda)\\
\pi, \rho &::= \alpha \mid v.\pi \mid [t]\pi && (\Pi)\\
p, q &::= t * \pi && (\Lambda \times \Pi)
\end{aligned}$$

Terms and values form a variation of the $\lambda\mu$-calculus [14] enriched with ML-like constructs (i.e. records and variants[8]). For technical purposes that will become clear later on, 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[9], and has no consequence for the programmer as we can define syntactic sugars as

$$t.l \ :=\ (\lambda x\, x.l)\, t \qquad \text{or} \qquad C[t] \ :=\ (\lambda x\, C[x])\, t$$

to hide the restriction. We will follow the usual notational convention: application is left associative and both λ-abstraction and μ-abstraction bind stronger than application[10].

Definition 2. *Given a value, term, stack or process ψ we denote $FV_\lambda(\psi)$ (resp. $FV_\mu(\psi)$, $TV(\psi)$) the set of free λ-variables (resp. free μ-variables, term variables) contained in ψ. We say that ψ 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.*

[6] A record is a tuple which fields are name using *labels*.

[7] A *Constructor* is to be thought as a kind of wrapper that can be recognised during a case analysis.

[8] In $case_v\ B$, B is a list of patterns of the form $C[x] \to t$.

[9] In particular only β-reduction has an effect on the content of the stack.

[10] This means that $\lambda x\ t\ u\ v$ is to be read as $\lambda x\ ((t\ u)\ v)$.

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 π in the process $t * \pi$ contains the functional arguments to be fed 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 following rules.*

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

We will denote $(\succ^+)$ its transitive closure, $(\succ^*)$ its reflexive-transitive closure and $(\succ^k)$ its k-fold application.

The first three rules are those that handle β-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 λ-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 μ-abstraction is reached, the current stack (i.e. the current evaluation context) is captured and stored into the corresponding μ-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 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 $a \in \mathcal{V}_\iota$ and $t \in \Lambda$, $p[a := t] \succ q[a := t]$.*

Consequently, if σ 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 $p = v * \alpha$,*
- δ-like *if there are values $v, w \in \Lambda_v$ and a stack $\pi \in \Pi$ such that $p = \delta_{v,w} * \pi$,*
- blocked *if there is no $q \in \Lambda \times \Pi$ such that $p \succ q$,*
- stuck *if it is not final nor δ-like, and if for every substitution σ, $p\sigma$ is blocked,*
- non-terminating *if there is no blocked process $q \in \Lambda \times \Pi$ such that $p \succ^* q$.*

Lemma 2. *Let p be a process and σ be a substitution for variables of any kind. If p is δ-like (resp. stuck, non-terminating) then $p\sigma$ is also δ-like (resp. stuck, non-terminating).*

Proof: Immediate by definition. ■

Lemma 3. *A stuck state is of one of the form:*

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

$$case_{\lambda x\ t}\ B * \pi \qquad case_{\{...\}}\ B * \pi$$

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

Proof: Simple case analysis. ■

Lemma 4. *A blocked process $p \in \Lambda \times \Pi$ is either stuck, final, δ-like, or of one of the forms:*

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

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

Proof: Straight-forward case analysis using lemma 3. This result was verified using the exhaustivity checker of OCaml's pattern matching. ■

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 $v \not\equiv w$. If $v \equiv w$ then $\delta_{v,w}$ is stuck. With this rule reduction and equivalence will become interdependent as equivalence will be defined using reduction.

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 $p \Uparrow_R$. We will use the notations $p \Downarrow_i$ and $p \Uparrow_i$ when working with indexed notation symbols like $(\twoheadrightarrow_i)$.*

Definition 6. *For every natural number i we define a reduction relation $(\twoheadrightarrow_i)$ and an equivalence relation $(\equiv_i)$ whose negation will be denoted $(\not\equiv_i)$.*

$$(\twoheadrightarrow_i) = (\succ) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid \exists j < i, v \not\equiv_j w\}$$

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

In particular, one can easily see that $(\twoheadrightarrow_0) = (\succ)$. For every natural number i, the relation $(\equiv_i)$ is indeed an equivalence relation as it can be seen as the intersection of equivalence relations. Its negation can be expressed as follows.

$$(\not\equiv_i) = \{(t,u),(u,t) \mid \exists j \leq i, \exists \pi, \exists \sigma, t\sigma * \pi \Downarrow_j \wedge u\sigma * \pi \Uparrow_j\}$$

Definition 7. *We define a reduction relation $(\twoheadrightarrow)$ and an equivalence relation $(\equiv)$ whose negation will be denoted $(\not\equiv)$.*

$$(\twoheadrightarrow) = \bigcup_{i \in \mathbb{N}} (\twoheadrightarrow_i) \qquad (\equiv) = \bigcap_{i \in \mathbb{N}} (\equiv_i)$$

