miniPML: classical logic and extensional choice on individuals

CHRISTOPHE RAFFALLI

We present a logical system, called *miniPML*, which is an extension of HOL with the Curry-Howard correspondence and a richer notion of individuals allowing in the base sort ι higher-order functionals over natural numbers (in HOL, individuals are in general limited to natural numbers). Propositions are used as types to classify individuals, following the *propositions-as-types* paradigm. We show that miniPML allows both classical logic and the extensional axiom of choice. The axiom of choice is limited to types i.e. proposition and can not be applied to choose predicates or elements of higher-order sorts. It is still a strong axiom as types in miniPML can represent higher-order functionals over the natural numbers. To our knowledge, this is the first system with any form of extensional axiom of choice. We also construct for any type a predicate that is a well-ordering on that type.

INTRODUCTION

Context

The Curry-Howard correspondence [10] consists in two paradigms:

- *proofs-as-programs*: a proof of $A \Rightarrow B$ is considered as a program building a proof of A from a proof of B.
- *propositions-as-types*: a proposition is the type of its proof.

This is natural for a constructive logic, surprisingly this can be extended to classical logic [3, 6, 8, 20] and even to the axiom of dependent choice, either using bar recursion [4, 25] or Krivines's classical realizability [12, 13]. Using bar recursion, Krivine was able to give a computational content to the well ordering of reals and as a consequence the continuum hypothesis [14].

It is important to understand that a proof of $\exists x \in \mathbb{N}, P(x)$ using classical logic or the dependent choice will not in general be a program computing x. However, this can be made true if P is a decidable property [11, 22]. The obtained program is in general more interesting than just an enumeration of all natural numbers. This may have some practical applications to help discover new algorithms hidden in non-constructive proofs. In Krivine's work, this also leads to new model of ZF [15]. Extending Curry-Howard correspondence to stronger axioms in various systems is a fruitful and challenging task.

In fact, using classical realizability, as remarked in [24], Krivine does more that the dependent choice: it does *non extensional choice*. Let us explain this further, using a formalism in HOL as in [24].

The non extensional axiom of choice establishes the existence of a choice predicate $C_P(X)$ such that the three following formulas have computational contents:

- $\forall P, \forall X, P(X) \Rightarrow \exists Y, C_P(Y)$ we choose Y,
- $\forall P, \forall X, C_P(X) \Rightarrow P(X)$ it has the intended property and
- $\forall P, \forall X, \forall Y, C_P(X) \Rightarrow C_P(Y) \Rightarrow X = Y$ we constantly do the same choice

where P is a unary predicate. In [22], we show that Krivine realizes the above axiom for X of any sort. However, this is not extensional because we do not know that C_P and C_Q do the same choice for two equivalent predicates P and Q.

The extensional axiom of choice replaces the last property with:

- $\forall P, \forall Q, (\forall X, P(X) \Leftrightarrow Q(X)) \Rightarrow \forall X, \forall Y, C_P(X) \Rightarrow C_Q(Y) \Rightarrow X = Y$

We claim that in HOL, this is the most general axiom of choice if we do not use any function symbol (there is another way of expressing the choice using Hilbert's epsilon).

Important remark: usually, the axiom of choice is classified by looking at the parameters of P for which we are extensional. In Krivine's work, the countable choice uses $P(X) = P_1(n, X)$ where n is a natural and we obtain a functional predicate that chooses X from n. But the choice is not extensional in the other parameters of P as this would give the general extensional axiom of choice.

In our system, the choice will be extensional over P itself, which implies that it is functional for any selection of parameters in P even up to an equivalence relation if P is compatible with that equivalence relation. But unfortunately we will require that X is a typed individual.

If we were in HOL, this would mean X is a natural number and the axiom would then be derivable by choosing the least X such that $P(X)$. However, in PML, individuals contain arbitrary arithmetic functionals (naturals, function from naturals to naturals, etc ...).

Our axiom of choice can be stated as:

- $\forall A, \forall P, \forall X \in A, P(X) \Rightarrow \exists Y \in A, C_P(Y)$
- $\forall A, \forall P, \forall X \in A, C_P(X) \Rightarrow P(X)$
- $\forall A, \forall P, \forall Q, (\forall X \in A, P(X) \Leftrightarrow Q(X)) \Rightarrow \forall X \in A, \forall Y \in A, C_P(X) \Rightarrow C_Q(Y) \Rightarrow X = Y$

where A is a type, i.e. a proposition like $\mathbb{N}$, $\mathbb{N} \Rightarrow \mathbb{N}$, ... and P is a predicate over A. From the view point of HOL, we have the extensional choice only for typed individuals, but our individuals are rich enough to develop a lot of mathematics. We think our result is a major progress, as this axiom is fully extensional and not limited to choosing a natural number.

Toward our solution for extensionality

Our method is very near to Krivine's. But it uses a quite non standard system featuring call-by-value, *membership types* and untyped programs as individual of the logic. We will now informally explain why extensionality naturally may lead to those ingredients.

First, in all the work using realizability, the axiom of choice is derived from a simpler axiom which is similar to (there are variations, for instance taking the contrapositive):

$$\forall X, P(X) \Rightarrow \exists n \in \mathbb{N}, P(\Psi(P, n))$$

Where Ψ is a function symbol that recovers an X such that $P(X)$ holds from a natural number. In previous works, this natural number was a code for a proof of $P(X)$ (or a code for a stack which is a counter proof when using the contrapositive of the axiom). The standard axiom of choice is derived using the minimum principle to choose the least integer such that $P(\Psi(P, n))$ holds. The problem to have extensionality is to deal with equivalent predicates P and Q and transform numbers for $P(\Psi(P, n))$ into numbers for $Q(\Psi(Q, n))$. This is rather complicated, and we did not succeed.

Our main idea is to number X itself, which means that we do not have to change the number when changing the predicate for an equivalent one. However, this implies two things:

- X should have a type and live in the world of proofs (i.e. programs) to be accessible to programs that need to number it.
- Our proofs should allow to represent higher-order functions not to be limited to choosing naturals.

To achieve this, our system has the *proofs-as-objects* paradigm, meaning that we will prove properties of proofs i.e. programs thanks to the Curry-Howard correspondence. This gives the basic ingredients of PML:

- An untyped programming language similar to ML in the case of PML and to Gödel's system T in the case of miniPML. Programs in this language will be the individuals of our logic. They will include natural numbers and functions.
- A language of expressions similar to HOL to express types (i.e. propositions, thanks to the propositions-as-types paradigm), predicates over types, etc.
- Expressions should at least include dependent products and sums otherwise we would be limited to a programming language like ML and not a logical system. Dependant products and sums are the usual way to have the *proofs-as-objects* paradigm. Indeed the elimination rule for dependent product is

$$\frac{t : \forall X \in A, P(X) \qquad u : A}{t\,u : P(u)}$$

  and we see that is allows programs to move inside logical expressions as u appears both to the left and to the right of the column.

There remain problems with those ideas. As we said, Krivine realizes the axiom $\forall X, P(X) \Rightarrow \exists n \in \mathbb{N}, P(\Psi(P, n))$. If we want to really give a number to X, we should drop the dependency over P in Ψ. Therefore, we will realize a formula like:

$$\forall P, \forall X \in A, P(X) \Rightarrow \exists n \in \mathbb{N}, \Psi(n) \in A \wedge P(\Psi(n)) \tag{1}$$

or even a simpler one:

$$\forall X \in A, \exists n \in \mathbb{N}, \Psi(n) \equiv X \tag{2}$$

In the above formula $t \equiv u$ denotes equality over individuals (i.e. programs). It is some kind of observational equivalence of programs, but it is provably equivalent to Leibniz equality. We use a different notation to emphasis the fact that t and u are individuals when we write $t \equiv u$.

The axiom (2) may seem contradictory as it implies that Ψ is a surjection from $\mathbb{N}$ to any type A. The first version (1) also implies this by taking $P(X) = (X \equiv Y)$. But Ψ is not itself of sort ι, it is not a program. It is of sort $\iota \to \iota$. In miniPML, the type $\mathbb{N} \Rightarrow \mathbb{N}$ whose elements are of sort ι is smaller than the sort $\iota \to \iota$ which is itself smaller than the functional predicates of sort $\iota \to \iota \to o$. There are no typed individuals in the type $\mathbb{N} \Rightarrow \mathbb{N}$ which can compute Ψ. In miniPML, $\mathbb{N} \Rightarrow \mathbb{N}$ is fortunately not equipotent to $\mathbb{N}$ if we require the bijection to live in the type $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. But, in some sense, (2) implies that the sort $\iota \to \iota$ *knows* that we have a countable model for the sort ι, this is no contradiction. In contrast, in ZF, the comprehension axiom allows to recover a set from any predicate and (2) would immediately lead to a contradiction.

This surprising situation can be useful! In the last section of this article, we actually build a predicate giving a injection from a type A to $\mathbb{N}$ which is an inverse of Ψ and we use this indexing to build a predicate that is a well-ordering on A.

However, some restriction are needed for (2) not to imply a contradiction. We must not be able to find a program χ_1 of type $\forall X \in A, \mathbb{N}$ with $\forall X \in A, \Psi(\chi_1\, X) \equiv X$ derivable[1]. A naive (and contradictory) type

[1]In this formula $\chi_1\, X$ denotes the application of the two terms χ_1 and X of sort ι which is itself of sort ι.

system with dependent sums inhabited by pairs would allow this, using the standard rules available in most type theories.

A first possibility is to encode existential by $(\exists X \in A, P(X)) = \forall K, (\forall X \in A, P(X) \Rightarrow K) \Rightarrow K$, which is standard in classical realizability. We think our ideas could allow to realize (2) in a modified version of miniPML using call-by-name. As we explain below this will require other modifications of miniPML.

We now have to justify our use of call-by-value in miniPML. There are some discussions in the programming language community about the choice of call-by-value versus call-by-name evaluation (more precisely lazy evaluation) and most programming languages available use call-by-value. But there are also a few logical reasons to consider call-by-value.

- We have to use storage operators [11] to actually compute natural numbers in classical logic with call-by-name.
- There is an alternative, using a type for *native* natural numbers, but it must only occur in the left side of an implication. The type $\mathbb{N} \Rightarrow \mathbb{N}$ is no more a valid type. Addition will have type $\mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \forall X, (\mathbb{N} \Rightarrow X) \Rightarrow X$. The same will happens if we want a type $A \times B$ whose elements are native pairs.
- As we have the proofs-as-individuals paradigm, it would be nice to have a *membership type* $x \in A$ in the logic, reflecting the typing relation. Membership types are similar to singleton types which have already been considered [2, 9]. They can be used to encode dependent products and sums from unbounded quantifications, implications and products with $\forall X \in A, P(X)$ defined as $\forall X, X \in A \Rightarrow P(X)$ and $\exists X \in A, P(X)$ as $\exists X, X \in A \times P(X)$. Unfortunately, we do not know how to justify the rules for this connective, even when it only appears in the left side of an implication.
- To get simpler program, we would like to have equality with no computational content using *restriction types* [21] denoted $A \mid t \equiv u$. This type is interpreted as the type A if the equation holds and the *empty type* $\forall X, X$ otherwise. In classical call-by-name realizability, again, we do not know how to justify rules for such a type.

Let us explain the problem further with membership type. It should have three rules:

$$\frac{\vdash t : A}{\vdash t : t \in A} \qquad \frac{\vdash t : t' \in A}{\vdash t : A} \qquad \frac{x \equiv t, x : A \vdash u : B}{x : t \in A \vdash u : B}$$

In classical realizability, a type A is interpreted by its falsity value: a set of stacks $|A|^{\perp}$. From $|A|^{\perp}$ we construct the truth value $||A||$ which is the set of all terms that *behave* well when applied to stacks in $|A|^{\perp}$. Good behaviours is defined using an *orthogonality* relation: we say that a term behaves well when applied to a stack when it is orthogonal to that stack.

Following this method, $|t \in A|^{\perp}$ should be a set of stacks which only accept t or terms equivalent to t in some sense when t is in A. The natural definition would be all stacks orthogonal to t if $t \in ||A||$ and all stacks if $t \notin ||A||$. But it does not justify the third rule because knowing that $u \in ||t \in A||$ does not imply $t \equiv u$. A similar problem arises with restriction $u \in ||A \mid t_1 \equiv t_2||$ does not force the equality $t_1 \equiv t_2$ to be true. We do not know how to solve these problems.

In contrast, call-by-value will support all the above features (types for native natural numbers, pairs, existential, membership and restriction). The price to pay is the so called *value restriction* further explained below. When using call-by-value a formula has three levels of semantic:

- $|A|^{o}$ a set of values (i.e. evaluated programs in some sense),
- $|A|^{\perp}$ the orthogonal of $|A|^{o}$, the set of stacks orthogonal to all values in $|A|^{o}$ and
- $||A||$ the orthogonal of $|A|^{\perp}$, the set of terms orthogonal to all stacks in $|A|^{\perp}$.

The adequacy theorem states that if $\vdash t : A$ then we have $t \in ||A||$ but if t is a value we also must have $t \in |A|^o$. This is stronger in general as $|A|^o \subset ||A||$ but not the opposite.

This requires the so called *value restriction* which was considered for ML in [1] and for call-by-value and classical proofs in [7]. It is needed because bi-orthogonals do not commute with intersections or unions[2]. Thus for each quantifier, one of the two rules must be restricted to value (the ∀-introduction and the ∃-elimination). This is also the case for the first of the three rules for membership type.

Value restriction will then make a difference between $\chi(v)$ and $n + m$ because, contrary to the latter, the former will never be equivalent to some value.

A problem arises: value restriction makes dependent product almost unusable: we can only deduce $P(v)$ from $\forall x \in A, P(x)$ and $v \in A$ when v is syntactically a value. So no way to deduce $P(n + m)$. A solution is to allow replacing terms by values and vice-versa when they are equal. This can be achieved with two rules like:

$$\frac{\vdash \mathrm{t} : \mathrm{A} \qquad \vdash \mathrm{t} \equiv \mathrm{v}}{\vdash \mathrm{v} : \mathrm{A}} \qquad\qquad \frac{\vdash \mathrm{v} : \mathrm{A} \qquad \vdash \mathrm{t} \equiv \mathrm{v}}{\vdash \mathrm{t} : \mathrm{A}}$$

The second rule is easy to justify from $|A|^o \subset ||A||$. But not the first one which requires $v \in ||A||$ implies $v \in |A|$ for any value v. Fortunately a model with this property is available in Lepigre's work [16, 18]. Moreover, we are able to prove in miniPML that functions like addition always returns values.

