Higher-Order Symb Comput
DOI 10.1007/s10990-012-9083-6

Path resolution for nested recursive modules

Jacques Garrigue · **Keiko Nakata**

Abstract The ML module system facilitates the modular development of large programs, through decomposition, abstraction and reuse. To increase its flexibility, a lot of work has been devoted to extending it with recursion, which is currently prohibited. The introduction of recursion adds expressivity to the module system. However it also creates problems that a non-recursive module system does not have.

In this paper, we address one such problem, namely resolution of path references. Paths are how one refers to nested modules in ML. Without recursion, well-typedness guarantees termination of path resolution, in other words, we can statically determine the module that a path refers to. This does not always hold with recursive module extensions, since the module system then can encode a lambda-calculus with recursion, whose termination is undecidable regardless of well-typedness. We formalize this problem of path resolution by means of a rewrite system on paths and prove that the problem is undecidable even without higher-order functors, via an encoding of the Turing machine into a calculus with just recursive modules, first-order functors, and nesting. Motivated by this result, we introduce a further restriction on first-order functors, limiting polymorphism on functor parameters by requiring signatures for functor parameters to be non-recursive, and show that this restriction is decidable and admits terminating path resolution.

Keywords The ML module system · Recursive modules · Ground term rewriting · Decidability · Termination

1 Introduction

Modularity is an important factor in the smooth development and maintenance of large programs. Many modern programming languages have mechanisms to support modular development of programs. Among such mechanisms, the ML module system is well-known for

J. Garrigue
Graduate School of Mathematics, Nagoya University, Chikusa-ku, Nagoya 464-8602, Japan
E-mail: garrigue@math.nagoya-u.ac.jp

K. Nakata
Institute of Cybernetics, Tallinn Univ. of Tech., Akadeemia tee 21, Tallinn, 12618, Estonia
E-mail: keiko@cs.ioc.ee

its strong support of program structuring [21,18]. Two of its important features are nested structures and functors. ML modules are nestable, that is, a module can contain submodules, as well as type and value definitions; nesting is a simple but powerful way to organize program codes and namespaces hierarchically. ML supports functions on modules, so-called functors, which facilitate code reuse in a modular way.

Despite this flexibility, ML prohibits recursion between modules. That is to say, recursive type or function definitions may not cross module boundaries. As a result of this constraint, programmers may have to consolidate conceptually separate components into a single module, intruding on modular programming [29]. The absence of recursive modules also hinders extensible program development [25].

Introducing recursive modules is a natural way out of this predicament. Recursion is a powerful language construct, hence its addition certainly increases the expressivity of the module system. At the same time the addition might jeopardize the static guarantees on safety that current ML enjoys, such as decidable type checking or error-free module initialization. Indeed, extending ML with recursive modules poses non-trivial problems and a lot of work has been devoted to investigate extensions that retain desirable safety guarantees while not introducing too many constraints [2,29,6,1,12,5].

In our previous work [25], we have studied an applicative module system [16] with polymorphic functors and recursion based on paths. Whereas several different approaches to accounting for ML-style modules have been proposed [21,30,17,11,9], we found a path-based approach natural from the programmer's viewpoint. One problem we encountered is resolution of path references. Paths, also known as qualified identifiers, are the way ML refers to nested modules and their contents. In the absence of recursive modules, success of path resolution is guaranteed via type checking, which in turn ensures that the runtime will find the module that the path refers to. Technically this is akin to the strong normalization property enjoyed by the simply typed lambda calculus. However this property does not hold once recursion is introduced, since the module system can then encode a lambda calculus with recursion, whose termination is undecidable regardless of well-typedness. The possibility for divergence during path resolution impacts safety. Type checking may not terminate, since determining type equality would require path resolution. For instance, the type checker of OCaml diverges on the following program:

```
module type M = sig
 module rec F : functor(X : sig type t end) → sig type t = F(F(X)).t end
end
```

Furthermore, resolving path references in order to prepare code for execution may diverge, which would contradict phase distinction [11].

In this paper, we examine and address the problem of path resolution. We formalize path resolution by defining a rewrite system on paths (Section 3 and 4). Then we prove that termination of path resolution is undecidable even without higher-order functors, via an encoding of any Turing machine into a calculus with just recursive modules, first-order functors, and nesting (Section 5). The result is interesting since it attests to the expressivity of nested structures, which are a distinguishing feature of ML modules but often receive less attention than functors. This result, together with the observations on decidable subsystems given in Section 5.1, lead us to a further restriction on first-order functors, by requiring signatures for functor parameters to be non-recursive. In Section 6, we formalize this restriction and show that path resolution is terminating when the restriction is enforced. We then develop a terminating algorithm which verifies that the restriction holds and that there are no dangling or diverging paths. These two technical results, namely proof of the undecidability of a first-order subsystem and introduction of a decidable restriction, are the main contributions of this paper.

2 Background

In our previous work [25], we studied a type system for recursive ML-style modules with applicative functors, which we named *Traviata*. In this section we give an overview of this type system, which motivates our study of path resolution in this paper.

The module system of OCaml [20] adopts applicative functors [16] and has been extended with recursive modules [19]. The type system of *Traviata* is very much inspired by that of OCaml. In particular it features both applicative functors and recursive modules. Indeed, we worked on *Traviata* to address deficiencies of OCaml's applicative functors, and to formally study the use of "paths" as central concept for an extension of the applicative module system with recursion.

The key extension[1] we made for *Traviata*, in comparison to OCaml, is that the signature language keeps an account of module abbreviations and that module path equality is determined in terms of which modules the paths refer to, instead of syntactically comparing those paths. Below we elaborate on these points.

Consider the following module definitions (written in OCaml-like syntax):

```
module Int = struct type t = Int of int end
module I = Int
```

In *Traviata*, Int and I have the following signatures respectively:

```
module Int : sig type t = Int of int end
module I : Int
```

In the above signature, "module I : Int" expresses both an equivalence between paths (I is equivalent to Int) and signatures (I has the same signature as Int).

The main reason for strengthening path equality is to circumvent a source of incompleteness in OCaml's applicative functors, which arises from the fact that the type system does not record module abbreviations, but only keeps track of (core) type equalities introduced by those abbreviations. For instance consider the following program:

```
module F =
  functor(X : sig type t end) → struct type t = A of X.t end
module AofInt1 = F(Int)
module AofInt2 = F(Int)
module AofInt3 = F(I)
```

The two types Int.t and I.t are equivalent thanks to type strengthening [16]. Functors being applicative, AofInt1.t and AofInt2.t are equivalent too. However, the two types AofInt1.t and AofInt3.t are not equivalent in OCaml, because equality on module paths is purely syntactic, and as a result does not take into account the fact that I is an abbreviation for Int.

[1] In *Traviata*, structures as well as signatures for structures are also extended with declarations of self variables, or recursion variables. That extension is not essential with respect to OCaml, but rather a design choice at the level of the surface syntax, which facilitates the formal study.

This kind of incompatibility is counter-intuitive. Indeed, the type system already keeps track of (core) type abbreviations and unfolds them when necessary. Thus from the programmer's viewpoint type abbreviations are just a notational convenience and do not affect typability; why should module abbreviations be any different? This deficiency of OCaml's type system was addressed in *Traviata*. The idea is simple: we extend the type system to keep track of module abbreviations too. As a result AofInt1.t and AofInt3.t are equivalent in *Traviata*, since I is unfolded to Int. As seen above, this is handled by allowing module paths inside signatures; in the above case, the signature of I is Int, allowing us to recover the path equality from the signature alone. This is also true for functors. For instance, the functor

```
module Id = functor (X : sig end) → X
```

will be given the signature

```
module Id : functor (X : sig end) → X
```

meaning that it fully implements identity: for any module M, Id(M) will be equivalent to M. This is not the case in OCaml or Standard ML, where a module variable only stands for the components explicitly included in its signature (*i.e.* the signature of Id would be the useless functor (X : sig end) → sig end), but this comes naturally if we track abbreviations of modules. We can of course obtain the weaker type in *Traviata* too, by ascribing the body of the functor with an opaque signature. Note also that this polymorphism (but not the type equality) can be simulated in OCaml by using an abstract signature:

```
module Id : functor (X : sig module type S module M : S end) → X.S
          = functor (X : sig module type S module M : S end) → X.M
```

Like in system F, this functor can be applied to a module containing any signature S and any module M satisfying S, and returns M itself. This is less flexible than what we propose, since abstract signatures can only be completely abstract, while in our approach we could give a more precise signature to X, allowing access to its components in the body of Id. We still think that, at least in the absence of recursive modules, it should be possible to encode *Traviata*'s polymorphism using abstract signatures, so that this difference is more a question of flexibility than expressivity.

The strengthened module path equality is all the more necessary once recursion is introduced. Since the same module may be accessed through different recursion variables, we can no longer rely on the unicity of syntactic paths.

```
module L = struct (Z)
  module M = struct (Z′)
    module N = struct type t = Int of int end
  end
end
```

In the above example, Z and Z′ are recursion variables as in Moscow ML [29]. Given the functor F defined earlier, one would expect the type equality F(Z.M.N).t = F(Z′.N).t to hold inside N for the very reason that Z.M.N and Z′.N refer to the same module. Note that inside N both Z and Z′ are in scope. This shows how weak syntactic module path equality is in the presence of recursion.

```
module TreeForest =
 functor(X : sig type t val compare : t → t → bool end) →
  struct (TF)
    module S = Set.Make(X)
    module Tree = struct
     module F = TF.Forest
     type t = Leaf of X.t | Node of X.t * F.t
     let split = λx.case x of Leaf i ⇒ [Leaf i]
         | Node (i, f) ⇒ (Leaf i) :: f
     let labels = λx.case x of Leaf i ⇒ TF.S.singleton i
         | Node (i, f) ⇒ TF.S.add i (F.labels f)
    end
    module Forest = struct
     module T = TF.Tree
     type t = T.t list
     let sweep = λx.case x of [] ⇒ []
         | (T.Leaf y) :: tl ⇒ (T.Leaf y) :: (sweep tl)
         | (T.Node y) :: tl ⇒ sweep tl
     let labels = λx.case x of [] ⇒ TF.S.empty
         | hd :: tl ⇒ TF.S.union (T.labels hd) (labels tl)
     let incr = λf.λt.let l1 = labels f in
         let l2 = T.labels t in
         if TF.S.diff l1 l2 != TF.S.empty then (t :: f) else f
    end
   end
```

Fig. 1 Trees and Forest

The strengthened module path equality also fixes another source of incompleteness in OCaml's applicative functors, in that the type system cannot establish some type equalities that hold with generative functors [2]. Below is an example of this incompleteness.

```
module M = struct
  module N = struct type t = int let compare x y = x − y end
  module S = Set.Make(N)
  let empty = S.empty
end
module M′ = M
let _ = M′.empty = M.empty
```

The last line does not type check in OCaml since M′.empty has type MakeSet(M′.N).t, whereas M.empty has type MakeSet(M.N).t. It type checks in *Traviata*: the module path equality between M′ and M is kept in the signature, from which it follows that M′.N and M.N refer to the same module.

In Figure 1 we give a typical use of recursive modules: Tree and Forest refer to each other recursively, defining recursive data types and functions on them that cross module boundaries. The module TreeForest is given the signature in Figure 2, keeping module path

[2] This problem alone could be fixed by adapting Dreyer's proposal [7]. But as motivated by the earlier examples, we believe our approach is more robust in the presence of recursion.

```
module TreeForest :
 functor(X : sig type t val compare : t → t → bool end) →
  struct (TF)
    module S : Set.Make(X)
    module Tree : sig
     module F : TF.Forest
     type t = Leaf of X.t | Node of X.t * F.t
     val split : t → F.t
     val labels : t → S.t
    end
    module Forest : sig
     module T : TF.Tree
     type t = T.t list
     val sweep : t → t
     val labels : t → S.t
     val incr : t → t
    end
   end
```

Fig. 2 Signature of *TreeForest*

$$\begin{array}{lcll}
\textit{Expressions} & e & ::= & \{m_1 = e_1 \cdots m_n = e_n\} \mid \lambda(x)e \mid p \\
\textit{Paths} & p,q & ::= & \varepsilon \mid x \mid p.m \mid p_1(p_2) \\
\textit{Program} & P & ::= & \{m_1 = e_1 \cdots m_n = e_n\}
\end{array}$$

Fig. 3 Syntax

equalities between S and MakeSet(S), F and TF.Forest, and T and TF.Tree. These equivalences are necessary to type check TreeForest.