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

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

$$(\not\equiv) = \{(t,u),(u,t) \mid \exists i, \exists \pi, \exists \sigma, t\sigma * \pi \Downarrow_i \wedge u\sigma * \pi \Uparrow_i\}$$

$$(\twoheadrightarrow) = (\succ) \cup \{(\delta_{v,w} * \pi, v * \pi) \mid v \not\equiv w\}$$

Remark. *Obviously $(\twoheadrightarrow_i) \subseteq (\twoheadrightarrow_{i+1})$ and $(\equiv_{i+1}) \subseteq (\equiv_i)$. As a consequence the construction of $(\twoheadrightarrow_i)_{i\in\mathbb{N}}$ and $(\equiv_i)_{i\in\mathbb{N}}$ converges. In fact $(\twoheadrightarrow)$ and $(\equiv)$ form a fixpoint at ordinal ω. Surprisingly this property is not necessary here.*

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

Proof: We suppose that $t \equiv u$ and we take $\pi_0 \in \Pi$ and a substitution σ_0. By symmetry we can assume that $t\sigma_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$ and show that $u\sigma_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$. 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 σ 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 $u\sigma_0 * \pi_0 \Downarrow_{\twoheadrightarrow}$. ∎

Remark. *The converse implication is not true in general. If $p \Downarrow_{\twoheadrightarrow}$, we do not necessarily have $p \Downarrow_i$ for a given $i \in \mathbb{N}$.*

Corollary 1. *Let t and u be terms and π be a stack. If $t \equiv u$ and $t * \pi \Downarrow_{\twoheadrightarrow}$ then $u * \pi \Downarrow_{\twoheadrightarrow}$.*

Proof: Direct consequence of theorem 1 using π 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 terms 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 λ-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 $v \not\equiv w$. By definition there is $i \in \mathbb{N}$, $\pi \in \Pi$ and a substitution σ 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 $dom(\sigma)$, we can suppose $E\sigma[x := v\sigma] * \pi \Downarrow_i$ and $E\sigma[x := w\sigma] * \pi \Uparrow_i$. In order to show $v \not\equiv w$ we need to find $i_0 \in \mathbb{N}$, $\pi_0 \in \Pi$ and a substitution σ_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 $v\sigma_0 * \pi_0 \twoheadrightarrow E\sigma[x := v\sigma] * \pi_{i_0} \Downarrow_{i_0}$ and $w\sigma_0 * \pi_0 \twoheadrightarrow E\sigma[x := w\sigma] * \pi_{i_0} \Uparrow_{i_0}$. ∎

Lemma 5. *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 blocked state p such that $s \succ^* p$ and either*

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

Proof: Let σ 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 blocked process p such that $s \succ^* p$ since $(\succ) \subseteq (\twoheadrightarrow_k)$. Using lemma 1 we get $s\sigma \succ^* p\sigma$ from which we obtain $p\sigma\Downarrow_k$. The process p cannot be stuck, otherwise $p\sigma$ would also be stuck 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 π. Since $\delta_{v\sigma,w\sigma} * \pi \Downarrow_k$ there must be $i < k$ such that $v\sigma \not\equiv_j w\sigma$, otherwise this would contradict $\delta_{v\sigma,w\sigma} * \pi \Downarrow_k$. In this case we necessarily have $k > 0$, otherwise there would be no possible candidate for i. According to lemma 4 we need to rule out four more forms of therms: $x.l * \pi$, $x * v.\pi$, $case_x\ B * \pi$ and $b * \pi$ in the case where $b \neq a$. If p was of one of these forms the substitution σ would not be able to unblock the reduction of p, which would contradict again $p\sigma\Downarrow_k$. ∎

