The PML_2 Language: Proving Programs in ML

Rodolphe Lepigre

LAMA, CNRS, Université Savoie Mont Blanc, France
Inria (Deducteam), LSV, CNRS, Université Paris-Saclay, France
`rodolphe.lepigre@inria.fr`

Abstract

We present the PML_2 language, which provides a uniform environment for programming, and for proving properties of programs in an ML-like setting. The language is Curry-style and call-by-value, it provides a control operator (interpreted in terms of classical logic), it supports general recursion and a very general form of (implicit, non-coercive) subtyping. In the system, equational properties of programs are expressed using two new type formers, and they are proved by constructing terminating programs. Although proofs rely heavily on equational reasoning, equalities are exclusively managed by the type-checker. This means that the user only has to choose which equality to use, and not where to use it, as is usually done in mathematical proofs. In the system, writing proofs mostly amounts to applying lemmas (possibly recursive function calls), and to perform case analyses (pattern matchings).

1998 ACM Subject Classification F.3.1 Specifying, Verifying and Reasoning about Programs

Keywords and phrases program proving, classical logic, ML-like language, termination, Curry-style quantification, implicit subtyping

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction: joining programming and proving

In the last thirty years, significant progress has been made in the application of type theory to computer languages. The Curry-Howard correspondence, which links the type systems of functional programming languages to mathematical logic, has been explored in two main directions. On the one hand, proof assistants such as Agda [24] or Coq [23] are based on very expressive logics [22, 7]. To establish their consistency, the underlying programming languages need to be restricted to provably terminating programs. As a result, they forbid the most general forms of recursion. On the other hand, functional programming languages such as Haskell, SML or OCaml are well-suited for programming, as they impose no restriction on recursion. However, their type systems are inconsistent when considered as logics, which means that they cannot be used for proving mathematical formulas.[1]

The aim of PML_2 is to provide a uniform environment in which programs can be designed, specified and proved. The idea is to combine a full-fledged ML-like programming language, with an enriched type system allowing the specification of computational behaviours.[2] The obtained system can thus be used as ML for type-safe general programming, and as a proof assistant for proving properties of ML programs. The uniformity of the framework implies that programs can be incrementally refined to obtain more and more guarantees. In particular, there is no syntactic distinction between programs and proofs. The only difference is that the

[1] This particular point will be explained in more detail in Section 5.
[2] On might argue that PML_2 is not a *full-fledged ML-like language* as it does not have mutable references. It is nonetheless effectful as it provides a control operator similar to Scheme's *call/cc*.

latter must be typed-checked against the consistent core of the system, which only accepts programs that can be proved terminating. In the current implementation, programs must also be proved terminating to be directly accepted by the type-checker. It would however be possible to accept programs that do not pass the termination check, and we would then need to make sure that such programs are not used to write proofs.[3] Note however that the system can already be used to reason about arbitrary programs, including untyped ones and those whose termination cannot be established (an example will be given in Section 5.3).

1.1 Program proving principles

In PML_2, program properties may be specified with types containing equations of the form `t ≡ u`, where `t` and `u` are terms of the language itself. By quantifying over the free variables of these terms, we can express properties such as the following.

```
// "add" is commutative.
∀n∈nat, ∀m∈nat, add n m ≡ add m n

// "reverse" is involutive.
∀a, ∀l∈list⟨a⟩, reverse (reverse l) ≡ l

// "sort" produces sorted lists.
∀l∈list⟨nat⟩, sorted (sort l) ≡ true

// All natural numbers are equal to "Zero".
∀n∈nat, n ≡ Zero.

// Sorted lists are not affected by "sort".
∀l∈list⟨nat⟩, (sorted l ≡ true) ⇒ (sort l ≡ l)
```

Of course, such a specification may be inaccurate, in which case it will not be provable. Note that it is possible to observe complex behaviours using predicates such as `sorted`, which correspond to boolean-valued functions.

The PML_2 language relies on two main ingredients for proving that programs meet their specifications. First, the type system of the language can be considered as a (classical) logic through the Curry-Howard correspondence, although it is only consistent for terminating programs. This (usual) part of the type system provides basic reasoning principles, which are used to structure the proofs. The second ingredient is an automatic decision procedure for the equational theory of the language. It is used to manage a context of equational assumptions, and eventually prove equations in this context. The decision procedure is driven by the type-checker, without any direct interaction with the user. As a consequence, the user only has to care about the structure of the proof, and not about the details of where equations should be applied. In fact, equality types of the form `t ≡ u` are computationally irrelevant in the system. More precisely, `t ≡ u` is equivalent to the unit type when the denoted equality holds (and can be proved), and it is empty otherwise. As a consequence, a proof will generally consist of a (possibly recursive) program that calls other programs and performs pattern matching but eventually just returns a completely uninteresting result. Nonetheless, writing proofs in this way is a very similar experience to writing functional

[3] For example, we could have two different function types: one used for functions whose termination has been established, and another one that that may be used for any function (and that would be a supertype of the former).

programs. We can hence hope that our approach to program proving will feel particularly intuitive to functional programmers.

1.2 Previous work on the language

PML_2 is based on many ideas introduced by Christophe Raffalli in the PML language [27]. Although this first version of the system was very encouraging on the practical side, it did not stand on solid theoretical grounds. On the contrary, PML_2 is based on a call-by-value, classical realizability model designed by the author [16]. This framework provides a satisfactory way of combining call-by-value evaluation and effects with dependent function types.[4] The proposed solution relies on a relaxed form of value restriction (called *semantical value restriction*), which takes advantage of our notion of program equivalence and its decision procedure.[5] In particular, it allows the application of a dependent function to a term which is not a value, under the condition that it can be proved equivalent to some value. This is especially important because dependent functions are an essential component of PML_2. Indeed, they enable a form of typed quantification without which many program properties could not be expressed (see Section 3).

Another important specificity of PML_2's type system is that it relies on the notion of *local subtyping*, which was introduced in joint work with Christophe Raffalli [19]. This framework can be used to give a syntax-directed formulation of the typing and subtyping rules of the system[6], despite its Curry-style nature. In particular, it provides a very general notion of infinite (circular) proof, that is used for handling inductive and coinductive types in subtyping derivations, and recursion and termination checking in typing derivations. Of course, infinite proofs are only valid if they are well-founded (this is ensured using the size-change principle [15]). The combination of local subtyping [19] and the realizability model of the system has been addressed in the author's PhD thesis [17, Chapter 6].

Last but not least, the implementation of PML_2 [18] was initiated as part of the author's thesis [17], and continues with the collaboration of Christophe Raffalli. The implemented system is intended to remain very close to the theoretical type system, and every part of the implementation is justified by the formal semantics of PML_2 [17]. Note that all the examples given in this paper are accepted by the version `2.0.2_types2017` of PML_2, which can be downloaded at the following URL.

`https://github.com/rlepigre/pml/archive/pml_2.0.2_types2017.tar.gz`

1.3 Disclaimer: the aim of this paper

This document is intended to be an introductory paper for the PML_2 system. Its aim is not to give the details of the realizability semantics, nor to prove new theoretical results, but rather to list the principles and ideas on which PML_2 is based. In particular, Section 6 contains an extensive description of several ideas that we would like to investigate in the near future, and that will be necessary for achieving the goals of PML_2 completely. For technical details, the reader should refer to the author's thesis [17], and related papers [16, 19].

[4] Due to the soundness issues explained in previous work [16], the application of dependent functions is usually restricted to value arguments. This is commonly called *value restriction* in the context of ML.

[5] Intuitively, two terms are (observationally) equivalent if they have the same computational behaviour (i.e., they both converge or they both diverge) in every possible evaluation context [16, 17].

[6] This means that exactly one typing rule applies for every term constructor, and only one subtyping rule applies for every pair of type constructors (up to commutation and circular proof construction).