With all this features, we think miniPML is a rich type system suitable for developing large programs in practice and featuring both classical logic and an extensional choice limited to choosing programs. PML is even richer, with support for both inductive and co-inductive types. In fact miniPML, introduced in this paper, is somehow the minimal system where we can explain our realization of the axiom of choice.

Contents

We first present (section 1) the untyped calculus, its reduction and observational equivalence. Next (section 2) we give the syntax of types and their semantic. The section 3 gives the typing rules and the adequacy (theorem 3.1) of the realizability semantic. Then, theorem 3.2 and corollary 3.3 explain when and how we can safely run a program to get a useful result. We also get the consistency of miniPML.

The axiom of choice is introduced in section 4. We first show how to realize the small axiom (2) above and then derive the extensional choice from it using the rules of miniPML. Finally (section 5), as applications, we give a construction of quotient types by choosing an element in each equivalent class. We also give a construction of a predicate well-ordering any type.

Except the realization of (1), all proofs in these two last sections have been machine checked with the latest version of PML. The code is given in appendix.

Note about PML: It started in 2007 [23] and was completely rewritten with solid theoretical foundation thanks to our collaboration with Rodolphe Lepigre [16–19]. it is a large system, available from github: `https://github.com/rlepigre/pml`. It features inductive and co-inductive types, subtyping, using the size change principle to prove program termination, etc. For simplicity, in this paper we decided not to use full PML and designed *miniPML* which we fully present with the complete construction of its semantic. Therefore, this paper should be self-content.

[2]However, the orthogonal of a union is the intersection of the orthogonals, which makes universal quantifications work in call-by-name realizability.

1 TERMS, REDUCTION AND EQUIVALENCE

The set Λ of terms, Λ_ι of values and Π of stacks are defined by the following BNF, using natural numbers, value variables in Λ_ι^v, term variables in Λ^v, stack variables in Π^v and a set F of total functions from natural numbers to natural numbers of various arities.

$$\begin{array}{rcl}
\Lambda_\iota & ::= & x \mid \lambda x.t \mid (v_1,\dots,v_n) \mid n \\
\Lambda & ::= & a \mid v \mid t\,u \mid v.n \mid \mu\alpha.t \mid [\pi]t \mid \varphi a.t \mid v[t,u] \mid f(v_1,\dots,v_n) \mid \chi(v) \mid \\
 & & \delta(v,w,t) \mid \phi(v,t) \mid \rho(v,t) \mid \nu(v,t) \\
\Pi & ::= & \varepsilon \mid \alpha \mid [t-]\pi \mid [-v]\pi
\end{array}$$

with the following convention used throughout this paper (with possible added subscript or apostrophe):

- x, y, z are value variables in Λ_ι^v,
- a, b are term variables in Λ^v,
- α, β are stack variables in Π^v,
- v, w are values in Λ_ι,
- r, s, t, u are terms in Λ,
- π, π' are stacks in Π,
- n, p, q are natural numbers in $\mathbb{N}$ and
- f is a function from $\mathbb{N}^n$ to $\mathbb{N}$ in the set F, always used with the correct number of arguments. If a function in F as a usual infix notation we will use this notation in terms. For instance, if addition is in F, we will consider that $v + w$ is a term.

The constructions $\lambda x.t$, $\mu\alpha.t$, $Ya.t$ bind respectively value, stack and term variables in terms. As usual, we work up to α-equivalence and we use capture free substitutions written $[x := v]$, $[a := t]$ or $[\alpha := \pi]$. Application and composition of substitutions will be written with juxtaposition: $t\sigma$ denotes the application of the substitution σ to t and $\sigma\tau$ is the composition of two substitutions with $t(\sigma\tau) = (t\sigma)\tau$. We will use the same conventions for quantifiers in expressions in the next section.

The usual call-by-value λ-calculus is present in our calculus, with value variables, λ-abstraction ($\lambda x.t$) and application ($t\,u$). The μ-variables $\alpha, \beta, \dots$, the constructions $\mu\alpha.t$ and $[\alpha]t$ are those of Parigot's $\lambda\mu$-calculus [20]. The call-by-value machine requires two ways to push elements on the stack: pushing computed arguments of functions with $[-v]\pi$ and pushing terms while their arguments are being computed with $[t-]\pi$.

A fixpoint combinator $\varphi a.t$ is also present in our calculus and require terms variables a in the calculus. We have $(v_1,\dots,v_n)$ and $v.n$ to build and project tuples and n, $f(v_1,\dots,v_n)$ and $v[t,u]$ to build and test natural numbers. The construction $\chi(v)$ is the *clock* instruction related to the axiom of choice.

The term constructors $\delta(v,w,t), \phi(v,t), \rho(v,t)$ and $\nu(n,t)$ will not have any typing rule, they are just added to have a strong enough observational equivalence to have the desired properties of reduction and equivalence. In an implementation, these constants would not even need to be present. These are further explained with the reduction rules.

We assume that F contains at least the function S for successor, a function *not* exchanging 0 and 1 and two tests $v \le w$ and $v < w$, defined as $\mathit{not}(w \le v)$. Those tests return 1 if the relation is true, 0 if it is false.

The fact that we restrict to values in many cases simplifies a few things, because only application and abstraction interact with the stack. This does not restrict the expressive power of the untyped calculus as

$$
\begin{array}{rcll}
t u * \pi * c & \succ_0 & u * [t-]\pi * c & \\
v * [t-]\pi * c & \succ_0 & t * [-v]\pi * c & \\
\lambda x.t * [-v]\pi * c & \succ_0 & t[x := v] * \pi * c & \\
(v_1, \dots, v_0).k * \pi * c & \succ_0 & v_k * \pi * c & \text{for } 1 \leq k \leq n \\
\varphi a.t * \pi * c & \succ_0 & t[a := \varphi a.t] * \pi * c & \\
m[t, u] * \pi * c & \succ_0 & t * \pi * c & \text{if } m = 1 \\
m[t, u] * \pi * c & \succ_0 & u * \pi * c & \text{if } m = 0 \\
f(m_1, \dots, m_0) * \pi * c & \succ_0 & m * \pi * c \text{if } f(m_1, \dots, m_0) = m & \\
\mu\alpha.t * \pi * c & \succ_0 & t[\alpha := \pi] * \pi * c & \\
[\pi]t * \pi' * c & \succ_0 & t * \pi * c & \\
\chi(v) * \pi * (n, l) & \succ_0 & c * \pi * (n+1, l-1) & \text{if } l > 0 \\
\phi(\lambda x.t, u) * \pi * c & \succ_0 & u * \varepsilon * c & \\
\rho((v_1, \dots, v_0), u) * \pi * c & \succ_0 & u * \varepsilon * c & \\
\nu(n, u) * \pi * c & \succ_0 & u * \varepsilon * c & \text{if } n \in \mathbb{N}
\end{array}
$$

Fig. 1. One step reduction reduction, except δ

we can use β-expansion. For instance if F contains addition, we can write $(\lambda x.\lambda y.(x + y))\, t\, u$ for $t + u$ when t and u are terms. But, this restriction are important regarding the value restriction in the type system.

The symbols Λ^c, Λ^c_ι and Π^c denote respectively the set of closed terms, values and stacks.

Reduction is a binary relation on processes which are elements of $\Lambda^c \times \Pi^c \times (\mathbb{N} \times \overline{\mathbb{N}})$ where $\overline{\mathbb{N}} = \mathbb{N} \cup \infty$. We restrict to closed processes to avoid reduction blocked by free variables. This is not really important but avoid useless cases in proofs, because in the next sections, we only need to consider closed terms, values and stack.

Processes are denoted $t * \pi * c$ where t is the term being evaluated, π the current stack and $c = (t, l) \in \mathbb{N} \times \overline{\mathbb{N}}$ is the current *clock* t and a timeout l possibly limiting the number of application of the clock. In the rule of the clock, we consider that $\infty - 1 = \infty$ when there is no timeout.

One step reduction $(\mathrm{t}_1 * \pi_1 * \mathrm{n}_1 \succ \mathrm{t}_2 * \pi_2 * \mathrm{n}_2)$, term equivalence $(\mathrm{t} \equiv \mathrm{u})$ and convergence $(\mathrm{t}{\downarrow}\mathrm{v})$ needs an inductive definition to have the desired properties.

We start by giving in figure 1 the basic one step reduction $\succ_0$. The three first lines implement call-by-value β-reduction. Next come the rule for projection of tuples, the fixpoint rule and the three rules for natural numbers.

The two rules for $\mu\alpha.t$ and $t * \pi$ correspond to the computational content of classical logic.

The *clock* is read and incremented by the χ instruction. The timeout is used to block the reduction of χ after some application because we need a strict control of the clock in our observational equivalence.

The constants ϕ, ρ and ν allow to recognise λ-abstractions, tuples and natural numbers respectively.

We write R^* for the transitive and reflexive closure of a relation R.

Next, we define by mutual induction over n reduction $\succ_n$, equivalence $\equiv_n$, convergence $\downarrow_n$ and divergence $\uparrow_n$ at level n.

- We define $\overline{\Lambda}^c_\iota = \Lambda^c_\iota \cup \{\chi(v) \mid v \in \Lambda^c_\iota\}$.
- $(\succ_0) \subset (\succ_n)$: basic one-step reduction are present at all levels
- $\delta(v, w, t) * \pi * c \succ_{n+1} t * \pi * c$ if $v \not\equiv_n w$: δ reduces only if the two values are not equivalent at the previous level.
- $t * \pi * c \downarrow_n v$ if $t * \pi * c \succ^*_n v * \varepsilon * c'$ with $v \in \Lambda^c_\iota$ and $c' \in \mathbb{N}$. A process *converges* to a closed value if it reduces to that value with an empty stack.

- $t * \pi * c \downarrow_n \chi(v)$ if $t * \pi * c \succ_n^* \chi(v) * \pi * (n, 0)$ with $v \in \Lambda_\iota^c$ and $\pi' \in \Pi^c$. A process *clock-converges* if it is blocked on the clock instruction with a null timeout. This can only happens if the initial timeout was not ∞.
- $t * \pi * c \uparrow_n$ if there is no $v \in \overline{\Lambda}_\iota^c$ such that $t * \pi * c \downarrow_n v$. A process diverges if we are not in the two previous cases.
- $t \equiv_n u$ if $\forall p < n, t \equiv_p u$ and if for any closed stack $\pi \in \Pi^c$ and any clock $c \in \mathbb{N} \times \overline{\mathbb{N}}$, one of the following three cases holds:

$$\begin{array}{lll}
t * \pi * c \downarrow_n v, & \text{and} & u * \pi * c \downarrow_n w \text{ and } \forall p < n, v \equiv_p w \text{ with } v, w \in \Lambda_\iota^c \\
t * \pi * c \downarrow_n \chi(v) & \text{and} & u * \pi * c \downarrow_n \chi(w) \text{ and } \forall p < n, v \equiv_p w \text{ with } v, w \in \Lambda_\iota^c \\
t * \pi * c \uparrow_n & \text{and} & u * \pi * c \uparrow_n
\end{array}$$

Here are two properties of these relations

LEMMA 1.1.

(1) *$\equiv_n$ is a decreasing in n and $\succ_n$ and $\downarrow_n$ are increasing.*
(2) *If $t * \pi * c \downarrow_n v$ for $v \in \overline{\Lambda}_\iota^c$, then $t * \pi * c \downarrow_p v$ for all $p \geq n$*

PROOF.

(1) The fact that $\equiv_n$ is decreasing is enforced in the definition. Hence $\not\equiv_n$ is increasing, $\succ_n$ is increasing, $\downarrow_n$ is increasing and $\uparrow_n$ is decreasing.
(2) $t * \pi * c \downarrow_n v$ means that $t * \pi * c \succ_n^* v * \varepsilon * c'$ for $v \in \Lambda_\iota^c$ or $t * \pi * c \succ_n^* \chi(w) * \pi' * (k, 0)$. As reduction is deterministic and increases with n, this will still hold for all $p \geq n$ with the same closed value v.$\square$

The idea of this induction to get the proper reduction rule for δ in lemma 1.2.3 is due to Rodolphe Lepigre [16, 18]. We modified it to ensure a stronger property of equivalence 1.2.4.

Now that we have defined reduction, convergence and equivalence for each level $n \in \mathbb{N}$, we define the final notions:

- $t * \pi * c \succ t' * \pi' * c'$ if $\exists n \in \mathbb{N}, t * \pi * c \succ_n t' * \pi' * c'$
- $t * \pi * c \downarrow v$ if $\exists n \in \mathbb{N}, t * \pi * c \downarrow_n v$
- $t * \pi * c \downarrow \chi(v)$ if $\exists n \in \mathbb{N}, t * \pi * c \downarrow_n \chi(v)$
- $t * \pi * c \uparrow$ if $\forall n \in \mathbb{N}, t * \pi * c \uparrow_n$
- $t \equiv u$ if $\forall n \in \mathbb{N}, t \equiv_n u$
- $v \equiv_+ w$ for $v, w \in \overline{\Lambda}_\iota^c$ iff
  - v and w are both values and $v \equiv w$ or
  - $v = \chi(v')$ and $w = \chi(w')$ with $v' \equiv w'$.
- $t * \pi \gg t' * \pi'$ if $\forall c \in \mathbb{N} \times \overline{\mathbb{N}}, t * \pi * c \succ^* t' * \pi' * c$.

The last line defines *clock free* reduction which will be very useful, thanks to lemma 2.3 to prove the correctness of the semantic.

Here are the two fundamental properties of our reduction. Once established, we will not have to consider anymore the indexed relations which are only a technical solution to obtain the properties below.

LEMMA 1.2.

