

Path Resolution for Nested Recursive Modules

Jacques Garrigue

*Graduate School of Mathematics, Nagoya University**

Keiko Nakata

Institute of Cybernetics, Tallinn University of Technology†

Draft of 2010/02/23 — Submitted for Publication.

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

In this paper, we address one of such problems, 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 its strong support of program structuring (Milner et al., 1997a; Leroy, 2000). Two of its important features are nested structures and functors. Firstly 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 namespace hierarchically. Secondly ML

* Chikusa-ku, NAGOYA 464-8602, Japan

† Akadeemia tee 21, TALLINN, EE-12618, Estonia

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 (Russo, 2001). The absence of recursive modules also hinders extensible program development (Nakata and Garrigue, 2006).

Introducing recursive modules is a natural way out of this predicament. Recursion is a powerful language construct, hence its addition certainly increases the expressiveness of the module system. At the same time the addition might be at the risk of static guarantees on safety that the current ML enjoys, such as decidable type checking or error-free module initialization. Indeed extensions with recursion of the module system pose non-trivial problems and much work has been devoted to investigate extensions that retain desirable safety guarantees while not introducing too many constraints (Crary et al., 1999; Russo, 2001; Dreyer, 2005; Boudol, 2004; Hirschowitz and Leroy, 2002; Dreyer, 2004).

In our previous work (Nakata and Garrigue, 2006), we have studied an applicative module system (Leroy, 1995) with polymorphic functors and recursion based on paths. Whereas several different approaches to accounting for ML-style modules have been proposed (Milner et al., 1997b; Russo, 1999; Leroy, 1996; Harper et al., 1990; Dreyer et al., 2003), we found arguably a path-based approach natural from the programmer's viewpoint. One problem raised 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 with a recursive module extension, 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. Worse, the runtime may diverge while resolving path references in order to prepare code for execution. Module mechanisms arising from the fundamental process of linking (Cardelli, 1997), diverging path resolution corresponds to divergence of the linking process, which is certainly undesirable.

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 are often paid less attention. This result, together 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 it 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 (Nakata and Garrigue, 2006) we studied a type system for recursive ML-style modules with applicative functors, by designing a language named *Traviata*. In this section we overview the type system to motivate our study of path resolution in the paper. The type system is formalized in (Nakata and Garrigue, 2006), where an interested reader can find the details.

The module system of OCaml (Leroy et al., 2008) adopts applicative functors (Leroy, 1995) and supports a recursive module extension (Leroy, 2003). 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 an extension of the applicative module system with recursion by taking “paths” as the central concept.

The key extension 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 ¹. 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*" denotes both an equivalence between paths (*I* is equivalent to *Int*) and signatures (*I* has the same signature as *Int*).

The main reason for this extension is to circumvent an aspect of incompleteness in OCaml's applicative functors, which arises due to the fact that the type system does not record module abbreviations, but only keeps track of (core) type equality 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 of course equivalent. Functors being applicative, *AofInt1.t* and *AofInt2.t* are equivalent too. Since the type system does not take into account the fact that *I* is an abbreviation for *Int*, however, the two types *AofInt1.t* and *AofInt3.t* are not equivalent in OCaml. This kind of incompatibilities are 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 is a deficiency in OCaml's type system that we addressed in our previous work. The idea is simple: we extend the type system to keep track of type as well as module abbreviations. As a result *AofInt1.t* and *AofInt3.t* are equivalent in *Traviata*, since *I* is unfolded to *Int*. As seen

¹ In *Traviata*, structures as well as signatures for structures are extended with declarations of self variables, or recursion variables. This 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.

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

$$\text{module } Id = \text{functor } (X : \text{sig } \text{end}) \rightarrow X$$

will be given the signature

$$\text{module } Id : \text{functor } (X : \text{sig } \text{end}) \rightarrow X$$

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 $\text{functor } (X : \text{sig } \text{end}) \rightarrow \text{sig } \text{end}$), 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 can be simulated in OCaml by using an abstract signature:

$$\begin{aligned} \text{module } Id = \\ \text{functor } (X : \text{sig } \text{module type } S \text{ module } M : S \text{ end}) \rightarrow X.M \end{aligned}$$

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 in a recursive module extension where, due to recursion variables, syntactically different paths may refer to the same module, as illustrated by the next example.

$$\begin{aligned} \text{module } L = \text{struct } (Z) \\ \text{module } M = \text{struct } (Z') \\ \text{module } N = \text{struct type } t = \text{Int of int end} \\ \text{end} \\ \text{end} \end{aligned}$$

In the above example, Z and Z' are recursion variables as in Moscow ML (Russo, 2001). 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 of Z and Z' are in the scope. This shows how weak is syntactic module path equality in the presence of recursion.

These extensions also fix another aspect 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 let compare x y = 0 end
  module S = MakeSet(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*: since the module path equality of M' to M is kept in the signature, it is possible to deduce that $M'.N$ and $M.N$ refer to the same module.

Having module paths in signatures allows for a limited form of recursive signatures. But the resulting signature language is less powerful than recursively dependent signatures (Crary et al., 1999). In Figure 1 we give a typical use of recursive modules: *Tree* and *Forest* refer to each other recursively, implementing recursive data types and functions on them across module boundaries. The module *TreeForest* is given the signature in Figure 2, retaining module path equalities between S and $MakeSet(S)$, F and $TF.Forest$, and T and $TF.Tree$. These equivalences are used to type check *TreeForest*.

More generally, given equalities on module paths, we have to decide if, for any given paths p and q , p and q are equivalent, that is, if p and q refer to the same module. For instance $p.M$ and $q.M$ are equivalent if p is an abbreviation of q and q has a submodule named M . Moreover, $F(M)$ and M are equivalent if F is an identity functor. Path normalization is a standard way to check equality, so that we can identify the module that a path refers to without further unfolding abbreviations. For the type checking to be decidable, path normalization must be terminating, which is the subject of the study in this paper.

² This problem alone could be fixed by adapting Dreyer's proposal (Dreyer, 2007a). 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 = MakeSet(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

```

Figure 1. Trees and Forest

The need for path normalization during type checking is not specific to the design choices of *Traviata*. The type system of OCaml has its own normalization algorithm, for checking type equality, which may diverge on some programs (Nakata, 2005). Neither is the problem of path resolution specific to an applicative module system with recursion. Sophisticated object systems such as ν Obj (Odersky et al., 2003), the theoretical underpinning of the Scala programming language (Programming Methods Laboratory, EPFL, 2007), share the same problem (Cremet et al., 2006).