2 Functional programming in PML$_2$

Our first goal in designing PML$_2$ is to obtain a practical, functional programming language. Out of the many possible technical choices, we decided to consider a call-by-value language similar to OCaml or SML, as they have proved to be highly practical and efficient. Our language provides polymorphic variants [8] and SML-style records, which are convenient for encoding data types. As an example, the type of lists can be defined as follows,[7] together with the corresponding `iter` and `append` functions.

```
type rec list⟨a⟩ = [Nil ; Cns of {hd : a ; tl : list}]

val rec iter : ∀a, (a ⇒ {}) ⇒ list⟨a⟩ ⇒ {} =
  fun f l {
    case l {
      Nil    → {}
      Cns[c] → f c.hd; iter f c.tl
    }
  }

val rec append : ∀a, list⟨a⟩ ⇒ list⟨a⟩ ⇒ list⟨a⟩ =
  fun l1 l2 {
    case l1 {
      Nil    → l2
      Cns[c] → Cns[{hd = c.hd ; tl = append c.tl l2}]
    }
  }
```

Note that the `iter` and `append` functions are polymorphic, which means, for instance, that they can be applied to lists with elements of an arbitrary type. In our syntax, this is explicitly materialised using universally quantified type variables. Note also that the `iter` function relies on the type `{}`, which contains records with no fields. It plays the same role as OCaml's `unit` type, and its unique inhabitant is denoted `{}` as well.

▶ Remark. As in System F [9, 29], polymorphism can be used anywhere in types, and it is not limited to *let-polymorphism* (or prenex polymorphism) as in most ML-like languages.

2.1 Control operator and classical logic

The programming languages of the ML family generally include effectful operations such as references (i.e., mutable variables). Our system is no exception since it provides a control operator similar to *call/cc*.[8] On the programming side, it may be used to encode a form of exception mechanism. For instance, we can define the following `exists` function, which tests whether there is an element satisfying a given predicate in a given list, and stops as soon as possible if such an element is found.

[7] Note that `list` is used without its type parameter in the type of the `Cns` constructor. This is dues to the fact that the "`type rec`" syntax is desugared to an inductive type (or least fixed point), and `list⟨a⟩` is actually defined as "`μ list, [Nil; Cns of {hd : a; tl : list}]`". In particular, the current version of PML$_2$ does not support polymorphically recursive types.

[8] This instruction can be used to capture the current continuation (or evaluation context), so that it can be restored later. It was first introduced in the Scheme language.

```
val exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    save k {
      iter (fun e { if pred e { restore k true } }) l;
      false
    }
  }
```

Here, the continuation is saved in a variable `k` before calling the `iter` function, and it is restored with the value `true` if an element satisfying the predicate is found. In this case, the evaluation of `iter` is simply aborted. To obtain a similar behaviour without using a continuation would require the user to write an independent recursive function (i.e., one that does not rely on `iter`). A more interesting example that cannot be written without a control operator will be given in Section 4.

As is now well-known, control operators such as ours can be used to give a computational content to classical theorems, thus extending the Curry-Howard correspondence to classical logic [10]. It is hence possible to define programs with a type corresponding to Peirce's law, or to the law of the excluded middle.

```
val peirce : ∀a b, ((a ⇒ b) ⇒ a) ⇒ a =
  fun x {
    save k { x (fun y { restore k y }) }
  }

// Disjoint sum of a and b (logical disjunction)
type either⟨a,b⟩ = [InL of a ; InR of b]

// Usual encoding of negation
type neg⟨a⟩ = a ⇒ ∀x,x

val excl_mid : ∀a, {} ⇒ either⟨a, neg⟨a⟩⟩ =
  fun _ {
    save k { InR[fun x { restore k InL[x] }] }
  }
```

Note that the definition of `excl_mid` requires a dummy function constructor due to the call-by-value evaluation strategy. Indeed, `excl_mid` would not be a value if it did not start with an abstraction, and it would thus save its continuation right away, unlike `peirce` which must first be given an argument to trigger the computation. This is related to value restriction [35, 34], which is required in presence of control operators [11].[9]

From a computational point of view, manipulating continuations using control operators can be understood as "cheating". For example `excl_mid` (or rather, `excl_mid {}`) saves the continuation and immediately returns a (possibly false) proof of `neg⟨a⟩`. Now, if this proof is ever applied to a proof of `a` (which would result in absurdity), the program backtracks and returns the given proof of `a`. This interpretation has been well-known for a long time, and an account is given in the work of Wadler [33, Section 4], for example.

[9] Value restriction is a sufficient (but not necessary) condition for correctness.

2.2 Non-coercive subtyping

In PML_2, subtyping plays a very important role as it allows us to give a mostly syntax-directed presentation of the system [19, 17]. Although it is less widespread than polymorphism in mainstream languages, subtyping can be exploited to improve code modularity. Note that we here consider a *non-coercive* form of subtyping, which means that if `a` is a subtype of `b`, then any value of type `a` is also a value of type `b` (i.e., no coercion is required).

In the system, there are many forms of subtyping that may interact. In particular, subtyping is used to handle all the connectives that do not have algorithmic contents (i.e., no counterpart in the syntax of terms). Such connectives include quantifiers as well as inductive types, but also the equality types of PML_2. Subtyping also plays an important role with variants and records. For instance, it implies that a record can always have more fields than required. Moreover, subtyping enables many commutations of connectives.

As a first example, we can show that the type corresponding to the (classical) double negation elimination principle can in fact be seen as an instance of Peirce's law. Indeed, it can be defined as follows in PML_2.

```
val dneg_elim : ∀a, neg⟨neg⟨a⟩⟩ ⇒ a =
  peirce
```

It is relatively easy to see that the type of `peirce` is indeed a subtype of that of `dneg_elim`. The corresponding subtyping derivation is sketched below.[10]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{a0} \subseteq \texttt{a0}}}{\texttt{a0} \Rightarrow \forall\texttt{x,x} \subseteq \texttt{a0} \Rightarrow \forall\texttt{x,x}} \quad \dfrac{\dfrac{}{\texttt{a0} \subseteq \texttt{a0}}}{\forall\texttt{x,x} \subseteq \texttt{a0}}}{(\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \forall\texttt{x,x} \subseteq (\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a0}} \quad \dfrac{}{\texttt{a0} \subseteq \texttt{a0}}}{((\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a0}) \Rightarrow \texttt{a0} \subseteq ((\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a0}}}{\forall\texttt{b}, ((\texttt{a0} \Rightarrow \texttt{b}) \Rightarrow \texttt{a0}) \Rightarrow \texttt{a0} \subseteq ((\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a0}}}{\forall\texttt{a}, \forall\texttt{b}, ((\texttt{a} \Rightarrow \texttt{b}) \Rightarrow \texttt{a}) \Rightarrow \texttt{a} \subseteq ((\texttt{a0} \Rightarrow \forall\texttt{x,x}) \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a0}}}{\forall\texttt{a}, \forall\texttt{b}, ((\texttt{a} \Rightarrow \texttt{b}) \Rightarrow \texttt{a}) \Rightarrow \texttt{a} \subseteq \forall\texttt{a}, ((\texttt{a} \Rightarrow \forall\texttt{x,x}) \Rightarrow \forall\texttt{x,x}) \Rightarrow \texttt{a}}$$

Intuitively, universal quantification on the right of the inclusion can be eliminated by introducing a fresh constant. On the left, variables that are quantified over can be replaced by anything. As usual, the subtyping rule for handling the arrow type reverses the inclusion between the domains due to the contra-variance of the arrow type.

We will now consider the extension of the type of lists with an additional constructor allowing constant time concatenation. In PML_2, the corresponding type of "append lists" can be defined in such a way that it admits the type of regular lists as a subtype.

```
type rec alist⟨a⟩ =
  [Nil ; Cns of {hd : a; tl : alist} ; App of alist × alist]

// Constant time "append" function.
val alist_append : ∀a, alist⟨a⟩ ⇒ alist⟨a⟩ ⇒ alist⟨a⟩ =
  fun l1 l2 { App[(l1,l2)] }
```

Although regular lists are a special case of "append lists", the converse is not true. To transform an "append list" into a list, it is necessary to define the following recursive, flattening function.