(1) *$(\succ_0) \subset (\succ)$*
(2) *If $p' \succ^* p$ and $p \downarrow v$ for $v \in \overline{\Lambda}_\iota^c$ then $p' \downarrow v$.*
(3) *For v, w values, t a term and π a stack, $\delta(v, w, t) * \pi * c \succ t * \pi * c$ iff $v \not\equiv w$.*

(4) *For t, u terms, $t \equiv u$ implies $\forall\pi \in \Pi^c, \forall c \in \mathbb{N}$, on of the following three cases hold:*

$$\begin{array}{lll} t * \pi * c \downarrow v & \textit{and} & u * \pi * c \downarrow w \textit{ and } v \equiv w \textit{ for } v, w \in \Lambda^c_\iota, \\ t * \pi * c \downarrow \chi(v) & \textit{and} & u * \pi * c \downarrow \chi(w) \textit{ and } v \equiv w \textit{ for } v, w \in \Lambda^c_\iota, \\ t * \pi * c \uparrow & \textit{and} & u * \pi * c \uparrow \end{array}$$

(5) *If for any stack π, we have $t * \pi \gg u * \pi$ then $t \equiv u$.*

Remark, the two first cases in the item 4 above can be written as one, thanks to our notation:

$$t * \pi * c \downarrow v \text{ and } u * \pi * c \downarrow w \text{ and } v \equiv_+ w \text{ for } v, w \in \overline{\Lambda}^c_\iota$$

Proof.

(1) immediate.
(2) $p' \succ^* p$ means that there exists $n_1 \in \mathbb{N}$ such that $p' \succ^*_{n_1} p$ (take the maximum over all level for each one step reductions). $p \downarrow v$ implies $p \downarrow_{n_2} v$ for some $n_2 \in \mathbb{N}$. Hence taking $n = max\, n_1\, n_2$, we have $p' \downarrow_n v$.
(3) $\delta(v, w, t) * \pi * c \succ t * \pi * c$ holds if and only if $\exists n \in \mathbb{N}$, s.t. $\delta(v, w, t) * \pi * c \succ_n t * \pi * c$ holds which is by definition equivalent to $\exists n \in \mathbb{N}, v \not\equiv_n w$, which is the negation of the definition of $v \equiv w$.
(4) $t \equiv u$ implies that $\forall n \in \mathbb{N}, \forall\pi \in \Pi^c, \forall c \in \mathbb{N} \times \overline{\mathbb{N}}$, we are in one the three cases of the definition of $\equiv_n$:

$$\begin{array}{lll} t * \pi * c \downarrow_n v, & \text{and} & u * \pi * c \downarrow_n w \text{ and } \forall p < n, v \equiv_p w \text{ with } v, w \in \Lambda^c_\iota \\ t * \pi * c \downarrow_n \chi(v) & \text{and} & u * \pi * c \downarrow_n \chi(w) \text{ and } \forall p < n, v \equiv_p w \text{ with } v, w \in \Lambda^c_\iota \\ t * \pi * c \uparrow_n & \text{and} & u * \pi * c \uparrow_n \end{array}$$

If we are in the last case for all n, then $t * \pi * c \uparrow$ and $u * \pi * c \uparrow$ holds. We can assume that we are in one of the two first cases for some $n \in \mathbb{N}$. From lemma 1.1.2 we know we remain in that case for all $m \geq n$. Moreover, in both cases, the closed values v and w do not depend upon m. This means we know that $v \equiv w$.
(5) Consequence of the previous item. □

The last part of the proof shows that $t \equiv u$ is a bit more precise than observational equivalence for the relation $\succ$ as it can observe the *level* of convergence.

Lemma 1.3 (extensionality). *Let t be a term with only one free value variable x and v_1 and v_2 be two closed values. If $v_1 \equiv v_2$ then $t[x := v_1] \equiv t[x := v_2]$. Let t be a term with only one free term variables a and u_1 and u_2 be two closed terms. If $u_1 \equiv u_2$ then $t[a := u_1] \equiv t[a := u_2]$.*

Proof. Let us assume $u_1 \equiv u_2$ and $t[x := v_1] \not\equiv t[x := v_2]$. This means we can find a stack π and a clock c such that $t[x := v_1] * \pi * c$ and $t[x := v_2] * \pi * c$ have different behaviour. We have $v_i * [\lambda x.t-]\pi * c \succ^* t[x := u_i] * \pi * c$ for $i = 1, 2$. The stack $[\lambda x.t-]\pi$ and the clock c establishes $u_1 \not\equiv u_2$ leading to a contradiction.

The case for terms is more complicated. A detailed proof can also be found In [16, 18], as we changed both equivalence and reduction we reprove it.

We need to extend reduction to allow one free term variable a. Reduction is blocked if the process is $a * \pi * c$ but also if it is $\delta(v, w, t) * \pi * c$ with a free in v or w as equivalence is only defined for closed terms. We do have $t * \pi * c \succ t' * \pi' * c'$ implies $t[a := u] * \pi[a := u] * c \succ t'[a := u] * \pi'[a := u] * c$. Convergence, equivalence are not used below when a is free. Only reduction needs to be extended.

We assume $u_1 \equiv u_2$ and $t[a := u_1] \not\equiv t[a := u_2]$ and define $\sigma_i = [x := u_i]$ for $i = 1, 2$.

We can find a stack π_0 and a clock c_0 which distinguishes $t\sigma_1$ and $t\sigma_2$. Without loss of generality, we may assume that $t\sigma_1 * \pi_0 * c_0 \downarrow v_1$ with $v_1 \in \overline{\Lambda}^c_\iota$.

We define a sequence $(s_n, t_{1,n}, t_{2,n}, \pi_n, c_n)$ with

- $s_0 = t_{1,0} = t_{2,0} = t$ and for $n > 0$
- $t_{1,n-1} * \pi_{n-1} * c_{n-1} \succ^* s_n * \pi_n * c_n$, a process blocked because of a.
- with $s_n = a$ and $t_{i,n} = u_i$ for $i = 1, 2$
- or $s_n = \delta(v, w, t)$ with a free in v or w and $t_{i,n} = \delta(v\sigma_i, w\sigma_i, t)$ for $i = 1, 2$ (no substitution for t).

This sequence is not defined for $n + 1$ and stops if $t_{1,n} * \pi_n * c_n$ does not reduce to a process in the form $a * \pi * c$ nor $\delta(v, w, t) * \pi * c$ with a free in v or w.

We have immediately $s_n\sigma_i = t_{i,n}\sigma_i$ for $i = 1, 2$.

For $n > 0$, as $t_{1,n}$ is not a nor of the form $\delta(v, w, t)$ with a free in v and w, each reduction $t_{1,n} * \pi_n * c_n \succ^* u_{n+1} * \pi_{n+1} * c_n$ has at least one step of reduction. For $n = 0$, if $t = a$ or $t = \delta(v, w, t)$ with a free in v or w, there is no reduction and $s_1 = s_0 = t$.

From this and $t_{1,n}\sigma_1 * \pi_n\sigma_1 * c_n \succ^* t_{1,n+1}\sigma_1 * \pi_{n+1}\sigma_1 * c_n$ we deduce that the sequence is finite otherwise $t\sigma_1 * \pi_0 * c_0$ diverges. Thus, we have $t_{1,n}\sigma_1 * \pi_n\sigma_1 * c_n \downarrow v_1$ for all n Let us name N the last index for which the sequence is defined.

We have $t_{1,N} * \pi_N * c_N \downarrow v_0$ with $v_0\sigma_1 = v_1$ Indeed, if $t_{1,N} * \pi_N * c_N \uparrow$ then $t_{1,N}\sigma_1 * \pi_N\sigma_1 * c_N\sigma_1 \uparrow$ which is not possible and if $t_{1,N} * \pi_N * c_N \downarrow v_0$, then $t_{1,N}\sigma_1 * \pi_N\sigma_1 * c_N \downarrow v_0\sigma_1$ and $v_0\sigma_1 = v_1$ because reduction is deterministic.

We define $v_{2,N+1} = v_0\sigma_2$ and from the same lemma with values, we have $v_1 \equiv_+ v_{2,N+1}$.

Now we prove by reverse induction on n (first case is N) that $s_n\sigma_2 * \pi_n\sigma_2 * c_n \downarrow v_{2,n}$ for some $v_{2,n} \in \overline{\Lambda}^c_\iota$ with $v_{2,n} \equiv_+ v_1$. This will end the proof as the case $n = 0$ gives $t\sigma_2 * \pi_0 * c_0 \downarrow v_{2,0}$ with $v_{2,0} \equiv_+ v_1$ contradicting the fact that π_0 and c_0 are distinguishing $t\sigma_1$ and $t\sigma_2$.

We have $t_{1,n} * \pi_n * c_n \succ^* p$ with $p = s_{n+1} * \pi_{n+1} * c_{n+1}$ when $n < N$ or $p = v_0 * \varepsilon * c_{N+1}$ if $n = N$. In both cases, either from induction hypothesis or from above, we have $p\sigma_2 \downarrow v_{2,n+1}$ with $v_{2,n+1} \equiv_+ v_1$. This gives $t_{1,n}\sigma_2 * \pi_n\sigma_2 * c_n \downarrow v_{2,n+1}$

Let us now distinguish three cases. First, if $n = 0$, $s_0 = t_{1,0} = t_{2,0}$ and $t_{1,0}\sigma_2 * \pi_0\sigma_2 * c_0 \downarrow v_{2,1}$ gives immediately the result with $v_{2,0} = v_{2,1} \equiv_+ v_1$.

Second, if $n > 0$, $s_n = a$ and $t_{i,n} = u_i$. Then, we have $u_1 * \pi_n\sigma_2 * c_n \succ^* p\sigma_2 \downarrow v_{2,n+1}$. But from $u_1 \equiv u_2$, we find $v_{2,n}$ such that $u_2 * \pi_n\sigma_2 * c_n \downarrow v_{2,n}$ with $v_{2,n} \equiv_+ v_{2,n+1} \equiv_+ v_1$.

In the last case, we have $s_n = \delta(v, w, t)$ with a free in v or w and $t_{i,n} = \delta(v\sigma_i, w\sigma_i, t)$ As the reduction continues, $t_{1,n} * \pi_n * c_n$ is not blocked and this implies that $v\sigma_1 \not\equiv w\sigma_1$. As we have $v\sigma_1 \equiv v\sigma_2$ and $w\sigma_1 \equiv w\sigma_2$ from the same lemma with values, we conclude $v\sigma_2 \not\equiv w\sigma_2$. Thus, we have $t_{2,n} * \pi_n * c_n \succ t * \pi_n * c_n \succ^* p$ and we have $u_n\sigma_2 * \pi_n\sigma_2 * c_n = t_{2,n}\sigma_2 * \pi_n\sigma_2 * c_n \succ t\sigma_2 * \pi_n\sigma_2 * c_n \succ^* p\sigma_2 \downarrow v_{2,n+1}$. Hence, we can take $v_{2,n} = v_{2,n+1} \equiv_+ v_1$. □

The following lemma ensure that equivalent values are similar enough.

Lemma 1.4. *For any $v \in \Lambda^c_\iota$ and $n \in \mathbb{N}$, if $v \equiv n$, then $v = n$.*

For any $v \in \Lambda^c_\iota$ and $t \in \Lambda$ with one free variable x, if $v \equiv \lambda x.t$, then $v = \lambda x.u$ and for all $w \in \Lambda^c_i$, $t[x := w] \equiv u[x := w]$.

For any $v, w_1, \dots, w_n \in \Lambda^c_\iota$ if $v \equiv (w_1, \dots, w_n)$, then $v = (v_1, \dots, v_n)$ with $w_i \equiv v_i$ for $1 \leq i \leq n$.

Proof. For natural numbers, assume $v \equiv n$. If v is not a natural number, consider the stack $\pi = [\lambda x.\nu(x, v_0)-]\varepsilon$ with $v_0 \in \Lambda^c_\iota$ and any clock $c \in \mathbb{N}$. We have $n*\pi*c \succ^* v_0*\varepsilon*c \downarrow v_0$ and $v*\pi*c \succ^* \nu(v, v_0)*\varepsilon*c$ which can not be further reduced. Hence we have $v \not\equiv n$.

If $v = m \neq n$ with $m \in \mathbb{N}$, consider the stack

$$\pi = [\lambda z.(\lambda x.\lambda y.x[y[v_0, \delta(0, 0, v_0)], \delta(0, 0, v_0)])(z \le n)(n \le z)-]\varepsilon$$

with $v_0 \in \Lambda^c_\iota$ and any clock $c \in \mathbb{N}$. We have

$$n * \pi * c \succ^* 1[1[v_0, \delta(0, 0, v_0)], \delta(0, 0, v_0)] * \varepsilon * c \succ^* v_0 * \varepsilon * c \downarrow v_0$$

and

$$m * \pi * c \succ^* n_1[n_2[v_0, \delta(0, 0, v_0)], \delta(0, 0, v_0)] * \varepsilon * c \succ^* \delta(0, 0, v_0) * \varepsilon * c \uparrow$$

because either $n_1 = 0$ or $n_2 = 0$ and this process can not be further reduced. Hence, $m \not\equiv n$.

For abstraction, assume that $v \equiv \lambda x.t$. If v is not an abstraction, the stack $[\lambda x.\phi(x, v_0)-]\varepsilon$ with $v_0 \in \Lambda^c_\iota$ and any clock c yields a contradiction. So we know $v = \lambda x.u$ for some u. Now let us take $w \in \Lambda^c_i$, we have to show that $u[x := w] \equiv t[x := w]$. We assume $u[x := w] \not\equiv t[x := w]$ and take $\pi \in \Pi^c$ and $c \in \mathbb{N} \times \overline{\mathbb{N}}$ witnessing that. The stack $[-w]\pi$ with the same clock contradicts $v \equiv \lambda x.t$.

For product, take $w = (w_1, \ldots, w_n)$ with $w_1, \ldots, w_n \in \Lambda^c_\iota$ and $v \equiv w$. First we use the stack $[\lambda x.\rho(x, v_0)-]*\varepsilon$ with $v_0 \in \Lambda^c_\iota$ to establish that $v = (v_1, \ldots, v_m)$. Then, using the stack $[\lambda x.x.i-][\lambda y.v_0-]\varepsilon$ for $1 \le i \le \max n\, m$ we get $n = m$. Next if π, c are a stack and a clock establishing $v_i \not\equiv w_i$, the stack $[\lambda x.x.i-]\pi$ gives $v \not\equiv w$ with the same clock. Thus we have $v_i \equiv w_i$ for all $1 \le i \le n$.

□

Remark: we only need ρ to distinguish the unit tuple $()$.