```

module TreeForest :
  functor(X : sig type t val compare : t → t → bool end) →
  struct (TF)
    module S : MakeSet(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
end

```

Figure 2. Signature of *TreesForest*

<i>Expressions</i>	e	$::=$	$\{m_1 = e_1 \cdots m_n = e_n\} \mid \lambda(x)e \mid p$
<i>Paths</i>	p, q	$::=$	$\epsilon \mid x \mid p.m \mid p_1(p_2)$
<i>Program</i>	P	$::=$	$\{m_1 = e_1 \cdots m_n = e_n\}$

Figure 3. Syntax

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 to 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) any sequence of bindings in a structure does not bind the same name twice; 2) a program does not contain free variables, where free occurrence of variables is defined in the standard way.

The calculus is kept small 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 with the intuition of path reduction, the syntax we chose for this formal calculus is different. When translating a *Traviata* signature into a structure of this calculus, one just has to first drop all core language values and types, then replace all self references with paths starting from ϵ , and finally use the following conversion:

$$\begin{aligned} \overline{\text{sig module } M_1 : S_1 \dots \text{ module } M_n : S_n \text{ end}} &= \{M_1 = \overline{S_1} \dots M_n = \overline{S_n}\} \\ \overline{\text{functor } (X : S_1) \rightarrow S_2} &= \lambda(X)\overline{S_2} \\ \overline{p} &= p \end{aligned}$$

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 6.5.

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

4. Semantics

Path resolution rewrites paths into *source form*. Until we formally define it later, source form can be explained as a normal form not containing dangling references.

To deliver the intuition of path resolution, let us consider the following program:

$$\left\{ \begin{array}{l} \mathbf{m}_1 = \{\mathbf{n}_1 = \{\mathbf{n} = \{\}\}\} \quad \mathbf{n}_2 = \epsilon.\mathbf{m}_1.\mathbf{n}_1 \\ \mathbf{m}_2 = \lambda\mathbf{x}.\{\mathbf{n}_1 = \{\}\} \quad \mathbf{n}_2 = \mathbf{x}.\mathbf{n}_2 \quad \mathbf{n}_3 = \epsilon.\mathbf{m}_2(\mathbf{x}).\mathbf{n}_1 \\ \mathbf{m}_3 = \epsilon.\mathbf{m}_2(\epsilon.\mathbf{m}_1).\mathbf{n}_2 \end{array} \right\}$$

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

$$\left\{ \begin{array}{l} \epsilon.\mathbf{m}_1.\mathbf{n}_2 \rightarrow \epsilon.\mathbf{m}_1.\mathbf{n}_1, \quad \epsilon.\mathbf{m}_2(\mathbf{x}).\mathbf{n}_2 \rightarrow \mathbf{x}.\mathbf{n}_2, \\ \epsilon.\mathbf{m}_2(\mathbf{x}).\mathbf{n}_3 \rightarrow \epsilon.\mathbf{m}_2(\mathbf{x}).\mathbf{n}_1, \quad \epsilon.\mathbf{m}_3 \rightarrow \epsilon.\mathbf{m}_2(\epsilon.\mathbf{m}_1).\mathbf{n}_2 \end{array} \right\}$$

According to these rules, we can induce the reduction steps:

$$\epsilon.\mathbf{m}_3 \rightarrow \epsilon.\mathbf{m}_2(\epsilon.\mathbf{m}_1).\mathbf{n}_2 \rightarrow \epsilon.\mathbf{m}_1.\mathbf{n}_2 \rightarrow \epsilon.\mathbf{m}_1.\mathbf{n}_1$$

which reflects the previous informal explanation of path resolution for $\epsilon.\mathbf{m}_2(\epsilon.\mathbf{m}_1).\mathbf{n}_2$.

4.1. TERMINOLOGY

We first introduce the basic terminology and useful notations for our formalization.

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

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

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 $\text{dom}(\theta)$ to denote the domain of θ . Application of a substitution θ to a path p , written $\theta(p)$, is defined by:

$$\begin{aligned} \theta(\epsilon) &= \epsilon & \theta(x) &= \begin{cases} x & \text{when } x \notin \text{dom}(\theta) \\ p & \text{when } x \in \text{dom}(\theta) \text{ and } \theta(x) = p \end{cases} \\ \theta(p.m) &= \theta(p).m & \theta(p_1(p_2)) &= \theta(p_1)(\theta(p_2)) \end{aligned}$$

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.

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 $C[]$.

A *path rewrite* rule is a pair (p, p') of paths. It will be written $p \rightarrow p'$. A path rewrite system R is a set of path rewrite rules $\{p_1 \rightarrow p'_1, \dots, p_n \rightarrow p'_n\}$. 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 $p_i \rightarrow p'_i \in R$ such that $p = C[\theta(p_i)]$ and $p' = C[\theta(p'_i)]$. We write $p \rightarrow_R p'$ 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 \rightarrow_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 $p \rightarrow_R q$. 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 familiar in term rewriting. For instance, we could represent a path $\epsilon.m_1.m_2(\epsilon.m_3)(x).m_4$ as $m_4(\text{app}(\text{app}(m_2(m_1(\epsilon)), m_3(\epsilon)), x))$, 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 a normal form with respect to the path rewrite system corresponding to P . Second the normal form is checked to be in source form by making certain that it

$$\begin{aligned}
Rules(p, \{m_1 = e_1 \dots m_n = e_n\}) &= \bigcup_{i=1}^n Rules(p.m_i, e_i) \\
Rules(p, \lambda(x)e) &= Rules(p(x), e) \\
Rules(p, p') &= \{p \rightarrow p'\}
\end{aligned}$$

Figure 4. Path rewrite rules of a program

does not contain 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 does 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 $p'.m \rightarrow p$ (the last case).

We use a shorthand $Rules_P$ for $Rules(\epsilon, P)$ and call it the path rewrite system of P .

It is important to notice that $Rules_P$ does not contain overlapping rules for any P . According to the conventions of Section 3, the module names m_1, \dots, 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 $Rules_P$.

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 $\epsilon.m_1.n_2$ to $\epsilon.m_1.n_1$ and the path $\epsilon.m_2(x).n_1$ to $\{\}$, but would fail for the path $\epsilon.m_1.n_3$ or $\epsilon.m_1.n_2.n$; the former is dangling and the latter needs to be rewritten into $\epsilon.m_1.n_1.n$ 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

$$\frac{\overline{P \vdash \epsilon \mapsto (id, P)}}{P \vdash p \mapsto (\theta, \{\dots m = e \dots\})} \quad \frac{P \vdash p_1 \mapsto (\theta, \lambda(x)e)}{P \vdash p_1(p_2) \mapsto (\theta[x \mapsto p_2], e)}$$

Figure 5. Lookup

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 in terms of the lookup relation we define the notion of source form. A path p is in source form if any path contained in p refers to either 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. However, having both higher-order functors and recursion, termination

of path evaluation is generally undecidable; the problem amounts to determining termination of normalization for a lambda calculus with recursion, which is clearly undecidable.

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