[10] The proof does not contain all the necessary information to ensure its validity. The reader should refer to the author's thesis [17, Figure 6.5] to fill in the missing details.

```
val rec alist_to_list : ∀a, alist⟨a⟩ ⇒ list⟨a⟩ =
  fun l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = c.hd ; tl = alist_to_list c.tl}]
      App[c] → append (alist_to_list c.1) (alist_to_list c.2)
    }
  }
```

Another example of extension for a pre-existing type can be obtained by defining the type of red-black trees as a subtype of binary trees. More precisely, a red-black tree can be represented as a tree whose nodes have an extra color field. Of course, the presence of additional information in the form of a new record field does not prevent the use of tree functions such as binary search.

2.3 Toward the encoding of a module system

Despite its many different features, PML_2 remains a fairly small system, which can be implemented rather concisely. Its design is based on the principle that every feature should be orthogonal. For instance, there is only one notion of product type in PML_2: records. This is not the case in OCaml, for instance, which provides tuples, records, objects, modules, which all have common product type characteristics.

In PML_2, modules can be easily encoded using a combination of records for storing the values, functions for building functors, and existentials for type abstraction. However, the implementation does not yet provide a specific syntax for modules. For instance, there is still no way of "opening" a module so that its values are accessible in the scope. It is nonetheless possible to work with the target of the encoding directly. For example, we can define a type corresponding to the signature (or interface) of a simple module providing an abstract representation for the stack data structure with the corresponding operations.[11]

```
type stack_sig = ∃stack: o → o,
  { empty : ∀a, stack⟨a⟩;
    push  : ∀a, a ⇒ stack⟨a⟩ ⇒ stack⟨a⟩;
    pop   : ∀a, stack⟨a⟩ ⇒ [None ; Some of a × stack⟨a⟩] }
```

An implementation of this interface can then be defined by giving a corresponding record value. For example, we can implement stacks with lists as follows.

```
val stack_impl : stack_sig =
  { empty = (Nil : ∀a, list⟨a⟩);
    push  = fun e s { Cns[{hd = e; tl = s}] };
    pop   = fun s {
              case s {
                Nil    → None
                Cns[c] → Some[(c.hd, c.tl)]
              }
            } }
```

[11] Here, $o \to o$ corresponds to the sort of types with one type parameter. In particular, o is the sort of types (or propositions), and we will later encounter the sort of program values ι.

Note that we need to give at least some type annotation for the system to know what to instantiate the existential with. This could be done in a more systematic way with a syntax requiring the user to give the intended definition for the `stack` type.

▶ Remark. It is possible to define a dot-projection operation in order to access abstract types, so that it is possible to write `stack_impl.stack` to refer to the type of stacks. More details are given in previous work [19], and in the corresponding implementation.

3 Proofs of ML programs

PML_2 is not only a programming language, but also a proof assistant focusing on program proving. Its proof mechanism relies on equality types of the form `t ≡ u`, where `t` and `u` are arbitrary (possibly untyped) terms of the language itself. Such an equality type is inhabited by the term `{}`[12] if the denoted equivalence is true, and it is empty otherwise. Equivalences are managed using a partial decision procedure that is driven by the construction of programs. An equational context is maintained by the type checker to keep track of the equational assumptions during the construction of proofs. This context is extended whenever a new equation is learned (e.g., when a lemma is applied), and equations are proved by looking for contradictions (e.g., when two different variants are equated).

Terms not only appear in (equality) types, but also play the role of objects in the underlying logic. In particular, they can be quantified over in types, and thus form one particular domain of discourse. In fact, our system is based on a higher-order logic with several atomic sorts (including types and terms), which means that many different kinds of objects can be quantified over (universally and existentially) in our types. We can for example quantify over types with one type parameter (of sort $o \to o$), as in the signature used for the stack module given in the previous section.

3.1 (Un)typed quantification and unary natural numbers

To illustrate the proof mechanism, we will consider very simple examples of proofs on unary natural numbers. Their type is given below, together with the corresponding addition function defined using recursion on its first argument.

```
type rec nat = [Zero ; S of nat]

val rec add : nat ⇒ nat ⇒ nat =
  fun n m {
    case n {
      Zero → m
      S[k] → S[add k m]
    }
  }
```

As a first example, we will show that for all `n` we have `add Zero n ≡ n`. This property is expressed using the type `∀n:ι, add Zero n ≡ n`, and it is proved as follows.[13]

```
val add_Zero_n : ∀n:ι, add Zero n ≡ n =
  {} // immediate
```

[12] Recall that it denotes a record with no fields, or the unique inhabitant of a one-element type.
[13] Here, the domain of the quantification is the set of values of the language, whose sort is ι.

The proof is immediate (i.e., only `{}`) as we have `add Zero n ≡ n` by definition of `add`. Note that this equivalence holds for every value `n`, whether it corresponds to an element of the type `nat` or not. For instance, it can be used to show `add Zero true ≡ true` since the term `add Zero true` evaluates to `true`.

▶ Remark. Here, it is crucial that `n` ranges only over values of the language, as otherwise the definition of add could not be unfolded. Indeed, since we are in call-by-value, it is only possible to effectively apply a function when its arguments are all values.

Let us now show that for every `n` we have `add n Zero ≡ n`. Although this property looks similar to `add_Zero_n`, the following proof is invalid.

```
// val add_n_Zero : ∀n:ι, add n Zero ≡ n =
//   {} // invalid
```

Indeed, the equivalence `add n Zero ≡ n` does not hold when `n` is not a unary natural number. In this case, the computation of `add n Zero` will produce a runtime error. As a consequence, we need to rely on a form of quantification that only ranges over unary natural numbers. This can be achieved with the type `∀n∈nat, add n Zero ≡ n`, which corresponds to a (dependent) function taking as input a natural number `n` and returning a proof of `add n Zero ≡ n`. This property can then be proved using induction (i.e., using a recursive function) and case analysis (i.e., pattern matching) with the following program.

```
val rec add_n_Zero : ∀n∈nat, add n Zero ≡ n =
  fun n {
    case n {
      Zero → {}
      S[k] → add_n_Zero k
    }
  }
```

If `n` is `Zero`, then we need to show `add Zero Zero ≡ Zero`, which is immediate by definition of `add`. In the case where `n` is `S[k]` we need to show `add S[k] Zero ≡ S[k]`. By definition of `add`, this reduces to `S[add k Zero] ≡ S[k]`. We can then use the induction hypothesis `add_n_Zero k` to learn `add k Zero ≡ k` and conclude the proof.

▶ Remark. The dependent product type (or typed quantification) constructor is not primitive in PML_2. It is encoded using a membership type of the form `t∈a` which contains all the elements of type `a` that are equivalent to the term `t` (it can be seen as a form of singleton type). The dependent function type `∀x∈a, b` is then encoded as `∀x:ι, x∈a ⇒ b`, which corresponds to the relativised quantification scheme (see previous work [16, 17]).

It is important to note that, in our system, a program that is considered as a proof needs to go through a termination checker. Indeed, a looping program could be used to prove anything otherwise.[14] For example, the following proof is rejected.

```
// val rec add_n_Zero_loop : ∀n∈nat, add n Zero ≡ n =
//   fun n {
//     add_n_Zero_loop n
//   }
```

It is however easy to see that `add_Zero_n` and `add_n_Zero` are terminating, and hence valid. In the following, we will only consider programs that can be automatically proved terminating by the system.

[14] More details will be given in Section 5.