As we have shown, the equivalence of nominal types can be decided using our strengthened module path equality: given two module paths p and q and a type name t, the types $p.t$ and $q.t$ are equivalent if and only if p and q are equal. Handling equality at the module level leads to some simplifications, for instance it avoids the need for type strengthening (*i.e.* the addition of equations at the type level). This equality is strong. For instance $p.M$ and $q.M$ are equal if p is an abbreviation of q and q has a submodule named M. Moreover, $F(M)$ and M are equal if F is an identity functor. Path normalization is a natural way to check module equality, as normal paths can be compared syntactically. For type checking to be decidable, path normalization must be terminating, which is the subject of study in this paper.

3 Syntax

In Figure 3 we define a small record calculus for our formal study. We use m as a metavariable for field names of records and x for variables.

An expression, ranged over by e, is either a *structure*, a *functor* or a *path*. A structure $\{m_1 = e_1 \cdots m_n = e_n\}$ is a sequence of bindings, or a record of expressions e_i labeled with names m_i. A functor $\lambda(x)e$ represents a function on expressions; x is the name of the formal parameter and e is the body, in which x is bound.

Paths (ranged over by p and q) are the most interesting construct of the calculus. They are built from 1) the root path ε, which refers to the toplevel structure; 2) variables x; 3) the dot notation "$p.m$", representing access to the field named m of the structure that p refers to; and 4) functor application $p_1(p_2)$, which applies the expression that p_1 refers to to the expression that p_2 refers to. As we shall see in an example later, a path can refer to a field at any level of nesting within the toplevel structure regardless of field ordering. Thus paths introduce recursion into the calculus. We may call a binding $m = p$ in a structure an *abbreviation binding*[3].

A program, ranged over by P, is a (toplevel) structure. All occurrences of the root path ε in a program are considered to refer to the toplevel structure.

We assume the following two conventions: 1) No sequences of bindings in a structure bind the same name twice; 2) No programs contain free variables, where free occurrence of variables is defined in the standard way.

The calculus is kept small so as to focus on the core technical issues studied in the paper. It is meant to be an abstraction of the signature language of *Traviata*: expressions actually represent signatures and a program corresponds to the signature of a toplevel structure. In order to relate to the intuition of path reduction, the syntax we chose for this formal calculus is different. When translating a *Traviata* recursive signature into a structure of this calculus, one just has to replace all self references with paths starting from ε, applying the following conversion to the whole recursive signature:

$$\begin{aligned}
[\![S]\!] &= [\![S]\!]^{\varepsilon}_{id} \\
[\![sig(Z)\ module\ M_1 : S_1\ \ldots\ module\ M_n : S_n\ end]\!]^{p}_{\sigma} & \\
&= \{M_1 = [\![S_1]\!]^{p.M_1}_{\sigma[Z\mapsto p]} \ldots M_n = [\![S_n]\!]^{p.M_n}_{\sigma[Z\mapsto p]}\} \\
[\![functor\ (X : S_1) \to S_2]\!]^{p}_{\sigma} &= \lambda(X)[\![S_2]\!]^{p(X)}_{\sigma} \\
[\![q]\!]^{p}_{\sigma} &= \sigma(q)
\end{aligned}$$

Note that we drop the annotations on functor arguments. It may seem that in doing so we are changing the nature of the problem studied. One first remark is that, in *Traviata*, annotations on functor arguments can be omitted, relying on type inference to reconstruct them. So this calculus could be seen as a model of signature inference for *Traviata* programs without annotations.

But the deeper reason for starting with such an untyped calculus, is that we do not want to fix a specific type system too early on. We will see that our undecidability result in Section 5 is still valid in presence of some kind of type system for functor arguments. In Section 6 we will introduce another form of type system to recover decidability. This in turn should help us in designing a surface language with a powerful yet decidable type system. We will discuss this surface language in Section 6.4.

4 Semantics

Path resolution rewrites paths into *source form*. Until we formally define it later, source form can be explained as a normal form with no dangling references.

To provide some intuition for path resolution, let us consider the following program:

$$\begin{aligned}
\{\ & \mathtt{m}_1 = \{\mathtt{n}_1 = \{\mathtt{n} = \{\}\}\ \ \mathtt{n}_2 = \varepsilon.\mathtt{m}_1.\mathtt{n}_1\ \} \\
& \mathtt{m}_2 = \lambda \mathtt{x}.\{\mathtt{n}_1 = \{\}\ \ \mathtt{n}_2 = \mathtt{x}.\mathtt{n}_2\ \ \mathtt{n}_3 = \varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1\ \} \\
& \mathtt{m}_3 = \varepsilon.\mathtt{m}_2(\varepsilon.\mathtt{m}_1).\mathtt{n}_2\ \}
\end{aligned}$$

[3] The general form of an abbreviation binding for p is $m = \lambda(x_1)\ldots\lambda(x_n)p$.

The path $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$. Hence, the path $\varepsilon.\mathtt{m}_1.\mathtt{n}_2$, which is an abbreviation for $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$, refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$, too; we say that $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$ is the source form of $\varepsilon.\mathtt{m}_1.\mathtt{n}_2$. A path can contain functor applications. For instance, the path $\varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the body of the functor $\mathtt{m}_2$. We may need to perform computation to resolve path references. For instance the path $\varepsilon.\mathtt{m}_2(\varepsilon.\mathtt{m}_1).\mathtt{n}_2$ resolves to the source form $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$; by reducing the functor application, we obtain $\varepsilon.\mathtt{m}_1.\mathtt{n}_2$, which resolves to $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$, as we have explained above. Besides, paths may contain dangling references. For instance the path $\varepsilon.\mathtt{m}_1.\mathtt{n}_3$ is dangling since the structure $\mathtt{m}_1$ does not contain a field named $\mathtt{n}_3$.

In this section, we formalize path resolution by defining a rewrite system on paths. The intuition is straightforward. Continuing the above example, we extract the following four rewrite rules from it, by collecting abbreviation bindings:

$$\begin{array}{ll} \{\ \varepsilon.\mathtt{m}_1.\mathtt{n}_2 \rightarrow \varepsilon.\mathtt{m}_1.\mathtt{n}_1, & \varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_2 \rightarrow \mathtt{x}.\mathtt{n}_2, \\ \ \ \varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_3 \rightarrow \varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1, & \varepsilon.\mathtt{m}_3 \rightarrow \varepsilon.\mathtt{m}_2(\varepsilon.\mathtt{m}_1).\mathtt{n}_2\ \} \end{array}$$

According to these rules, we can induce the reduction steps

$$\varepsilon.\mathtt{m}_3 \rightarrow \varepsilon.\mathtt{m}_2(\varepsilon.\mathtt{m}_1).\mathtt{n}_2 \rightarrow \varepsilon.\mathtt{m}_1.\mathtt{n}_2 \rightarrow \varepsilon.\mathtt{m}_1.\mathtt{n}_1$$

which reflect the previous informal explanation of path resolution for $\varepsilon.\mathtt{m}_2(\varepsilon.\mathtt{m}_1).\mathtt{n}_2$.

4.1 Terminology

We first introduce some basic terminology and useful notation for our formalization.

For a path p, we write $args(p)$ to denote the set of paths that occur within p in functor argument positions, or:

$$\begin{array}{rlrl} args(\varepsilon) &= \emptyset & args(x) &= \emptyset \\ args(p.m) &= args(p) & args(p_1(p_2)) &= \{p_2\} \cup args(p_1) \end{array}$$

A path p is *ground* if p does not contain variables.

Substitutions, ranged over by θ, are finite mappings from variables to paths. We write $\mathrm{dom}(\theta)$ to denote the domain of θ. Application of a substitution θ to a path p, written $\hat{\theta}(p)$, is defined by:

$$\begin{array}{ll} \hat{\theta}(\varepsilon) = \varepsilon & \hat{\theta}(x) = \begin{cases} x & \text{when } x \notin \mathrm{dom}(\theta) \\ p & \text{when } x \in \mathrm{dom}(\theta) \text{ and } \theta(x) = p \end{cases} \\ \hat{\theta}(p.m) = \hat{\theta}(p).m & \hat{\theta}(p_1(p_2)) = \hat{\theta}(p_1)(\hat{\theta}(p_2)) \end{array}$$

We write $\theta[x \mapsto p]$ to denote a mapping extension. Precisely,

$$\theta[x \mapsto p](x') = \begin{cases} p & \text{when } x' = x. \\ \theta(x') & \text{when } x' \neq x \end{cases}$$

We write *id* to denote an identity mapping. In the rest of this paper we write $\theta(p)$ for $\hat{\theta}(p)$.

Path contexts, ranged over by $C[]$, are defined by:

$$C[] ::= [\cdot] \mid C[].m \mid C[](p) \mid p(C[])$$

where $[\cdot]$ denotes the empty context. We write $C[p]$ to denote the path obtained by placing p in the hole of the context $C[]$.

$$
\begin{array}{lcl}
\mathit{Rules}(p, \{m_1 = e_1 \ldots m_n = e_n\}) & = & \bigcup_{i=1}^{n} \mathit{Rules}(p.m_i, e_i) \\
\mathit{Rules}(p, \lambda(x)e) & = & \mathit{Rules}(p(x), e) \\
\mathit{Rules}(p, p') & = & \{p \to p'\}
\end{array}
$$

Fig. 4 Path rewrite rules of a program

A *path rewrite* rule is a pair (p, p') of paths. It will be written $p \to p'$. A path rewrite system R is a set of path rewrite rules $\{p_1 \to p'_1, \ldots, p_n \to p'_n\}$. A path *p rewrites into p′* in one step with respect to R if there is a substitution θ, a path context $C[]$ and a rewrite rule $p_i \to p'_i \in R$ such that $p = C[\theta(p_i)]$ and $p' = C[\theta(p'_i)]$. We write $p \to_R p'$ when p rewrites into p' in one step with respect to R and $p \xrightarrow{*}_R p'$ when p rewrites into p' in zero or more steps with respect to R, that is, $\xrightarrow{*}_R$ is the reflexive and transitive closure of $\to_R$. We may omit the subscript R when it is clear from the context. A path p is in *normal form* with respect to R if there is no q such that $p \to_R q$. A path q is a normal form of p with respect to R if $p \xrightarrow{*}_R q$ and q is in normal form with respect to R.

It may help to consider paths represented in a style closer to term rewriting. For instance, we could represent a path $\varepsilon.\mathtt{m}_1.\mathtt{m}_2(\varepsilon.\mathtt{m}_3)(\mathtt{x}).\mathtt{m}_4$ as $\mathtt{m}_4(\mathtt{app}(\mathtt{app}(\mathtt{m}_2(\mathtt{m}_1(\varepsilon)),\mathtt{m}_3(\varepsilon)),\mathtt{x}))$, where both ε and variables are of arity 0, names are of arity 1, and we have introduced a new function symbol app of arity 2.

4.2 Program evaluation

Given any path p, we *evaluate p* with respect to a program P in the following two steps. First the path p is rewritten into normal form with respect to the path rewrite system corresponding to P. Second the normal form is guaranteed to be in source form by making certain that it contains no dangling references. Below we formalize these steps in turn.

4.2.1 Path rewrite system of a program

In Figure 4, we define a function *Rules* for building a path rewrite system from a program. The functionality of *Rules* is straightforward. It traverses a program from the toplevel structure, collecting abbreviation bindings, as we have informally explained above. The first argument, a path, keeps track of the location of the second argument, an expression that *Rules* is currently examining. The path is extended with $.m$ when *Rules* takes up a field named m (the first case), and with functor application when it enters the body of a functor (the second case). When encountering an abbreviation binding $m = p$ at the location tracked as p', *Rules* introduces a path rewrite rule $p'.m \to\ p$ (the last case).

We use the shorthand Rules_P for $\mathit{Rules}(\varepsilon, P)$ and call it the path rewrite system of P.

It is important to notice that Rules_P contains no overlapping rules for any P. According to the conventions of Section 3, the module names $m_1, \ldots, m_n$ of the first case are pairwise distinct. As a result, the rewriting system is confluent. This is natural as we are considering a deterministic programming language, ML.

Definition 1 A path p is a *normal form* of q with respect to a program P if p is a normal form of q with respect to Rules_P.

$$\frac{}{P \vdash \varepsilon \mapsto (id, P)}$$

$$\frac{P \vdash p \mapsto (\theta, \{\ldots\ m = e\ \ldots\})}{P \vdash p.m \mapsto (\theta, e)} \qquad \frac{P \vdash p_1 \mapsto (\theta, \lambda(x)e)}{P \vdash p_1(p_2) \mapsto (\theta[x \mapsto p_2], e)}$$

Fig. 5 Lookup

4.2.2 Lookup

A program P can be regarded as a lookup table from paths to expressions, provided that the input paths are in an appropriate form. For instance, the example of this section would map the path $\varepsilon.\mathtt{m}_1.\mathtt{n}_2$ to $\varepsilon.\mathtt{m}_1.\mathtt{n}_1$ and the path $\varepsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1$ to $\{\}$, but would fail for the path $\varepsilon.\mathtt{m}_1.\mathtt{n}_3$ or $\varepsilon.\mathtt{m}_1.\mathtt{n}_2.\mathtt{n}$; the former is dangling and the latter needs to be rewritten into $\varepsilon.\mathtt{m}_1.\mathtt{n}_1.\mathtt{n}$ first.