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 $\{\mathfrak{m}_1 = \lambda(\mathbf{x})\epsilon.\mathfrak{m}_2 \quad \mathfrak{m}_2 = \epsilon.\mathfrak{m}_1(\epsilon.\mathfrak{m}_3)\}$ and $\{\mathfrak{m}_2 = \epsilon.\mathfrak{m}_1(\epsilon.\mathfrak{m}_2)\}$ do not terminate on path $\epsilon.\mathfrak{m}_2$, while $\{\mathfrak{m}_1 = \lambda(\mathbf{x})\epsilon.\mathfrak{m}_1\}$ and $\{\mathfrak{m}_1 = \lambda(\mathbf{x})\epsilon.\mathfrak{m}_2(\mathbf{x}) \quad \mathfrak{m}_2 = \lambda(\mathbf{x})\mathbf{x}\}$ always terminate. In order to decide termination, we build a non-deterministic push-down automaton with states the names of our modules and functors, and only one stack symbol, where the length of the stack tells us the number of arguments we have. Our first example gets encoded as $\{(\mathfrak{m}_1, \mathbf{n} + 1) \rightarrow (\mathfrak{m}_2, \mathbf{n}), (\mathfrak{m}_2, \mathbf{n}) \rightarrow (\mathfrak{m}_1, \mathbf{n} + 1), (\mathfrak{m}_2, \mathbf{n}) \rightarrow (\mathfrak{m}_3, \mathbf{n})\}$. We then check whether this push-down automaton may have infinite transitions starting from a finite stack, which is a decidable problem. Here we see that there is a transition from $(\mathfrak{m}_1, 1)$ to $(\mathfrak{m}_1, 1)$, 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 $\epsilon.m_1.m_2.\dots.m_i \rightarrow \epsilon.m'_1.m'_2.\dots.m'_j$. We can then encode our program as a deterministic push-down automaton with a single state, using names as stack alphabet, each rule defining a transition. Termination becomes equivalent to whether this push-down 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 (Dauchet and Tison, 1990).

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

Figure 6. A first-order fragment

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 push-down automaton. Since a Turing machine is essentially a push-down 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 easier to check the first one.

1. The path rewrite system of a program P does not yield 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 $\epsilon.m$ can appear in functor positions.
2. A program is a sequence of toplevel expressions, which are lambda abstraction 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 (Hopcroft et al., 2001), 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 losing 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 configuration $a_1 a_2 \cdots a_{i-1} q a_i a_{i+1} \cdots a_n$ of a Turing machine is encoded by a path

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

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 $\epsilon.q$ 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.

In the rest of the section, we present our encoding in the following three steps.

1. Firstly we explain how to construct a set of path rewrite rules R_M from any Turing machine M .
2. Then, we show that R_M encodes the Turing machine M .
3. Finally, we observe a program P whose path rewrite system is R_M .

Given a Turing machine M , we construct a path rewrite system R_M , which is the union of the following sets:

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

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

$$a_1 \cdots a_{i-1} q a_i a_{i+1} \cdots a_n \vdash a_1 \cdots a_{i-2} q' a_{i-1} a'_i a_{i+1} \cdots a_n$$

We have reductions

$$\begin{aligned} & \epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_i.a_{i+1}\cdots.a_n.\hat{b} \\ \rightarrow & \epsilon.a_{i-1}(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).q'.a'_i.a_{i+1}\cdots.a_n.\hat{b} \\ \rightarrow & \epsilon.q'(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_{i-1}.a'_i.a_{i+1}\cdots.a_n.\hat{b} \end{aligned}$$

2. When $i = 1$, then we have a move:

$$q a_1 a_2 \cdots a_n \vdash q' b a'_1 a_2 \cdots a_n$$

We have reductions

$$\begin{aligned} & \epsilon.q(\epsilon.\hat{b}(\epsilon)).a_1.a_2\cdots.a_n.\hat{b} \\ \rightarrow & \epsilon.\hat{b}(\epsilon).q'.a'_1.a_2\cdots.a_n.\hat{b} \\ \rightarrow & \epsilon.q'(\epsilon.\hat{b}(\epsilon)).b.a'_1.a_2\cdots.a_n.\hat{b} \end{aligned}$$

3. When $i = n$ and $a'_i = b$, then we have a move:

$$a_1 a_2 \cdots a_{n-1} q a_n \vdash a_1 a_2 \cdots a_{n-2} q' a_{n-1}$$

We have reductions

$$\begin{aligned} & \epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_n.\hat{b} \\ \rightarrow & \epsilon.a_{i-1}(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).q'.b.\hat{b} \\ \rightarrow & \epsilon.q'(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).b.\hat{b} \end{aligned}$$

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

It is also straightforward to construct a program whose path rewrite system is R_M . Firstly we recall that the path rewrite system of the program $\{q = \lambda(x)\{a = \epsilon.q'(\epsilon.a'(x))\}\}$ is $\{\epsilon.q(x).a \rightarrow \epsilon.q'(\epsilon.a'(x))\}$. The rule exactly corresponds to rules from the first set above. In general the toplevel structure of the program consists of the following definitions.

1. For each q in Q ,

$$\begin{aligned} q = \lambda(x)\{ & \\ & a_1 = \epsilon.q'_1(\epsilon.a'_1(x)) \cdots a_m = \epsilon.q'_m(\epsilon.a'_m(x)) \\ & b_1 = x.r'_1.b'_1 \cdots b_n = x.r'_n.b'_n \\ & \hat{b} = \epsilon.q(x).b.\hat{b} \\ & \} \end{aligned}$$

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

$$\{(b_1, r'_1, b'_1), \cdots, (b_n, r'_n, b'_n)\} = \{(b, r', b') \mid \delta(q, b) = (r', b', L)\}$$

2. $\hat{b} = \lambda(x)\{q_1 = \epsilon.q_1(\epsilon.\hat{b}(x)).b \cdots q_n = \epsilon.q_n(\epsilon.\hat{b}(x)).b\}$
where $\{q_1, \cdots, q_n\} = Q$.

3. For each a in Γ ,

$$a = \lambda(x)\{q_1 = \epsilon.q_1(x).a \ \cdots \ q_n = \epsilon.q_n(x).a\}$$

where $\{q_1, \dots, q_n\} = Q$.

It is worth noticing that this encoding 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 loose its Turing completeness.

This result was shown for the untyped calculus, but it is possible to extend it to a typed one. Suppose that we annotate functor arguments with (regular) recursive signatures. *I.e.* an argument signature indicates all the submodules an argument must provide. This seems a natural enough notion of typing, but this would not help obtain decidability. Indeed, supposing that the transition function of our Turing machine is total, we could just annotate all variables with the following signature T :

$$S = \mu X.\{a_1 : X \ \dots \ a_n : X \ \hat{b} : X\}$$

$$T = \{q_1 : S \ \dots \ q_n : S\}$$

where $Q = \{q_1, \dots, q_n\}$ and $\Sigma = \{a_1, \dots, a_n\}$. It is easy to verify that our encoding of a Turing machine is well-typed with respect to these annotations. All arguments to functors are either variables (which all share the same signature), or application of a toplevel functor $\epsilon.a_1, \dots, \epsilon.a_n, \epsilon.\hat{b}$ to a single argument. If we look at the body of these functors, they are all of the form $q_i = \epsilon.q_i(arg).a$, with arg either a variable or $\epsilon.\hat{b}(x)$, so that both arg and $\epsilon.a_i(x)$ satisfy T if $\epsilon.q_i(arg)$ satisfies S . If we look at the body of $\epsilon.q_i$, it is composed of bindings either of the form $b_i = x.r'_i.b'_i$, where the right-hand sides satisfy S since x satisfies T , or of the form $a_i = \epsilon.q'_i(\epsilon.a'_i(x))$, which satisfies S too, or of the form $\epsilon.q(x).b.\hat{b}$, which again satisfies S . So path resolution stays undecidable even if we restrict functor arguments through explicit regular signatures.

6. Terminating path evaluation