3.2 Building up an equational context

There are two main ways of learning new equations in the system. On the one hand, when a term `t` is matched in a case analysis, a given branch can only be reached when the corresponding pattern `C[x]` matches. In this case we can extend the equational context with `t ≡ C[x]`. On the other hand, it is possible to invoke a lemma by calling the corresponding function. In particular, this must be done to use the induction hypothesis in proofs by induction like in `add_Zero_n` or the following lemma.

```
val rec add_n_S_m : ∀n m∈nat, add n S[m] ≡ S[add n m] =
  fun n m {
    case n {
      Zero → {}
      S[k] → add_n_S_m k m
    }
  }
```

In this case, the equation corresponding to the conclusion of the used lemma is directly added to the context. Of course, more complex results can be obtained by combining more lemmas. For example, the following proves the commutativity of addition using a proof by induction with `add_n_Zero` and `add_n_S_m`.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero → add_n_Zero m
      S[k] → add_comm k m; add_n_S_m m k
    }
  }
```

▶ Remark. Note that terms can be put in sequence with a semicolon. In the above proof, the recursive call `add_comm k m` is performed first, before calling `add_n_S_m m k`. They are also type-checked in that order, and the corresponding equations are added to the context one after the other as a side-effect to type-checking. Here, the order in which equations are added is not significant (the resulting equational context is the same either way), but that is not always the case (lemmas may require some equations to hold to be applied).

3.3 Detailed proofs using type annotations

Although the above proof of commutativity is perfectly valid, it might not be easy enough to read by a human. This problem arises in most proof assistants. For instance, it is almost impossible to understand a Coq [23] proof without replaying it step by step in a compatible editor. In PML_2, it is possible to annotate proofs to highlight the corresponding thought process. For example, we can reformulate `add_comm` as follows.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero → show add Zero m ≡ add m Zero using add_n_Zero m; qed
      S[k] → show add k m ≡ add m k using add_comm k m;
             deduce add S[k] m ≡ S[add m k];
             show add S[k] m ≡ add m S[k] using add_n_S_m m k; qed
    }
  }
```

Note that no addition to the system is required for such annotations to be supported, it is only syntactic sugar. For instance, `qed` is a synonym of `{}`, and `show u1 ≡ u2 using p` is translated to `p : u1 ≡ u2`, which amounts to a type coercion.

▶ Remark. Many examples of proofs and programs are provided with the implementation of the system. Each of the examples given here has been automatically checked upon the generation of the document, they are hence correct with respect to the implementation.

3.4 Mixing proofs and programs

We will now see that the programming and the proving features of PML_2 can be mixed when constructing proofs or programs. In fact, there is no obvious distinction between the world of the usual programs, and the world of proofs (remember that proofs are programs in PML_2). For instance, it is possible to combine proofs with programs for them to transport properties (e.g., addition carrying its own commutativity). This can be achieved using restriction types, which are in fact used to encode equality types. In PML_2, the type `a | t ≡ u` is equivalent to `a` if `t ≡ u` is true, and to the empty type otherwise. The type `t ≡ u` is thus encoded as `{} | t ≡ u`, where `{}` is the unit type. Intuitively, the restriction type can be seen as a form of conjunction with no algorithmic contents.

When combined with existential quantification and the membership type, restriction can be used to encode a *set type* syntax similar to that of NuPrl [6]. Indeed, we can define `{x ∈ a | t ≡ u}`, which contains all the elements of type `a` such that `t ≡ u` holds, as `∃x:ι, x∈(a | t ≡ u)`. This provides a very useful scheme for defining the set of terms of `a` that satisfy some property. For example, we can encode the type of vectors (i.e., lists of a given length) by taking every list `l` that has size `s`. The type of vectors will hence have two parameters: the type of the elements contained in the vectors and a term indicating the size of the vector.

```
val rec length : ∀a:o, list⟨a⟩ ⇒ nat =
  fun l {
    case l {
      Nil     → Zero
      Cns[c] → S[length c.tl]
    }
  }

type vec⟨a:o, s:τ⟩ = {l ∈ list⟨a⟩ | length l ≡ s}
```

▶ Remark. In the definition of `vec`, the second parameter must have sort τ (the sort of terms) and not ι (the sort of values). Indeed, it is often required to work with vectors whose sizes are of the form `add n m` (see the definition of the `app` function below).

▶ Remark. There is no constraint on the type of `s` in the definition of `vec`. This means that it is possible to consider the type of vectors of size `true` for example, but it will be empty since the `length` function only returns natural numbers.

Let us stress that vectors can always be used as lists, independently of their size. The type of vectors is a subtype of the type of lists, as shown by the following function.

```
val vec_to_list : ∀a:o, ∀s:τ, vec⟨a,s⟩ ⇒ list⟨a⟩ =
  fun x { x }
```

Note that we will never need to use the function `vec_to_list` to turn a vector into a list. A vector can be seen as a list directly, without relying on any form of coercion.

We will now define a concatenation function app on vectors. It produces a vector whose length is the sum of the lengths of its two arguments. Note that we are first required to define the length_total function for a technical reason that will be explained in Section 5.[15]

```
val rec length_total : ∀a:o, ∀l∈list⟨a⟩, ∃v:ι, v ≡ length l =
  fun l {
    case l {
      Nil    → {}
      Cns[c] → let ind = length_total c.tl; {}
    }
  }

val rec app : ∀a:o, ∀m n:ι, vec⟨a, m⟩ ⇒ vec⟨a, n⟩ ⇒ vec⟨a, add m n⟩ =
  fun l1 l2 {
    case l1 {
      Nil    → l2
      Cns[c] → length_total c.tl;
               Cns[{hd = c.hd; tl = app c.tl l2}]
    }
  }
```

Thanks to the Curry-style nature of our system, the sizes of the argument vectors do not need to be provided as arguments. This may be surprising for readers that are used to manipulating equivalent types in Agda or Coq, for example.

▶ Remark. In PML_2, the proof mechanism can also be used to eliminate unreachable (or dead) code. Indeed, if an equational contradiction is triggered only by learning equations along the way, then the code in that branch cannot be accessed during evaluation. In this case, a special value ✂ (to be pronounced "scissors") can be used. Note that reachability information would be particularly useful to efficiently compile PML_2 programs down to assembly code.

4 Programs extracted from classical proofs

We will now consider an example of a program that can only be written in a classical setting (i.e., with control operators). We are going to define a function on streams of natural numbers, that extracts from its input a stream of odd numbers, or a stream of even numbers. First, we need to define odd and even numbers in our language, using our set type syntax.

```
val rec is_odd : nat ⇒ bool =
  fun n {
    case n {
      Zero → false
      S[m] →
        case m {
          Zero → true
          S[p] → is_odd p
        }
    }
  }
```

[15] We have good hopes of simplifying this particular point in future work, for example by automatically obtaining length_total from the definition of length as they have a similar structure.

```
type odd  = {v∈nat | is_odd v ≡ true }
type even = {v∈nat | is_odd v ≡ false}
```

As for the `length` function of the previous section, we will need to show that the `is_odd` function is total for a technical reason (see Section 5 for more details). Intuitively, this will allow us to reason by cases on the oddness (or evenness) of a given number of the input stream. Indeed, the totality of `is_odd` implies that this function always produces a result value, and hence that we can pattern match on its result.

```
val rec odd_total : ∀n∈nat, ∃v:ι, is_odd n ≡ v =
  fun n {
    case n {
      Zero → {}
      S[m] →
        case m {
          Zero → {}
          S[p] → odd_total p
        }
    }
  }
```

We also need to define the type of streams, together with a related type corresponding to streams with an explicit size annotation (or ordinal) `s`. Intuitively, this size annotation indicates the number of elements that are available in the stream (see Section 5 for more details on sized-types).

```
type corec stream⟨a⟩ = {} ⇒ {hd : a; tl : stream}
type sized_stream⟨s,a⟩ = ν_s stream, {} ⇒ {hd : a; tl : stream}
```

We can now define the `itl_aux` function, which will be used to build our main infinite tape lemma function. Note that this function uses `abort`, which logically amounts to the *ex falso quodlibet* principle. Size annotations are also required on the type of `itl_aux`, for our type-checking algorithm to prove its termination.

```
val abort : ∀y, (∀x,x) ⇒ y = fun x { x }

val rec itl_aux : ∀a b,
    neg⟨sized_stream⟨a,even⟩⟩ ⇒
    neg⟨sized_stream⟨b,odd ⟩⟩ ⇒ neg⟨stream⟨nat⟩⟩ =
  fun fe fo s {
    let {hd ; tl} = s {};
    use odd_total hd;
    if is_odd hd {
      fo (fun _ {
        {hd = hd; tl = save oc {
          abort (itl_aux fe (fun x { restore oc x }) tl)}}
      })
    } else {
      fe (fun _ {
        {hd = hd; tl = save ec {
          abort (itl_aux (fun x { restore ec x }) fo tl)}}
      })
    }
  }
```

Intuitively, the `itl_aux` function looks at the head of its third argument (a stream of natural numbers). Depending on whether this number is odd or even, the function then calls one of its first two arguments, which corresponds to a partially constructed stream of even or odd numbers, in the form of continuations.[16] The read number is then added to this stream, and a recursive call is made to continue the construction.