Lemma 1.5. *If* $t_1 * \pi_1 * (k_1, l_1) \succ \chi(v) * \pi_2 * (k_2, l_2)$ *then we have* $t_1 * \pi_1 * (k_1, k_2 - k_1) \downarrow \chi(v)$.

Proof. We only need to remark that during reduction, the sum of the clock and the timeout is invariant and that the timeout is only decreased by reduction and never observed. Hence we can change the initial timeout with any value large enough not to reach zero and the reduction will be preserved with a constant shift of the timeout compared to the previous reduction. □

2 TYPES AND REALIZABILITY SEMANTIC

Expressions are simply typed λ-terms using the base type ι for individuals and o for propositions. The figure 2 gives the rules to form expressions which are quotiented by $\alpha\beta$-equivalence.

We call *sorts* the types of expressions to avoid confusion with the types of our system, i.e. expressions of sort o. It is important to notice that individuals, expressions of sort ι contains all terms, not only natural numbers.

This system is an extension of HOL, a.k.a. Church's simple theory of type [5]. In HOL existentials and products can be encoded, thus the essential extensions are *membership types* $t \in e$ and *restriction* $e \mid t \equiv u$ together with the use of terms as individual. Ψ is a constant function symbol used for the axiom of choice. We also use usual natural numbers as values rather than some encoding, but this is mainly to simplify the presentation of the system. In PML, we have full inductive and co-inductive types as in ML and natural numbers are defined.

An essential difference of PML compared to HOL is the way we formalise mathematics, especially functions. In HOL, functions from natural numbers to natural numbers will be represented by expressions of sort $\iota \to \iota$. In PML we will use expression of sort ι and of type $\mathbb{N} \Rightarrow \mathbb{N}$. We can express properties of

$\lambda\xi.e$	:	$s_1 \to s_2$	if $e : s_2$ assuming $\xi : s_1$
ef	:	s_2	if $e : s_1 \to s_2$ and $f : s_1$
t	:	ι	if $t \in \Lambda$
$A \Rightarrow B$	:	o	if $A, B : o$
$\forall\xi^s, A$	:	o	if $A : o$ assuming $\xi : s$
$\exists\xi^s, A$	:	o	if $A : o$ assuming $\xi : s$
$A_1 \times \ldots \times A_n$	:	o	if $A_1 : o, \ldots, A_n : o$
$\mathbb{U}$	:	o	$\mathbb{U}$ denotes the empty product type
$t \in A$	:	o	if $t \in \Lambda$ and $A : o$
$A \mid t \equiv u$	:	o	if $A : o$ and $t, u \in \Lambda$
$A \mid t \downarrow$	:	o	if $A : o$ and $t \in \Lambda$
$\mathbb{N}$	:	o	
Ψ	:	$\iota \to \iota$	

Fig. 2. formation of expression of miniPML

elements of sort ι thanks to membership type and restriction. Our main point is that mathematics can be developed within the sort ι in PML, including real numbers and much more as we have classical logic and the axiom of choice. We will begin to see this in the section 5.

We use a few syntactic sugars and priority rules:

- Quantifiers have the largest possible scope: $\forall\xi^s, A \Rightarrow B$ means $\forall\xi^s, (A \Rightarrow B)$ and $\exists\xi^s, A \times B$ means $\exists\xi^s, (A \times B)$
- Implication is right associative for expression and sorts: $A \Rightarrow B \Rightarrow C$ means $A \Rightarrow (B \Rightarrow C)$.
- Product has a greater priority than implication: $A_1 \times A_2 \Rightarrow B_1 \times B_2$ means $(A_1 \times A_2) \Rightarrow (B_1 \times B_2)$.
- Membership and restriction have a greater priority than implication and product: $A \times B \mid t \equiv u$ means $A \times (B \mid t \equiv u)$.
- Membership as priority over restriction: $t \in A \mid u_1 \equiv u_2$ means $(t \in A) \mid u_1 \equiv u_2$.
- We can write $t \equiv u$ for $\mathbb{U} \mid t \equiv u$ to use equivalence as a formula.
- We can write $t \downarrow$ for $\mathbb{U} \mid t \downarrow$ to use convergence as a formula.
- We can write $f(n_1, \ldots, n_m)$ for $f(n_1, \ldots, n_m) \equiv 1$ typically, this allows to write $\mathbb{N} \mid p < n$ for $\mathbb{N} \mid (p < n) \equiv 1$ or even $p < n$ for $\mathbb{U} \mid (p < n) \equiv 1$.

The semantic of an expression of sort s lives in a set $|s|$ defined below. It is parameterised by a non empty set of closed values $\perp\!\!\!\perp_\iota$ closed by equivalence and an initial process $p_0 \in \Lambda^c \times \Pi^c \times \{(0, \infty)\}$ (p_0 is a closed process using the clock 0 and no timeout to start). It is important to remark that only the definition $A \Rightarrow B$ and the definition of Ψ depend upon $\perp\!\!\!\perp_\iota$ and p_0.

We first define the *thread* of p_0 denoted $\mathrm{th}(p_0)$, as the smallest set such that

- $p_0 \in \mathrm{th}(p_0)$,
- if $p \in \mathrm{th}(p_0)$ and $p \succ q$, then $q \in \mathrm{th}(p_0)$ and
- if $t * \pi * c \in \mathrm{th}(p_0)$ and $t \equiv u$, then $u * \pi * c \in \mathrm{th}(p_0)$.

This definition is due to Krivine [13] and extended here to be compatible with equivalence.

Lemma 2.1. *If any element in th(p_0) converges, then all elements in th(p_0) converge to the same $v \in \overline{\Lambda}^c_\iota$ up to equivalence.*

Proof. We first show that if $p \in \mathrm{th}(p_0)$ and $p \downarrow v$ for $v \in \overline{\Lambda}^c_\iota$, then $p_0 \downarrow w$ with $v \equiv_+ w$. We prove the result by induction on the definition of $\mathrm{th}(p_0)$. The base case is immediate. If $q \in \mathrm{th}(p_0)$ and $q \succ p$ and

$p \downarrow v$, then we have $q \downarrow v$ by lemma 1.2.2. If $t * \pi * c \in \mathrm{th}(p_0)$ and $t \equiv u$ and $u * \pi * c \downarrow v$, then $t * \pi * c \downarrow w$ with $w \equiv_+ v$ by lemma 1.2.4.

Then, we get $p_0 \downarrow v$ implies that $\forall p \in \mathrm{th}(p_0), p \downarrow w$ with $v \equiv_+ w$ using a similar induction. □

LEMMA 2.2. *If $\chi(v) * \pi * c \in th(p_0)$ and $\chi(w) * \pi' * c \in th(p_0)$, with the same clock, then $v \equiv w$.*

PROOF. We define up-to-equivalence reduction ($\succcurlyeq$) with $p \succ q$ implies $p \succcurlyeq q$ and $t * \pi * c \succcurlyeq u * \pi * c$ if $t \equiv u$.

By definition of $\mathrm{th}(p_0)$, $p \in \mathrm{th}(p_0)$ if and only if $p_0 \succcurlyeq p$.

If $p_0 = t_0 * \pi_0 * (0, \infty) \succcurlyeq \chi(v) * \pi * (k, \infty)$ then we have $t_0 * \pi_0 * (0, k) \succcurlyeq \chi(v) * \pi * (k, 0)$ by lemma 1.5. Then we prove that $t_0 * \pi_0 * (0, k) \downarrow \chi(v_0)$ for some $v_0 \equiv v$. We proceed by induction on the number of applications of the equivalence rule in the definition of $\succcurlyeq$.

If we do not use equivalence, then we have directly $t_0 * \pi_0 * (0, k) \succ \chi(v) * \pi * (k, 0) \downarrow \chi(v)$ and we can take $v_0 \equiv v$.

Otherwise, we have $t_0 * \pi_0 * (0, k) \succcurlyeq t_1 * \pi_1 * c$, $t_1 \equiv t_2$ and $t_2 * \pi_1 * c \succ \chi(v) * \pi * (k, 0) \downarrow \chi(v)$. Using lemma 1.2.4, we conclude that $t_1 * \pi_1 * c \succ \chi(v_1) * \pi' * (k, 0)$ with $v \equiv v_1$. Thus we have $t_0 * \pi_0 * (0, k) \succcurlyeq \chi(v_1) * \pi' * (k, 0)$, using one equivalence less and we can conclude using the induction hypothesis and the transitivity of equivalence.

Using the same argument with $\chi(w) * \pi' * (k, \infty) \in \mathrm{th}(p_0)$, we get $v \equiv v_0 \equiv w$. □

We define the *pole* used for orthogonal.

$$\begin{aligned}
\perp\!\!\!\perp &= \perp\!\!\!\perp_\infty \cap \perp\!\!\!\perp_0 \\
\perp\!\!\!\perp_0 &= \{t * \pi \in \Lambda^c * \Pi^c \mid t * \pi * (0,0) \downarrow v \text{ with } l \in \mathbb{N}, v \in \perp\!\!\!\perp_\iota \cup \{\chi(w) \mid w \in \Lambda^c_\iota\}\} \\
\perp\!\!\!\perp_\infty &= \{t * \pi \in \Lambda^c * \Pi^c \mid \exists c \in \mathbb{N}, t * \pi * c \in \mathrm{th}(p_0) \text{ implies } p_0 \downarrow v \text{ with } v \in \perp\!\!\!\perp_\iota\}
\end{aligned}$$

The pole $\perp\!\!\!\perp$ is the intersection of two poles: the pole $\perp\!\!\!\perp_0$ express that program should behave well with a null timeout. And the pole $\perp\!\!\!\perp_\infty$, only looks at the thread p_0. $\perp\!\!\!\perp_\infty$ is all pairs $t * \pi$ if p_0 converges to a value in $\perp\!\!\!\perp_\iota$. Otherwise, it is the complement of $\mathrm{th}(p_0)$ with the clock removed.

Thanks to lemmas 1.2.4, 2.1 and the closure of $\perp\!\!\!\perp_\iota$ by equivalence, we could replace $p_0 \downarrow v$ in the above definition by $t * \pi * c \downarrow v$.

Here are three lemmas about our pole.

LEMMA 2.3. *If $t * \pi \in \perp\!\!\!\perp$ and $t' * \pi' \gg t * \pi$, then $t' * \pi' \in \perp\!\!\!\perp$.*

PROOF. Let use assume $t * \pi \in \perp\!\!\!\perp$ and $t' * \pi' \gg t * \pi$.

First, we have $t * \pi * (0,0) \downarrow v$ implies $t' * \pi' * (0,0) \downarrow v$ with $v \in \perp\!\!\!\perp_\iota \cup \{\chi(w) \mid w \in \Lambda^c_\iota\}$ by lemma 1.2.2. Thus we have $t' * \pi' \in \perp\!\!\!\perp_0$.

Second, if there is no c such that $t' * \pi' * c \in \mathrm{th}(p_0)$, then $t' * \pi' \in \perp\!\!\!\perp$. Otherwise, we have $t' * \pi' * c \in \mathrm{th}(p_0)$ for some clock c. This implies $t * \pi * c \in \mathrm{th}(p_0)$ and as we have $t * \pi \in \perp\!\!\!\perp$, we have $p_0 \downarrow v$ with $v \in \perp\!\!\!\perp_\iota$. □

LEMMA 2.4. *If $\forall c \in \mathbb{N} \times \overline{\mathbb{N}}$, $t * \pi * c \downarrow v$ with $v \in \perp\!\!\!\perp_\iota$, then $t * \pi \in \perp\!\!\!\perp$.*

PROOF. Immediate. □

LEMMA 2.5. *If $t * \pi \in \perp\!\!\!\perp$ and $t \equiv u$, then $u * \pi \in \perp\!\!\!\perp$.*

PROOF. From lemma 1.2.4, using the fact that $\perp\!\!\!\perp_\iota$ is closed by equivalence for $\perp\!\!\!\perp_0$. Using the definition of $\mathrm{th}(p_0)$, we have $t * \pi * c \in \mathrm{th}(p_0)$ if and only if $u * \pi * c \in \mathrm{th}(p_0)$. This allows to conclude for $\perp\!\!\!\perp_\infty$. □

$$
\begin{array}{rcl}
|i|^s & = & i \text{ if } i \in |s| \qquad \text{(this includes } |\mathbb{N}|^o = \mathbb{N}) \\
|t|^\iota & = & \{u \in \Lambda^c \mid u \equiv t\} \text{ the equivalence class of } t \text{ for } \equiv \\
|\lambda\xi.e|^{s_1 \to s_2} & = & g \in |s_1 \to s_2| \text{ s.t. } \forall i \in |s_1|, g(i) = |e[\xi := i]|^{s_2} \\
|ef|^{s_2} & = & |e|^{s_1 \to s_2}|e|^{s_1} \\
|A \Rightarrow B|^o & = & \{\lambda x.t \mid \forall v \in |A|^o, t[x := v] \in ||B||\} \\
|\forall \xi^s, A|^o & = & \bigcap_{i \in |s|} |A[\xi := i]|^o \\
|\exists \xi^s, A|^o & = & \bigcup_{i \in |s|} |A[\xi := i]|^o \\
|a_1 \times \ldots \times A_n|^o & = & \{(v_1, \ldots, v_n) \mid \forall 1 \leq i \leq n, v_i \in |A_i|^o\} \\
|t \in A|^o & = & \{v \in |A|^o \mid v \equiv t\} \\
|A \mid t \equiv u|^o & = & |A|^o \text{ if } t \equiv u \text{ and } \emptyset \text{ otherwise} \\
|A \mid t \downarrow|^o & = & |A|^o \text{ if } \exists v \in \Lambda^c_\iota, t \equiv v \text{ and } \emptyset \text{ otherwise} \\
|A|^\perp & = & \{\pi \in \Pi^c \mid \forall v \in |A|^o, v * \pi \in \perp\!\!\!\perp\} \\
||A|| & = & \{t \in \Lambda^c \mid \forall \pi \in |A|^\perp, t * \pi \in \perp\!\!\!\perp\} \\
|\Psi|^{\iota \to \iota} & = & g \in |\iota \to \iota| \text{ s.t. } \begin{cases} g(\overline{c}) = \overline{v}, \text{ if } c \in \mathbb{N} \text{ and } \chi(v) * \pi * c \in \mathrm{th}(p_0) \\ g(\overline{t}) = \overline{()} \text{ otherwise} \end{cases}
\end{array}
$$

Fig. 3. Semantic of expressions

The definition of $|s|$ follows:

$$
\begin{array}{rcll}
|\iota| & = & \Lambda^c/\equiv & \text{the set of closed terms quotiented by equivalence} \\
|o| & = & \mathcal{P}(\Lambda^c_\iota) & \text{the set of set of closed values} \\
|s_1 \to s_2| & = & |s_2|^{|s_1|} & \text{set of all functions from } |s_1| \text{ to } |s_2|
\end{array}
$$

We did not define $|o|$ as the powerset of $\Lambda^c_i/\equiv$ to have a better control of the elements of $|e|^o$. We will actually prove that it is closed by equivalence.