We formalize this view of a program as a lookup table by defining the *lookup relation* in Figure 5. Arguments are accounted for by building a substitution from formal parameters (in the source program) to actual arguments (in the path looked up). The judgment $P \vdash p \mapsto (\theta, e)$ means that, with respect to the program P, the path p refers to the expression e, where variables x appearing in e are bound to $\theta(x)$. Observe that the relation is decidable and deterministic for any program P and path p. In other words, given P and p, we can search in a terminating way e and θ such that $P \vdash p \mapsto (\theta, e)$ holds and they are unique if they exist.

We write $P \vdash p \not\mapsto$ when $P \vdash p \mapsto (\theta, e)$ does not hold for any (θ, e).

Finally, we use the lookup relation to define the notion of source form. A path p is in source form if every path contained in p refers to a structure or a functor.

Definition 2 A path p is in *source form* with respect to a program P if the following two conditions hold.

1. There exists a pair (θ, e) such that $P \vdash p \mapsto (\theta, e)$ holds, and e is not a path.
2. For any q in $args(p)$, q is in source form with respect to P.

Since the lookup relation is decidable, we can determine, for any program P and path p, whether p is in source form with respect to P; this is easily proved by induction on the structure of p.

We end this section with a formal definition of the evaluation of paths.

Definition 3 A path p *evaluates into* q with respect to a program P if p rewrites into a normal form q and q is in source form with respect to P.

5 Undecidability

We are interested in the decidability of path evaluation. In other words, given a path p and a “program” P (which actually encodes a recursive signature), we want to evaluate p with respect to P in a terminating way and signal an error when p contains cyclic or dangling references. This problem is clearly undecidable if we include both higher-order functors and recursion, as it then amounts to determining termination of normalization for a lambda calculus with (unrestricted) recursion.

In this section, we prove that termination remains undecidable with only first-order functors and nested modules, i.e., without higher-order functors. The restricted calculus is much less powerful than a lambda calculus with recursion. Indeed, systems with respectively only recursive first-order functors (without nested modules) and only nested recursive modules (without functors) have decidable termination. Below we first look at these systems.

5.1 Decidability of subsystems

In the absence of nested modules, since our rewriting rules do not depend on arguments, non-terminating programs are exactly those containing effective recursion. By effective recursion, we mean that a recursive occurrence of a module or functor is called dynamically with the same number of arguments as in its definition, or more. Here we allow functors to return functors, but they should not be passed as argument. For instance both $\{\mathtt{m}_1 = \lambda(\mathtt{x})\varepsilon.\mathtt{m}_2 \ \ \mathtt{m}_2 = \varepsilon.\mathtt{m}_1(\varepsilon.\mathtt{m}_3)\}$ and $\{\mathtt{m}_2 = \varepsilon.\mathtt{m}_1(\varepsilon.\mathtt{m}_2)\}$ do not terminate on path $\varepsilon.\mathtt{m}_2$, while $\{\mathtt{m}_1 = \lambda(\mathtt{x})\varepsilon.\mathtt{m}_1\}$ and $\{\mathtt{m}_1 = \lambda(\mathtt{x})\varepsilon.\mathtt{m}_2(\mathtt{x}) \ \ \mathtt{m}_2 = \lambda(\mathtt{x})\mathtt{x}\}$ always terminate. In order to decide termination, we build a non-deterministic pushdown automaton. Its states are the names of the modules and functors, and it has only one stack symbol. The length of the stack tells us the number of arguments we have. Our first example gets encoded as $\{(\mathtt{m}_1,\mathtt{n}+\mathtt{1}) \to (\mathtt{m}_2,\mathtt{n}), (\mathtt{m}_2,\mathtt{n}) \to (\mathtt{m}_1,\mathtt{n}+\mathtt{1}), (\mathtt{m}_2,\mathtt{n}) \to (\mathtt{m}_3,\mathtt{n})\}$. We then check whether this pushdown automaton may have infinite transitions starting from a finite stack, which is a decidable problem (this is a direct consequence of the pumping lemma for context free languages, which are equivalent to pushdown automata [13].) Here we see that there is a transition from $(\mathtt{m}_1,\mathtt{1})$ to $(\mathtt{m}_1,\mathtt{1})$, so there is a non-terminating path.

For the case with only nested modules, since there are no functors, there are no variables, and termination is clearly decidable. Indeed if a program P does not contain functors at all, every path rewrite rule in $Rules_P$ is of the form $\varepsilon.m_1.m_2.\cdots.m_i \to \varepsilon.m'_1.m'_2.\cdots.m'_j$. We can then encode our program as a deterministic pushdown automaton with a single state, using names as stack alphabet, each rule defining a transition. Termination becomes equivalent to whether this pushdown automaton can generate infinite words, which is decidable. More directly, reduction in P amounts to head-reduction for a string rewrite system, whose termination is known to be decidable [4].

5.2 The first-order path calculus is Turing complete

We prove undecidability for the case with both first-order functors and nested modules by encoding any Turing machine into a first-order fragment of our calculus. As we have seen above, both features provide us with a pushdown automaton. Since a Turing machine is essentially a pushdown automaton with two stacks, the trick will be to use respectively functor application and submodule access to encode each stack.

To preclude the potential use of higher-order functors during path rewriting, it is enough to ensure the following two conditions. The second condition is overly restrictive, but it makes it easier to check the first one.

1. The path rewrite system of a program P yields no paths of the forms $x(p)$ or $x.m(p)$ during rewriting. Thus variables or their submodules cannot be applied.
2. Only the toplevel structure can define functors $\lambda(x)e$, where e must not be or contain a functor. Thus functors cannot return functors.

We enforce these conditions by confining ourselves to a fragment of the calculus defined in Figure 6. The new syntax is restricted in the following three ways.

1. Only paths of the form $\varepsilon.m$ can appear in functor positions.

Paths	p	$::=$	$\varepsilon \mid x \mid \varepsilon.m(p) \mid p.m$
Toplevel expression	te	$::=$	$\lambda(x)\{m_1 = p_1 \cdots m_n = p_n\ \}$
Program	P	$::=$	$\{m_1 = te_1 \cdots m_n = te_n\}$

Fig. 6 A first-order fragment

2. A program is a sequence of toplevel expressions, which are lambda abstractions of structures.
3. A toplevel expression only contains abbreviation bindings. In particular, it does not contain functors.

Let $M = (Q,\Sigma,\Gamma,\delta,q_0,b,F)$ be a Turing machine [13], where Q is the set of states; $\Sigma \subseteq \Gamma$ is the set of input symbols; Γ is the set of tape symbols; δ is the transition function; $q_0 \in Q$ is the start state; b is the blank symbol, which is in Γ but not in Σ; F is the set of final states, which we assume to be empty without loss of generality. In particular, the arguments of $\delta(q,a)$ are a state $q \in Q$ and a tape symbol $a \in \Gamma$. The value of $\delta(q,a)$, if it is defined, is a triple (q',a',D), where q' is the next state; a' is the symbol in Γ to be written in the scanned cell of the tape; D is a direction, which is either R (for right) or L (for left). A Turing machine M halts on an empty tape if M halts on an initial configuration q_0.

Proposition 1 *For any Turing machine $M = (Q,\Sigma,\Gamma,\delta,q_0,b,F)$, there exists a program P and a path p such that M halts on an empty tape if and only if the evaluation of p with respect to P terminates.*

Proof.
A configuration $a_1a_2\cdots a_{i-1}qa_ia_{i+1}\cdots a_n$ of the Turing machine M is encoded by a path

$$\varepsilon.q(\varepsilon.a_{i-1}(\cdots(\varepsilon.a_2(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon))))\cdots)).a_i.a_{i+1}.\cdots.a_n.\hat{b}$$

where the special symbol $\hat{b}$ is not contained in Q or Γ. The intuition is that the right hand side of the tape is encoded with the dots and the left hand side with functor applications. The head part $\varepsilon.q$ of the path represents the current state and a_i, which follows the head part by a dot, is the symbol to be read next. We put $\hat{b}$ at the inner most functor application and the outermost dot to represent the right and left limits of input symbols on the tape. The initial configuration q_0 (with an empty input tape) of the Turing machine is represented by $\varepsilon.q_0(\varepsilon.\hat{b}(\varepsilon)).\hat{b}$.

The rest of the proof is structured according to the following three steps.

1. We define a set of path rewrite rules R_M from the Turing machine M.
2. We show that R_M encodes the Turing machine M.
3. We give a program P whose path rewrite system is R_M.

A path rewrite system R_M, encoding the transition function δ, is defined as the union of the following sets:

1. $\{\varepsilon.q(x).a \to \varepsilon.q'(\varepsilon.a'(x)) \mid \delta(q,a) = (q',a',R)\}$
2. $\{\varepsilon.q(x).a \to x.q'.a' \mid \delta(q,a) = (q',a',L)\}$
3. $\{\varepsilon.q(x).\hat{b} \to \varepsilon.q(x).b.\hat{b} \mid q \in Q\}$
4. $\{\varepsilon.\hat{b}(x).q \to \varepsilon.q(\varepsilon.\hat{b}(x)).b \mid q \in Q\}$
5. $\{\varepsilon.a(x).q \to \varepsilon.q(x).a \mid a \in \Gamma, q \in Q\}$

The first two sets of rules encode transitions of M. The rules from third and fourth sets are for elongating the tape, moving the edge by adding a blank symbol to the left or right extremity on demand. Finally, the rules from the last set commute a tape symbol with the current state, to allow the next move to take place. A transition of M can be simulated either by a rule of 1, potentially followed by a rule of 3, or by a rule of 2 followed by a rule of 4 or 5.

It is straightforward to prove that these rules encode the Turing machine. Suppose $\delta(q,a_i) = (q',a'_i,L)$:

1. When $i \neq 1$, or $i = n$ and $a'_i \neq b$, then we have a move
$$a_1 \cdots a_{i-1} q a_i a_{i+1} \cdots a_n \vdash a_1 \cdots a_{i-2} q' a_{i-1} a'_i a_{i+1} \cdots a_n$$
reflected by the reduction sequence
$$\begin{array}{rl} & \varepsilon.q(\varepsilon.a_{i-1}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).a_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \varepsilon.a_{i-1}(\varepsilon.a_{i-2}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).q'.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \varepsilon.q'(\varepsilon.a_{i-2}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).a_{i-1}.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \end{array}$$
2. When $i = 1$, then we have a move
$$q a_1 a_2 \cdots a_n \vdash q' b a'_1 a_2 \cdots a_n$$
reflected by the reduction sequence
$$\begin{array}{rl} & \varepsilon.q(\varepsilon.\hat{b}(\varepsilon)).a_1.a_2\cdots.a_n.\hat{b} \\ \to & \varepsilon.\hat{b}(\varepsilon).q'.a'_1.a_2\cdots.a_n.\hat{b} \\ \to & \varepsilon.q'(\varepsilon.\hat{b}(\varepsilon)).b.a'_1.a_2\cdots.a_n.\hat{b} \end{array}$$
3. When $i = n$ and $a'_i = b$, then we have a move
$$a_1 a_2 \cdots a_{n-1} q a_n \vdash a_1 a_2 \cdots a_{n-2} q' a_{n-1}$$
reflected by the reduction sequence
$$\begin{array}{rl} & \varepsilon.q(\varepsilon.a_{n-1}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).a_n.\hat{b} \\ \to & \varepsilon.a_{n-1}(\varepsilon.a_{n-2}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).q'.b.\hat{b} \\ \to & \varepsilon.q'(\varepsilon.a_{n-2}(\cdots(\varepsilon.a_1(\varepsilon.\hat{b}(\varepsilon)))\cdots)).b.\hat{b} \end{array}$$

The cases where $\delta(q,a_i) = (q',a'_i,R)$ are similar.

Finally we give a program P whose path rewrite system is R_M. We then have that M halts on an empty tape if and only if the evaluation of $\varepsilon.q_0(\varepsilon.\hat{b}(\varepsilon)).\hat{b}$ with respect to P is terminating. Recall that the path rewrite system of the program $\{q = \lambda(x)\{a = \varepsilon.q'(\varepsilon.a'(x))\}\}$ is $\{\varepsilon.q(x).a \to \varepsilon.q'(\varepsilon.a'(x))\}$. The rule exactly corresponds to rules from the first set above. In general the toplevel structure of P consists of the following definitions.