▶ Remark. It may seem surprising that our prototype implementation is able to establish the termination of `itl_aux` as an element is added to one of two streams at each call. Moreover, this example does not satisfy the usually required semi-continuity condition [1]. It is here accepted because our termination test depends more finely on the structure of programs than previous approaches [19].

Using `itl_aux`, it is then possible to define the `itl` function corresponding to our infinite tape lemma as follows.

```
val itl : stream⟨nat⟩ ⇒ [InL of stream⟨even⟩; InR of stream⟨odd⟩] =
  fun s {
    save a {
      InL[save ec { restore a InR[save oc {
        abort (itl_aux (fun x { restore ec x})
          (fun x { restore oc x }) s)
      } ] } ]
    }
  }
```

This function starts by saving two continuations, corresponding to the constructors `InL` and `InR` of the return type, and then calls `itl_aux` on the input stream. The very fact that we can write `itl` proves that it is possible to extract a stream of odd numbers or a stream of even numbers from any stream of natural numbers.

Of course, it is only possible to observe a finite prefix of a stream using a terminating program. As a consequence, we may want to consider a finite version of `itl`, which result is a vector of a given size `n` instead of a stream.

```
val rec prefix : ∀a, ∀n∈nat, stream⟨a⟩ ⇒ vec⟨a,n⟩ =
  fun n s {
    case n {
      Zero → Nil
      S[k] → let {hd ; tl} = s {};
             Cns[{hd ; tl = prefix k tl}]
    }
  }

val finite_itl : ∀n∈nat,
    stream⟨nat⟩ ⇒ [InL of vec⟨even,n⟩; InR of vec⟨odd,n⟩] =
  fun n s {
    case itl s {
      InL[s] → InL[prefix n s]
      InR[s] → InR[prefix n s]
    }
  }
```

[16] Logical negation is intuitively used to type continuations represented in the form of a function. A continuation of type `neg⟨a⟩` can thus be called with a value of type `a` in any context since it yields a logical contradiction (or an element of type `∀x,x`).

▶ Remark. It is possible to give an equivalent definition of `finite_itl` in an intuitionistic setting (i.e., without using a control operator). Indeed, at most `2 × n` elements of the input stream need to be considered to construct the result.

▶ Remark. Of course, the type of `itl` or `finite_itl` does not directly imply that the result of these functions is a substream of their input. It is nonetheless easy to convince oneself that this is indeed the case, and we could certainly prove it in PML_2 with some effort.

To conclude this section, we will consider the result returned by `itl` (or rather `finite_itl`) on two particular streams. The former will be the stream of all natural numbers, which can be defined as follows, and is called `naturals`.

```
val rec naturals_from : nat ⇒ stream⟨nat⟩ =
  fun n _ {
    {hd = n; tl = naturals_from S[n]}
  }

val naturals : stream⟨nat⟩ = naturals_from Zero
```

The latter will be a stream of ones, prefixed by three zeroes. It can be defined as follows, and is called `three_zeroes_then_ones`.

```
val rec ones : stream⟨nat⟩ =
  fun _ {
    {hd = S[Zero]; tl = ones}
  }

val three_zeroes_then_ones : stream⟨nat⟩ =
  let add_Zero : stream⟨nat⟩ ⇒ stream⟨nat⟩ = fun tl _ { {hd = Zero; tl} };
  add_Zero (add_Zero (add_Zero ones))
```

The results of the application of `finite_itl` on `naturals` and `three_zeroes_then_ones` may be displayed using the following printing functions.

```
val rec print_nat : nat ⇒ {} =
  fun n {
    case n {
      Zero → print "0"
      S[k] → print "S"; print_nat k
    }
  }

val rec print_list : ∀a, (a ⇒ {}) ⇒ list⟨a⟩ ⇒ {} =
  fun pelt l {
    case l {
      Nil          → print "\n"
      Cns[{hd;tl}] → pelt hd; print " "; print_list pelt tl
    }
  }

val print_res : [InL of list⟨nat⟩; InR of list⟨nat⟩] ⇒ {} =
  fun e {
    case e {
      InL[l] → print "  InL "; print_list print_nat l
      InR[l] → print "  InR "; print_list print_nat l
    }
  }
```

▶ Remark. Although the above is boiler-plate code, it is provided so that the examples in this document are completely self-contained, and can be type-checked and evaluated by PML_2 without any modification.

We have now all the components that are required to run some tests, and to display prefixes of the streams produced by the `itl` function. We will display, for each example, prefixes of increasing size. We will thus rely on the following `test` function, taking as input a stream of natural numbers, and showing the result of applying the `finite_itl` function on this stream with various prefix lengths (from zero to four).

```
val test : stream⟨nat⟩ ⇒ {} =
  fun s {
    print_res (finite_itl Zero s);
    print_res (finite_itl S[Zero] s);
    print_res (finite_itl S[S[Zero]] s);
    print_res (finite_itl S[S[S[Zero]]] s);
    print_res (finite_itl S[S[S[S[Zero]]]] s)
  }
```

Let us first consider the output that is produced by applying the above `test` function to `three_zeroes_then_ones`, which contains three zeroes followed by infinitely many ones.

```
InL
InL 0
InL 0 0
InR S0 S0 S0
InR S0 S0 S0 S0
```

As one should expect, the computation of the smallest prefixes yields a list of even numbers. However, if more elements of the input stream are read, the `itl` function eventually backtracks and produces a list of odd numbers instead. Indeed, the input stream only contains three even numbers.

Note that one could expect the fourth line of the output to show a list of even numbers with three zeroes. The produced result is due to the definition of `itl`, which looks ahead one element further than strictly necessary in the input stream. It would be possible to avoid doing so, but the function would be even more complex than it already is. Note that this also has consequences on the result obtained by running `test` on `naturals`.

```
InL
InL 0
InL 0 SS0
InL 0 SS0 SSSS0
InL 0 SS0 SSSS0 SSSSSS0
```

Here, one would expect the prefixes to alternate between lists of even numbers, and lists of odd numbers. Indeed, the stream of all the natural numbers both contain an infinite sub-stream of even numbers, and an infinite sub-stream of odd numbers.