Lemma 6. *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 $E[a := t_1] \equiv_k E[a := t_1]$. By symmetry we can assume that we have $i \leq k$, $\pi \in \Pi$ and a substitution σ 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 is does not appear in $dom(\sigma)$, $TV(\pi)$, $TV(t_1)$ 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 $E'\sigma_1 * \pi \twoheadrightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1$ 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, π_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 5 to obtain a blocked 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 $(E' * \pi)\sigma_1 \twoheadrightarrow_k^* E_0\sigma_1 * \pi_0\sigma_1$ 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 $E'\sigma_1 * \pi \twoheadrightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1 = (E_i\sigma_1 * \pi_i)\sigma_1$ in l_i steps. Using lemma 5 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 α 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 5 that there is a blocked 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 $E_i\sigma_1 * \pi_i\sigma_1 \twoheadrightarrow_k p\sigma_1$ 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 5. It cannot be of the form $a * \pi$ as we assumed that a does not appear in t_1 or σ. 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 $E_i\sigma_1 * \pi_i = \delta(v\sigma_1, w\sigma_1) * \pi_i \twoheadrightarrow_k v\sigma_1 * \pi_i$ by definition. Moreover $E_i\sigma_1 * \pi_i\sigma_1 \twoheadrightarrow_k v\sigma_1 * \pi_i\sigma_1$ by lemma 1. Since $E'\sigma_1 * \pi \twoheadrightarrow_k^* E_i\sigma_1 * \pi_i\sigma_1$ in l_i steps we obtain that $E'\sigma_1 * \pi \twoheadrightarrow_k^* v\sigma_1 * \pi_i\sigma_1$ 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 5 there is a blocked 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 $v\sigma_1 * \pi_i\sigma_1 \twoheadrightarrow_k^* p\sigma_1$ 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 δ-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 $E_n\sigma_2 * \pi_n\sigma_2 = v\sigma_2 * \alpha$ from which we trivially obtain $E_n\sigma_2 * \pi_n\sigma_2\Downarrow_k$. We now suppose that $E_{i+1}\sigma_2 * \pi_i\sigma_2\Downarrow_k$ 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[11]:

- If $E_i = a$ then $t_1\sigma * \pi_i \twoheadrightarrow_k^* E_{i+1}\pi_{i+1}$. Using lemma 1 we obtain $t_1\sigma * \pi_i\sigma_2 \twoheadrightarrow_k E_{i+1}\sigma_2 * \pi_i\sigma_2$ 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 $v * \pi_i \twoheadrightarrow_k E_{i+1} * \pi_{i+1}$ and hence $v\sigma_2 * \pi_i\sigma_2 \twoheadrightarrow_k E_{i+1}\sigma_2 * \pi_{i+1}$ by lemma 1. Using the induction hypothesis we obtain $v\sigma_2 * \pi_i\sigma_2\Downarrow_k$. It remains to show that $\delta(v\sigma_2, w\sigma_2) * \pi_i\sigma_2 \twoheadrightarrow_k^* v\sigma_2 * \pi_i\sigma_2$. 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 $w\sigma_1 \equiv_m v\sigma_2$. Now if $v\sigma_2 \equiv_m w\sigma_2$ then $v\sigma_1 \equiv_m v\sigma_2 \equiv_m w\sigma_2 \equiv_m w\sigma_1$ contradicts $v\sigma_1 \not\equiv w\sigma_1$. 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 $E[a{:=}t_1] \equiv E[a{:=}t2]$.*

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 6. ■

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 second-order type system that will be defined shortly. Our abstract machine slightly differs from the mainstream presentation of *Krivine's classical realizability* which is usually call-by-name. Although call-by-value presentations have rarely been published, such developments are well-known among a restricted community of experts. We do not claim any credit on this point. The addition of the δ 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 (i.e. stacks).

Definition 8. *A* pole *is a set of processes $\perp\!\!\!\perp \subseteq \Lambda \times \Pi$ which is closed under backward reduction*[12]*. More formally, if $q \in \perp\!\!\!\perp$ and $p \twoheadrightarrow q$ then $p \in \perp\!\!\!\perp$.*

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

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

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

[11] Only $E_n * \pi_n$ can be of the form $v * \alpha$.

[12] It is said to be *saturated* in Krivine's original work.

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 is given in the following definition.

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

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

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

Lemma 7. *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 8. *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: Immediate by definition of orthogonality. ∎

The construction involving δ 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 4. *If $\Phi \subset \Lambda_v$ is a set of values closed under $(\equiv)$, then $\Phi^{\perp\perp} \cap \Lambda_v = \Phi$.*

Proof: The direction $\Phi \subset \Phi^{\perp\perp} \cap \Lambda_v$ is straight-forward using lemma 7. We are going to show that $\Phi^{\perp\perp} \cap \Lambda_v \subset \Phi$, 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 π_0 such that $v \star \pi_0 \notin \perp\!\!\!\perp$ and for every value $w \in \Phi$, $w \star \pi_0 \in \perp\!\!\!\perp$. We take $\pi_0 = [\lambda x\ \delta(x, v)]\ \alpha$ and show that is is suitable. By definition of the reduction relation $v \star \pi_0$ reduces to $\delta(v, v) \star \alpha$ which is not in $\perp\!\!\!\perp$ (it is stuck as $v \equiv v$ by reflexivity). Let us now take $w \in \Phi$. Again by definition, $w \star \pi_0$ reduces to $\delta(w, v) \star \alpha$, but this time we have $w \not\equiv v$ since Φ was supposed to be closed under $(\equiv)$ and $v \notin \Phi$. Hence $w \star \pi_0$ reduces to $w \star \alpha \in \perp\!\!\!\perp$. ∎

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.