1. For each q in Q,
$$\begin{array}{l} q = \lambda(x)\{ \\ \quad a_1 = \varepsilon.q'_1(\varepsilon.a'_1(x)) \ \cdots \ a_m = \varepsilon.q'_m(\varepsilon.a'_m(x)) \\ \quad b_1 = x.r'_1.b'_1 \ \cdots \ b_n = x.r'_n.b'_n \\ \quad \hat{b} = \varepsilon.q(x).b.\hat{b} \\ \} \end{array}$$
where
$$\{(a_1,q'_1,a'_1),\cdots,(a_m,q'_m,a'_m)\} = \{(a,q',a') \mid \delta(q,a) = (q',a',R)\}$$
and
$$\{(b_1,r'_1,b'_1),\cdots,(b_n,r'_n,r'_n)\} = \{(b,r',b') \mid \delta(q,b) = (r',b',L)\}$$
2. $$\hat{b} = \lambda(x)\{q_1 = \varepsilon.q_1(\varepsilon.\hat{b}(x)).b \ \cdots \ q_n = \varepsilon.q_n(\varepsilon.\hat{b}(x)).b\}$$
where $\{q_1,\cdots,q_n\} = Q$.
3. For each a in Γ,
$$a = \lambda(x)\{q_1 = \varepsilon.q_1(x).a \ \cdots \ q_n = \varepsilon.q_n(x).a\}$$
where $\{q_1,\cdots,q_n\} = Q$. □

Taking a closer look at the proof of Proposition 1, one notices that our encoding of Turing machines fully exploits the abilities of the first-order path calculus. A Turing machine requires at least two stacks, which the calculus barely provides through functor applications for one, and nesting for the other. Of course, since it is Turing complete, it can also encode any computation, including an arbitrary number of stacks, but this would require all the tricks one uses with Turing machines. This observation suggests that even a weak restriction would be sufficient for the calculus to lose its Turing completeness, as we will see in the next section.

The undecidability result was shown for the untyped calculus, but it is possible to obtain it in a typed setting. For simplicity, we assume that the transition function δ of our Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$ is total. Consider the following (regular) recursive signatures S and T:

$$\begin{aligned} S &= \mu X.\{a_1 : X \dots a_n : X \;\; \hat{b} : X\} \\ T &= \{q_1 : S \;\; \dots q_n : S\} \end{aligned}$$

where $Q = \{q_1, \dots, q_n\}$ and $\Sigma = \{a_1, \dots, a_n\}$. It is easy to see that, in our encoding of the Turing machine, the toplevel functors $\varepsilon.a_i$'s and $\varepsilon.\hat{b}$ have signatures $T \rightarrow T$, and $\varepsilon.q_i$'s have signatures $T \rightarrow S$. In particular, all variables are of signature T. The signature for a functor argument indicates all the submodules an argument must provide, which seems a natural enough notion of typing. So, path resolution stays undecidable even if we restrict functor arguments through explicit regular signatures.

6 Terminating path evaluation

In the last paragraph of the previous section, we have seen that encoded Turing machines are typable by (regular) recursive signatures. The recursion is needed, as the way we encode Turing machines requires unfettered access to submodules of functor arguments to simulate an (unbounded) stack. The signature T we assigned to functor arguments is only usable because it allows us to access arbitrarily deep nested modules, $x.q.a_1.a_2$ and $x.q.a_1.a_2.a_3$, etc...

These remarks suggest that we might escape undecidability by restricting such accesses, and moreover that such a restriction could be naturally expressed by annotating functor arguments by (non-recursive) signatures. Following this idea, in the next subsection we introduce *access signatures*, which have only finitely deep nesting, and require that a submodule of a functor argument can only be accessed if it is present in its access signature. We then present a terminating path normalization algorithm, named *semi-ground normalization*, which implements this restriction, and prove it correct and complete. All proofs can be found in appendix A. Some technically subtle lemmas and theorems have been checked using the Coq proof assistant, and the proof scripts are available at the following location.

`http://www.math.nagoya-u.ac.jp/~garrigue/papers/Path-coq.zip`

6.1 Access signatures

In Figure 7, we revise our language to extend it with access signatures. An access signature maps module names available for access to their own access signatures. There is no recursion in access signatures; *i.e.* an access signature is a finite tree whose leaves are empty

Access signatures	$S ::= \{m_1 : S_1 \cdots m_n : S_n\}$
Expressions	$e ::= \{m_1 = e_1 \cdots m_n = e_n\} \mid \lambda(x : S)e \mid p$
Paths	$p ::= vp \mid rp$
Variable paths	$vp ::= x \mid vp.m$
Rooted paths	$rp ::= \varepsilon \mid rp(p) \mid rp.m$
Access paths	$ap ::= \varepsilon \mid ap.m$

Fig. 7 Revised syntax

signatures. Since we do not have higher-order functors, access signatures only contain signatures for structures. Abstracted variables are annotated with access signatures. Paths are restricted to be either *variable* or *rooted*, where only the latter may contain functor applications in accordance with the absence of higher-order functors. We call sequences of module names *access paths*. Concatenation of a path p and an access path ap, noted $p.ap$, is defined as the path obtained by replacing the leading ε in ap by p, *i.e.* $p.ap$ is p suffixed by the module names of ap. We can see an access signature as a prefix-closed set of access paths, and we will sometimes write $ap \in S$ to denote that the access path ap is included in the access signature S.

To simplify the presentation, we assume that all bound variables in a program are distinct, and write $\mathsf{sig}_P(x)$ to denote the access signature of x in the program P.

A well-typed program should not access a submodule of a variable not allowed by the variable's access signature or pass an argument which does not satisfy the access signature. We will use this restriction to keep path evaluation terminating. Recall, however, that we are dealing with a program whose well-typedness is not yet known. Since path evaluation is necessary to decide type equality during type checking, we cannot rely on the well-typedness of the whole program to preclude ill-typed uses of variables during path evaluation: the dependency between type checking and path evaluation would be circular. Therefore our terminating path evaluation must enforce well-typed uses of variables, while expanding paths simultaneously.

Syntactic restrictions will not be sufficient to eliminate invalid accesses to variables during path evaluation. For instance, consider the following program.

$$\{\mathtt{m}_1 = \lambda(\mathtt{x} : \{\})\{\mathtt{m}_2 = \mathtt{x}\ \ \mathtt{m}_3 = \varepsilon.\mathtt{m}_1(\mathtt{x}).\mathtt{m}_2.\mathtt{m}_4\}\ \ \mathtt{m}_5 = \{\}\}$$

Superficially, no attempt is made to access submodules of x. Once we recognize that the path $\varepsilon.\mathtt{m}_1(\mathtt{x}).\mathtt{m}_2$ actually refers to x, however, we see that $\mathtt{m}_3$ is an abbreviation for $\mathtt{x}.\mathtt{m}_4$. Yet, even looking at the evaluation of a path would not expose the problem, since variables are immediately substituted:

$$\varepsilon.\mathtt{m}_1(\varepsilon.\mathtt{m}_5).\mathtt{m}_3 \longrightarrow \varepsilon.\mathtt{m}_1(\varepsilon.\mathtt{m}_5).\mathtt{m}_2.\mathtt{m}_4 \longrightarrow \varepsilon.\mathtt{m}_5.\mathtt{m}_4$$

To effectively enforce well-typed uses of variables as specified by their access signatures, we need to make sure that evaluating any of the paths appearing in a program requires no ill-typed access to variables.

$$\textsf{r-src}\ \frac{P \vdash p \mapsto (\theta, e) \quad P \vdash \theta\ \textsf{safe} \quad e \text{ not a path}}{P \vdash p \downarrow p} \qquad \textsf{r-vp}\ \frac{ap \in \textsf{sig}_P(x)}{P \vdash x.ap \downarrow x.ap}$$

$$\textsf{r-exp}\ \frac{P \vdash p \mapsto (\theta, p') \quad P \vdash \theta\ \textsf{safe} \quad P \vdash \theta(p') \downarrow q}{P \vdash p \downarrow q}$$

$$\textsf{r-dot}\ \frac{P \vdash p \downarrow p' \quad P \vdash p'.m \downarrow q}{P \vdash p.m \downarrow q} \qquad \textsf{r-app}\ \frac{P \vdash p_1 \downarrow p'_1 \quad P \vdash p'_1(p_2) \downarrow q}{P \vdash p_1(p_2) \downarrow q}$$

$$\textsf{s-subst}\ \frac{P \vdash p_i : \textsf{sig}_P(x_i) \quad (1 \leq i \leq n)}{P \vdash [x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]\ \textsf{safe}}$$

$$\textsf{s-rec}\ \frac{P \vdash p \downarrow q \quad P \vdash q.m_i : S_i\ (1 \leq i \leq n)}{P \vdash p : \{m_1 : S_1\ \ldots\ m_n : S_n\}}$$

Fig. 8 Safe reduction, safety for a signature and safe substitutions

6.2 Safe programs

In this subsection we formalize the above restrictions by the notion of *safe program*, defined through the derivability of a judgment, in the style of natural semantics [15]. We then prove that if a program is safe then path evaluation with respect to this program terminates for any ground path (Proposition 2). In the next subsection, we will prove the decidability of program safety. In Appendix A.3, we will provide an alternative definition of safety based on term rewriting.

We define safe programs in terms of safe reduction, safety for a signature and safe substitutions.

Definition 4 A path p *reduces safely* to q with respect to a program P when $P \vdash p \downarrow q$ is derivable. A path p is *safe for a signature* S with respect to P when $P \vdash p : S$ is derivable. A substitution θ is *safe* with respect to P when $P \vdash \theta\ \textsf{safe}$ is derivable. Specifically, we may say a path p is *safe* with respect to P when there is a path q such that $P \vdash p \downarrow q$, or equivalently $P \vdash p : \{\}$. The derivable judgments are given in Figure 8.

Intuitively a path p is safe with respect to a program P when p reduces to a *head normal form* q, and for all functor applications appearing during this reduction, path arguments are safe with respect to their access signatures. By *head normal form* we mean either a variable path, or a rooted path which does not point to another path.

If we ignore the $P \vdash \theta\ \textsf{safe}$ in the premises of r-src and r-exp, the r-rules define a big step reduction relation in the standard way. Rules r-src and r-vp end reduction for head normal forms. r-exp implements a rewriting step: p is rewritten into $\theta(p')$, and we further check that $\theta(p')$ reduces to q. r-dot and r-app are congruence rules, which allow applying rewriting steps to a prefix of the current path.

Furthermore, functor applications are checked to verify that each argument satisfies the signature of the corresponding parameter, via the judgment $P \vdash \theta\ \textsf{safe}$ implemented by rule s-subst. It ensures that the argument contains the required submodules, which must be safe themselves, as specified by the access signature of the parameter, via the judgment $P \vdash p : S$ implemented by rule s-rec. This check is carried out before each expansion step (rule r-exp), and once more on the final path (rule r-src), ensuring that all subpaths in a derivation satisfy their safety requirements.

$$\begin{array}{lll} \mathit{Paths} & p & ::= vp \mid rp \\ \mathit{Expressions} & e & ::= \{m_1 = e_1 \ \cdots \ m_n = e_n \ \} \mid \lambda(x:S)e \mid p^i \end{array}$$

Fig. 9 Expressions with integer labels

Note that we do not normalize path arguments in the resulting path; we just check their safety before reducing to head normal form. From the point of view of safety, path arguments do not just represent one path, but rather all paths in their access signatures. Therefore keeping one normal form would not be sufficient to avoid repeated checks.

This inference system defines our notion of safe program:

Definition 5 A program P is *safe* if all the paths it contains can be reduced safely, or equivalently if $P \vdash p : \{\}$ for all the paths p appearing in P.

Safety for a signature is preserved by both safe reduction and safe substitution.

Lemma 1 (typing equivalence) *If $P \vdash p : S$, $P \vdash p \downarrow q$, and $P \vdash p' \downarrow q$, then $P \vdash p' : S$. (Coq-checked)*

Lemma 2 (substitution) *If $P \vdash p : S$ and $P \vdash \theta$ safe, then $P \vdash \theta(p) : S$. Moreover, if $P \vdash p \downarrow q$ for a rooted path q, then $P \vdash \theta(p) \downarrow \theta(q)$. (Coq-checked)*

We can see the effectiveness of our restriction in the following proposition, which states that if a program is safe, then evaluation of any ground path (*i.e.* containing no variables) terminates.

Proposition 2 *If P is safe and q is a ground path, then the evaluation of q with respect to P terminates.*

6.3 Semi-ground normalization

The last step is to prove that program safety is decidable, by exhibiting an algorithm that normalizes a path, either providing a proof of safety, or returning an error if the program is unsafe.