5 Termination and internal totality proofs

We will now look more deeply into the relation between proofs and termination checking in PML_2. Technically, the termination of PML_2 programs (and thus of PML_2 proofs) is established using circular proof techniques introduced in joint work with Christophe Raffalli [19] and adapted in the author's thesis [17]. The idea is to type recursive programs, or more

precisely the fixed-point combinator used by PML_2, using a simple unfolding rule. In other words, instances of the fixed-point construction of the language are typed assuming that they can be typed, thus leading to a circular structure. Of course, proofs constructed in this way may be invalid (i.e., not well-founded). To rule out such invalid proofs, a test based on the size-change principle [15] is used. When it is able to show that the structure of a proof is indeed well-founded, termination then follows from a standard semantic proof by realizability.[17]

5.1 Termination and consistency

As mentioned in the introduction, practical functional programming languages like OCaml or Haskell cannot be used to prove mathematical formulas, since their type system is not consistent when seen as a logic. More precisely, the "empty type" is inhabited by a simple looping program in these systems. Any formula can thus be proved through the *ex falso quodlibet* principle, as demonstrated by the following piece of Haskell code.[18]

```
type Empty = forall a. a

bad :: Empty
bad = bad

ex_falso :: Empty → a
ex_falso e = e
```

A similar example can also be given in OCaml, but a slightly more complex definition is required for `bad`.[19] This is due to the call-by-value evaluation strategy of the language, which restricts the use of the `let rec` construct to the definition of functions.

```
type empty = { any : 'a.'a }

let bad : empty =
  let rec bad_aux : unit → empty = fun () → bad_aux () in
  bad_aux ()

let ex_falso : type a. empty → a =
  fun e → e.any
```

Of course, a similar example can be written in PML_2. This is why the current implementation requires every program (not only proofs) to pass the termination check.

As PML_2 can be used to prove program equivalences, the inconsistency that would be introduced by possible non-termination would allow proving any program equivalence by inhabiting the corresponding type. Moreover, a non-terminating program would allow invalid program equivalences to be added to the equational context. The following invalid program (first given in Section 3) gives an example of such a scenario.

```
// val rec add_n_Zero_loop : ∀n∈nat, add n Zero ≡ n =
//   fun n {
//     add_n_Zero_loop n
//   }
```

[17] In this case, adequacy can still be proved by well-founded induction on the structure of the typing proof.
[18] Note that the `Rank2Types` (or `RankNTypes`) extension is required for the definition of `Empty`.
[19] Note that `empty` is encoded using a polymorphic record field.

Here, the recursive call `add_n_Zero_loop n` brings into the equational context the equivalence `add n Zero` $\equiv$ `n`, which exactly corresponds to the goal of the proof.

▶ Remark. The example `add_n_Zero_loop` must be rejected because the underlying program (and hence proof structure) is non terminating (and hence not well-founded). The management of equivalences being correct by construction, incorrect equations can only be proved in a contradictory equational context. In the example, a faulty equation is learned when the non-well-founded recursive call is made.

5.2 Sized types

The termination checking technology used by PML_2 is based on a notion of size that is attached to typing judgments. In fact, inductive and coinductive types are annotated using an ordinal size indicating the number of times their definition can be unfolded (this is usual in the context of sized types [1, 13, 30, 19]). Inductive or coinductive types, such as lists or streams, can then be seen as sized types annotated by a large enough (limit) ordinal.[20]

In practice, the implementation of PML_2 introduces sizes automatically when typing recursive functions. This means that it replaces (some) inductive and coinductive types, which carry a limit ordinal, with universal quantification over all possible ordinals. Note that it is only possible to do so when the obtained type is more general that the one that was given by the user. To enforce this invariant, we only introduce quantification on inductive types in negative position, and on coinductive types in positive position. In practice, this heuristic works well on simple functions, but the user is sometimes required to annotate the functions with explicit quantifications for termination to be established. Moreover, manual annotation may lead to more precise types, leading to more examples passing the termination check. For example, the following `map` function is accepted by the implementation.

```
val rec map : ∀a b, (a ⇒ b) ⇒ list⟨a⟩ ⇒ list⟨b⟩ =
  fun fn l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = fn c.hd; tl = map fn c.tl}]
    }
  }
```

However, if the user writes a complex recursive function containing a recursive call through the `map` function, it will not be possible to establish its termination (despite the fact that `map` does not change the size of the list it is applied to). To solve this problem, the user may rather use a more precise sized type, which is a subtype of the former type.

```
type slist⟨s:κ, a:o⟩ = μ_s slist, [Nil ; Cns of {hd : a; tl : slist}]

val rec map : ∀s, ∀a b, (a ⇒ b) ⇒ slist⟨s,a⟩ ⇒ slist⟨s,b⟩ =
  fun fn l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = fn c.hd; tl = map fn c.tl}]
    }
  }
```

[20] In practice, ω is sufficient for most of the usual data types, but this is not true in general. Nonetheless, there exists an ordinal that is large enough for all the inductive and coinductive types to converge [19].

▶ Remark. The type `slist⟨s,a⟩` should not be confused with the type of vectors `vec⟨a,s⟩` defined in Section 3.4. Although they are both subtypes of regular lists, the former carries an ordinal s (of sort κ) that can be used by the termination checker to establish size relations, while the latter contains a term (of sort τ) corresponding to the size of the list (as computed by the `length` function) which cannot be used by the termination checker. Note however that these two types could be easily combined.

▶ Remark. The above type of `map` only enforces that the output list is at most as long as the input list. For instance, we could give the same type to a function taking the same two arguments and always returning an empty list.

A similar scheme can be applied to *insertion sort* for example, but not for *quick sort*, as discussed in previous work [19]. Indeed, a richer language of ordinals would be required to express the fact that the partition function preserves the number of elements of its input.[21]

5.3 Proof by equivalence to a terminating function

Although the current implementation of PML_2 checks the termination of all programs (not only proofs), it is possible to use the specific features of the system to write termination proofs. Indeed, the equivalence relation on which the system relies can be used to substitute one term with another, provided that they are equivalent. This means that if we want to establish the termination of a program that does not pass the termination check directly, then we can instead establish the termination of any equivalent program. We will here consider the example of the well-known *McCarthy 91* function, which termination cannot be established by most of the existing termination criteria (if not all).

```
include lib.nat
include lib.nat_proofs

def mccarthy91_hard =
  fix fun mccarthy91 n {
    if gt n u100 {
      minus n u10
    } else {
      mccarthy91 (mccarthy91 (add n u11))
    }
  }

// val mccarthy91 : nat ⇒ nat =
//   mccarthy91_hard
```

Note that here, the value `mccarthy91_hard` is not defined as a usual, type checked and termination checked value, but as a value object (using the `def` keyword). This means that this function can be manipulated as an object of the logic, but not evaluated directly. Note also that we rely on some functions (and constants) defined in the standard library of PML_2. The `minus` function computes the difference, and the `gt` function tests whether its first argument is strictly greater than its second argument.

Although PML_2 is not able to prove the termination of the commented version of `mccarthy91`, we can give the following alternative (but equivalent) definition.

[21] It is nonetheless possible to show that quick sort is size-preserving using a PML_2 proof.

```
val mccarthy91_easy : nat ⇒ nat =
  fun n {
    if gt n u100 {
      minus n u10
    } else {
      u91
    }
  }
```

This second definition passes our termination check (it is not even recursive), but it does not really correspond to the traditional definition of the *McCarthy 91 function*, which is a shame. We can nonetheless write a PML_2 proof showing that both definitions are equivalent, which will then allow us to replace one with the other. To prove the equivalence, we first need to show that `mccarthy91_hard n` has value `u91` for all numbers that are not greater than `u100`.

```
val hard_lemma : ∀n∈nat, gt n u100 ≡ false ⇒ mccarthy91_hard n ≡ u91 =
  fun n eq {
    {- ... -} // Can be done by enumerating the domain.
  }
```

We do not give the full proof for lack of space, but it can be easily completed since the domain of quantification is finite. One simply needs to explore the domain by pattern matching on `n`, obtaining a trivial proof for all numbers less or equal to `u100`. In the case of numbers greater that `u100`, the presence of an additional successor produces a contradiction with the hypothesis `gt n u100 ≡ false` which allows the enumeration to remain finite.

▶ Remark. This brute-force approach, although it could be easily automated, yields a proof that it rather inefficient. A better solution would be to write a proof by "induction", which is what one would do on paper.