Theorem 5. *The pole $\perp\!\!\!\perp$ is consistent, which means that for every closed term t there is a stack π such that $t * \pi \notin \perp\!\!\!\perp$.*

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

B. 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 $\mathcal{V}_n = \{X_n, Y_n, Z_n ...\}$ of n-ary predicate variables.

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

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

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. Besides 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 provides 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.

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

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[13]. In order to handle free term variables and free predicate variables in formulas we introduce valuations.

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

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

Given a formula A we denote $FV(A)$ the set of its free variables. Given a valuation ρ such that $dom(\rho) \subset FV(A)$ we write $A[\rho]$ the closed formula built by applying ρ to A. In

[13]Here, $\perp$ and $\top$ can be obtain using the usual second-order encoding: $\perp = \forall X_0\ X_0$ and $\top = \exists X_0\ X_0$.

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 ρ such that $A[\rho]$ is closed, we define the* raw semantics $[\![A]\!]_\rho \subseteq \Lambda_v/\equiv$ *of A under the valuation ρ as follows.*

$$
\begin{aligned}
[\![X_n(t_1,...,t_n)]\!]_\rho &= \rho(X_n)(t_1\rho,...,t_n\rho)\\
[\![A \Rightarrow B]\!]_\rho &= \{\lambda x\ t \mid \forall v \in [\![A]\!]_\rho, t[x:=v] \in [\![B]\!]_\rho^{\perp\perp}\}\\
[\![\forall a\ A]\!]_\rho &= \cap_{t\in\Lambda}\ [\![A]\!]_{\rho,a:=t}\\
[\![\exists a\ A]\!]_\rho &= \cup_{t\in\Lambda}\ [\![A]\!]_{\rho,a:=t}\\
[\![\forall X_n\ A]\!]_\rho &= \cap_{P\in\Lambda^n\to\mathcal{P}(\Lambda_v/\equiv)}\ [\![A]\!]_{\rho,X_n:=P}\\
[\![\exists X_n\ A]\!]_\rho &= \cup_{P\in\Lambda^n\to\mathcal{P}(\Lambda_v/\equiv)}\ [\![A]\!]_{\rho,X_n:=P}\\
[\![\{l_i : A_i\}_{i\in I}]\!]^\rho &= \cap_{i\in I}\ \{v \in \Lambda_v \mid v.l_i \in [\![A_i]\!]_\rho^{\perp\perp}\}\\
[\![[C_i : A_i]_{i\in I}]\!]_\rho &= \cup_{i\in I}\ \{C_i[v] \mid v \in [\![A_i]\!]_\rho\}\\
[\![t \in A]\!]_\rho &= \{v \in [\![A]\!]_\rho \mid t\rho \equiv v\}\\
[\![A \restriction t \equiv u]\!]_\rho &= \begin{cases} [\![A]\!]_\rho & \text{if } t\rho \equiv u\rho\\ \emptyset & \text{otherwise}\end{cases}
\end{aligned}
$$

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 ρ a valuation such that $A[\rho]$ is closed. We say that:*

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

C. Contexts and typing rules

Before giving the typing rules 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 λ-variables and μ-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 Γ and*
- *$\Gamma \vdash t : A$ meaning that the term t has type A in the context Γ.*

The typing rules of the system are given in Fig. 4. Although most of them are fairly usual, our type system differs in several ways. For instance the last eight rules are related 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 in their left-hand 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.

An 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 straightforward. Going the other direction is the main motivation for our model as this will allow us to lift the value restriction expressed in the syntax to a restriction expressed in terms of equivalence. As an example, the rule

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

can be derived as follows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, t \equiv v \vdash t : A}{\Gamma, t \equiv v \vdash v : A}\equiv_{t,t,l}}{\Gamma, t \equiv v \vdash_v v : A}\ v\downarrow \qquad a \notin FV(\Gamma)}{\Gamma, t \equiv v \vdash_v v : \forall a\ A}\ \forall_i}{\dfrac{\Gamma, t \equiv v \vdash v : \forall a\ A}{\Gamma, t \equiv v \vdash t : \forall a\ A}\equiv_{t,t,l}}\ v\uparrow$$