To detect cyclic abbreviations, we label paths appearing inside expressions with integers in the syntax of our calculus, in Figure 9. We assume that a program contains no duplicate occurrences of the same integer label and write $\mathit{Labels}(P)$ to denote the set of integer labels occurring in the program P. Note that for any program P, $\mathit{Labels}(P)$ is finite.

By preventing the same label from being expanded twice, we can easily define a terminating evaluation algorithm. The main difficulty is how to detect non-termination of path evaluation in a complete way. We will achieve completeness by expanding paths using a call-by-value strategy, while η-expanding functor arguments to uncover errors in subpaths on the fly. By adopting a call-by-value strategy, we can minimize the depth of the call stack, avoiding false-positives when a function is called twice in a non-recursive way. By applying η-expansion, we can make path normalization semi-ground, in the sense that we need not normalize paths after applying substitution. To avoid transforming the source program, we will perform "virtual" η-expansion during path normalization.

Let us see on examples why these techniques are needed, and how they work in practice. For readability, we omit the ε prefix to toplevel modules in subsequent examples.

Here is a simple example, applying a functor twice.

$$\{f = \lambda(x)x^1 \;\; a = \{\} \;\; n = f(f(a))^2\}$$

If we were to adopt a call-by-name strategy, the evaluation of n would proceed as follows (keeping labels while expanding abbreviations).

$$n \to f(f(a))^2 \to (f(a)^1)^2 \to ((a^1)^1)^2$$

The final path $((a^1)^1)^2$ contains two occurrences of 1, showing that we would encounter 1 twice in the same calling stack. Failing on this example clearly loses completeness, as n reduces safely to a.

This can be solved by using a call-by-value strategy and discarding locations on normal forms with special $\sharp$-*steps*.

$$n \to f(f(a))^2 \to f(a^1)^2 \xrightarrow{\sharp} f(a)^2 \to (a^1)^2$$

Compare the above call-by-value evaluation with the previous call-by-name evaluation: at every step, the call stack contains each label at most once.

This is not the whole story: due to the presence of variable paths, we may not be interested in the argument itself, but in one of its submodules. Here is a concrete example.

$$\begin{array}{l} \{\; f = \lambda(x:\{m:\{\}\})\{m = x.m^1\} \\ \;\;\; a = \{m = \{\}\} \;\;\; n = f(f(a)).m^2 \;\} \end{array}$$

Call-by-value evaluation of n would proceed as follows.

$$n \to f(f(a)).m^2 \to (f(a).m^1)^2 \to ((a.m^1)^1)^2$$

While n safely reduces to $a.m$, the final path $((a.m^1)^1)^2$ contains two occurrences of 1. The trick of evaluating arguments first did not work here, because $f(a)$ is already a normal form and we are interested in $f(a).m$ rather than $f(a)$. Our solution is to η-expand the parameter of f through currying, introducing a distinct argument for each access path in the access signature. All calls to f need to be transformed correspondingly.

$$\begin{array}{ll} \{\; f = \lambda(x_\varepsilon:\{\})\lambda(x_m:\{\}).\{m = (x_m)^1\} & a = \{m = \{\}\} \\ \;\;\; n = f(f(a)(a.m))(f(a)(a.m).m).m^2 & \} \end{array}$$

This makes the evaluation succeed.

$$\begin{array}{l} n \to f(f(a)(a.m))(f(a)(a.m).m).m^2 \\ \;\;\; \to f(f(a)(a.m))(a.m^1).m^2 \\ \;\;\; \xrightarrow{\sharp} f(f(a)(a.m))(a.m).m^2 \to (a.m^1)^2 \xrightarrow{\sharp} a.m \end{array}$$

Our normalization algorithm will simulate this behavior by computing an *access substitution* from the lookup substitution on the fly, without actually transforming the source program. An access substitution maps each variable path in the access signature of an argument to the normal form of a suitable subpath of the corresponding argument.

We now define an algorithmic version of path evaluation, called semi-ground normalization in Figure 10. Again this algorithm reduces a path to head normal form, verifying that all involved paths are safe. We use π as a metavariable for sets of integer labels. We call π a *lock*, as we are not allowed to expand abbreviations in it. The semi-ground normalization

```
1:   sgnlz$(P,\pi,q)$ =
2:     match $q$ with
3:     | $\mathtt{x} \Rightarrow \mathtt{x}$
4:     | $\varepsilon \Rightarrow \varepsilon$
5:     | $q_1.m \Rightarrow$ expand$(P,\pi,$sgnlz$(P,\pi,q_1).m)$
6:     | $q_1(q_2) \Rightarrow$ expand$(P,\pi,$sgnlz$(P,\pi,q_1)(q_2))$

7:   expand$(P,\pi,\mathtt{x}.ap)$ =
8:     if $ap \in$ sig$_P(\mathtt{x})$ then $\mathtt{x}.ap$ else error
9:   expand$(P,\pi,rp)$ =
10:    let $(\theta,e)$ = lookup$(P,rp)$ in
11:    let $\rho$ = vp_subs$(P,\pi,\theta)$ in
12:    match $e$ with
13:    | $q^i \Rightarrow$ if $i \in \pi$ then error else subs$(\theta,\rho,$sgnlz$(P,\{i\} \cup \pi,q))$
14:    | $\{m_1 = e_1 \ldots m_n = e_n\} \Rightarrow rp$
15:    | $\lambda(x)e' \Rightarrow rp$

16:  subs$(\theta,\rho,rp) = \theta(rp)$
17:  subs$(\theta,\rho,vp)$ = if $vp \in \mathrm{dom}(\rho)$ then $\rho(vp)$ else $vp$

18:  vp_subs$(P,\pi,id) = id$
19:  vp_subs$(P,\pi,\theta[\mathtt{x} \mapsto p])$ =
20:    sig_subs$(P,\pi,\mathtt{x},p,$sig$_P(\mathtt{x})) \cup$ vp_subs$(P,\pi,\theta)$

21:  sig_subs$(P,\pi,vp,rp,\{\})$ = $[vp \mapsto$ sgnlz$(P,\pi,rp)]$
22:  sig_subs$(P,\pi,vp,\mathtt{x}.ap,\{\})$ =
23:    if $ap \in$ sig$_P(\mathtt{x})$ then $[vp \mapsto \mathtt{x}.ap]$ else error
24:  sig_subs$(P,\pi,vp,p,\{m_1 : S_1\} \uplus S)$ =
25:    sig_subs$(P,\pi,vp.m_1,p.m_1,S_1) \cup$ sig_subs$(P,\pi,vp,p,S)$
```

Fig. 10 Semi-ground normalization

uses an auxiliary function lookup, which is an algorithmic version of the lookup relation defined in Section 4. Precisely,

$$\mathsf{lookup}(P,p) = \begin{cases} (\theta,e) & \text{when } P \vdash p \mapsto (\theta,e) \\ \mathsf{error} & \text{otherwise} \end{cases}$$

We recall that the lookup relation is decidable and deterministic. Hence lookup is well-defined.

Semi-ground normalization implements the ideas outlined above using four mutually recursive functions sgnlz, expand, vp_subs and sig_subs and one auxiliary function subs. These four functions keep track of which abbreviation bindings in the program are under expansion using π. Therefore π is passed around between recursive calls. In particular, π constitutes the measure for the termination of the algorithm. It is consulted and incremented on line 13 when expand unfolds an abbreviation binding. We simply prohibit expand from revisiting the same abbreviation twice to avoid looping without losing completeness. Splitting the algorithm into separate routines effectively implements discarding integer labels on normal forms.

Now we look at the working of each function in more detail. sgnlz recurses structurally on its input path, calling expand on the results of the recursive calls. expand checks safety of the input path and unfolds abbreviations. A variable path is safe if it conforms to the variable's access signature (line 8). For a rooted path to be safe, its arguments must be safe as well. This is checked by the call to vp_subst on line 11. $\mathsf{vp\_subst}(P,\pi,\theta)$ returns an access substitution ρ, which maps a variable path $x.ap$ such that x is in $\mathrm{dom}(\theta)$ and ap is in $\mathsf{sig}_P(x)$ to the result of calling sgnlz on $\theta(x).ap$. It signals an error when θ is not safe, and hence such a substitution does not exist. Importantly, when unfolding an abbreviation binding, expand expands the right hand side of the binding *without substituting its variables*, and only applies the substitution afterwards. This is justified by Lemma 3: if $\mathsf{subs}(\theta,\rho,\mathsf{sgnlz}(P,\{i\}\cup\pi,q))$ is successful then so is $\mathsf{sgnlz}(P,\pi,\theta(q))$ and their results are identical. The auxiliary function subs performs case analysis on the input path, and applies θ when the input is a rooted path, and ρ when the input is a variable path in the domain of ρ (*i.e.* its head variable is in the domain of θ).

We have proved correctness, termination, and completeness of the semi-ground normalization.

Theorem 1 (correctness) *For any program P, lock π and path p, if* $\mathsf{sgnlz}(P,\pi,p) = q$ *then* $P \vdash p \downarrow q$. *(Coq-checked)*

Theorem 2 (termination) *For any program P, lock π and path p,* $\mathsf{sgnlz}(P,\pi,p)$ *is terminating.*

Lemma 3 (postponement) *If p is a rooted path,* $\mathsf{sgnlz}(P,\pi',p) = q$ *and* $\mathsf{vp\_subs}(P,\pi,\theta) = \rho$ *with* $\pi \subset \pi'$*, then* $\mathsf{sgnlz}(P,\pi,\theta(p)) = \mathsf{subs}(\theta,\rho,q)$. *(Coq-checked)*

Theorem 3 (completeness) *For any program P and path p, if P is safe and* $P \vdash p : \{\}$*, then* $\mathsf{sgnlz}(P,\emptyset,p) \neq \mathsf{error}$.

Combining correctness and completeness, sgnlz provides a decidable check for the safety of programs: for each abbreviation (binding) of P, we check in turn if its right hand side is normalizable by running sgnlz. If all the abbreviations in P are normalizable then P is safe, otherwise P is unsafe.

6.4 Towards a practical language

As we mentioned earlier, the calculus we studied in this paper is intended to describe a signature language. Access signatures therefore actually denote "signatures of signatures". We have also made a number of technical design choices intended to make the formalization and proofs simpler. For all these reasons, this calculus is a theoretical one, and it needs to be adapted for use in a practical language.

One of these choices was about using absolute paths, starting from the root ε. For this paper, the simplicity this provides was a major gain. With a single root and the convention on bound variables (requiring their names to be distinct), we were able to formalize and prove Lemmas 2 and 3 and Theorem 1 in Coq, and this in a reasonably short time. Our experience with formalized proofs for lambda-calculi suggests that a syntax with binders would have required much more effort to prove those lemmas in a proof assistant. Yet, in a practical language recursive references should be supported through recursion variables, as in Moscow ML [28] and in our previous works [25,14].

The main reason to prefer recursion variables is modularity and separate compilation. The type system then only needs to know the signatures of the recursion variables mentioned inside each separately compiled module, rather than the source code of the whole program. Note however that the system we have presented in this paper is already about signatures. In order to apply our algorithm, we only need to know (partial) signatures for all recursive modules in the program, which does not break modularity. For a stronger support of separate compilation, the same algorithm could also be used in an incremental way, considering strongly connected signatures one group at a time, assuming that already checked groups are safe. So, the fact that our algorithm uses absolute paths prevents neither modularity nor separate compilation.

We still need to apply our algorithm to actual programs using recursion variables. Starting from a language having signatures similar to *Traviata*, we extend the translation of Section 3 in two respects. First we need to extract access signatures from functor argument signatures, rather than just dropping them. Second, since functor argument signatures may be recursive, we shall check them too, independently of the above extracted version. This can be done by changing the translation for functors and applications as

$$\begin{aligned} [\![\mathit{functor}\ (X : S_1) \rightarrow S_2]\!]^p_\sigma &= \\ &\{\mathit{arg} = [\![S_1]\!]^{p.\mathit{arg}}_\sigma\ \mathit{app} = \lambda(X : \overline{S_1})[\![S_2]\!]^{p.\mathit{app}(X)}_\sigma\} \\ \sigma(p_1(p_2)) &= \sigma(p_1).\mathit{app}(\sigma(p_2)) \end{aligned}$$

where $\overline{S_1}$ is the access signature obtained from $[\![S_1]\!]$ by replacing all path references with the empty signature $\{\}$. Each functor is converted into a module containing a submodule *arg*, which is the translation of the type of the argument, and a submodule *app*, which is a functor returning the type of the result when provided with an argument type. Accordingly, functor application extracts the *app* submodule. The idea is to separate the information about the type of the functor argument from the behaviour of the functor itself. The *arg* submodule is only used to check its safety and is never accessed, whereas the *app* submodule may be referred to from other parts of the program.