We now define the semantic $|e|^s \in |s|$ for closed expressions of sort s in figure 3. In the particular case of o we define three sets: $|A|^o \in |o|$ a set of closed values, $|A|^\perp$ a set of closed stacks and $||A||$ a set of closed terms. To simplify the presentation, we allow elements in $|s|$ to be used directly as expressions of sort s, so called *expressions with parameters in the model*. This allows writing $|e[\xi := i]|^s$ to give the interpretation of the variable ξ if sorts agree. This is expressed by the first line in figure 3.

The definition is correct and we indeed have $|e|^s \in |s|$. Only the last line requires two comments: a term can be equivalent to at most one natural number by lemma 1.4 and, for any $c \in \mathbb{N}$, there are at most one closed value v, up to equivalence such that $\chi(v) * \pi * c \in \mathrm{th}(p_0)$ by lemma 2.2.

LEMMA 2.6. *For any expression $e : o$, $||e||$ is closed by equivalence.*

PROOF. From lemma 2.5. □

Now comes a lemma expressing that equivalence does not equalise to much values and we have the control over values in $|e|^o$. Without this, the correctness theorem 3.2 would be mostly useless.

LEMMA 2.7. *The interpretation $|A|^o$ is closed for equivalence restricted to closed values.*

PROOF. This is proved by induction on A. Most cases are immediate. The cases for $A = \mathbb{N}$, $A = B \Rightarrow C$ and $A = B_1, \ldots, B_n$ are deduced from the induction hypothesis using lemma 1.4. □

LEMMA 2.8. *For any set of value V closed by equivalence, $V \subset ||V||$. If for a closed value v, we have $v \in ||V||$, then $v \in V$.*

Thanks to our convention for using formula with parameter in the model, we can write $||V||$ and do not need a syntax for orthogonality. The relation $V \subset ||V||$ is standard in classical realizability. The second property is less usual and essential. This was the purpose of adding the constant δ and defining reduction and equivalence by induction.

PROOF. To prove $V \subset ||V||$, we take $v \in V$ and prove $v \in ||V||$. For this we take $\pi \in |V|^{\perp}$ and must prove $v * \pi \in \perp\!\!\!\perp$ which is true by definition $|V|^{\perp}$.

Let us assume that $v \in ||V||$, $v \notin V$. Consider the stack $\pi = [\lambda x.\delta(x, v, v_0)-] * \varepsilon$ with $v_0 \in \perp\!\!\!\perp_{\iota}$ which is assumed non empty. As $v \notin V$, for any w in V, as V is closed by equivalence, we have $v \not\equiv w$ and therefore $w * \pi * c \succ v_0 * \varepsilon * c \downarrow v_0$ for any clock $c \in \overline{\mathbb{N}}$. This implies $w * \pi \in \perp\!\!\!\perp$ by lemma 2.4 and therefore $\pi \in |V|^{\perp}$. As $v \in ||V||$ we should have $v * \pi * (0,0) \downarrow v_1$ with $v_1 \in \perp\!\!\!\perp_{\iota} \cup \{\chi(w_0) \mid w_0 \in \Lambda^c_{\iota}\}$. This is obviously not the case as $v \equiv v$ implies that the rule for δ does not apply and the process $\delta(v, v, v_0) * \varepsilon$ can not be further reduced. $\square$

LEMMA 2.9. *If V and W are two sets of closed values with $V \subset W$, then we have $|W|^{\perp} \subset |V|^{\perp}$ and $||V|| \subset ||W||$.*

PROOF. We take $\pi \in |W|^{\perp}$ and prove $\pi \in |V|^{\perp}$. For this, we take $v \in V$ and must prove $v * \pi \in \perp\!\!\!\perp$. This holds as $v \in V \subset W$ and $\pi \in |W|^{\perp}$.

We take $t \in ||V||$ and prove $t \in ||W||$. For this, we take $\pi \in |W|^{\perp}$ and prove $t * \pi \in \perp\!\!\!\perp$. This holds as $\pi \in |W|^{\perp} \subset |V|^{\perp}$ and $t \in ||V||$. $\square$

The next theorem is really essential as it validates the substitution rule which is required to use dependent products with terms that are equivalent to values and not only equal values.

THEOREM 2.10. *For any $A : o$, we have $|A|^o \subset ||A||$ and $||A|| \cap \Lambda^c_{\iota} \subset |A|^o$.*

PROOF. Direct from lemma 2.7 and 2.8. $\square$

LEMMA 2.11. *The semantic as the following property regarding subsitution: If $e : s_1$ and $f : s_2$ are expressions, if e has only one free variable ξ of sort s_2 and if f is closed, then $|e[\xi := f]|^{s1} = |e[\xi := |f|^{s_2}]|^{s_1}$ which means we can compute the semantic before of after performing a substitution.*

If e is an expression with only one free variable a of sort ι, and if $t \equiv u$, then $|e[a := t]|^s = |e[a := u]|^s$.

PROOF. Easy induction on the construction of e, using extensionality (lemma 1.3) for the sort ι and the type constructors $t \in e$, $e \mid t \equiv u$ and $e \mid t \downarrow$. $\square$

3 ADEQUACY OF THE TYPING RULES

The typing rules of our system are given in figure 4. The sequent have the form $\gamma; \Gamma \vdash t : A$ with:

- γ a set of equations $t \equiv u$. Remark: an equation of the form $x \equiv u$ implies that u is equivalent to some value.
- Γ a set of declarations for value variables $x : A$ with $A : o$ or
- declarations for stack variables $\alpha : A^{\perp}$ with $A : o$.

The notation $A^{\perp}$ is not part of the syntax of formula, but is here to recall that an hypothesis of the form $\alpha : A^{\perp}$ means that we assume the negation of A, not A itself.

In premises, when we write $\gamma \vdash t \equiv u$ we mean that it is true that $t\sigma \equiv u\sigma$ for all substitution such that all equations in $\gamma\sigma$ holds. In our implementation, we have a partial decision procedure for this, but for a theoretical paper, truth is enough there.

In the rules, in quite a few places, we impose a *value restriction.* Value restriction can be circumvent using β-expansion. But one should note that when we type a redex $(\lambda x.t)u$, is u is not known to be a value,

$$\frac{}{\gamma;\Gamma, x : A \vdash x : A}$$

$$\frac{\gamma;\Gamma, x : A \vdash t : B}{\gamma;\Gamma \vdash \lambda x.t : A \Rightarrow B} \qquad \frac{\gamma;\Gamma \vdash t : A \Rightarrow B \qquad \gamma;\Gamma \vdash u : A}{\gamma;\Gamma \vdash t\,u : B}$$

$$\frac{\gamma;\Gamma, \alpha : A^{\perp} \vdash t : A}{\gamma;\Gamma \vdash \mu\alpha.t : A} \qquad \frac{\gamma;\Gamma, \alpha : A^{\perp} \vdash t : A}{\gamma;\Gamma, \alpha : A^{\perp} \vdash [\alpha]t : B}$$

$$\frac{\gamma;\Gamma \vdash v : A}{\gamma;\Gamma \vdash v : \forall\xi^{s}, A}\,* \qquad \frac{\gamma;\Gamma \vdash t : \forall\xi^{s}, A \qquad e : s}{\gamma;\Gamma \vdash t : A[\xi := e]}$$

$$\frac{\gamma;\Gamma, x : A \vdash t : B}{\gamma;\Gamma, x : \exists\xi^{s}, A \vdash t : B}\,* \qquad \frac{\gamma;\Gamma \vdash t : A[\xi := e] \qquad e : s}{\gamma;\Gamma \vdash t : \exists\xi^{s}, A}$$

$$\frac{\gamma;\Gamma \vdash v_1 : A_1 \qquad \ldots \qquad \gamma;\Gamma \vdash v_n : A_n}{\gamma;\Gamma \vdash (v_1, \ldots, v_n) : A_1 \times \ldots \times A_n} \qquad \frac{\gamma;\Gamma \vdash v : A_1 \times \ldots \times A_n \qquad 1 \leq i \leq n}{\gamma;\Gamma \vdash v.i : A_i}$$

$$\frac{\gamma, x \equiv u; x : A \vdash t : B}{\gamma, x : u \in A \vdash t : B} \qquad \frac{\gamma;\Gamma \vdash v : A}{\gamma;\Gamma \vdash v : v \in A}$$

$$\frac{\gamma, u_1 \equiv u_2;\Gamma, x : A \vdash t : B}{\gamma, \Gamma, x : A \mid u_1 \equiv u_2 \vdash t : B} \qquad \frac{\gamma;\Gamma \vdash t : A \qquad \gamma \vdash u_1 \equiv u_2}{\gamma;\Gamma \vdash t : A \mid u_1 \equiv u_2}$$

$$\frac{\gamma, u \equiv y;\Gamma, x : A \vdash t : B}{\gamma, \Gamma, x : A \mid u \downarrow\ \vdash t : B}\,* \qquad \frac{\gamma;\Gamma \vdash t : A \qquad \gamma \vdash u \equiv v}{\gamma;\Gamma \vdash t : A \mid u \downarrow}$$

$$\frac{\gamma, \Gamma \vdash t[a := u_1] : B[a := u_1] \qquad \gamma \vdash u_1 \equiv u_2}{\gamma, \Gamma \vdash t[a := u_2] : B[a := u_2]}$$

$$\frac{\gamma;\Gamma, z : \forall y^{\iota}, (y \in \mathbb{N} \mid y < x) \Rightarrow A\,y, x : \mathbb{N} \vdash t[a := z] : A\,x}{\gamma;\Gamma \vdash \varphi a.\lambda x.t : \forall x^{\iota}, x \in \mathbb{N} \Rightarrow A\,x} \qquad \frac{\mathrm{n} \in \mathbb{N}}{\gamma;\Gamma \vdash n : \mathbb{N}}$$

$$\frac{\gamma;\Gamma \vdash v_1;\mathbb{N} \qquad \ldots \qquad \gamma;\Gamma \vdash v_n : \mathbb{N}}{\gamma;\Gamma \vdash f(v_1, \ldots, v_n) : \mathbb{N}}$$

$$\frac{\gamma;\Gamma \vdash v : \mathbb{N} \mid v \leq 1 \qquad \gamma, v \equiv 1;\Gamma \vdash t : A \qquad \gamma, v \equiv 0;\Gamma \vdash u : A}{\gamma;\Gamma \vdash v[t, u] : A}$$

$$\frac{}{\gamma, 0 \equiv Sv;\Gamma \vdash t : A} \qquad \frac{\gamma, v \equiv w \vdash t : B \qquad \gamma, v \equiv w;\Gamma \vdash v : \mathbb{N} \qquad \gamma, v \equiv w;\Gamma \vdash w : \mathbb{N}}{\gamma, Sv \equiv Sw;\Gamma \vdash t : B}$$

In the rule marked with $*$, the variable ξ and y must not be free in the conclusion of the rule. The rules for fixpoint and case analysis assume $\leq$ and $<$ in F as function returning 0 or 1.

Fig. 4. Rules of miniPML

t can not by typed under the hypothesis that $x \equiv u$. Thus the typing of t must not the specific value of u but only its type.

The substitution rule allows substitution on both sides of the column. This permits to relax value restriction when a term is known to be equivalent to a value. We can replace the term by a value, apply the rule with the value restriction and put back the term. This is very important as value restriction would make the system mostly useless otherwise.

It is also important to notice that replacing a value by a term often require two steps: to replace v by t in $v.1$, we first replace $v.1$ by $(\lambda x.\,x.1)v$ and then we can replace v by t. This is possible because $(\lambda x.\,x.1)v \equiv v.1$ using lemma 1.2.5.

Our system allows encoding of dependent product and pairs as $\forall x^\iota, x \in A \Rightarrow B\,x$ and $\exists x^\iota, (x \in A \times B\,x)$. This encoding appears in the rule for the fixpoint. Let us derive the elimination rule for dependant product to see how value restriction is important here:

$$\dfrac{\dfrac{\gamma,\Gamma \vdash t : \forall x^\iota, x \in A \to B\,x}{\gamma,\Gamma \vdash t : u \in A \to B\,u} \qquad \dfrac{\dfrac{\dfrac{\gamma,\Gamma \vdash u : A \qquad \gamma \vdash u \equiv v}{\gamma,\Gamma \vdash v : A}}{\gamma,\Gamma \vdash v : v \in A} \qquad \gamma \vdash u \equiv v}{\gamma,\Gamma \vdash u : u \in A}}{\gamma,\Gamma \vdash t\,u : B\,u}$$

Hence the derived rule is

$$\dfrac{\gamma \vdash u \equiv v \qquad \gamma,\Gamma \vdash t : \forall x^\iota, x \in A \to B\,x \qquad \gamma,\Gamma \vdash u \in A}{\gamma,\Gamma \vdash t\,u : B\,u}$$

This semantical value restriction and the above derivation is due to Rodolphe Lepigre [16, 18].

We say that a substitution σ validates a context $\gamma;\Gamma$ if it makes terms and types in the context closed and if for all $t \equiv u$ in γ, $t\sigma \equiv u\sigma$ is true, for all $x : A$ in Γ, $x\sigma \in |A\sigma|^o$ and for all $\alpha : A^\perp$, $\alpha\sigma \in |A\sigma|^\perp$.

Theorem 3.1 (adequacy). *If $\gamma,\Gamma \vdash v : A$ and if σ validates the context, then $v\sigma \in |A\sigma|^o$. If $\gamma,\Gamma \vdash t : A$ and if σ validates the context, then $t\sigma \in ||A\sigma||$.*

Proof. We proceed by induction considering each rule. In each case, we take σ validating the context of the conclusion.