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

D. Adequacy

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

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

- *for every $x : A$ in Γ we have $\sigma(x) \in [\![A]\!]_\rho$,*
- *for every $\alpha : \neg A$ in Γ we have $\sigma(x) \in [\![A]\!]_\rho^\perp$,*
- *for every $t \equiv u$ in Γ we have $t\sigma\rho \equiv u\sigma\rho$ and*
- *for every $t \not\equiv u$ in Γ we have $t\sigma\rho \not\equiv u\sigma\rho$.*

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

- *If $\Gamma \vdash_v v : A$ then $v\sigma \in [\![A]\!]_\rho$,*
- *if $\Gamma \vdash t : A$ then $t\sigma \in [\![A]\!]_\rho^{\perp\perp}$.*

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

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

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

$$\frac{}{\Gamma, x : A \Vdash_v x : A}\, Ax \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \Vdash_v \lambda x\ t : A \Rightarrow B} \Rightarrow_i \qquad \frac{\Gamma \vdash t : A \Rightarrow B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B} \Rightarrow_e$$

$$\frac{\Gamma, \alpha : \neg A \vdash t : A}{\Gamma \vdash \mu\alpha\ t : A}\, \mu \qquad \frac{\Gamma, \alpha : \neg A \vdash t : A}{\Gamma, \alpha : \neg A \vdash t * \alpha : B}\, *$$

$$\frac{\Gamma \Vdash_v v : A \qquad a \notin FV(\Gamma)}{\Gamma \Vdash_v v : \forall a\ A}\, \forall_i \qquad \frac{\Gamma \vdash t : \forall a\ A}{\Gamma \vdash t : A[a := u]}\, \forall_e$$

$$\frac{\Gamma \vdash t : A[a := u]}{\Gamma \vdash t : \exists a\ A}\, \exists_i \qquad \frac{\Gamma, y : A \vdash t : B \qquad a \notin FV(\Gamma, x : A)}{\Gamma, y : \exists a\ A \vdash t : B}\, \exists_e$$

$$\frac{\Gamma \Vdash_v v : A \qquad X_n \notin FV(\Gamma)}{\Gamma \Vdash_v v : \forall X_n\ A}\, \forall_I \qquad \frac{\Gamma \vdash t : \forall X_n\ A}{\Gamma \vdash t : A[X_n := P]}\, \forall_E$$

$$\frac{\Gamma \vdash t : A[X_n := P]}{\Gamma \vdash t : \exists X_n\ A}\, \exists_I \qquad \frac{\Gamma, x : A \vdash t : B \qquad X_n \notin FV(\Gamma, x : A)}{\Gamma, x : \exists X_n\ A \vdash t : B}\, \exists_E$$

$$\frac{\{\Gamma \Vdash_v v_i : A_i\}_{i=1}^n}{\Gamma \Vdash_v \{l_i = v_i\}_{i=1}^n : \{l_i : A_i\}_{i=1}^n}\, \times_i \qquad \frac{\Gamma \Vdash_v v : \{l_i : A_i\}_{i=1}^n}{\Gamma \vdash v.l_i : A_i}\, \times_e$$

$$\frac{\Gamma \Vdash_v v : A_i}{\Gamma \Vdash_v C_i[v] : [C_i : A_i]_{i=1}^n}\, +_i \qquad \frac{\Gamma \Vdash_v v : [C_i : A_i]_{i=1}^n \qquad \{\Gamma, x : A_i, C_i[x] \equiv v \vdash t_i : B\}_{i=1}^n}{\Gamma \vdash case_v\ [C_i[x] \to t_i]_{i=1}^n : B}\, +_e$$

$$\frac{\Gamma \Vdash_v v : A}{\Gamma \Vdash_v v : v \in A}\, \in_i \qquad \frac{\Gamma \vdash t : u \in A}{\Gamma \vdash t : A}\, \in_e \qquad \frac{\Gamma, u_1 \equiv u_2 \vdash t : A}{\Gamma, u_1 \equiv u_2 \vdash t : A \restriction u_1 \equiv u_2}\, \restriction_i$$

$$\frac{\Gamma, x : A, u_1 \equiv u_2 \vdash t : B}{\Gamma, x : A \restriction u_1 \equiv u_2 \vdash t : B}\, \restriction_e \qquad \frac{\Gamma \Vdash_v v : A}{\Gamma \vdash v : A}\, v\!\uparrow \qquad \frac{\Gamma \vdash v : A}{\Gamma \Vdash_v v : A}\, v\!\downarrow$$

$$\frac{\Gamma, w_1 \equiv w_2 \Vdash_v v[x := w_1] : A}{\Gamma, w_1 \equiv w_2 \Vdash_v v[x := w_2] : A} \equiv_{v,v,l} \qquad \frac{\Gamma, t_1 \equiv t_2 \Vdash_v v[a := t_1] : A}{\Gamma, t_1 \equiv t_2 \Vdash_v v[a := t_2] : A} \equiv_{v,t,l}$$