Using the above lemma, we can then show that the two implementations of the *McCarthy 91 function* produce the same result on every natural number as follows.

```
val hard_is_easy : ∀n∈nat, mccarthy91_easy n ≡ mccarthy91_hard n =
  fun n {
    use gt_total n u100;
    if gt n u100 {
      deduce mccarthy91_easy n ≡ minus n u10;
      deduce mccarthy91_hard n ≡ minus n u10;
      qed
    } else {
      deduce mccarthy91_easy n ≡ u91;
      show mccarthy91_hard n ≡ u91 using hard_lemma n {};
      qed
    }
  }
```

The proof is straightforward[22] since the two implementations have the same structure, and they share the same "then" branch. In the case of the "else" branch, `hard_lemma` can be used to conclude. We can then type check and prove the termination of the original version of the *McCarthy 91 function* as follows.

[22] None of the `deduce` annotations are necessary, they are only provided for clarity. The `gt_total` lemma is defined in the standard library, more detail about its purpose will be given in the next section.

```
val mccarthy91 : nat ⇒ nat =
  fun n {
    check mccarthy91_easy n  // Term used for type-checking.
      for mccarthy91_hard n  // Actual term used in the definition.
      because hard_is_easy n // Proof that they are equal.
    // The above really is "mccarthy91_hard n" (up to erasure).
  }
```

The annotation used in `mccarthy91` instructs the type-checker to substitute `mccarthy_hard n` with `mccarthy_easy n` in the construction of the typing proof. This is only possible because these two terms are equivalent (when `n` has type `nat`), as witnessed by `hard_is_easy n`. However, the term used for the computation will indeed be `mccarthy_hard n` after the annotations are erased.

▶ Remark. As all the types of PML_2 are closed under equivalence, it is always possible to replace a term by another equivalent term. This technique can not only be used for proving termination of functions such as `mccarthy91`, but also for typing terms that would not be typable otherwise (but that are, for example, more efficient).

▶ Remark. Note that we did not prove `mccarthy_hard` ≡ `mccarthy_easy`, which may not even be true. Indeed, equivalence considers these two terms as untyped, and it is very well possible that they can be distinguished by a certain evaluation context.[23] A simpler example arises when comparing different implementations of the identity function on natural numbers: we have `(fun n { case n { Zero ⇒ n | S[_] ⇒ n } }) k` ≡ `(fun n { n }) k` for all `k` in `nat`, but `fun n { case n { Zero ⇒ n | S[_] ⇒ n } }` ≡ `fun n { n }` is false. These two functions can be distinguished using the argument `false`, which yields a pattern matching failure on the former, while the latter successfully returns `false`.

5.4 Internal totality proofs

In this last section, we will give more explanations about the so-called "totality proofs" that are currently required in PML_2. A function is said to be *total* if it computes some value, when applied to any value of its domain. In PML_2, the totality of functions can be expressed inside the system using an existential quantification. We can thus write internal totality proofs such as the following.

```
val rec add_total : ∀n m∈nat, ∃v:ι, add n m ≡ v =
  fun n m {
    case n {
      Zero → qed
      S[k] → use add_total k m; qed
    }
  }
```

▶ Remark. Note that the value that is obtained by applying the function is not relevant here, nor is its type. We could however modify the definition of `add_total` to make sure that an element of type `nat` is returned. In this case, we could even use `add_total` as `add`.

[23] In PML_2, the equivalence `t` ≡ `u` being provable implies that `t` and `u` are observationally equivalent, which means that they have the same "observable behaviour" in every possible evaluation context. Note that we only observe termination, versus divergence or runtime error [17, 16].

The reason why totality proofs are required in the system is strongly related to the call-by-value evaluation strategy of the language. Indeed, in call-by-value, a function can only be applied when all of its arguments are values. More precisely, it only makes sense to reduce a β-redex if the term in argument position is a syntactic value. To understand where the notion of totality is really required, let us consider the following proof example showing the associativity of addition.

```
val rec add_assoc : ∀m n p∈nat, add m (add n p) ≡ add (add m n) p =
  fun m n p {
    use add_total n p;
    case m {
      Zero → qed
      S[k] → use add_assoc k n p; use add_total k n; qed
    }
  }
```

Let aside the first call to `add_total`, the proof starts by a case analysis on variable `m`. Let us consider the `Zero` case, which already illustrates very well the necessity for the totality proof. In this branch, the automatic decision procedure learns the equation `m ≡ Zero`. As a consequence, the goal simplifies to `add Zero (add n p) ≡ add (add Zero n) p`, and even as `add Zero (add n p) ≡ add n p` since both `Zero` and `n` are values (the function can thus be applied). However, the left-hand side of the equation cannot reduce further because `add n p` is not a value. We can then only proceed using the totality proof produced by `add_total n p`, which gives us a value `v` such that `add n p ≡ v`. As a consequence, this allows us to obtain `add Zero (add n p) ≡ add n p` as follows.

$$\texttt{add Zero (add n p)} \equiv \texttt{add Zero v} \equiv \texttt{v} \equiv \texttt{add n p}$$

▶ Remark. It is clear that the totality proof corresponding to a given function has a similar structure as the definition of the function itself. We may thus hope that totality proofs can be generated and called automatically, at least in most cases.

6 Future work

The current implementation of PML_2 already allows for several convincing examples. However, theoretical work and implementation work remain to be done for the language to become fully practical, both as a programming language and as a proof assistant.

6.1 Mixing termination and non-termination

As mentioned earlier, termination checking is only necessary for PML_2 programs that are considered as proofs. In the theory, proving that a program terminates amounts to showing that its typing derivation has a well-founded circular structure. In this case, a standard semantic proof can be used to prove normalisation [19], the essential point being that the adequacy of the type system can be established by induction on the circular structure of proofs[24], provided that they are well-founded.

Alternatively, it is possible to type programs with standard (non-circular) typing proofs,[25] to the expense of losing normalisation since our termination criterion works by analysing

[24] Circularity is introduced by the typing rule for the fixed-point combinator.
[25] This feature is not available in the current implementation, which only accepts terminating programs.

the circular structure of proofs. Note that lack of termination checking implies the loss of soundness, but type-safety is nonetheless preserved. As it is hard to automatically prove the termination of programs, it is clear that a user will not want to be restricted to programs that can be proved terminating. For this purpose, it is important to allow arbitrary (type-safe) programs to be written, if only to prove them terminating later (examples of such programs can be found in previous work [27]).

As programs that can be proven terminating can be typed in both ways, it is natural to consider a way of mixing the two approaches in the theory. This has actually already been implemented in a particular branch of our implementation (called *totality*) [18]. The corresponding extension of the theory has also been checked informally.

6.2 Other forms of effects, mutation

One of the distinguishing features of PML_2 is the possibility for programs to manipulate their own continuation (or evaluation context). This is achieved using a construct similar to Scheme's *call/cc*, or rather Michel Parigot's μ-abstraction [25], which triggers a form of effect. As shown in Section 2.1, it can be used to realize theorems which only hold classically, by extracting a program from their proofs [26].

Although PML_2 is the first proof system based on a programming language with effects and a classical realizability model [17], one may argue that control structures only have a limited interest for writing practical ML programs.[26] Other forms of effects however, for example input/output directives or mutable cells, are essential to ML programmers. Although it should be relatively easy to extend the system with the former, the latter poses a real technical challenge. Indeed, it is not yet known how to account for mutation in a classical realizability model.

6.3 Subject reduction and strong safety

The theory of PML_2 is based on a realizability model, which has the major advantage of being flexible. More precisely, the adequacy lemma, which is the keystone of the development, only needs to be modified locally to encompass a new typing or subtyping rule. However, we have not yet proved any subject reduction result for the system, and thus we only have a weak form of type safety.

6.4 Extensible variants and records (better inference)

The current type-system of PML_2 requires a relatively small amount of type annotations (at least for programs). Nonetheless, the system relies on unification in several places, and it may happen that the system guesses the wrong types. This situation arises most often with variant and record types, for which some fields or constructors might be left out. This problem can be solved using extensible variant types and record types, but we will need to make sure that this does not pose any problem in the theory.