In the previous section we have seen that, while path evaluation is undecidable in the presence of recursion, first-order functors and nested modules, it is decidable if we drop any of them. Indeed termination of path evaluation is then decidable, and since determining whether a

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	$rp ::= \epsilon \mid rp(p) \mid rp.m$
Access paths	$ap ::= \epsilon \mid ap.m$

Figure 7. Revised syntax

path is in source form or not is decidable too, so is path evaluation. By examining the proof of the undecidability result, in particular the encoding of a Turing machine by a program, we notice that this encoding relies essentially on unfettered access to submodules of functor arguments. Namely, if we use a signature to describe the way we access the functor argument x of module $q \in Q$ in the toplevel structure, we need a recursive signature to type unfettered accesses $x.q.a_1$, $x.q.a_1.a_2$ and $x.q.a_1.a_2.a_3$, etc. . . If we restrict such accesses, we may well recover decidability.

In this section, we formalize such a restriction, called *bounded submodule access*: a submodule of a functor argument may be accessed only if it is present in the *access signature* of this argument, where an access signature can only have finitely deep nesting. We present and prove correct and complete a terminating path evaluation algorithm, named *semi-ground normalization*, which implements this restriction.

6.1. ACCESS SIGNATURES

The starting point of our restriction is to annotate abstracted variables with access signatures, indicating which submodules can be accessed. We therefore revise our language as given in Figure 7. 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 signatures. Since we do not allow higher-order functors, access signatures only contain signatures for structures. Paths distinguish *variable paths* and *rooted paths*, where only the latter may contain functor applications in accordance with the absence of higher-order functors. An *access path* is just a sequence of module names. We define concatenation of a path p and an access path ap , noted $p.ap$, 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 .

In order to simplify definitions, we assume that all bound variables in a program are distinct, and we define $\text{sig}_P(x)$ as the access signature associated to x in P . If the signature is omitted in a variable binding, then an empty access signature is assumed.

As expected, our restriction stipulates that attempting to access a submodule of a variable not in its access signature is an error, as is passing an argument which does not satisfy the access signature. Hence we must at least reject variable paths in which accessed modules are not in the access signature of the variable. We are also syntactically rejecting paths of the form $x(p)$. Unfortunately, these purely syntactic restrictions are not sufficient to eliminate invalid accesses during evaluation. For instance, look at the following program.

$$\{\mathfrak{m}_1 = \lambda(\mathbf{x} : \{\})\{\mathfrak{m}_2 = \mathbf{x} \quad \mathfrak{m}_3 = \epsilon.\mathfrak{m}_1(\mathbf{x}).\mathfrak{m}_2.\mathfrak{m}_4\} \quad \mathfrak{m}_5 = \{\}\}$$

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

$$\epsilon.\mathfrak{m}_1(\epsilon.\mathfrak{m}_5).\mathfrak{m}_3 \rightarrow \epsilon.\mathfrak{m}_1(\epsilon.\mathfrak{m}_5).\mathfrak{m}_2.\mathfrak{m}_4 \rightarrow \epsilon.\mathfrak{m}_5.\mathfrak{m}_4$$

To effectively enforce well-typed uses of variables as stipulated by their access signatures, we need to make sure that evaluating any of the paths appearing in a program does not require ill-typed access to variables. Recall that we are dealing with a program whose well-typedness is not known yet. 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.

In the next subsection we introduce a notion of “safe programs”, a specialized notion of well-typedness for path evaluation, by means of derivability of a judgment. We then prove that if a program is safe then path evaluation against this program is successful for any ground path (Proposition 1). In Section 6.3 we provide an alternative definition of safety, based on path rewriting systems, and prove that the two definitions are equivalent. In Section 6.4 we develop and prove correct and complete a terminating algorithm, deciding the safety of any given program, while enforcing well-typed uses of variables simultaneously.

$$\begin{array}{c}
\text{r-src} \frac{P \vdash p \mapsto (\theta, e) \quad P \vdash \theta \text{ safe} \quad e \text{ not a path}}{P \vdash p \downarrow p} \quad \text{r-vp} \frac{ap \in \text{sig}_P(x)}{P \vdash x.ap \downarrow x.ap} \\
\text{r-exp} \frac{P \vdash p \mapsto (\theta, p') \quad P \vdash \theta \text{ safe} \quad P \vdash \theta(p') \downarrow q}{P \vdash p \downarrow q} \\
\text{r-dot} \frac{P \vdash p \downarrow p' \quad P \vdash p'.m \downarrow q}{P \vdash p.m \downarrow q} \quad \text{r-app} \frac{P \vdash p_1 \downarrow p'_1 \quad P \vdash p'_1(p_2) \downarrow q}{P \vdash p_1(p_2) \downarrow q} \\
\text{s-subst} \frac{P \vdash p_i : \text{sig}_P(x_i) \quad (1 \leq i \leq n)}{P \vdash [x_1 \mapsto p_1, \dots, x_n \mapsto p_n] \text{ safe}} \\
\text{s-rec} \frac{P \vdash p \downarrow q \quad P \vdash q.m_i : S_i \quad (1 \leq i \leq n)}{P \vdash p : \{m_1 : S_1 \dots m_n : S_n\}}
\end{array}$$

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

$$\begin{array}{c}
\text{d-dot} \frac{P \vdash_{\text{t}} rp \downarrow rp \quad P \vdash_{\text{t}} rp.m \not\downarrow}{P \vdash_{\text{t}} rp.m \downarrow rp.m} \\
\text{d-app} \frac{P \vdash_{\text{t}} rp \downarrow rp \quad P \vdash_{\text{t}} p : \{\} \quad P \vdash_{\text{t}} rp(p) \not\downarrow}{P \vdash_{\text{t}} rp(p) \downarrow rp(p)}
\end{array}$$

Figure 9. Safety for termination

6.2. SAFE PROGRAMS

We define safe programs in terms of safe reduction, safety for a signature and safe substitutions. We also introduce a variation, safety for termination, which allows dangling paths.

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 $P \vdash \theta \text{ safe}$ 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.

Definition 5. A path p reduces safely for termination to q with respect to a program P when $P \vdash_{\text{t}} p \downarrow q$ is derivable, \vdash_{t} being the inference system obtained by adding the rules **d-dot** and **d-app** of Figure 9 to the rules in Figure 8. A path p is safe for termination for

the signature S with respect to P when $P \vdash_{\mathfrak{t}} p : S$ is derivable, it is just *safe for termination* if $P \vdash_{\mathfrak{t}} p : \{\}$. A substitution θ is *safe for termination* with respect to P when $P \vdash_{\mathfrak{t}} \theta$ safe is derivable.

Intuitively p is safe with respect to P when p is expanded to a head normal form q , and all the paths that the expansion goes through, including q , are safe. In particular, functor applications are checked to verify that each argument satisfies the signature constraint on the corresponding parameter (via the judgment $P \vdash \theta$ safe), *i.e.* that the argument contains submodules, which must be safe themselves, as stipulated by the access signature of the parameter (via the judgment $P \vdash p : S$). This check is done before each expansion step, and once more on the final path, ensuring that all subpaths in a derivation satisfy their safety requirements.