In real programs, not all modules need to be recursive. So we shall investigate whether it would be possible to syntactically distinguish recursive modules declaring self variables and non-recursive modules, as in Moscow ML, in order to allow the latter to use higher-order functors. This would allow us to keep backwards compatibility with existing non-recursive code.

7 Related work

OCaml supports a recursive module extension on top of a ML module system with applicative functors [19], which is our starting point. In Section 2, we have discussed deficiencies of OCaml's applicative functors: the type system cannot deduce some desired type equalities [23] and may diverge on some programs while resolving path references [24]. In our previous work [25], we addressed these deficiencies by enriching the signature language with module abbreviations, allowing for strengthened module path equality, and by developing a decidable path resolution. Having module paths in signatures can express a limited form of recursive signatures. But the resulting signature language is less powerful than recursively dependent signatures of [2]: our signature language does not include abstract signatures, which limits polymorphic signatures. To obtain decidability in [25] we gave up any access to submodules of functor parameters. In this paper we lifted this restriction, which is a major improvement.

The undecidability result of Section 5 was first presented at the 9th International Workshop on Termination [26]. The normalization algorithm and decidability results presented here are new.

The problem of path resolution has been addressed in [3], which studied the problem of decidability of Scala type checking. They presented an algorithmic Featherweight Scala, FS_{alg}, and gave a decidable type checking algorithm for it. The calculus FS_{alg} has type path abbreviations and abstract type members in classes, which can be instantiated in a way similar to functor application. Having these two together, they had to resolve path references and rule out cyclic definitions of class members statically. They do not preclude higher-order functors, but simply avoid using the same abbreviation binding twice during the head normalization of the same path. In practice, their algorithm cannot handle our first example of Section 6.3. Here is the encoding of our example in Scala, where `x` is the functor input, and `res` its output:

```
trait f { type x; type res = x; };
val b1 = new f { type x = Int; };
type b = b1.res;
val n1 = new f { type x = b; };
```

This part of the encoding is typable by the Scala interpreter (version 2.9.1), but here is the result of accessing `n1.res`.

```
scala> type n = n1.res;
error: cyclic aliasing or subtyping involving type res
```

This is basically the same example as those given to us through the Scala mailing list [31]. It would be interesting to see whether our algorithm can help improve the situation.

We have also previously studied path resolution in the context of an object system with type parameters and type members [27]. This gave us our first insights in the problem, and how to rule out cycles while keeping expressivity.

Recursive modules extensions of ML-style modules with generative functors have been proposed by several authors [2,29,6,8,22,14]. In particular, several solutions to the double vision problem have been given. The double vision problem refers to the inability to identify the external name of an abstract type with its internal name inside the recursive module where the type is defined. It has proved difficult to solve. Our solution [14] introduces *path substitutions* in the context of a path-based formalization of recursive modules; although this was done for an ML module system with generative functors, we believe it scales to a module system with applicative functors. (Detailed comparison of different solutions to the double vision problem is given in [14].) Notably, our solution uses structural type equality for recursive types without imposing syntactic contractiveness. (Hence, our type equivalence relation is different from that of [10]. Both systems allow opaque type cycles, but the interpretations of types differ.) This is important for us to model the type system of OCaml which supports structural recursive types (i.e. object types and polymorphic variants) and does not impose syntactic contractiveness. Our path-based approach also closely follows the current implementation of OCaml modules.

8 Conclusion

In this paper we have examined the problem of resolving path references in presence of recursion, motivated by recent work on adding recursion to the ML module system. We have

$$\textsf{d-dot}\ \frac{P \vdash_{\mathsf{t}} rp \downarrow rp \quad P \vdash rp.m \not\mapsto}{P \vdash_{\mathsf{t}} rp.m \downarrow rp.m}$$

$$\textsf{d-app}\ \frac{P \vdash_{\mathsf{t}} rp \downarrow rp \quad P \vdash_{\mathsf{t}} p : \{\} \quad P \vdash rp(p) \not\mapsto}{P \vdash_{\mathsf{t}} rp(p) \downarrow rp(p)}$$

Fig. 11 Safety for termination

formalized the problem by defining a rewrite system on paths and proved that the problem is undecidable even if we allow only first-order functors and submodule access to functor arguments, in the absence of higher-order functors.

This result and some observations on the decidability of subsystems led us to design a terminating path resolution algorithm, by requiring functors to be first-order and restricting access to submodules of functor arguments to a finite depth. The algorithm is directly applicable to our recursive module calculus, *Traviata* [25].

This is a major improvement over the original *Traviata*, where all accesses to submodules of functor arguments were prohibited. If we see this calculus as a successor to the ML module system, restricting arguments to a finite depth is not a problem, as this restriction is already present implicitly in ML programs. However, we still need to work on integration with higher-order functors, which are now fully part of ML. We hope to find an appropriate way to separate recursive and non-recursive uses of modules, so that this limitation would apply only to recursive ones.

Acknowledgements We thank the referees for helping us improve this article, and Olivier Danvy for his editorial support. James Chapman made helpful comments on the final version.

K. Nakata's research was supported by the European Regional Development Fund (ERDF) through the Estonian Centre of Excellence in Computer Science (EXCS), the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12, and the Estonian Science Foundation grant no. 9398.

A Proofs for terminating path evaluation

In this appendix, we give proofs for the properties and theorems of section 6.

A.1 Safe programs

We first define a notion of *safety for termination*, which will be needed to prove Proposition 2.

Definition 6 A path p *reduces safely for termination* to q with respect to a program P when $P \vdash_{\mathsf{t}} p \downarrow q$ is derivable, $\vdash_{\mathsf{t}}$ being the inference system obtained by adding the rules d-dot and d-app of Figure 11 to the rules in Figure 8 where $\vdash$ is replaced by $\vdash_{\mathsf{t}}$. A path p is *safe for termination for the signature S* with respect to P when $P \vdash_{\mathsf{t}} p : S$ is derivable. It is just *safe for termination* if $P \vdash_{\mathsf{t}} p : \{\}$. A substitution θ is *safe for termination* with respect to P when $P \vdash_{\mathsf{t}} \theta$ safe is derivable.

Lemma 1 (typing equivalence) *If $P \vdash p : S$, $P \vdash p \downarrow q$, and $P \vdash p' \downarrow q$, then $P \vdash p' : S$.*

Proof. (Coq-checked) By inversion of s-rec. □

Lemma 2 (substitution) *If $P \vdash p : S$ and $P \vdash \theta$ safe, then $P \vdash \theta(p) : S$. Moreover, if $P \vdash p \downarrow q$, then $P \vdash \theta(p) \downarrow \theta(q)$ when q is rooted, and otherwise $P \vdash \theta(p) \downarrow r$ and $P \vdash \theta(q) \downarrow r$ for the same path r. Both results also hold for $\vdash_{\mathsf{t}}$.*

Proof. (Coq-checked) We prove these statements by mutual induction on the structure of the derivation of $P \vdash p : S$ and the safe reductions and safe substitutions involved. The last rule is necessarily s-rec, with first premise $P \vdash p \downarrow q$. We perform case analysis on q.

Case 1: q is a rooted path.
We first show that $P \vdash \theta(p) \downarrow \theta(q)$ by induction on the derivation of $P \vdash p \downarrow q$ with case analysis on the last rule used.
Case r-src: We have $p = q$, $P \vdash p \mapsto (\theta', e)$ and $P \vdash \theta'$ safe for some θ' and e. After substitution we have $P \vdash \theta(p) \mapsto (\theta(\theta'), e)$. By induction hypothesis, for any x in domθ' we have $P \vdash \theta(\theta'(x)) : \mathsf{sig}_P(x)$, so that $P \vdash \theta(\theta')$ safe, and we conclude.
Case r-exp: The hypotheses are $P \vdash p \mapsto (\theta', p')$, $P \vdash \theta'$ safe and $P \vdash \theta'(p') \downarrow q$. If $\theta'(p')$ were a variable path, we would have $P \vdash \theta'(p') \downarrow \theta'(p')$, and q would be a variable path too, which contradicts our assumption. Therefore we have $P \vdash \theta(p) \mapsto (\theta(\theta'), p')$, $P \vdash \theta(\theta')$ safe and, by induction hypothesis, $P \vdash \theta(\theta'(p')) \downarrow \theta(q)$, and we conclude.
Cases r-dot and r-app: Immediate. Note that p' and p'_1 cannot be variable paths.

Finally, we need to show that $P \vdash \theta(q) : S$, relying on the second premise of s-rec ($P \vdash q.m_i : S_i$). By induction hypothesis we have $P \vdash \theta(q.m_i) : S_i$, and since $\theta(q.m_i) = \theta(q).m_i$ we conclude $P \vdash \theta(q) : S$ by s-rec.

Case 2: q is a variable path.
Suppose $q = x.m_1 \dots m_n$. If $\theta(x)$ is a variable path, then $\theta(q)$ is a variable path, and $P \vdash \theta(q) \downarrow \theta(q)$ by $P \vdash \theta$ safe and r-vp. We can prove by an easy induction that $P \vdash \theta(p) \downarrow \theta(q)$ (which also gives us the extra result with $r = \theta(q)$), and we obtain $P \vdash \theta(p) : S$ like in Case 1.

If $\theta(x)$ is not a variable path, we must be more careful, as new reduction steps may appear, both in the reductions of p and of its submodules for s-rec. For this reason we prove $P \vdash \theta(p) : S$ by induction on the derivation of $P \vdash p \downarrow q$, assuming $P \vdash p : S$ as extra hypothesis. We perform case analysis on the last rule used.
Case r-src: impossible.
Case r-vp: immediate by $P \vdash \theta$ safe; we trivially have convergence since $p = q$.
Case r-exp: We have $P \vdash p \mapsto (\theta', p')$ and $P \vdash \theta'(p') \downarrow q$. By applying the induction hypothesis to the latter, we obtain $P \vdash \theta(\theta'(p')) : S$ and conclude by r-exp.
Case r-dot: By hypothesis $P \vdash p.m : S$, so that $P \vdash p : \{m : S\}$, and by induction hypothesis $P \vdash \theta(p) : \{m : S\}$, thus $P \vdash \theta(p.m) : S$. For convergence, either p' is rooted, and the new left premise is $P \vdash \theta(p) \downarrow \theta(p')$, so that we can conclude using the induction hypothesis for the right premise; or p' is a variable path, and there exists a path r such that $P \vdash \theta(p) \downarrow r$ and $P \vdash \theta(p') \downarrow r$. We know that $P \vdash r : \{m : S\}$ by typing equivalence, so that there is a r' such that $P \vdash r.m \downarrow r'$, and as result both $P \vdash \theta(p.m) \downarrow r'$ and $P \vdash \theta(p'.m) \downarrow r'$ by the uniqueness of safe reductions.
Case r-app: p'_1 cannot be a variable path, since $p'_1(p_2)$ would not be valid. Thus we have $P \vdash \theta(p_1) \downarrow \theta(p'_1)$. From the hypothesis $P \vdash p_1(p_2) : S$, we obtain $P \vdash p'_1(p_2) : S$, since they both reduce to the same q. By induction hypothesis we obtain $P \vdash \theta(p'_1(p_2)) : S$, thus $P \vdash \theta(p_1(p_2)) : S$ by r-app. We obtain convergence in the same way.

The same proof applies to $\vdash_{\mathsf{t}}$: a dangling path, as in the conclusion of rules d-dot and d-app, is a rooted path such that none of its prefixes refers to an abbreviation, but itself cannot be looked up. This property is preserved by substitution. □

The following lemma relates reduction steps and safety derivations. It is used in the ensuing proof of termination.

Lemma 4 *If P is safe, $P \vdash q : S$, and $q \rightarrow q'$ by a reduction step of $Rules_P$, then $P \vdash q' : S$, and the size of its derivation is strictly smaller than the size of $P \vdash q : S$. This also holds for P safe for termination and $\vdash_{\mathsf{t}}$.*

Proof. First note that it is sufficient to prove this property for $S = \{\}$: $P \vdash q : S$ is equivalent to $\forall ap \in S, P \vdash q.ap : \{\}$, and if $q \rightarrow q'$ then $q.ap \rightarrow q'.ap$.

We prove it by induction on the structure of q.

If the reduction step was on q itself (*i.e.* not on one of its arguments), then there can be only one such redex, and this reduction corresponds exactly to the first use of r-exp in our derivation. After reduction, this r-exp step disappears, replaced by its third premise, and the derivation is otherwise unchanged, so the size of the derivation is strictly smaller.