Axiom rule $x\sigma \in |A\sigma|^\sigma$ as σ validates $\gamma,\Gamma,x : A$.

Introduction of implication We take $v \in |A\sigma|^o$ and have to show $t\sigma[x := v] \in ||B\sigma||$. As σ only produces closed terms, up to a renaming of x, we have $t\sigma[x := v] = t[x := v]\sigma$ and this substitution validates the context of the premise. Hence we have $t[x := v]\sigma \in ||B\sigma||$ by induction hypothesis.

Elimination of implication By induction hypothesis, we have $t\sigma \in ||A\sigma \Rightarrow B\sigma||$ and $u\sigma \in ||A\sigma||$. We use lemma 2.3. Let us take $\pi \in |B\sigma|^\perp$. We have $t\sigma u\sigma * \pi \gg u\sigma * [t\sigma -]\pi$. Thus it is enough to prove $[t\sigma -]\pi \in |A\sigma|^\perp$. For this we take a value $v \in |A\sigma|$. We have $v * [t\sigma -]\pi \gg t\sigma * [-v]\pi$ thus it suffices to prove $[-v]\pi \in |\sigma A \Rightarrow B\sigma|^\perp$. For this, we take $\lambda x.u \in |A\sigma \Rightarrow B\sigma|^o$. We have $\lambda x.u * [-v]\pi \gg u[x := v] * \pi$ and we have $u[x := v] * \pi \in \perp\!\!\!\perp$ because $\lambda x.u \in |A\sigma \Rightarrow B\sigma|^o$ implies $u[x := v] \in ||B\sigma||$ by definition.

Classical logic (rule typing μ) Take a stack $\pi \in |A\sigma|^\perp$. We have $\mu\alpha.t\sigma * \pi \gg t\sigma[\alpha := \pi] * \pi$. The induction hypothesis and lemma 2.3 allows to conclude as $\sigma[\alpha := \pi]$ validates the context of the premise.

Contradiction (rule typing $[\alpha]t$) Take $\pi \in |B\sigma|^\perp$. We have $\alpha\sigma \in |A\sigma|^\perp$ because σ validates the context and $t\sigma \in ||A\sigma||$ by induction hypothesis. Hence, $([\alpha\sigma]t\sigma) * \pi \gg t\sigma * \alpha\sigma \in \perp\!\!\!\perp$.

For all introduction To prove $v\sigma \in |\forall \xi^s, A\sigma|^o$, we take any $i \in |s|$. As ξ is not free in the conclusion $\sigma[\xi := i]$ validates the context of the premise. Hence we have $v\sigma \in |A\sigma[\xi := i]|^o$ by induction hypothesis. We may need a renaming of ξ to have $(\forall \xi^s, A)\sigma = \forall \xi^s, A\sigma$.

For all elimination From lemma 2.11, we have $|\forall \xi^s, A\sigma|^o \subset |A\sigma[\xi := |e\sigma|^s]|^o = |A[\xi := e]\sigma|^o$. We get $||\forall \xi^s, A\sigma|| \subset ||A[\xi := e]\sigma||$ by lemma 2.9.

Existential left introduction We have $x\sigma \in |\exists \xi^s, A\sigma|^o$ hence we find $i \in |s|$ such that $x\sigma \in |A\sigma[\xi := i]|^o$. Thus, $\sigma[\xi := i]$ validates the context of the premise and we can conclude by induction hypothesis.

Existential right introduction From lemma 2.11, we get $|A\sigma[\xi := e]|^o = |A\sigma[\xi := |e\sigma|^s]|^o \subset |\exists \xi^s, A\sigma|^o$. We get $||A\sigma[\xi := e]|| \subset ||\exists \xi^s, A\sigma||$ by lemma 2.9.

Product introduction By induction hypothesis.

Product elimination From $v\sigma \in |A_1\sigma \times \ldots \times A_n\sigma|^o$, we know that $v\sigma = (v_1, \ldots, v_n)$ with $v_i \in |A_i|^o$. Take $\pi \in |A_i|^\perp$, we have $v\sigma.i * \pi \gg v_i * \pi \in \perp\!\!\!\perp$.

Singleton left introduction From $x\sigma \in |u\sigma \in A\sigma|^o$, we have that $x\sigma \equiv u\sigma$ and $x\sigma \in |A\sigma|^o$. Hence σ validates the context of the premise and we conclude by induction hypothesis.

Singleton right introduction Immediate by reflexivity of $\equiv$.

Restriction left introduction From $x\sigma \in |A\sigma|u_1\sigma \equiv u_2\sigma|^o$, we have $u_1\sigma \equiv u_2\sigma$ and $x\sigma \in |A\sigma|^o$. Hence σ validates the context of the premise and we conclude by induction hypothesis.

Restriction right introduction The right premise gives $u_1\sigma \equiv u_2\sigma$ and therefore $|A\sigma|u_1\sigma \equiv u_2\sigma|^o = |A\sigma|^o$ which implies $||A\sigma|u_1\sigma \equiv u_2\sigma|| = ||A\sigma||$.

Convergence left introduction From $x\sigma \in |A\sigma|u\sigma \downarrow|^o$, we have $u\sigma \equiv v$ for some value and $x\sigma \in |A\sigma|^o$. Hence $\sigma[y := v]$ validates the context of the premise and we conclude by induction hypothesis.

Convergence right introduction The right premise gives $u\sigma \equiv v\sigma$ and therefore $|A\sigma|u\sigma \downarrow|^o = |A\sigma|^o$ which implies $||A\sigma|u_1\sigma \downarrow|| = ||A\sigma||$.

Substitution rule The right premise gives $u_1\sigma \equiv u_2\sigma$. We have $|A[x := u_1]\sigma|^o = |A[x := u_2]\sigma|^o$ using lemma 2.11 and therefore $||A[x := u_1]\sigma|| = ||A[x := u_2]\sigma||$. Using lemma 1.3 we have $t[a := u_1]\sigma \equiv t[a := u_2]\sigma$. When $t[a := u_1]\sigma$ and $t[a := u_2]\sigma$ are both terms, we conclude using lemma 2.6. When they are both values, we use 2.7. When $t[a := u_1]\sigma$ is a value and $t[a := u_2]\sigma$ is a term, we need $|A[a := u_2]\sigma|^o \subset ||A[a := u_2]\sigma||$ from the first part of theorem 2.10 and lemma 2.6 or 2.7. Finally, when $t[a := u_1]\sigma$ is a term and $t[a := u_2]\sigma$ is a value we need $||A[a := u_2]\sigma|| \cap \Lambda_\iota^c \subset |A[a := u_2]\sigma|^o$ from the second part of theorem 2.10 and lemma 2.6 or 2.7.

Introduction of natural numbers Immediate.

Fixpoint rule For all stack π, we have $\varphi a.\lambda x.t * \pi \gg \lambda x.t[a := \varphi a.\lambda x.t] * \pi$. This implies $\varphi a.\lambda x.t \equiv \lambda x.t[a := \varphi a.\lambda x.t]$ using lemma 1.2.5. Let us define $v = \lambda x.t[a := \varphi a.\lambda x.t]$. Then, it is enough to show that $v\sigma \in |\forall x^\iota, x \in \mathbb{N} \Rightarrow A\,x|^o$ by lemma 2.7. For a value w which is not a natural number, $|w \in \mathbb{N}|^o$ is empty and therefore $|w \in \mathbb{N} \Rightarrow A\,w|^o$ contains all λ-abstractions.

To finish, we show by induction on n that $v\sigma \in |n \in \mathbb{N} \Rightarrow A\,n|^o$. For this, we have to show $t\sigma[a := \varphi a.\lambda x.t\sigma][x := n] \in ||A\,n||$ which using extensionality is a consequence of $t\sigma[a := v\sigma][x := n] = t[a := z]\sigma[z := v\sigma][x := n] \in ||A\,n||$. The induction hypothesis $v\sigma \in |p \in \mathbb{N} \Rightarrow A\,p|^o$ for all $p < n$ means $v\sigma \in |(p \in \mathbb{N} \mid p < n) \Rightarrow A\,p|^o$ for all $p \in \mathbb{N}$. Hence, we have $v\sigma \in |\forall y^s, (y \in \mathbb{N} \mid y < n) \Rightarrow A\,y|^o$. This means that the substitution $\sigma[z := v\sigma][x := n]$ satisfies the context of the premise. Hence we get $t[a := z]\sigma[z := v\sigma][x := n] \in ||An||$ by the main induction hypothesis.

Rule to introduce function symbol By induction hypothesis, we know that $v_1\sigma, \ldots, v_n\sigma$ are natural numbers. We can conclude as f is assumed total using $|\mathbb{N}|^o \subset ||\mathbb{N}||$.

Case analysis . By induction hypothesis, $v\sigma$ is a natural number and $(v\sigma \leq 1) \equiv 1$ Hence, $v\sigma$ is 0 or 1. If it is 1 then $v\sigma[t\sigma, u\sigma] \equiv t\sigma$ otherwise $v\sigma[t\sigma, u\sigma] \equiv u\sigma$. In both cases we can conclude using the induction hypothesis.

First arithmetic axiom There are no value satisfying $0 \equiv Sv$ if S is interpreted by the successor function. Hence there are no substitution satisfying the context.

Second arithmetic axiom All substitution satisfying $v\sigma \in \mathbb{N}$, $w\sigma \in \mathbb{N}$ and $Sv\sigma \equiv Sw\sigma$ also satisfy $v\sigma \equiv w\sigma$.

□

The next theorem is out safety/correctness theorem. It establishes that terms evaluate to values in the intended type (semantically) when the type is simple enough:

THEOREM 3.2. *If A is a type not using the symbol $\Rightarrow$ nor Ψ with $|A|^o$ non empty, and if $\vdash t : A$, then $t * \varepsilon * 0 \succ v * \varepsilon * c$ for some $c \in \mathbb{N} \times \overline{\mathbb{N}}$ and $v \in |A|^o$.*

PROOF. First we remark that if A does not use the symbol $\Rightarrow$ nor Ψ, the definition of A does not use nor depend upon the set $\perp\!\!\!\perp_\iota$ nor the initial process p_0.

Therefore, we can take $\perp\!\!\!\perp_\iota = |A|^o$ (because it is non empty) and $p_0 = t * \varepsilon * (0, \infty)$. With this definition, using lemma 2.4, we have $v * \varepsilon \in \perp\!\!\!\perp$ for any value $v \in \perp\!\!\!\perp_\iota = |A|^o$ as $v * \varepsilon * c \downarrow v$ for any clock $c \in \mathbb{N} \times \overline{\mathbb{N}}$. This implies $\varepsilon \in |A|^\perp$. Hence, we know that $t * \varepsilon \in \perp\!\!\!\perp$ and as $t * \varepsilon * (0, \infty) = p_0 \in \mathrm{th}(p_0)$, we have $p_0 \downarrow v$ with $v \in \perp\!\!\!\perp_\iota = |A|^o$ and therefore $p_0 \succ v * \varepsilon * c$. $\square$

This theorem with the definition of $|A|^o$ gives the following (and similar results for pairs of naturals, ...).

COROLLARY 3.3. *If $\vdash t : \exists x^\iota, x \in \mathbb{N} | f(x) \equiv 0$, then $t * \varepsilon * (0, \infty) \succ n * \varepsilon * c$ for some $n \in \mathbb{N}$ such that $f(n) = 0$.*

This more explicitly shows that if we use classical logic (and in the next section the axiom of choice) to establish the existence of a natural number with some given properties, then this natural can be computed by reduction.

It also gives the coherence of our system:

THEOREM 3.4. *miniPML is consitent.*

PROOF. We assume $\vdash t : \forall X^o, X$, we can not apply directly theorem 3.2 as $|\forall X^o, X| = \emptyset$. But we immediately get $\vdash t : \exists n^\iota, n \in \mathbb{N} \mid n \equiv 0$ and $\vdash t : \exists n^\iota, n \in \mathbb{N} \mid n \equiv 1$.Hence theorem 3.2 implies that $t * \varepsilon * (0, \infty) \succ 0 * \varepsilon * c$ and $t * \varepsilon * (0, \infty) \succ 1 * \varepsilon * c$ which is not possible since reduction in deterministic. $\square$

4 DERIVATION OF THE AXIOM OF CHOICE

First, we prove the following lemma to type the $\chi(v)$ term construction.

LEMMA 4.1. $\lambda x.\chi(x) \in |\forall A^o \forall x^\iota, x \in A \Rightarrow \exists n^\iota.n \in \mathbb{N} \mid \Psi n \equiv x|^o$

This lemma is much more direct that the lemma for the *quote instruction* in [12]. The main difference is that we directly give a number to any value. Remark: it is essential that $\chi(v)$ is not equivalent to a value, because if we could show that the above function always returns values, we would have an injection from any type A to natural numbers, which would lead to inconsistency.

PROOF. Let us take $A \in |o|$, $v \in |A|^o$ and prove $\chi(v) \in ||\exists n^\iota.n \in \mathbb{N} \mid \Psi n \equiv v||$. We take a stack $\pi \in |\exists n^\iota.n \in \mathbb{N} \mid \Psi n \equiv v|^\perp$.

First, we have $\chi(v) * \pi * (0, 0) \downarrow \chi(v)$, hence $\chi(v) \in \perp\!\!\!\perp_0$.