This inference system embodies our notion of safe programs:

Definition 6. 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 . It is only *safe for termination* if we use the $\vdash_{\mathfrak{t}}$ variant.

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. By inversion of s-rec.

We will now prove the substitution lemma.

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 $P \vdash \theta(p) \downarrow \theta(q)$. Both results also hold for $\vdash_{\mathfrak{t}}$.

Proof. By induction on the structure of the derivation of $P \vdash p : S$. 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 $\text{dom } \theta'$ we have $P \vdash \theta(\theta'(x)) : \text{sig}_P(x)$, 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 $P \vdash \theta(p) \downarrow \theta(q)$, 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.

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 $P \vdash \theta(p.m) : S$.

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**.

The same proof applies to $\vdash_{\mathbf{t}}$: 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 kept by substitution.

We can see the effectiveness of our restriction in the following proposition, which states that if a program is safe for termination, then evaluation of any ground path (*i.e.* containing no variables) terminates. It requires a lemma relating reduction steps and safety derivations.

Lemma 3. 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 $\vdash_{\mathbf{t}}$.

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.

Proposition 1. If P is 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. We first prove that for any signature S used in P , $P \vdash_{\mathfrak{t}} q : S$ can be derived by induction on q . This amounts to proving that there is a path q'' such that $P \vdash_{\mathfrak{t}} q.ap \downarrow q''$, for any access path ap in S . We prove it by induction on $q' = q.ap$.

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

Next we show that the evaluation of q terminates by induction on the size of the derivation of $P \vdash_{\mathfrak{t}} q : \{\}$. This size is finite, and lemma 3 provides the induction step.

6.3. SAFE REWRITING

In this section we provide an alternative definition of safety, in terms of termination of a term rewriting system, and show that the two definitions are equivalent. This is beneficial in two respects. First, it gives a more computational view of what safety means, which may be

more intuitive for many. Second, we will actually use this equivalence when proving the completeness of our decision algorithm, *i.e.* we will prove that when the decision algorithm fails for a program, one can build an infinite reduction in this rewriting system, hence the program is unsafe according to the definition of the previous subsection.

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) $\mathbf{x}.ap.m \rightarrow \mathbf{x}.ap.m$ for any \mathbf{x} , ap , and m , s.t. $ap \in \text{sig}_P(\mathbf{x})$ and $ap.m \notin \text{sig}_P(\mathbf{x})$
- (2) $\mathbf{x}.ap(x) \rightarrow \mathbf{x}.ap(x)$ for any path variable \mathbf{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
- (5) $\text{snd}(x, y) \rightarrow y$

Here \mathbf{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.

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

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 $\text{snd}(\text{chk}(q.ap_1, \dots, q.ap_n), p!(q)).p'$. Here snd and chk are added to evaluation contexts,

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

so that one can reduce any of the $q.ap_i$, checking the safety of q as argument of p . Then one can use rule (5) of $Errors_P$ to discard $\text{chk}(q.ap_1, \dots, \blacksquare$

$q.ap_n$). 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 $Rules_P \cup Errors_P$.

Theorem 1. Definitions 6 and 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 $Rules_P \cup Errors_P$. For a term t of our path rewriting system, containing possibly some `snd` and `chk`, we define the *pure path* of t , noted $pp(t)$, 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 $pa(t)$, 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 $pps(t)$, by induction on the structure of t :

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

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 $pps(t)$. We prove that any reduction step $t \rightarrow t'$ will keep the typability of the paths in $pps(t')$, and decrease this measure according to the multiset ordering. If the reduction is one of the original ones, then according to lemma 3, each path in $pps(t)$ is either reduced into a corresponding path in $pps(t')$, with its derivation size reduced, or it is kept unchanged, and at least one pure path is reduced. If the new rule $p(x) \rightarrow \text{snd}(\text{chk}(x.ap_1, \dots, x.ap_n), p!(x))$ is applied, then the number of pure applications of $p(q)$ is reduced by 1, and we replace this $p(q)$ by $q.ap_1, \dots, q.ap_n$, which were already contained in the derivation of $P \vdash p(q) : \{\}$, so that they are safe, and the measure decreases. If $\text{snd}(x, y) \rightarrow y$ is applied, then we discard the first argument, which leaves $pp(t')$ unchanged, and the measure decreases. Rules (1)-(4) cannot apply, since $pp(t)$ is safe.

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

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

Figure 10. Expressions with integer labels

Conversely, we prove that if there is no infinite reduction using $Rules_P \cup Errors_P$, then P must be safe. We define the relation $p \succ q$ iff p can be rewritten into a path q_0 such that $C[\theta(q)] \in pps(q_0)$ for some path context C and substitution θ . This relation must be antisymmetric 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.

6.4. 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.

The main difficulty is how to detect non-termination of path evaluation, and this is in a complete way. To detect cycles, we label paths appearing inside expressions with integers in the syntax of our calculus, in Figure 10. We assume that a program does not contain duplicate occurrences of the same integer label and write $Labels(P)$ to denote the set of integer labels occurring in the program P . Note that for any program P , $Labels(P)$ is finite.

By preventing the same label from being expanded twice, we can easily define a terminating evaluation algorithm. However, making it complete with respect to our notion of safety is more complex. First, we need to restrict the format of the programs we handle. We explain

on an example. For readability, we omit the ϵ prefix to toplevel modules in examples.

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

The evaluation of n would proceed as follows (keeping labels while we expand abbreviations).