$$\frac{\Gamma, w_1 \equiv w_2 \Vdash_v v : A[x := w_1]}{\Gamma, w_1 \equiv w_2 \Vdash_v v : A[x := w_2]} \equiv_{v,v,r} \qquad \frac{\Gamma, t_1 \equiv t_2 \Vdash_v v : A[a := t_1]}{\Gamma, t_1 \equiv t_2 \Vdash_v v : A[a := t_2]} \equiv_{v,t,r}$$

$$\frac{\Gamma, w_1 \equiv w_2 \vdash t[x := w_1] : A}{\Gamma, w_1 \equiv w_2 \vdash t[x := w_2] : A} \equiv_{t,v,l} \qquad \frac{\Gamma, t_1 \equiv t_2 \vdash t[a := t_1] : A}{\Gamma, t_1 \equiv t_2 \vdash t[a := t_2] : A} \equiv_{t,t,l}$$

$$\frac{\Gamma, w_1 \equiv w_2 \vdash t : A[x := w_1]}{\Gamma, w_1 \equiv w_2 \vdash t : A[x := w_2]} \equiv_{t,v,r} \qquad \frac{\Gamma, t_1 \equiv t_2 \vdash t : A[a := t_1]}{\Gamma, t_1 \equiv t_2 \vdash t : A[a := t_2]} \equiv_{t,t,r}$$

Fig. 4. Second-order type system.

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

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

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

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

($\forall_e$) We need to show $t\sigma \in [\![A[a := u]]\!]_\rho^{\perp\perp}$ for some $u \in$

Λ. By induction hypothesis we know $t\sigma \in [\![\forall a\ A]\!]_\rho^{\perp\perp}$ hence we only need to show that $[\![\forall a\ A]\!]_\rho^{\perp\perp} \subseteq [\![A]\!]_{\rho[a:=u]}^{\perp\perp}$. Since $[\![\forall a\ A]\!]_\rho \subseteq [\![A]\!]_{\rho[a:=u]}$ we can conclude using lemma 8.

($\exists_i$) The proof for this rule is similar to the one for *($\forall_e$)*. We need to show that $[\![A[a:=u]]\!]_\rho^{\perp\perp} \subseteq [\![\exists a\ A]\!]_\rho^{\perp\perp}$. By definition $[\![A[a:=u]]\!]_\rho \subseteq [\![\exists a\ A]\!]_\rho$ hence we use lemma 8.

($\exists_e$) We need to show that $t\sigma \in [\![B]\!]_\rho^{\perp\perp}$. To be able to use the induction hypothesis σ must realizes $(\Gamma, y : A)[\rho]$. By hypothesis σ realizes $(\Gamma, y : \exists a\ A)[\rho]$ hence it remains to show $\sigma(y) \in [\![A]\!]_\rho$. Since $\sigma(y) \in [\![\exists a\ A]\!]_\rho = \bigcup_{u\in\Lambda} [\![A[a := u]]\!]_\rho$ 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 $\{l_i = v_i\sigma\}_{i=1}^n \in [\![\{l_i : A_i\}_{i=1}^n]\!]_\rho$ so we take $0 \leq i \leq n$ and show $\{...\ l_i = v_i\sigma\ ...\}.l_i \in [\![A_i]\!]_\rho^{\perp\perp}$. We take $\pi \in [\![A_i]\!]_\rho^\perp$ and show $\{...\ l_i = v_i\sigma\ ...\}.l_i * \pi \in \perp\!\!\!\perp$. We then take one step of reduction and show $v_i\sigma * \pi \in \perp\!\!\!\perp$. It remains to show that $v_i\sigma \in [\![A_i]\!]_\rho^{\perp\perp}$. By induction hypothesis $v_i\sigma \in [\![A_i]\!]_\rho$ hence we can conclude using lemma 7.

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

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

($+_e$) We need to show $case_{v\sigma}\ [C_i[x] \to t_i\sigma]_{i=0}^n \in [\![B]\!]_\rho^{\perp\perp}$. By induction hypothesis $v\sigma \in [\![[C_i of A_i]_{i=0}^n]\!]_\rho$ which means that there is $0 \leq i \leq n$ and $w \in [\![A_i]\!]_\rho$ such that $v\sigma = C_i[w]$. We take $\pi \in [\![B]\!]_\rho^\perp$ and show $case_{C_i[w]}\ [C_i[x] \to t_i\sigma]_{i=0}^n * \pi \in \perp\!\!\!\perp$. We then take a reduction step and show $t_i\sigma[x := w] * \pi \in \perp\!\!\!\perp$. It remains to show that $t_i\sigma[x := w] \in [\![B]\!]_\rho^{\perp\perp}$. 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 σ realizes $\Gamma[\rho]$, $w \in [\![A_i]\!]_\rho$ and $C_i[w] \equiv v\sigma$ by reflexivity.