If the reduction was done on an argument of q, by inversion of $P \vdash q : \{\}$, this argument must be safe for a signature S appearing in P. By induction hypothesis, after reduction this proof of safety becomes smaller. Moreover, if this argument is required for reducing q, *i.e.* if by some use of r-exp it becomes the head of the path we are reducing, then the next step in the derivation was necessarily to reduce it, so that a second occurrence of r-exp disappears. $\square$

Proposition 2 *If P is safe (or safe for termination) and q is a ground path, then the evaluation of q (in the sense of Definition 3) with respect to P terminates.*

Proof. Note that any safe program is also safe for termination, so we only consider the second case.

We first prove that for any signature S used in P, $P \vdash_{\mathsf{t}} q : S$ can be derived by induction on q. This amounts to proving that there is a path q'' such that $P \vdash_{\mathsf{t}} q.ap \downarrow q''$, for any access path ap in S. We prove it by induction on $q' = q.ap$.

- If $q' = \varepsilon$, $P \vdash_{\mathsf{t}} \varepsilon \downarrow \varepsilon$ by r-src.
- If $q' = p.m$, then by induction hypothesis $P \vdash_{\mathsf{t}} p \downarrow p'$, and either $p'.m$ is dangling thus $P \vdash_{\mathsf{t}} p.m \downarrow p'.m$ holds by d-dot, or there exist θ and e such that $P \vdash p'.m \mapsto (\theta, e)$, and since $P \vdash_{\mathsf{t}} p \downarrow p'$, the arguments of p', *i.e.* θ, are safe for termination. If e is a path, then by the safety of P, $P \vdash_{\mathsf{t}} e : \{\}$, and by our substitution lemma, $P \vdash_{\mathsf{t}} \theta(e) : \{\}$. Otherwise $P \vdash p'.m \downarrow p'.m$ by r-src.
- If $q' = p_1(p_2)$, then by induction hypothesis $P \vdash_{\mathsf{t}} p_1 \downarrow p'_1$ and $P \vdash_{\mathsf{t}} p_2 : S_2$ for any S_2 in P, and either $p'_1(p_2)$ is dangling, and $P \vdash_{\mathsf{t}} p_1(p_2) \downarrow p'_1(p_2)$ by d-app, or there exist θ and e s.t. $P \vdash p'_1(p_2) \mapsto (\theta, e)$, and from our two induction hypotheses, the arguments of $p'_1(p_2)$, *i.e.* θ, are safe for termination, which lets us conclude as in the $p.m$ case.

Next we show that the evaluation of q terminates by induction on the size of the derivation of $P \vdash_{\mathsf{t}} q : \{\}$. This size is finite, and lemma 4 provides the induction step. $\square$

A.2 Correctness of semi-ground normalization

Theorem 1 (correctness) *For any program P, lock π and path p, if $\mathsf{sgnlz}(P, \pi, p) = q$ then $P \vdash p \downarrow q$. Moreover, for any substitution θ, if $\mathsf{vp\_subs}(P, \pi, \theta) = \rho$, then $P \vdash \theta$ safe, and for any $x \in \mathrm{dom}(\theta)$ and $ap \in \mathsf{sig}(x)$, $P \vdash \theta(x.ap) \downarrow \rho(x.ap)$.*

Proof. (Coq-checked) We prove both properties by functional induction.

If $p = x$ or $p = \varepsilon$, then $P \vdash p \downarrow p$.

If p is a variable path $x.ap$, then sgnlz will call expand, which in turn checks that ap is in the signature of x, and returns p itself, and $P \vdash p \downarrow p$.

If $p = p_1(p_2)$, we assume $\mathsf{sgnlz}(P, \pi, p_1) = q_1$. By induction hypothesis on p_1, $P \vdash p_1 \downarrow q_1$. We call expand on $q_1(p_2)$. First we call lookup to obtain θ and e. Then we call $\mathsf{vp\_subs}$ to obtain ρ, so that $P \vdash \theta$ safe by induction hypothesis. If e is not a path, we have $P \vdash q_1(p_2) \downarrow q_1(p_2)$ by $\mathsf{r\text{-}src}$, and $P \vdash p_1(p_2) \downarrow q_1(p_2)$ by $\mathsf{r\text{-}app}$. Let e be the path q^i. By induction hypothesis, we have $P \vdash q \downarrow q'$ with $q' = \mathsf{sgnlz}(P, \{i\} \cup \pi, q)$. If q' is a rooted path, by the substitution lemma, we have $P \vdash \theta(q) \downarrow \theta(q')$, so that $P \vdash q_1(p_2) \downarrow \theta(q')$ by $\mathsf{r\text{-}exp}$. If q' is a variable path $x.ap$, by inversion we have $ap \in \mathsf{sig}_P(x)$. If $x \notin \mathrm{dom}(\theta)$, by the substitution lemma we have $P \vdash \theta(q) \downarrow x.ap$ $(= \rho(x.ap)$ since $x.ap \notin \mathrm{dom}(\rho))$. Otherwise, by induction hypothesis on $\mathsf{vp\_subs}$, we have $P \vdash \theta(x.ap) \downarrow \rho(x.ap)$, so that $P \vdash \theta(q) \downarrow \rho(x.ap)$ by the substitution lemma. In both cases we conclude that $P \vdash q_1(p_2) \downarrow \rho(x.ap)$ by $\mathsf{r\text{-}exp}$.

If $p = p_1.m$, we prove the property in the same way, replacing occurrences of $p_1(p_2)$ and $q_1(p_2)$ by $p_1.m$ and $q_1.m$ respectively, and $\mathsf{r\text{-}app}$ by $\mathsf{r\text{-}dot}$.

Next we prove the property on $\mathsf{vp\_subs}$. This amounts to proving that for each $x \mapsto p$ in θ, if $\mathsf{sig\_subs}(P, \pi, x, p, \mathsf{sig}(x)) = \rho$, then $P \vdash p : \mathsf{sig}_P(x)$ and for all $ap \in \mathsf{sig}_P(x)$, $P \vdash p.ap \downarrow \rho(x.ap)$. Since the latter implies the former, we just need to prove the latter.

If p is a variable path $y.ap'$, then we just check that $y.ap'.ap$ is valid, and return $\rho(x.ap) = y.ap'.ap$, so that $P \vdash y.ap'.ap \downarrow \rho(x.ap)$ by $\mathsf{r\text{-}vp}$.

If p is a rooted path, then $\mathsf{sgnlz}(P, \pi, p.ap) = q$ for some q, $P \vdash p.ap \downarrow q$, and $\rho = [x.ap \mapsto q]$. As a result, $P \vdash p.ap \downarrow \rho(x.ap)$. □

Theorem 2 (termination) *For any program P, lock π and path p,* $\mathsf{sgnlz}(P, \pi, p)$ *is terminating.*

Proof. Termination is guaranteed since any recursive call $\mathsf{sgnlz}(P, \pi, p)$ is strictly decreasing with respect to a well-founded lexicographic ordering $\prec$ on pairs (π, p) of a path and a lock, where the two constituent ordering $\prec_\pi$ on locks and $\prec_p$ on paths are respectively defined as follows.

- $\pi_1 \prec_\pi \pi_2$ if $\pi_2 \subset \pi_1 \subset \mathit{Labels}(P)$.
- $p_1 \prec_p p_1.m$ for any field name m
- $p_1 \prec_p p_1(p_2)$ for any path p_2
- $p_2.ap \prec_p p_1(p_2)$ for any path p_1 and access path $ap \in S$, where S is the union of all access signatures of P.

Since $\mathit{Labels}(P)$ and S are finite, both $\prec_\pi$ and $\prec_p$ are well-founded, thus its lexicographic combination $\prec$ is well-founded too. □

A.3 Safe rewriting

Before proving the completeness of semi-ground normalization, we first need to introduce an alternative definition of safety, in terms of termination of a term rewriting system. We will show that the two definitions of safety are equivalent (Proposition 3). Then, we prove that when semi-ground normalization fails for a program, one can build an infinite reduction in this rewriting system, hence the program is unsafe according to the definition in Section 6.2. This alternative definition is also interesting for itself, since it gives a more computational view of what safety means, which may be more intuitive.

A way to formalize invalid paths concisely in the framework of path rewriting systems is to transform invalid paths into non-termination. That is, we introduce conditional path rewriting rules $Errors_P$ for a program P that cause non-termination whenever invalid paths appear during reduction:

(1) $\mathtt{x}.ap$	$\rightarrow \mathtt{x}.ap$	for any $\mathtt{x}$ and ap s.t. $ap \notin \mathsf{sig}_P(\mathtt{x})$
(2) $\mathtt{x}.ap(x)$	$\rightarrow \mathtt{x}.ap(x)$	for any path variable $\mathtt{x}$ and access path ap
(3) $p.m$	$\rightarrow p.m$	if $P \vdash p \mapsto (\theta, e)$, e not a path, and $P \vdash p.m \not\mapsto$
(4) $p(x)$	$\rightarrow p(x)$	if $P \vdash p \mapsto (\theta, e)$, and e is a structure

Here $\mathtt{x}$ indicates a path variable appearing in paths to be rewritten, not to be confused with the x used for rewriting rule variables. The first two rule sets cause non-termination when a variable is either decomposed beyond its access signature or applied; both cases break our syntactic restriction. The third one transforms dangling references to non-termination. The fourth one ensures that only functors are applied; otherwise it causes non-termination.

We also need to enforce the safety check on path arguments. For this we modify rewrite rules generated for functors, using the last rule of $Errors_P$ to let reduction progress.

$Rules(p, \lambda(x:S)e) =$
$Rules(p!(x), e) \cup \{p(x) \rightarrow \mathrm{snd}(\mathrm{chk}(x.ap_1, \ldots, x.ap_n), p!(x))$
where $ap_1 \ldots ap_n$ are the maximal access paths of S, *i.e.* all the paths s.t. $ap_i \in S \wedge \forall m\ ap_i.m \notin S$.

The idea is that $p(q).p'$ is a path where q has not been checked for safety as argument of p yet, and $p!(q).p'$ is the same path appearing in a context where the safety of q may have been checked. Actual path expansion occurs in two steps. First we rewrite $p(q).p'$ into $\mathrm{snd}(\mathrm{chk}(q.ap_1, \ldots, q.ap_n), p!(q)).p'$. Here snd and chk are added to evaluation contexts, with an extra rule for snd,

$$C[] ::= \ldots \mid \mathrm{snd}(C[], p) \mid \mathrm{chk}(\ldots, C[], \ldots)$$

$$(snd) \qquad\qquad \mathrm{snd}(x, y) \rightarrow y$$

so that one can reduce any of the $q.ap_i$, checking the safety of q as argument of p. Note that depending on which submodules of q are actually accessed in $p!(q).p'$, rewriting $q.ap_i$ may introduce a non-termination behavior that would not appear otherwise. One can then use rule (snd) to discard $\mathrm{chk}(q.ap_1, \ldots, q.ap_n)$. Then we are just left with $p!(q).p'$, which can be rewritten according to the definitions in P once all the applications have been converted. We also add the following rule to the lookup predicate, so that rules (3) and (4) of $Errors_P$ will work on both forms of applications.

$$\frac{P \vdash p(q) \mapsto (\theta, e)}{P \vdash p!(q) \mapsto (\theta, e)}$$

Definition 7 A program P is *safe for evaluation* if for all paths p appearing as right hand side of an abbreviation binding in P, there is no infinite reduction using $Rules_P \cup Errors_P$.