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

While n reduces safely to a , the above reduction shows that we have expanded abbreviation 1 twice. Failing on this example would clearly lose completeness. Fortunately the solution is simple enough: arguments should be expanded independently, so that the labels they generate can be discarded before expanding the path they appear in. In order to do this, arguments need their own location labels. Rather than adding labels inside paths, we choose to put a requirement on programs we are to check: all path arguments should be either variable paths, or lookup-able with an identity substitution. We call programs satisfying this requirement *rewritable*. Here is the rewritable version of the above program.

$$\{f = \lambda(x)x^1 \quad a = \{\} \quad b = f(a)^2 \quad n = f(b)^3\}$$

We allow labels on normal forms to be discarded with special $\#$ -steps. Then evaluation succeeds in a call-by-value strategy (expanding arguments first).

$$n \rightarrow f(b)^3 \rightarrow f(f(a)^2)^3 \rightarrow f((a^1)^2)^3 \xrightarrow{\#} f(a)^3 \rightarrow (a^1)^3 \xrightarrow{\#} a$$

In practice, this new requirement does not restrict the programs we can handle. Any program can be automatically converted to rewritable form, by defining a functor for each path argument, taking all the necessary parameters. To distinguish those argument paths, we put them in a substructure at $\epsilon.arg$. For instance,

$$\{m_1 = \lambda(x_1 : S_1)\{m_2 = \lambda(x_2 : S_2)\epsilon.m_3(\epsilon.m_4(x_1)(x_2))\}\}$$

can be converted to

$$\left\{ \begin{array}{l} m_1 = \lambda(x_1 : S_1)\{m_2 = \lambda(x_2 : S_2)\epsilon.m_3(\epsilon.arg.a_1(x_1)(x_2))\} \\ arg = \{a_1 = \lambda(x_1 : S_1)\lambda(x_2 : S_2)\epsilon.m_4(x_1)(x_2)\} \end{array} \right\}$$

Note that, in order to ensure that the lookup substitution is the identity, we have to reuse the variables of the original path in the generated functor, and as a result the converted program does not satisfy our requirement of distinctness of bound variables. Here, since these reused

variables have the same signatures as original variables and are only substituted by themselves, this introduces no complications.

We will decide safety of programs in this rewritable form, but we still need a further refinement for our algorithm. Due to the presence of variable paths, we may not be interested in the argument itself, but in one of its subpaths (by subpath we mean the argument suffixed by an access path). Here is a concrete example.

$$\{ f = \lambda(x : \{m : \{\}\})\{m = x.m^1\} \\ a = \{m = \{\}\} \quad b = f(a)^2 \quad n = (f(b).m)^3 \}$$

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

$$n \rightarrow f(b).m^3 \rightarrow f(f(a)^2).m^3 \xrightarrow{\#} f(f(a)).m^3 \rightarrow (f(a).m^1)^3 \rightarrow ((a.m^1)^1)^3$$

While n safely reduces to $a.m$, if we look at the labels we see that again we have expanded twice the abbreviation 1. So if this were the only information we use, we would have to fail in this case, losing completeness. The trick of evaluating arguments first did not work here, because $f(a)$ is a normal form. However, here we are not interested in $f(a)$, but in $f(a).m$. Since we know that submodules of x can only be accessed according to its signature, we might transform the above program as follows.

$$\{ f = \lambda(x)\lambda(x_m).\{m = (x_m)^1\} \quad a = \{m = \{\}\} \\ b = f(a)(a.m)^2 \quad b_m = (b.m)^3 \quad n = (f(b)(b_m).m)^4 \}$$

Then evaluation would succeed.

$$n \rightarrow f(b)(b_m)^4 \xrightarrow{*} \xrightarrow{\#} f(f(a)(a.m))(f(a)(a.m).m^3)^4 \\ \xrightarrow{*} f(f(a)(a.m))((a.m^1)^3).m^4 \\ \xrightarrow{\#} f(f(a)(a.m))(a.m).m^4 \rightarrow (a.m^1)^4 \xrightarrow{\#} a.m$$

The transformation suggested in this example would change the arity of functors, which might create problems. A more practical approach is to require that all arguments point to *access structures*. An access structure is a structure filled with paths according to an access signature.

$$acc_struc(p, \{\}) = \{it = p\} \\ acc_struc(p, \{m_1 : S_1\} \uplus S) = \\ \{m_1 = acc_struc(p.m_1, S_1)\} \uplus acc_struc(p, S)$$

Inside each submodule of the access structure, a special member it gives the path accessing the corresponding submodule of the original

argument. In order to provide locations for all subpaths, we will replace the abbreviation associated to an argument with an access structure for this abbreviation. Since there is no way to know the required access signature for an argument before normalizing the path leading to it, we shall build access structures containing all the access paths of all signatures in a program.

For instance, assuming $S_1 = \{m_1 : \{\}\}$ and $S_2 = \{m_2 : \{\}\}$, our second example would now become:

$$\left\{ \begin{array}{l} m_1 = \lambda(x_1 : S_1)\{m_2 = \lambda(x_2 : S_2)\epsilon.m_3(\epsilon.arg.a_1(x_1)(x_2))\} \\ arg = \{a_1 = \lambda(x_1 : S_1)\lambda(x_2 : S_2)\{it = \epsilon.m_4(x_1)(x_2) \\ \qquad m_1 = \{it = \epsilon.m_4(x_1)(x_2).m_1\} \\ \qquad m_2 = \{it = \epsilon.m_4(x_1)(x_2).m_2\}\}\} \end{array} \right\}$$

Here $arg.a_1(x_1)(x_2)$ points to the access structure decomposing the argument $\epsilon.m_4(x_1)(x_2)$ according to the access signature $\{m_1 : \{\} m_2 : \{\}\}$, which combines S_1 and S_2 . Here is our third example after transformation (following our convention, b was moved inside arg).

$$\left\{ \begin{array}{l} f = \lambda(x : \{m : \{\}\})\{m = x.m\} \quad a = \{m = \{\}\} \quad n = \epsilon.f(\epsilon.arg.b).m \\ arg = \{b = \{it = \epsilon.f(\epsilon.arg.a_1) \quad m = \{it = \epsilon.f(\epsilon.arg.a_1).m\}\} \\ \quad a_1 = \{it = \epsilon.a \quad \quad \quad m = \{it = \epsilon.a.m\}\} \quad \quad \quad \} \end{array} \right\}$$

Note that the transformed programs cannot be evaluated using our original definitions: access structures introduce the name it , which does not appear in arguments. Our algorithmic version of normalization will insert it automatically where needed, so that evaluation will agree with that of non-transformed programs. Since access structures are built from a maximal signature, they contain meaningless paths, but we will just ignore them (*i.e.* it is sufficient that only used paths are safe).

The following definition formalizes our requirement on path arguments.

Definition 8. A path p is in *access form* if either p is a variable, or $P \vdash p \mapsto (\theta, e)$ with e a full access structure, and all arguments of p are also in access form. It is in *access source form* if additionally $\theta = id$, *i.e.* its arguments are the variables appearing in its source definition. A path p is in *located form* if all its arguments are in access form. It is in *located source form* if it does not start with $\epsilon.arg$, and all its arguments are in access source form. A program P is in located source form if all its abbreviations are in located source form.

We will require P to be in located source form.

The following definition relates a program in located source form with its rewritable form. As we have just seen, the located source form