($\in_i$) We need to show $v\sigma \in [\![v\sigma \in A]\!]_\rho$. By definition we need to show that $v\sigma \in [\![A]\!]_\rho$ 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 $t\sigma \in [\![A]\!]_\rho^{\perp\perp}$. By induction hypothesis $t\sigma \in [\![u \in A]\!]_\rho^{\perp\perp}$ hence we can conclude using lemma 8 since $[\![u \in A]\!]_\rho \subseteq [\![A]\!]_\rho$.

($\restriction_i$) We need to show $t\sigma \in [\![A \restriction u_1 \equiv u_2]\!]_\rho^{\perp\perp}$. By hypothesis $u_1\rho\sigma \equiv u_2\rho\sigma$ hence $[\![A \restriction u_1 \equiv u_2]\!]_\rho = [\![A]\!]_\rho$. By lemma 7 it is enough to show $t\sigma \in [\![A]\!]_\rho^{\perp\perp}$ which is exactly the induction hypothesis.

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

($v\uparrow$) Direct consequence of lemma 7.

($v\downarrow$) Direct consequence of theorem 4.

($\equiv_{v,v,l}$) We need to show that $v\sigma[x := w_1\rho\sigma] \in [\![A]\!]_\rho$. 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 $[\![A]\!]_\rho$ is closed under $(\equiv)$ it is enough to show $v\sigma[x := w_2\rho\sigma] \in [\![A]\!]_\rho$ 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). ■

Remark. *For the sake of simplicity we fixed a pole $\perp\!\!\!\perp$ at the beginning of the current section. However, many of the properties presented here (including the adequacy lemma) remain valid with similar poles. We will make use of this fact in the proof of the following theorem.*

Theorem 7. (Safety) *Let A be a formula, Γ be a context and ρ be a valuation such that $\Gamma[\rho]$ is a closed context and $A[\rho]$ is a closed formula. We also require that $A[\rho]$ is pure (i.e. does not contain any $\_ \Rightarrow \_$). If σ is a substitution realizing $\Gamma[\rho]$ and t is a term such that $\Gamma \vdash t : A$ then for every stack $\pi \in [\![A]\!]_\rho^\perp$ there is a value $v \in [\![A]\!]_\rho$ and $\alpha \in \mathcal{V}_\mu$ such that $t\sigma * \pi \twoheadrightarrow^* v * \alpha$.*

Proof: We do a proof by realizability using the pole $\perp\!\!\!\perp_A = \{p \in \Lambda \times \Pi \mid p \twoheadrightarrow^* v * \alpha \ \wedge\ v \in [\![A]\!]_\rho\}$. It is well-defined as $A[\rho]$ is pure and hence $[\![A]\!]_\rho$ does not depend on the pole. Using the adequacy lemma (theorem 6) with $\perp\!\!\!\perp_A$ we obtain that $t\sigma \in [\![A]\!]_\rho^{\perp\perp}$. Hence for every stack $\pi \in [\![A]\!]_\rho^\perp$ we have $t\sigma * \pi \in \perp\!\!\!\perp_A$. ■

Theorem 8. (Consistency) *There is no t such that $\vdash t : \perp$.*

Proof: Let us suppose that $\vdash t : \perp$. Using the adequacy lemma (theorem 6) we obtain that $t \in [\![\perp]\!]^{\perp\perp}$. Since $[\![\perp]\!] = \emptyset$ we know that $[\![\perp]\!]^\perp = \Pi$ by definition. Now using theorem 5 we obtain that $[\![\perp]\!]^{\perp\perp} = \emptyset$. This is a contradiction. ■

III. Further Developments

The model presented in the previous sections is intended to be used as the basis for the design of *PML*. A first prototype (with a different theoretical foundation) was implemented by Christophe Raffalli [20]. Strong from this experience, the design of a new version of the language with a clean theoretical basis can now be undertaken. The core of the system will consist in three independent components: a type-checker, a termination checker and a decision procedure for program equivalence.

A. A partial type-checking algorithm

Working with a Curry style language has the disadvantage of making type-checking undecidable. While most proof systems avoid this problem by switching to Church style, it is possible to use heuristics making most Curry style programs that arise in practice directly typable. Christophe Raffalli implemented such a system [21] and from his experience it would seem that very few help from the user is required in general. In particular, if a term is typable then it is possible for the user to provide hints (e.g. the type of a variable) so that type-checking may succeed. This can be seen as a kind of completeness.