6.5 Support for mutually recursive function

In the current implementation, PML_2 lacks the possibility of defining mutually recursive functions. Although it is always possible to encode mutual recursion using additional

[26] They can however be used to encode a form of exception mechanism.

parameters, this method does not perform very well when combined with our termination checking technology. We thus need to consider a different fixed-point instruction for our abstract machine, which does not seem to pose any theoretical problem. The idea is to replace the current fixed-point instruction with a term constructor $\varphi a.v$, binding the term variable a into the value v, and with the reduction rule $\varphi a.v \to v[a := \varphi a.v]$.[27] Mutual recursion can then be encoded using a value v that is a record containing several λ-abstractions, which can still be typed using a simple unfolding rule (as in previous work [19, 17]).

6.6 Certificates using proof traces for equivalences

For now, it is not possible to formally check the proofs produced by PML_2 in another system. Although the system already records the proof trees that are produced during type-checking, the decision procedure for program equivalence yet lacks the ability of producing a proof trace. However, there is no theoretical evidence that it would not be possible for the decision procedure to record enough information for an external prover (for example Coq [23] or Dedukti [31]) to check the proofs produced by PML_2.

7 Similar systems

To conclude this paper, we will compare PML_2 to other proof systems and languages that can be used to formalise and prove program properties, or that rely on similar principles.

7.1 Dependent types in ML

To our knowledge, the combination of call-by-value evaluation, side-effects and dependent products has never been achieved before. At least not for a dependent product fully compatible with effects and call-by-value. For example, the Aura language [14] forbids dependency on terms that are not values in dependent applications. Similarly, the $F^\star$ language [32] relies on (partial) let-normal forms to enforce values in argument position. Daniel Licata and Robert Harper have defined a notion of positively dependent types [20] which only allow dependency over strictly positive types. Finally, in languages like ATS and DML [36, 37], dependencies are limited to a specific index language.

7.2 Tools based on intuitionistic type theory

The most actively developed proof assistants following the Curry-Howard correspondence are Agda and Coq [24, 23]. The former is based on Martin-Löf's dependent type theory and the latter on Coquand and Huet's calculus of constructions [7, 22]. These two constructive theories provide dependent types, which allow the definition of very expressive specifications. Contrary to PML_2, Coq and Agda do not directly give a computational interpretation to classical logic. Classical reasoning can only be done through a negative translation or with the definition of axioms such as the law of the excluded middle. In particular, these two languages are not effectful. However, they are logically consistent, which means that they only accept terminating programs. As termination checking is a difficult (and undecidable) problem, many terminating programs are rejected. Although this is not a problem for formalizing mathematics, this makes programming tedious. In PML_2, only proofs really need

[27] Term variables should not be confused with value variables (or λ-variables). In particular, the former can be substituted with any term, while the latter can only be substituted with values.

to be shown terminating, and it is in any case possible to reason about non-terminating and even untyped programs as they can be manipulated as objects in types.

7.3 NuPrl and refinement types

The NuPrl system [6] has many similarities with PML_2 on the theoretical side, although it is inconsistent with classical logic. NuPrl accommodates an observational equivalence relation similar to ours (Howe's *squiggle* relation [12]), which is partially reflected in the syntax of the system. Being based on a Kleene-style realizability model, NuPrl can also be used to reason about untyped terms. Another major difference between PML_2 and NuPrl is that the latter is based on refinement types, which means that it does not have an automatic way of building typing derivations for programs. Indeed, typing derivations are built interactively using a specific interface, and the user must say what typing rule should be applied first.

7.4 Partially consistent languages

The TRELLYS project [3] aims at providing a language in which a consistent core interacts with type-safe dependently typed programming with general recursion. Although the language is call-by-value and effectful, it suffers from value restriction like Aura [14]. The value restriction does not appear explicitly but is encoded into a well-formedness judgement appearing as the premise of the typing rule for application. Apart from value restriction, the main difference between the language of the TRELLYS project and ours resides in the calculus itself. Their calculus is Church-style (or explicitly typed) while ours is Curry-style (or implicitly typed). In particular, their terms and types are defined simultaneously, while our type system is constructed on top of an untyped calculus.

7.5 Systems aimed at proving programs in ML

Several systems have been proposed for proving ML programs. ProPre [21] relies on a notion of *algorithms*, corresponding to equational specifications of programs. It is used in conjunction with a type system based on intuitionistic logic. Although it is possible to use classical logic to prove that a program meets its specification, the underlying programming language is not effectful. Similarly, the PAF! system [2] implements a logic supporting proofs of programs, but it is restricted to a purely functional subset of ML. Another approach for reasoning about purely functional ML programs is given in the work of Yann Regis-Gianas [28], where Hoare logic is used to specify program properties. Finally, it is also possible to reason about ML programs (including effectful ones) by compiling them down to higher-order formulas [4, 5], which can then be manipulated using an external prover such as Coq [23]. In this case, the user is required to master at least two languages, contrary to our system in which programming and proving take place in a uniform framework.

References

1 Andreas Abel. Semi-continuous sized types and termination. *Logical Methods in Computer Science*, 4(2), 2008.

2 Sylvain Baro. *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML*. PhD thesis, Paris Diderot University, France, 2003.

3 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, pages 33–46. ACM, 2014.

4 Arthur Charguéraud. Program verification through characteristic formulae. In *ICFP*, pages 321–332. ACM, 2010.

5 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011.

6 Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

7 Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

8 Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

9 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

10 Timothy Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58. ACM Press, 1990.

11 Robert Harper and Mark Lillibridge. ML with callcc is unsound. Posted to the SML mailing list, 1991. URL: `http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html`.

12 Douglas J. Howe. Equality in lazy computation systems. In *LICS*, pages 198–203. IEEE Computer Society, 1989.

13 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423. ACM Press, 1996.

14 Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: a programming language for authorization and audit. In *ICFP*, pages 27–38. ACM, 2008.

15 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92. ACM, 2001.

16 Rodolphe Lepigre. A classical realizability model for a semantical value restriction. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

17 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, Grenoble Alpes University, France, 2017.

18 Rodolphe Lepigre and Christophe Raffalli. Implementation of the PML_2 language. The source code is available on GitHub at `https://github.com/rlepigre/pml`, 2016.

19 Rodolphe Lepigre and Christophe Raffalli. Practical Subtyping for Curry-Style Languages. *to appear in ACM Trans. Program. Lang. Syst.*, 2018. Draft and joined implementation available at `https://rlepigre.github.io/subml/`.

20 Daniel R. Licata and Robert Harper. Positively dependent types. In *PLPV*, pages 3–14. ACM, 2009.

21 Pascal Manoury, Michel Parigot, and Marianne Simonot. ProPre A Programming Language with Proofs. In *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 484–486. Springer, 1992.

22 Per Martin-Löf. Constructive mathematics and computer programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 198.

23 The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2018. Version 8.8.0. URL: `http://coq.inria.fr`.

24 Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

25 Michel Parigot. Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

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

27 Christophe Raffalli. PML: a new proof assistant. Prototype implementation available at `http://lama.univ-savoie.fr/~raffalli/pml`, talk at the TYPES workshop, 2007.

28 Yann Régis-Gianas. *From types to logical assertions : automatic or assisted proofs of property about functional programs. (Des types aux assertions logiques : preuve automatique ou assistée de propriétés sur les programmes fonctionnels).* PhD thesis, Paris Diderot University, France, 2007.

29 John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

30 Jorge Luis Sacchini. Type-based productivity of stream definitions in the calculus of constructions. In *LICS*, pages 233–242. IEEE Computer Society, 2013.

31 Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique).* PhD thesis, Mines ParisTech, France, 2015.

32 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

33 Philip Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Notices*, 38(9):189–201, 2003.

34 Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

35 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

36 Hongwei Xi. Applied type system: Extended abstract. In *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2003.

37 Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL*, pages 214–227. ACM, 1999.