```

1:  sgnlz( $P, \pi, q$ ) =
2:    match  $q$  with
3:    |  $x \Rightarrow x$ 
4:    |  $\epsilon \Rightarrow \epsilon$ 
5:    |  $q_1.m \Rightarrow \text{expand}(P, \pi, \text{sgnlz}(P, \pi, q_1).m)$ 
6:    |  $q_1(q_2) \Rightarrow \text{expand}(P, \pi, \text{sgnlz}(P, \pi, q_1)(q_2))$ 
7:   $\text{expand}(P, \pi, x.ap) =$ 
8:    if  $ap \in \text{sig}_P(x)$  then  $x.ap$  else error
9:   $\text{expand}(P, \pi, rp) =$ 
10:   let  $(\theta, e) = \text{lookup}(P, rp)$  in
11:   let  $\rho = \text{vp\_subs}(P, \pi, \theta)$  in
12:   match  $e$  with
13:   |  $q^i \Rightarrow$  if  $i \in \pi$  then error else  $\text{subs}(\theta, \rho, \text{sgnlz}(P, \{i\} \cup \pi, q))$ 
14:   |  $\{m_1 = e_1 \dots m_n = e_n\} \Rightarrow rp$ 
15:   |  $\lambda(x)e' \Rightarrow rp$ 
16:   $\text{subs}(\theta, \rho, rp) = \theta(rp)$ 
17:   $\text{subs}(\theta, \rho, vp) = \rho(vp)$ 
18:   $\text{vp\_subs}(P, \pi, id) = id$ 
19:   $\text{vp\_subs}(P, \pi, \theta[x \mapsto p]) =$ 
20:    $\text{sig\_subs}(P, \pi, x, p, \text{sig}_P(x)) \cup \text{vp\_subs}(P, \pi, \theta)$ 
21:   $\text{sig\_subs}(P, \pi, vp, rp, \{\}) = [vp \mapsto \text{sgnlz}(P, \pi, rp.it)]$ 
22:   $\text{sig\_subs}(P, \pi, vp, x.ap, \{\}) =$ 
23:   if  $ap \in \text{sig}_P(x)$  then  $[vp \mapsto x.ap]$  else error
24:   $\text{sig\_subs}(P, \pi, vp, p, \{m_1 : S_1\} \uplus S) =$ 
25:    $\text{sig\_subs}(P, \pi, vp.m_1, p.m_1, S_1) \cup \text{sig\_subs}(P, \pi, vp, p, S)$ 

```

Figure 11. Semi-ground normalization

of a program can be generated from its rewritable form, and vice-versa, so we can see the two forms as different representations for the same program.

Definition 9. We note \overline{P} for the rewritable form of P , a program in located source form. \overline{P} is obtained by replacing any access structure located at path p by the abbreviation located at $p.it$. I.e. if $P \vdash p \mapsto (\theta, e)$ with e a full access structure, and $P \vdash p.it \mapsto (\theta, q^i)$ then $\overline{P} \vdash p \mapsto (\theta, q)$. Paths outside of $\epsilon.arg$ are unchanged.

We define an algorithmic version of path evaluation, called semi-ground normalization in Figure 11. The semi-ground normalization works on programs in located source form, and expects its path argument to be in located form. We use π as a metavariable for sets of integer labels. The semi-ground normalization uses an auxiliary function `lookup`, which is an algorithmic version of the lookup relation defined in Section 4. Precisely,

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

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

Semi-ground normalization implements the idea 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 23 when `expand` unfolds an abbreviation binding. Since the input program is in located source form, we will 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. `vp_subst`(P, π, θ) returns a substitution ρ , which maps a variable path $x.ap$ such that x is in $\text{dom}(\theta)$ and ap is in $\text{sig}_P(x)$ to the result of calling `snlz` on $\theta(x).ap.it$. Note the insertion of *it*. 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 5: if `subs`($\theta, \rho, \text{sgnlz}(P, \{i\} \cup \pi, q)$) is successful then so is `sgnlz`($P, \pi, \theta(q)$) and their results are same. The auxiliary function `subs` performs case analysis on the input path, and applies θ (resp. ρ) when the input is a rooted path (resp. variable path). Note that $\rho(vp)$ is a total function: if vp is not in $\text{dom}(\rho)$, then $\rho(vp) = vp$.

To check the safety of a given program P in located source form, we run $\text{sgnlz}(P, \emptyset, p)$ for the right hand side p of all the abbreviation bindings in P in an arbitrary order. We will see that if $\text{sgnlz}(P, \pi, q)$ fails, then either of P or q is unsafe (Theorem 4), which ensures completeness.

Below we prove correctness, termination, and completeness of the semi-ground normalization. We first need to ensure that access forms are preserved by substitution.

Lemma 4. If p and θ are in access form, then $\theta(p)$ is in access form.

Proof. By induction on the structure of p .

Theorem 2. (correctness) For any program P in located source form, lock π and path p in located form, if $\text{sgnlz}(P, \pi, p) = q$ then $\bar{P} \vdash p \downarrow q$ and q is in located form. Moreover, for any substitution θ in access form, if $\text{vp_subs}(P, \pi, \theta) = \rho$, then $\bar{P} \vdash \theta$ safe, ρ is in located form, and for any $x \in \text{dom}(\theta)$ and $ap \in \text{sig}(x)$, $\bar{P} \vdash \theta(x.ap) \downarrow \rho(x.ap)$.

Proof. We prove both properties by functional induction.

If $p = x$ or $p = \epsilon$, then $\bar{P} \vdash p \downarrow p$.

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 $\bar{P} \vdash p \downarrow p$.

If $p = p_1(p_2)$, we assume $\text{sgnlz}(P, \pi, p_1) = q_1$. By induction hypothesis on p_1 , $\bar{P} \vdash p_1 \downarrow q_1$. We call expand on $q_1(p_2)$. First we call lookup to obtain θ and e . Then we call vp_subs to obtain ρ , so that $\bar{P} \vdash \theta$ safe by induction hypothesis. If e is not a path, we have $\bar{P} \vdash q_1(p_2) \downarrow q_1(p_2)$ by r_src , and $\bar{P} \vdash p_1(p_2) \downarrow q_1(p_2)$ by r_app . If e is a path q' , then it is in located form. By induction hypothesis, we have $\bar{P} \vdash q \downarrow q'$ with $q' = \text{sgnlz}(P, \{i\} \cup \pi, q)$. If q' is a rooted path, by the substitution lemma, we have $\bar{P} \vdash \theta(q) \downarrow \theta(q')$, so that $\bar{P} \vdash q_1(p_2) \downarrow \theta(q')$ by r_exp , and the arguments of $\theta(q')$ are in access form by lemma 4. If q' is a variable path $x.ap$, by inversion we have $ap \in \text{sig}_P(x)$. If $x \notin \text{dom}(\theta)$, by the substitution lemma we have $\bar{P} \vdash \theta(q) \downarrow x.ap (= \rho(x.ap))$ since $x.ap \notin \text{dom}(\rho)$. Otherwise, by induction hypothesis on vp_subs , we have $\bar{P} \vdash \theta(x.ap) \downarrow \rho(x.ap)$, so that $\bar{P} \vdash \theta(q) \downarrow \rho(x.ap)$. In both cases we conclude that $\bar{P} \vdash q_1(p_2) \downarrow \rho(x.ap)$ by r_exp , and $\rho(x.ap)$ is in located form.

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 r_app by r_dot .

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

If p is a variable y , then we just check that $y.ap$ is valid, and return $\rho(x.ap) = y.ap$, so that $\bar{P} \vdash y.ap \downarrow \rho(x.ap)$ by r_vp .

If p is a rooted path, then $\text{sgnlz}(P, \pi, p.ap.it) = q$ for some q , and $\rho = [x.ap \mapsto q]$. Since p points to the top of an access structure, either $p.ap$

is already in weak source form, or it causes a lookup error, which would contradict the existence of q . Since $p.ap$ is in weak source form, unfolding $\text{sgnlz}(P, \pi, p.ap.it)$ results in calling $\text{expand}(P, \pi, p.ap.it)$. Since $p.ap$ points inside an access structure, $P \vdash p.ap.it \mapsto (\theta', e)$ implies that there is path q_0 such that $e = q_0.ap^i$ and $\bar{P} \vdash p \mapsto (\theta', q_0)$. By induction hypothesis, the let $\rho' = \text{vp_subs}(P, \pi, \theta')$ ensures that $\bar{P} \vdash \theta'$ safe, ρ' is in located form, and that for all (appropriate) vp' , $\bar{P} \vdash \theta'(vp') \downarrow \rho'(vp')$. We then call $\text{sgnlz}(P, \{i\} \cup \pi, q_0.ap)$. If the result is a rooted path rp' , then we return $\rho(x.ap) = \theta'(rp')$, $\theta'(rp')$ is in located form, by the substitution lemma $\bar{P} \vdash \theta'(d) \downarrow \theta'(rp')$, and $\bar{P} \vdash p.ap \downarrow \rho(x.ap)$ by $r\text{-exp}$. If it is a variable path vp' , then $\rho(x.ap) = \rho'(vp')$, and either the root variable of vp' is in the domain of θ' , $\rho'(vp')$ is in located form, and $\bar{P} \vdash \theta'(vp') \downarrow \rho'(vp')$ applies, or it is not, vp' is in located form (as any variable path), and $\rho'(vp') = vp' = \theta'(vp')$, so that $\bar{P} \vdash \theta'(vp') \downarrow \rho'(vp')$ is true again, and $\bar{P} \vdash p.ap \downarrow \rho(x.ap)$ by $r\text{-exp}$ in both cases.