Proof assistants like Coq [1] or Agda [2] both have decidable type-checking algorithms. However these systems provide mechanisms for handling implicit arguments or meta-variable which introduce some incompleteness. This does not make these system any less usable in practice. We conjecture that going even further (i.e. full Curry style) provides a similar user experience.

B. Termination checking and inductive types

In order to obtain a practical programming language we will need support for recursive programs. For this purpose we plan on adapting Pierre Hyvernat's termination checker based on size change termination [22] that has been used for the first implementation of *PML*.

The type system will also be extended with inductive types [23], [24]. They can be added to the system in the form of fixpoints $\mu X\ A$ and $\nu X\ A$. The corresponding typing rules can be proved safe without difficulty.

C. Deciding program equivalence

The type system given in figure 4 does not provide any way of discharching an equivalence from the context. As a consequence the truth of an equivalence cannot be used. To address this problem the following rule is necessary.

$$\frac{\Gamma, u_1 \equiv u_2 \vdash t : A \qquad Eq_\Gamma(u_1, u_2)}{\Gamma \vdash t : A}\ Eq$$

The right premise $Eq_\Gamma(u_1, u_2)$ can be proved if and only the procedure deciding equivalence is able to show $u_1 \equiv u_2$ in context Γ. Such a procedure can be easily implemented using Knuth-Bendix algorithm, provided that we are able to extract a set of equational axioms from the definition of $(\equiv)$. For instance it is easy to show that $(\lambda x\ t)v \equiv t[x := v]$ for every term t and value v. Of course many other equivalences need to be derived to obtain a usable system.

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 who suggested the encoding of the dependent product type using relativized quantification à la Krivine. Thank you also to Pierre Hyvernat, Tom Hirschowitz and the anonymous reviewers for their very helpful comments.

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] T. Coquand and G. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, no. 2-3, pp. 95–120, Feb. 1988.

[4] P. Martin-Löf, "Constructive mathematics and computer programming," in *Logic, Methodology and Philosophy of Science VI*, 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.

[5] A. K. Wright, "Simple imperative polymorphism," in *LISP and Symbolic Computation*, 1995, pp. 343–356.

[6] M. Tofte, "Type inference for polymorphic references," *Inf. Comput.*, vol. 89, no. 1, pp. 1–34, Sep. 1990.

[7] 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.

[8] 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.

[9] X. Leroy, "Polymorphism by name for references and continuations," in *20th symposium Principles of Programming Languages*. ACM Press, 1993, pp. 220–231.

[10] A. K. Wright and M. Felleisen, "A syntactic approach to type soundness," *Inf. Comput.*, vol. 115, no. 1, pp. 38–94, 1994.

[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] 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.

[13] R. Harper and M. Lillibridge, "ML with callcc is unsound," Jul. 1991. [Online]. Available: http://www.seas.upenn.edu/∼sweirich/types/archive/1991/msg00034.html

[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] J. 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.

[16] G. Munch-Maccagnoni, "Focalisation and classical realisability," in *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL*, 2009, pp. 409–423.

[17] A. Miquel, "Le calcul des constructions implicites : Syntaxe et smantique," Ph.D. dissertation, Université Paris VII, 2001.

[18] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas, "PVS: combining specification, proof checking, and model checking," in *Computer-Aided Verification, CAV '96*, ser. Lecture Notes in Computer Science, R. Alur and T. A. Henzinger, Eds., no. 1102, 1996, pp. 411–414.

[19] J. Krivine, "A call-by-name lambda-calculus machine," *Higher-Order and Symbolic Computation*, vol. 20, no. 3, pp. 199–207, 2007.

[20] C. Raffalli, *The PML programming language*, LAMA - Université Savoie Mont-Blanc, 2012. [Online]. Available: http://lama.univ-savoie.fr/tracpml/

[21] ——, "A normaliser for pure and typed λ-calculus," 1996. [Online]. Available: http://www.lama.univ-savoie.fr/∼raffalli/normaliser.html

[22] P. Hyvernat, "The size-change termination principle for constructor based languages," *Logical Methods in Computer Science*, vol. 10, no. 1, 2014.

[23] C. Raffalli, "L'arithmétiques fonctionnelle du second ordre avec points fixes," Ph.D. dissertation, Université Paris VII, 1994.

[24] N. P. Mendler, "Recursive types and type constraints in second-order lambda calculus," in *Proceedings of the Symposium on Logic in Computer Science (LICS) 1987*, 1987, pp. 30–36.