Then, if there exists c such that $\chi(v) * \pi * c \in \mathrm{th}(p_0)$, then we have $\chi(v) * \pi * c \succ c * \pi * (c + 1)$ and therefore $c * \pi * (c + 1) \in \mathrm{th}(p_0)$. Moreover, by definition of the semantics, we have $|\Psi(c)|^\iota = \overline{v}$ hence we have $c \in |\exists n^\iota.n \in \mathbb{N} \mid \Psi n \equiv v|^o$. This implies that $c * \pi \in \perp\!\!\!\perp \supset \perp\!\!\!\perp_\infty$ and therefore the convergence of p_0. $\square$

This lemma justifies the following typing rules:

$$\frac{}{\Gamma \vdash \lambda x.\chi(x) : \forall A^o \forall x^\iota, x \in A \Rightarrow \exists n^\iota.n \in \mathbb{N} \mid \Psi n \equiv x}$$

Theorem 4.2 (AC). *From the above axiom we can define a predicate $C : o \to (\iota \to o) \to \iota \to o$ and a proof of the following axiom of choice which consists in 3 propositions:*

$$\begin{array}{rcl} M & : & \forall A^o, \forall P^{\iota\to o}, (\exists x^\iota, (x \in A) \times P\,x) \Rightarrow (\exists x^\iota, C\,A\,P\,x) \\ K & : & \forall A^o, \forall P^{\iota\to o}, \forall x^\iota, C\,A\,f\,x \Rightarrow x \in A \times P\,x \\ U & : & \forall A^o, \forall P^{\iota\to o}, \forall Q^{\iota\to o}, \forall x^\iota, \forall y^\iota, P \Leftrightarrow_A Q \Rightarrow C\,A\,P\,x \Rightarrow C\,A\,Q\,y \to x \equiv y \\ \textit{with } P \Leftrightarrow_A Q & = & (\forall x^\iota, x \in A \Rightarrow P\,x \Rightarrow Q\,x) \times (\forall x^\iota, x \in A \Rightarrow Q\,x \Rightarrow P\,x) \end{array}$$

This theorem will be proved using the system rules and the above axiom throughout the rest of this section. Let us first see that this is indeed the extensional axiom of choice over any type. The predicate $C\,A\,P\,x$ must be read as "x is the chosen element in A such $P\,x$ holds". The first proposition ensures that the choice is possible if a candidate exists. The second one ensures that the choice is correct and the last ensures extensionality as the choice would be the same for any predicate Q equivalent to P. This is the fact that we allow for equivalence that makes this the axiom extensional.

What this axiom can not do is to choose a predicate of sort $\iota \to o$. But it can still choose a function in $\mathbb{N} \Rightarrow \mathbb{N}$ or a functional in $(\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. With PML, we have been able to formalise classical real numbers (as opposite to constructive reals). As we have a type of real numbers, we can choose a real or a function from the reals to the reals.

As an application, we construct quotient types in the next section.

The proof from here to the end of the paper have been machine checked in the current implementation of PML.

First, we derive a lemma:

Lemma 4.3. *We define two predicates:*

$$\begin{array}{rcl} B\,A\,P\,Q\,n & = & n \in \mathbb{N} \times (P \Leftrightarrow_A Q) \times \psi\,n \in A \times Q(\psi\,n) \times M\,A\,P\,n \\ M\,A\,P\,n & = & \forall Q^{\iota\to o}, \forall p^\iota, p \in \mathbb{N} \Rightarrow (P \Leftrightarrow_A Q) \Rightarrow \psi\,p \in A \Rightarrow Q(\psi\,p) \to n \le p \end{array}$$

and we find a term

$$C : \forall A^o, \forall P^{\iota\to o}, (\exists x^\iota, x \in A \times P\,x) \Rightarrow \exists Q^{\iota\to o}, \exists n^\iota, B\,A\,P\,Q\,n$$

The predicate $B\,A\,P\,Q\,n$ means that n the least integer such that $\Psi\,n$ is a good choice for A and Q, meaning that we have $\psi\,n \in A \times Q(\psi\,n)$ with Q equivalent to P on A. The fact that n is minimum is expressed by the predicate $M\,A\,P\,n$.

From our hypothesis on the function symbol in the systems, we have $(n \le p) \equiv 0 \vdash (p < n) \equiv 1$. Moreover, we must have a proof that $n \le p$ is a value, that is a proof that $\forall n^\iota, \forall p^\iota, n \in N \Rightarrow p \in N \Rightarrow n \le p \downarrow$. This can be proved in general by induction on n and p. This is necessary because when we do a test on $n \le p$ we must write with a redex $(\lambda x.x[t,u])(n \le p)$. Then, when typing this redex, if $n \le p$ is not known to be a value, we can not use the strong application. If is a value, when we typecheck $x[t,u]$, we will have $x \equiv n \le p$ in the context and therefore $(n \le p) \equiv 1$ (resp. 0) when typing t (resp. u).

Proof. Here is a possible C:

$$\begin{array}{rcl} C & = & \lambda h.(\lambda k.\mu\alpha.(\varphi r.\lambda n.\lambda e.\lambda x.\lambda t.(n,e,x,t,T))\,k\,R\,h.1\,h.2)(\chi\,h.1) \\ T & = & \lambda p.\lambda e'.\lambda y.\lambda u.(\lambda z.z[(),[\alpha]r\,p\,e'\,y\,u])(n \le p) \\ R & = & (\lambda x.\lambda y.y, \lambda x.\lambda y.y) \end{array}$$

To do the typing, we take $A : o$ and $P : \iota \to o$ and assume $h : \exists x^\iota, x \in A \times P\,x$. Using the left existential rule, we have $h : x \in A \times P\,x$. We have $\chi(h) : \exists n^\iota.n \in \mathbb{N} \mid \Psi\,n \equiv x$. Using a redex and the left rule for existential, membership and restriction, we have $k : \mathbb{N}$, $k \equiv n$ and $\Psi\,n \equiv x$.

We define $F = \exists Q^{\iota\to o}, \exists n^\iota, B\,A\,P\,Q\,n$ and find a term of that type. We use classical logic and therefore we assume $\alpha : F^\perp$.

To prove F, we first prove $\forall n : \iota, n \in \mathbb{N} \to H\,n$ by induction on n with:

$$H\,n = \forall Q^{\iota\to o}, (P \Leftrightarrow_A g) \to \psi\,n \in A \to Q(\psi\,n) \to \exists n^\iota, B\,A\,P\,Q\,n$$

Using the rule for φ, We assume $n : \mathbb{N}$, $r : \forall p : \iota, p \in \mathbb{N} \mid p < n \to H\,p$ and we must prove $H\,n$. Unrolling the definition of H, we assume $e : (P \Leftrightarrow_A Q)$, $x : \psi\,n \in a$, $t : Q(\psi\,n)$ and we must prove

$$\exists n^\iota, B\,A\,P\,Q\,n = \exists n^\iota, n \in \mathbb{N} \times (P \Leftrightarrow_A Q) \times \psi\,n \in A \times Q(\psi\,n) \times M\,A\,P\,n$$

Taking n for the existential, we can use the tuple (n, e, x, t, T) provided we find $T : M\,A\,P\,n$.

To contruct T, we take $Q' : \iota \to o$, assume $p : \mathbb{N}$, $e' : (P \Leftrightarrow_A Q')$, $y : \psi\,p \in A$ and $u : Q'(\psi p)$ and we must prove $n \leq p$. So we actually test for $n \leq p$. As this function is total, we can actually use $n \leq p$ is the first case and provide $() : n \leq p$. In the second case, we have $(n \leq p) \equiv 0 \vdash (p < n) \equiv 1$. Hence we can use the induction hypothesis with p, e', y and u to get a contradiction with $\alpha : F^\perp$.

Now that we have proved $\forall n : \iota, n \in \mathbb{N} \to H\,n$, we use it with k, R, $h.1$ and $h.2$ to get F where $R = (\lambda x.\lambda y.y, \lambda x.\lambda y.y) : P \Leftrightarrow_A P$. □

PROOF OF THEOREM AC. Now we can define the choice predicate by

$$C\,A\,P\,x = \exists n{:}\iota, n \in \mathbb{N} \times \psi\,n \in A \times P(\psi\,n) \times M\,A\,P\,n \mid x \equiv \psi\,n$$

and finish the proof of the axiom of choice.

We give the three terms where A is any proof of the anti-symmetry of $\leq$ on $\mathbb{N}$.

$$\begin{array}{rcl}
M & = & \lambda h.(\lambda c.(c.1, c.3, c.2.2\ c.3\ c.4, c.5))(C\,h) \\
K & = & \lambda c.(c.2, c.3) \\
U & = & \lambda c_1.\lambda c_2.\lambda e.A\ c_1.1\ c_2.1 \quad (c_1.4\ c_2.1\ e\ c_2.2\ c_2.3) \\
 & & \qquad\qquad\qquad\qquad\quad (c_2.4\ c_1.1\ (S\ e)\ c_1.2\ c_1.3) \\
S & = & \lambda c.(c.2, c.1) \\
A & : & \forall n^\iota, \forall p^\iota, n \in \mathbb{N} \Rightarrow p \in \mathbb{N} \Rightarrow n \leq p \Rightarrow p \leq n \Rightarrow n \equiv p
\end{array}$$

Above we wrote $c.2.2$ as an abbreviation for $(\lambda x.\,x.2)c.2$.

To type M, we assume $h : \exists x^\iota, x \in A \times P\,x$ and we use the previous lemma on h through a redex. Using left rule for existential, we have $c : B\,A\,P\,Q\,n$. Hence, $c.1 : \mathbb{N}$, $c.2 : P \Leftrightarrow_A Q$, $c.3 : \Psi\,n \in A$, $c.4 : Q(\Psi\,n)$ and $c.5 : M\,A\,P\,n$. Thus, $c.2.2\ c.3\ c.4 : P(\Psi\,n)$. This gives what we need with $x = \Psi\,n$.

In the typing of K, we have $c : C\,A\,P\,x$. Left rules for existential and restriction and the substitution rule, we get n such that $x \equiv \Psi\,n$, $c.2 : x \in A$ and $c.3 : P\,x$.

In the typing of U, similarly we get

$$\begin{array}{l}
x \equiv \Psi\,n,\ c_1.1 : n \in \mathbb{N},\ c_1.2 : x \in A,\ c_1.3 : P\,x,\ c_1.4 : M\,A\,P\,n, \\
y \equiv \Psi\,p,\ c_2.1 : p \in \mathbb{N},\ c_2.2 : y \in A,\ c_2.3 : Q\,y,\ c_2.4 : M\,A\,Q\,p \text{ and} \\
e : P \Leftrightarrow_A Q.
\end{array}$$

Thus we get $I = c_1.4\ c_2.1\ e\ c_2.2\ c_2.3 : n \leq p$, $S\,e : Q \Leftrightarrow_A P$, and $J = c_2.4\ c_1.1\ (S\,e)\ c_1.2\ c_1.3 : p \leq n$. Finally, we get $A\ c_1.1\ c_2.1\ I\ J : n \equiv p$, which implies $x \equiv \Psi\,n \equiv \Psi\,p \equiv y$. □

5 APPLICATIONS

5.1 quotient

To encode quotient, we first define a type $E\,A\,R$ (for $A : \iota \to o$ and $R : \iota \to \iota \to o$) saying that R is an equivalence relation over A:

$$\begin{array}{rcl} E\,A\,R & = & (\forall x^\iota, x \in A \Rightarrow R\,x\,x) \\ & \times & (\forall x^\iota, \forall y^\iota x \in A \Rightarrow y \in Y \Rightarrow R\,x\,y \Rightarrow R\,y\,x) \\ & \times & (\forall x^\iota, \forall y^\iota, \forall z^\iota x \in A \Rightarrow y \in A \Rightarrow z \in A \Rightarrow R\,x\,y \Rightarrow R\,y\,z \Rightarrow R\,x\,z) \end{array}$$

Then, we define a type $C'\,A\,R\,x\,y$ expressing that y is the chosen represent of the equivalence class of x for the relation R over the type A:

$$C'\,A\,R\,x\,y = C\,A\,(\lambda y.R\,x\,y)\,y$$

From this, we get the following three terms:

$$\begin{array}{ll} \lambda r.\lambda x.\,M\,(x, r.1\,x) & : \ \forall A^{\iota\to o}, \forall R^{\iota\to\iota\to o}, E\,A\,R \Rightarrow x \in A \Rightarrow \exists y^\iota, C'\,A\,R\,x\,y \\ \lambda r.\lambda x.\lambda c.\,(c.2, c.3) & : \ \forall A^{\iota\to o}, \forall R^{\iota\to\iota\to o}, \forall x^\iota, \forall y^\iota, C'\,A\,R\,x\,y \Rightarrow y \in a \times r\,x\,y \\ \multicolumn{2}{l}{\lambda r.\lambda x_1.\lambda x_2.\lambda c_1.\lambda c_2.\lambda e.U\,c_1\,c_2\,(\lambda z.\lambda p.\,r.3\,x_2\,x_1\,z\,(r.2\,x_1\,x_2\,e)\,p, \lambda z.\lambda p.\,r.3\,x_1\,x_2\,z\,e\,p)} \\ & : \ \forall A^{\iota\to o}, \quad \forall R^{\iota\to\iota\to o}, \forall x_1^\iota, \forall x_2^\iota, \forall y_1^\iota, \forall y_2^\iota, E\,A\,R \Rightarrow x_1 \in A \Rightarrow x_2 \in A \Rightarrow \\ & \qquad C'\,A\,R\,x_1\,y_1 \Rightarrow C'\,A\,R\,x_2\,y_2 \Rightarrow R\,x_1\,x_2 \Rightarrow y_1 \equiv y_2 \end{array}$$

The first term establishes the existence of a representent. The second term the fact that this representent is in A and equivalent to the original object. Finally the last term shows that representent are equal if the original objects were equivalent.

Typing is easy.

5.2 Indexing

In the introduction, we explain that our axiom implies that all type are countable if looked at with predicate. We actually derive this.

We define a predicate $I\,A\,x\,n$ that says that n is the index of x in A. This is only a simplification of the choice predicate and similarly, we take the least possible index.

$$I\,A\,x\,n = n \in \mathbb{N} \mid \Psi\,n \equiv x \times (\forall p^\iota, p \in \mathbb{N} \mid \Psi\,p \equiv x \to p \leq n)$$

Theorem 5.1. *We can derive in miniPML that ndex exists and are unique:*

$$\vdash \forall A^o, \forall x^\iota, x \in A \Rightarrow \exists n^\iota, I\,A\,x\,n \qquad \vdash \forall A^o, \forall x^\iota, \forall n^\iota, \forall m^\iota, I\,A\,x\,n \Rightarrow I\,A\,x\,m \Rightarrow n \equiv m$$

Proof. We give the two terms, using the same term A for anti-symmetry as in the axiom of choice.

$$\lambda x.(\lambda k.\mu\alpha.\varphi r.\lambda n.(n, \lambda p.(\lambda z.z[(), [\alpha]rp])n \leq p))(\chi x) \lambda i_x \lambda i_y.A\,i_x.1 i_y.2(i_x.2 i_y.1)(i_y.2 i_x.1)$$

□

From the index, it is clear that we can construct a well-ordering on any type. We define a predicate $L\,A\,x\,y$ expressing that x is less than y.