Proposition 3 *For any program P,* safety *(Definition 5) and* safety for evaluation *(Definition 7) are equivalent.*

Proof. We first show that if a program is safe, then it is safe for evaluation. *I.e.*, for any path q, if $P \vdash q : \{\}$ then there is no infinite reduction starting from q using the rules of $\mathit{Rules}_P \cup \mathit{Errors}_P$. For a term t of our path rewriting system, containing possibly some snd and chk, we define the *pure path* of t, noted $\mathrm{pp}(t)$, as t where all occurrences of snd were reduced (and, as a result, all occurrences of chk discarded), and all ! were removed. The number of *pure applications* in t, noted $\mathrm{pa}(t)$, is the number of path applications contained in t, after having reduced all the snd, and excluding the !-applications. We define the multiset of implicit paths of t, noted $\mathrm{pps}(t)$, by induction on the structure of t:

$$\begin{aligned}
\mathrm{pps}(t) &= \mathrm{pa}(t) \times (\{\mathrm{pp}(t)\} \cup \bigcup\{\mathrm{pps}(t') \mid t' \in \mathrm{chks}(t)\}) \\
\mathrm{chks}(\varepsilon) &= \emptyset \\
\mathrm{chks}(\mathtt{x}) &= \emptyset \\
\mathrm{chks}(t.m) &= \mathrm{chks}(t) \\
\mathrm{chks}(t_1(t_2)) &= \mathrm{chks}(t_1) \cup \mathrm{chks}(t_2) \\
\mathrm{chks}(\mathrm{snd}(\mathrm{chk}(t_1,\ldots,t_n),t)) &= \{t_1,\ldots t_n\} \cup \mathrm{chks}(t)
\end{aligned}$$

By $n \times S$ we mean that we duplicate n times the contents of the multiset S. We define the measure of a term t as the multiset of the sizes of the derivations of $P \vdash p : \{\}$, for p in $\mathrm{pps}(t)$. We prove that any reduction step $t \longrightarrow t'$ keeps the typability of the paths in $\mathrm{pps}(t')$, and decreases this measure according to the multiset ordering. If the reduction is one of the original ones, then according to lemma 4, each path in $\mathrm{pps}(t)$ is either reduced into a corresponding path in $\mathrm{pps}(t')$, with its derivation size reduced, or it remains unchanged, and at least one pure path is reduced. If the new rule $p(x) \longrightarrow \mathrm{snd}(\mathrm{chk}(x.ap_1,\ldots,x.ap_n),p!(x))$ is applied, then the number of pure applications of $p(q)$ is reduced by 1, and we replace this $p(q)$ by $q.ap_1,\ldots,q.ap_n$, which were already contained in the derivation of $P \vdash p(q) : \{\}$, so that they are safe, and the measure decreases. If $\mathrm{snd}(x,y) \longrightarrow y$ is applied, then we discard the first argument, which leaves $\mathrm{pp}(t')$ intact, and the measure decreases. Rules (1)-(4) cannot apply, since $\mathrm{pp}(t)$ is safe.

Since the multiset ordering is well-founded, this proves that there cannot be infinite reductions.

Conversely, we prove that if there is no infinite reduction using $\mathit{Rules}_P \cup \mathit{Errors}_P$, then P must be safe. We define the relation $p \succ q$ iff p can be rewritten into a path q_0 such that $C[\theta(q)] \in \mathrm{pps}(q_0)$ for some path context C and substitution θ. This relation must be anti-symmetric on the paths appearing in P (*i.e.* we never have both $p \succ q$ and $q \succ p$, otherwise we could easily build an infinite reduction.) As a result $\succ$ can be topologically extended into a total order on the paths appearing as right hand side of abbreviation bindings in P.

We prove that if there is no infinite reduction starting from p, then $P \vdash p : \{\}$, starting with the smallest paths in this topological order (those with no q in P such that $p \succ q$). We rewrite p using a customized strategy. That is, we basically use a call-by-name strategy (not reducing arguments), with the exception of the arguments of chk which we reduce to normal form before discarding them with snd. Thanks to this strategy, we are able to build our safety derivation straightforwardly. Namely, we obtain the safety of substitutions from the normalization of chk, and then actually apply the original rewriting rule on the non-reduced argument, as does our inference system. Moreover, since we check following a topological order, we can use our substitution lemma to build the third premise of r-exp rule. Other inference rules need just to be inserted as glue, as they perform no actual reduction.

□

A.4 Completeness proof

The following lemmas are used by the completeness proof. The first one proves that postponing substitution is correct.

Lemma 3 (postponement) *If p is a rooted path,* $\mathsf{sgnlz}(P,\pi',p) = q$ *and* $\mathsf{vp\_subs}(P,\pi,\theta) = \rho$ *with* $\pi \subset \pi'$*, then* $\mathsf{sgnlz}(P,\pi,\theta(p)) = \mathsf{subs}(\theta,\rho,q)$.

Proof. (Coq-checked) By functional induction on $\mathsf{sgnlz}(P,\pi',p)$. □

Lemma 5 (idempotence) *If* $\mathsf{sgnlz}(P,\pi,p) = q$*, then* $\mathsf{sgnlz}(P,\pi,q) = q$.

Proof. By functional induction, using lemma 3. □

Theorem 3 (completeness) *For any program P and path p, if P is safe and $P \vdash p : \{\}$, then* $\mathsf{sgnlz}(P,\emptyset,p) \neq \mathsf{error}$.

Proof. By proposition 3, $P \vdash p : \{\}$ implies that there is no infinite reduction from p using $Rules_P \cup Errors_P$.

We show by functional induction that $\mathsf{sgnlz}(P,\pi,p) = \mathsf{error}$ implies the existence of such an infinite reduction starting from either p or an abbreviation in P, assuming that for each $i \in \pi$, there are paths p_i and q_i such that $P \vdash p_i \mapsto (id,(q_i)^i)$, and reductions $q_i \xrightarrow{*} C_i[\theta_i(p)]$. In case of repeated attempts to expand the same abbreviation, we will use them to build an infinite reduction. Since some recursive calls to sgnlz will be on paths that do not appear in P, we also keep a reduction $q_0 \xrightarrow{*} C_0[p]$, q_0 being either the original p, or the last abbreviation in P we are currently normalizing. This last reduction will be used for errors, producing an infinite reduction with rules of $Errors_P$. For the sake of simplicity, we do not distinguish normal applications and !-applications in the paths we reduce. We just assume that where needed we can get the arguments of chk, in order to build an infinite reduction.

We start with $\pi = \emptyset$ and $q_0 = p$, so that our only reduction is the 0-step one from q_0 to p.

If $p = x$ or $p = \varepsilon$, then $\mathsf{sgnlz}(P,\pi,p) = p \neq \mathsf{error}$.

If $p = p_1.m$ or $p = p_1(p_2)$, then we first try to evaluate $\mathsf{sgnlz}(P,\pi,p_1)$. If it results in an error, then we can reuse our reductions $q_i \xrightarrow{*} C_i[\theta_i(p)]$ $(i \in \pi)$, as $C_i[\theta_i(p)]$ contains $\theta_i(p_1)$ (*e.g.* $C_i[\theta_i(p)] = C_i[\theta_i(p_1).m]$ in the first case), and $q_0 \xrightarrow{*} C_0[p]$ since p contains p_1. Using the induction hypothesis we conclude that there is an infinite reduction.

If $\mathsf{sgnlz}(P,\pi,p_1) = p'_1$, then we call $\mathsf{expand}(P,\pi,p')$, with p' either $p'_1.m$ or $p'_1(p_2)$, and since by correctness we have a reduction $p_1 \xrightarrow{*} p'_1$, we also have a reduction $p \xrightarrow{*} p'$, and we can complete all reductions $q_i \xrightarrow{*} C_i[\theta_i(p)]$ $(i \in \pi)$ into reductions $q_i \xrightarrow{*} C_i[\theta_i(p')]$, and $q_0 \xrightarrow{*} C_0[p']$. If $p' = x.ap$, then to have an error it must be $ap \notin \mathsf{sig}_P(x)$, and we can use a reduction (1) of $Errors_P$ to create an infinite reduction $q_0 \xrightarrow{*} C_0[p'] \rightarrow C_0[p'] \rightarrow \dots$ If p' is a rooted path, then we first call $\mathsf{lookup}(P,p')$. If this fails we can use either a reduction (3) or (4) of $Errors_P$ to create an infinite reduction. Otherwise, we obtain θ and e.

Then we call $\mathsf{vp\_subs}(P,\pi,\theta)$. From the reduction point of view, if p'_j is an argument of p', then there is a prefix $p_j(p'_j)$ of p', and we can find a step $p_j(p'_j) \rightarrow \mathrm{snd}(\mathrm{chk}(p'_j.ap_{j1},\dots,$ $p'_j.ap_{jn_j}), p_j!(p'_j))$ inside the reduction leading to p', allowing us to check all the required subpaths of the argument p'_j. This is also what $\mathsf{sig\_subs}(P,\pi,x_j,p'_j,\mathsf{sig}_P(x_j))$ does. If p'_j is a variable path $x.ap$, then it checks for each $ap' \in \mathsf{sig}_P(x_j)$ whether $ap.ap' \in \mathsf{sig}_P(x)$. Since all those ap' are prefixes of some of the ap_{jk}, if we have an error then we can build an infinite reduction starting from $p'_j.ap_{jk}$. If p'_j is a rooted path, then for each $ap' \in \mathsf{sig}_P(x_j)$, we call $\mathsf{sgnlz}(P,\pi,p'_j.ap')$. If one of them fails, we can construct a context C such that $p' \xrightarrow{*} C[p'_j.ap_{jk}]$, so that we can complete all our reductions into $q_i \xrightarrow{*} C_i[\theta_i(C[p'_j.ap_{jk}])]$ $(i \in \pi)$ and $q_0 \xrightarrow{*} C_0[C[p'_j.ap_{jk}]]$, and by induction hypothesis we have an infinite reduction starting from q_0.

If $\mathsf{vp\_subs}(P,\pi,\theta)$ succeeds, the next step is either to return successfully from expand, which contradicts the presence of an error, or we have $e = q^i$, and we check whether $i \in \pi$. If $i \in \pi$, then since $p' \to \theta(q) = \theta(q_i)$, we have a reduction $q_i \xrightarrow{*} C_i[\theta_i(p')] \to C_i[\theta_i(\theta(q_i))]$, so that we can build an infinite reduction by repeatedly appending it to itself, *i.e.* $q_i \xrightarrow{*} C_i[\theta_i(\theta(q_i))] \xrightarrow{*} C_i[\theta_i(\theta(C_i[\theta_i(\theta(q_i))]))] \xrightarrow{*} \ldots$

If $i \notin \pi$, we call $\mathsf{sgnlz}(P,\{i\}\cup\pi,q)$, which must fail. First we need to extend our reductions. We have $P \vdash p' \mapsto (\theta,q)$, and $p' \xrightarrow{*} \theta(q)$. We can combine this reduction with our other reductions to obtain $q_j \xrightarrow{*} C_j[\theta_j(\theta(q))]$ $(j \in \pi)$. Since q is an abbreviation in P, we update q_0 to q, with new reductions $q_i = q \xrightarrow{*} q$, and $q_0 = q \xrightarrow{*} q$. So by induction hypothesis, in case of error we have an infinite reduction starting from one of the q_i or q_0. □

References

1. G. Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14:263–315, 2004.
2. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.
3. Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Int. Symposium on Mathematical Foundations of Computer Science*, Springer LNCS, September 2006.
4. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *IEEE Symposium on Logic in Computer Science*, 1990.
5. D. Dreyer. A type system for well-founded recursion. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.
6. D. Dreyer. Recursive type generativity. In *ACM SIGPLAN International Conference on Functional Programming*, 2005.
7. D. Dreyer. Post to the Caml Mailing List, `http://caml.inria.fr/pub/ml-archives/caml-list/2007/03/73e1ea81e35002046fdce6f14c1d8848.en.html`, 2007.
8. D. Dreyer. A type system for recursive modules. In *ACM SIGPLAN International Conference on Functional Programming*, 2007.
9. D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249, 2003.
10. D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ACM SIGPLAN International Conference on Functional Programming*, pages 307–320, 2008.
11. R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354, 1990.
12. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, volume 2305 of *Springer LNCS*, pages 6–20, 2002.
13. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.
14. H. Im, K. Nakata, J. Garrigue, and S. Park. A syntactic type system for recursive modules. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 993–1012, 2011.
15. Gilles Kahn. Natural semantics. In *Symposium of Theoretical Aspects of Computer Science*, pages 22–39, 1987.
16. X. Leroy. Applicative functors and fully transparent higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, 1995.
17. X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

18. X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
19. X. Leroy. A proposal for recursive modules in Objective Caml. Available at `http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf`, 2003.
20. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.11. Software and documentation available on the Web, `http://caml.inria.fr/`, 2008.
21. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
22. B. Montagu and D. Rémy. Modeling abstract types in modules with open existential types. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 354–365, 2009.
23. M. Mottl. OCaml bug report. `http://caml.inria.fr/mantis/view.php?id=3476`, 2005.
24. K. Nakata. OCaml bug report. `http://caml.inria.fr/mantis/view.php?id=3674`, 2005.
25. K. Nakata and J. Garrigue. Recursive modules for programming. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2006.
26. K. Nakata and J. Garrigue. Path resolution for recursive nested modules is undecidable. In *Int. Workshop on Termination (WST)*, Paris, France, June 2007.
27. K. Nakata, A. Ito, and J. Garrigue. Recursive object-oriented modules. In *Int. Workshop on Foundations of Object-Oriented Languages*, 2005.
28. C. Russo. First-class structures for Standard ML. In *European Symposium on Programming*, volume 1782 of *Springer LNCS*, 2000.
29. C. Russo. Recursive structures for Standard ML. In *ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM Press, 2001.
30. C. V. Russo. Non-dependent types for standard ml modules. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, volume 1702 of *Springer LNCS*, pages 80–97, 1999.
31. G. A. Washburn. Scala mailing list. `http://article.gmane.org/gmane.comp.lang.scala/13573/`, 2008.