Theorem 5.2. *The following predicate define a well-ordering on any type:*

$$L\,A\,x\,y = \exists n^\iota, \exists m^\iota, I\,A\,x\,n \times I\,A\,y\,m \times n < m$$

Proof. Using this predicate, we can recover all properties of the ordering of naturals, including well-ordering. □

REFERENCES

[1] *Simple Imperative Polymorphism*, 1995.

[2] David Aspinall. Subtyping with Singleton Types. In *Computer Science Logic, CSL'94*, volume 933 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1995.

[3] Franco Barbanera and Stefano Berardi. A symmetric lambda calculus for classical program extraction. *Inf. Comput.*, 125(2):103–117, 1996.

[4] U. Berger and P. Oliva. Modified bar recursion and classical dependent choice. pages 89–107. Springer, 2005.

[5] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–80, 1940.

[6] Thierry Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60:325–337, 1995.

[7] Ken etsu Fujita. Polymorphic call-by-value calculus based on classical proofs. In *AISC*, pages 170–182, 1998.

[8] Timothy G. Griffin. A formulae as-as-types notion of control. In *17th annual ACM symposiuom on Principle of Programming Language*, pages 47–58. Oxford University Press, 1990.

[9] S. Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109:174–210, 94.

[10] W.A. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on combinatory logic, λ-calculus and formalism*, pages 479–490, 1980.

[11] Jean-Louis Krivine. Classical logic, storage operators and second-order lambda-calculus. *Annals of Pure and Applied Logic*, 68:225–260, 1994.

[12] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.

[13] Jean-Louis Krivine. Realizability in classical logic. Phd Course, To appear in Panoramas et synthèses, Société Mathématique de France., 2004.

[14] Jean-Louis Krivine. Realizability algebras : a program to well order r. *Log. Methods Comp. Sc.*, 7:1–47, 2011.

[15] Jean-Louis Krivine. Realizability algebras ii : new models of zf + dc. *Log. Methods Comp. Sc.*, 8:1–28, 2012.

[16] Rodolphe Lepigre. A Classical Realizability Model for a Semantical Value Restriction. In Peter Thiemann, editor, *25th European Symposium on Programming, ESOP 2016*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

[17] Rodolphe Lepigre. PML2: integrated program verification in ML. In *TYPES*, volume 104 of *LIPIcs*, pages 4:1–4:27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[18] Rodolphe Lepigre. *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes).* PhD thesis, Université Grenoble Alpes, France, 2017.

[19] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58, 2019.

[20] Michel Parigot. $\lambda\mu$-calculus an algorithmic interpretation of classical natural deduction. *Proceedings of Logic and Automatic Reasoning*, 1991. Lecture Notes in Computer Science Vol. 624.

[21] Michel Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94:335–356, 1992.

[22] Christophe Raffalli. Getting results from programs extracted from classical proofs. *Theor. Comput. Sci.*, 323(1-3):49–70, 2004.

[23] Christophe Raffalli. Pml: a new proof assistant. 2007.

[24] Christophe Raffalli and Frédéric Ruyer. Realizability of the axiom of choice in HOL. (an analysis of krivine's work). *Fundam. Inform.*, 84(2):241–258, 2008.

[25] M. Bezem S. Berardi and T. Coquand. On the coputational content of the axiom of choice. *J. Symb. Logic*, 63:600–622, 1998.

A PROOFS IN PML

We now give the formalisation of the proof in section 4 and 5 that are type-checked by PML.

```
include lib.either
include lib.nat
include lib.nat_proofs

/////////////////////////////// Axiom of choice ///////////////////////////////

// Definition and theorem for equivalence of unary predicates
type equiv⟨a,f,g⟩ = (∀x:ι, x∈a → f⟨x⟩ → g⟨x⟩) × (∀x:ι, x∈a → g⟨x⟩ → f⟨x⟩)

def idt : ι = fun x {x}

val eq_refl : ∀a,∀f, equiv⟨a,f,f⟩ = (fun _ {idt}, fun _ {idt})

val eq_symm : ∀a, ∀f g, equiv⟨a,f,g⟩ → equiv⟨a,g,f⟩
              = fun e { (e.2, e.1) }

val eq_tran : ∀a, ∀f g h, equiv⟨a,f,g⟩ → equiv⟨a,g,h⟩ → equiv⟨a,f,h⟩
  = fun e1 e2 { ( fun x p { e2.1 x (e1.1 x p) }
              , fun x p { e1.2 x (e2.2 x p) }) }

// Test reducing the type size when we know n<m, to help the termination checker
val rec leq_size : ∀o, ∀m∈nat^(o+1), ∀n∈nat, either⟨leq m n, n∈nat^o⟩
  = fun m n {
    case m {
      Zero → case n {
          Zero → InL
          S[n] → InL
        }
      S[m] →
        case n {
          Zero → InR[Zero]
          S[n] →
            case m {  // case for n because leq use compare
              Zero  → case n { Zero → InL | S[_] → InL}
              S[m'] →
                case leq_size S[m'] n {
                  InL    → InL
                  InR[p] → InR[S[p]]
                  }
              }
          }
      }
  }

// Derivation of the axiom of choice from the clock rule
// Predicate M and B in the paper
type mini⟨a,f:ι→o,n:ι⟩ =
  ∀h,∀p:ι, p∈nat → equiv⟨a,f,h⟩ → ψ⟨p⟩∈a → h⟨ψ⟨p⟩⟩ → n ≤ p
type best⟨a,f:ι→o,g:ι→o,n:ι⟩ =
 n ∈ nat × equiv⟨a,f,g⟩ × ψ⟨n⟩∈a × g⟨ψ⟨n⟩⟩ × mini⟨a,f,n⟩

// The lemma
val ac1
  : ∀a,∀f:ι→o, (∃x∈a, f⟨x⟩) → ∃g,∃n:ι, best⟨a,f,g,n⟩
  = fun h {
    let a,f such that h : ∃x∈a, f⟨x⟩;
    let (x,p) = h;
    let n : ∃n, n∈nat | ψ⟨n⟩ ≡ x = χ x;
    save s {
      let rec fn : ∀g,∀n:ι, n∈nat → equiv⟨a,f,g⟩ → ψ⟨n⟩∈a → g⟨ψ⟨n⟩⟩ →
              ∃n, best⟨a,f,g,n⟩
        = fun n e x h {
            let o such that n : nat^(o+1);
            (n, e, x, h,
                fun p e x h {
                  case leq_size (n:nat^(o+1)) p {
                    InL → qed
                    InR[p] → restore s (fn p e x h)
                  }
                })
          };
        fn n eq_refl x p
      }
    }

// Predicate C in the paper
type choice⟨a,f,x⟩ = ∃n:ι, n ∈ nat × ψ⟨n⟩ ∈ a × f⟨ψ⟨n⟩⟩ × mini⟨a,f,n⟩ | x ≡ ψ⟨n⟩

// The proof of the three theorems that makes AC
val mk_choice : ∀a,∀f, (∃x∈a, f⟨x⟩) → ∃x, choice⟨a,f,x⟩
     = fun v {
          let c = ac1 v;
          let a,f,g,n such that c : best⟨a,f,g,n⟩;
          let c : best⟨a,f,g,n⟩ = c;
          (c.1, c.3, c.2.2 c.3 c.4, c.5)
     }

val choice_correct : ∀a,∀f:ι→o,∀x:ι, choice⟨a,f,x⟩ → x ∈ a × f⟨x⟩
   = fun c { (c.2, c.3) }

val choice_unique
        : ∀a,∀f g,∀x y, choice⟨a,f,x⟩ → choice⟨a,g,y⟩ → equiv⟨a,f,g⟩ → x ≡ y
   = fun c1 c2 eq {
        let p1 = c1.4 c2.1 eq c2.2 c2.3;
        let p2 = c2.4 c1.1 (eq_symm eq) c1.2 c1.3;
        show c1.1 ≡ c2.1 using leq_anti c1.1 c2.1 p1 p2;
        qed
   }

/////////////////////////////// Quotient types ////////////////////////////////

// Record defining equivalence relation
type eq_rel⟨a,eq⟩ =
  { r : ∀x:ι, x∈a → eq⟨x, x⟩
  ; s : ∀x y:ι, x∈a → y∈a → eq⟨x, y⟩ → eq⟨y, x⟩
  ; t : ∀x y z:ι, x∈a → y∈a → z∈a → eq⟨x, y⟩ → eq⟨y, z⟩ → eq⟨x, z⟩ }

// Definition of the choosen representent
// C' in the paper
type class⟨a,eq:ι→ι→o,x:ι,y:ι⟩ = choice⟨a, (z:ι ↦ (eq⟨x,z⟩)), y⟩

// The three properties of quotients
val mk_class : ∀a,∀eq:ι→ι→o, eq_rel⟨a,eq⟩ → ∀x:ι, (x∈a → ∃y:ι, class⟨a,eq,x,y⟩)
   = fun e x {
        let a, eq such that e:eq_rel⟨a,eq⟩;
        mk_choice ((x, e.r x) : ∃z∈a, eq⟨x, z⟩)
   }

val class_correct :
   ∀a,∀eq:ι→ι→o, eq_rel⟨a,eq⟩ → ∀x y:ι, x∈a →
       class⟨a,eq,x,y⟩ → y ∈ a × eq⟨x, y⟩
          = fun e x c { (c.2, c.3) }

val class_equal :
   ∀a,∀eq:ι→ι→o, eq_rel⟨a,eq⟩ → ∀x_1 y_1 x_2 y_2:ι, x_1 ∈ a → x_2 ∈ a →
       class⟨a,eq,x_1,y_1⟩ → class⟨a,eq,x_2,y_2⟩ → eq⟨x_1, x_2⟩ → y_1 ≡ y_2
     = fun e x_1 x_2 c_1 c_2 ee {
         let a, eq such that e:eq_rel⟨a,eq⟩;
         let p : equiv⟨a,(z:ι ↦ (eq⟨x_1, z⟩)),(z:ι ↦ (eq⟨x_2, z⟩))⟩ =
           (fun z p { e.t x_2 x_1 z (e.s x_1 x_2 ee) p }
           ,fun z p { e.t x_1 x_2 z ee p});
        choice_unique c_1 c_2 p
      }

////////////////////// Injection from any type to nat /////////////////////////

// Predicate giving the number of a value in a type
// I in the paper
def mini2⟨a,x:ι,n:ι⟩ = (∀p:ι, p∈nat | ψ⟨p⟩ ≡ x → n ≤ p)
def index⟨a,x:ι,n:ι⟩ = n ∈ nat | ψ⟨n⟩ ≡ x × mini2⟨a,x,n⟩

// Term calculing the index
val index_exists
  : ∀a,∀x:ι,x ∈ a → ∃n:ι, index⟨a,x,n⟩
  = fun x {
    let a such that x : a;
    let n : ∃n, n∈nat | ψ⟨n⟩ ≡ x = χ x;
    save s {
      let rec fn : ∀n:ι, n∈nat | ψ⟨n⟩≡x → ∃n, index⟨a,x,n⟩
        = fun n {
            let o such that n : nat^(o+1);
            (n, fun p {
                  case leq_size (n:nat^(o+1)) p {
                    InL → qed
                    InR[p] → restore s (fn p)
                  }
                })
        };
      fn n
    }
  }

// Unicity of the index
val index_unique
  : ∀a, ∀x n m:ι, index⟨a,x,n⟩ → index⟨a,x,m⟩ → n ≡ m
  = fun xi yi {
      let n = xi.1; let m = yi.1;
      let ineq1 : n ≤ m = xi.2 m;
      let ineq2 : m ≤ n = yi.2 n;
      use leq_anti n m ineq1 ineq2;
      qed
  }

////////////////////////// well-ordering on any type /////////////////////////

type le⟨a,x,y⟩ = ∃n m:ι, index⟨a,x,n⟩ × index⟨a,y,m⟩ × n < m

val le_total : ∀a, ∀x y:ι, x∈a → y∈a → either⟨x ≡ y, either⟨le⟨a,x,y⟩,le⟨a,y,x⟩⟩⟩
  = fun x y {
    let a such that x : a;
    let ix = index_exists x;
    let iy = index_exists y;
    case compare ix.1 iy.1 {
      Eq → InL
      Ls → InR[InL[(ix,iy,{})]]
      Gr → InR[InR[(iy,ix,lt_gt iy.1 ix.1)]]
    }
  }

// Identity function (really identity) reducing the size if we know m < n
val reduce_size : ∀o, ∀n∈nat^(o+1), ∀m∈nat | m < n, m ∈ nat^o =
  fun n m {
    let o such that n : nat^(o+1);
    check { // check this term
      case leq_size (n : nat^(o+1)) m {
        InL    → show not (m < n) by leq_lt n m;
```

```
                qed // impossible case
        InR[p] → p
      }}
    for { m // but use m as they are equal }
  }

val le_well_founded : ∀a, ∀p, (∀x:ι, (∀y:ι, y∈a → le⟨a,y,x⟩ → p⟨y⟩) → x∈a → p⟨x⟩)
                                       → ∀x:ι, x∈a → p⟨x⟩
  = fun ind x0 {
    let a such that x0 : x0 ∈ a;
    let p such that _ : p⟨x0⟩;
    let ind : ∀x:ι, (∀y:ι, y∈a → le⟨a,y,x⟩ → p⟨y⟩) → x∈a → p⟨x⟩ = ind;
    let rec fn : ∀x:ι, ∀n:ι, x ∈ a → index⟨a,x,n⟩ → n ∈ nat → p⟨x⟩ =
      fun x ix n {
        deduce n ≡ ix.1;
        let o such that n : nat^(o+1);
        let gn : ∀y:ι, y∈a → le⟨a,y,x⟩ → p⟨y⟩ =
          fun y le {
            let iy = le.1;
            let ix' = le.2;
            let _ : n ≡ ix'.1 = index_unique ix ix';
            deduce iy.1 < n;
            let p = reduce_size (n : nat^(o+1)) iy.1;
            fn y iy p
            }
          };
        ind gn x
      };
    let ix = index_exists x0;
    fn x0 ix ix.1
  }
```