Theorem 3. (termination) For any program P in located source form, lock π and path p in located form, $\text{sgnlz}(P, \pi, p)$ is terminating.

Proof. Termination is guaranteed since any recursive call $\text{sgnlz}(P, \pi, p)$ 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_π on locks and \prec_p on paths are respectively defined as follows.

- $\pi_1 \prec_\pi \pi_2$ if $\pi_2 \subset \pi_1$.
- $p_1 \prec_p p_2$ if p_2 is either $p_1.m$ for some field name m , or $p_1(p)$ for some path p .

Technically, calls in sig_subs do not satisfy this order, but by theorem 2 all path arguments are in access form, so $rp.it$ points inside an access structure, and expand will be called immediately, causing the locking of a new location.

Since $\text{Labels}(P)$ is finite, both \prec_π and \prec_p are obviously well-founded, thus its lexicographic combination \prec is well-founded too.

The following lemmas are used by the completeness proof ⁴. The first one proves that postponing substitution is correct.

Lemma 5. If P is in located source form, p is a rooted path in located form, θ is in access form, $\text{sgnlz}(P, \pi', p) = q$ and $\text{vp_subs}(P, \pi, \theta) = \rho$ with $\pi \subset \pi'$, then $\text{sgnlz}(P, \pi, \theta(p)) = \text{subs}(\theta, \rho, q)$.

Proof. By functional induction on $\text{sgnlz}(P, \pi', p)$. P and π are constant throughout the proof. By hypothesis, p cannot be a variable. If $p = \epsilon$, then $\theta(\epsilon) = \epsilon$, and we conclude.

⁴ Note to the reviewers: we included all the proofs to make checking their validity easier, but we believe that the proofs of these three lemmas could be safely omitted in the final version.

If $p = p_1(p_2)$, we must have $\text{sgnlz}(P, \pi, \theta(p_1)) = q_1$. If q_1 is a variable path, then $\text{expand}(P, \pi, q_1(p_2))$ fails and we have a contradiction. So q_1 is a rooted path, and by induction hypothesis $\text{sgnlz}(P, \pi, \theta(p_1)) = \theta(q_1)$. Calling $\text{expand}(P, \pi, q_1(p_2))$, we have $P \vdash q_1(p_2) \mapsto (\theta', e)$, so that $P \vdash \theta(q_1(p_2)) \mapsto (\theta(\theta'), e)$. Then $\text{vp_subs}(P, \pi, \theta') = \rho'$. We pose $\rho'' = \text{vp_subs}(P, \pi, \theta(\theta'))$, and we need to prove that ρ'' is correctly defined, *i.e.* that for each $x \in \text{dom}(\theta')$, and each $ap \in \text{sig}_P(x)$, we have $\rho''(x.ap) = \text{subs}(\theta, \rho, \rho'(x.ap))$. If $\theta'(x)$ is a rooted path, then

$$\begin{aligned} \rho''(x.ap) &= \text{sgnlz}(P, \pi, \theta(\theta'(x).ap.it)) \\ &= \text{subs}(\theta, \rho, \text{sgnlz}(P, \pi, \theta'(x).ap.it)) \\ &= \text{subs}(\theta, \rho, \rho'(x.ap)) \end{aligned}$$

the second step using the induction hypothesis. If $\theta'(x)$ is a variable path $x'.ap'$, then $\rho'(x.ap) = x'.ap'.ap$. If $\theta(x') = x''.ap''$ (including $\theta(x') = x'$ if $x' \notin \text{dom}(\theta)$), then

$$\begin{aligned} \rho''(x.ap) &= x''.ap''.ap'.ap = \rho(x'.ap'.ap) \\ &= \text{subs}(\theta, \rho, x'.ap'.ap) = \text{subs}(\theta, \rho, \rho'(x.ap)) \end{aligned}$$

where $ap'.ap \in \text{sig}_P(x')$ and $ap''.ap'.ap \in \text{sig}_P(x'')$ since we have $\bar{P} \vdash \theta$ safe and $\bar{P} \vdash \theta'$ safe by the correctness theorem. If $\theta(x')$ is a rooted path, then

$$\begin{aligned} \rho''(x.ap) &= \text{sgnlz}(P, \pi, \theta(x').ap'.ap.it) = \text{vp_subs}(P, \pi, \theta)(x'.ap'.ap) \\ &= \rho(x'.ap'.ap) = \text{subs}(\theta, \rho, x'.ap'.ap) = \text{subs}(\theta, \rho, \rho'(x.ap)) \end{aligned}$$

where $ap'.ap \in \text{sig}_P(x')$ since $\bar{P} \vdash \theta$ safe. Thus ρ'' is correctly defined. Last we check the path returned by expand . If e is not a path, we return $\theta(q_1(p_2)) = \text{subs}(\theta, \rho, q_1(p_2))$. If e is a path q_0^i , we return

$$\begin{aligned} \text{sgnlz}(P, \pi, \theta(q_1(p_2))) &= \text{subs}(\theta(\theta'), \rho'', \text{sgnlz}(P, \{i\} \cup \pi, q_0)) \\ &= \text{sgnlz}(P, \pi, \theta(\theta'(q_0))) \\ &= \text{subs}(\theta, \rho, \text{sgnlz}(P, \pi, \theta'(q_0))) \\ &= \text{subs}(\theta, \rho, \text{subs}(\theta', \rho', \text{sgnlz}(P, \{i\} \cup \pi, q_0))) \\ &= \text{subs}(\theta, \rho, \text{sgnlz}(P, \pi, q_1(p_2))) \end{aligned}$$

by repeated uses of the induction hypothesis.

If $p = p_1.m$, then we reason as for $p = p_1(p_2)$, except that q_1 may be a variable path $x.ap$, so that $\text{sgnlz}(P, \pi, \theta(p_1)) = \rho(x.ap)$. In that case, we must have $ap.m \in \text{sig}_P(x)$. If $\theta(x) = x'.ap'$, then $ap'.ap.m \in \text{sig}_P(x')$ since $\bar{P} \vdash \theta$ safe, and expansion succeeds returning $\rho(x.ap.m)$. If $\theta(x)$ is a rooted path, then we are actually computing $\text{expand}(P, \pi, \theta(x).ap.m)$. $\theta(x)$ must point to the top of an access structure. Since $ap.m \in \text{sig}_P(x)$, m cannot be *it*, and lookup necessarily returns a functor or a structure. As a result, $\text{sgnlz}(P, \pi, \theta(x).ap.m) = \theta(x).ap.m = \rho(x.ap.m)$.

Lemma 6. (idempotence) If P and p are in located source form, and $\text{sgnlz}(P, \pi, p) = q$, then $\text{sgnlz}(P, \pi, q) = q$.

Proof. By functional induction. If p is a variable or ϵ , then $\text{sgnlz}(P, \pi, p) = p$, and the property stands.

If $p = p_1(p_2)$, then we first compute $\text{sgnlz}(P, \pi, p_1) = q_1$, and we have $\text{sgnlz}(P, \pi, q_1) = q_1$ by induction hypothesis. We then call $\text{expand}(P, \pi, q_1(p_2))$. Lookup succeeds, and we obtain θ and e such that $P \vdash q_1(p_2) \mapsto (\theta, e)$. We also obtain ρ by vp_subs , and by case analysis on vp_subs using our induction hypothesis, we obtain that for any $vp \in \text{dom}(\rho)$, $\text{sgnlz}(P, \pi, \rho(vp)) = \rho(vp)$. If e is not a path, then we just return $q_1(p_2)$, and we have $\text{sgnlz}(P, \pi, q_1(p_2)) = \text{expand}(P, \pi, \text{sgnlz}(P, \pi, q_1)(p_2)) = \text{expand}(P, \pi, q_1(p_2)) = q_1(p_2)$. If e is a path q_e^i , then we compute $\text{sgnlz}(P, \{i\} \cup \pi, q_e) = q_0$. By induction hypothesis $\text{sgnlz}(P, \{i\} \cup \pi, q_0) = q_0$. If q_0 is a rooted path, using lemma 5, we obtain $\text{sgnlz}(P, \pi, \theta(q_0)) = \text{subs}(\theta, \rho, q_0) = \theta(q_0) = q$. If q_0 is a variable path, then we return $q = \rho(q_0)$, and we have $\text{sgnlz}(P, \pi, \rho(q_0)) = \rho(q_0)$.

The case $p = p_1.m$ is similar.

Lemma 7. If P and p are in located source form, during the evaluation of $\text{sgnlz}(P, \pi, p)$, sig_subs may only fail on arguments that are in access source form.

Proof. By functional induction. Since p is in located source form, its arguments are in access source form.

Case $p = p_1.m$: We first call $\text{sgnlz}(P, \pi, p_1)$ with p_1 in located source form. If this fails, we conclude by induction hypothesis. If this succeeds, by lemma 6 we can normalize the result q_1 again, obtaining its arguments in θ_1 , and $\text{vp_subs}(P, \pi, \theta_1)$ will succeed. Since the arguments of $p_1.m$ are the same as p_1 , $\theta = \theta_1$. Finally either d is not a path, and normalization succeeds, or we call $\text{sgnlz}(P, \{i\} \cup \pi, q)$, with q a path in located source form, and we conclude by induction hypothesis.

Case $p = p_1(p_2)$: Similar to $p_1.m$, except that $\theta = \theta_1[x \mapsto p_2]$ for some x . Since p_2 is in access source form, the property still stands if $\text{sig_subs}(P, \pi, x, p_2, \text{sig}_P(x))$ fails. But we must also verify it for recursive calls to $\text{sgnlz}(P, \pi, p_2.ap.it)$ when p_2 is a rooted path. Since p_2 is in access form, we have $\overline{P} \vdash p_2 \mapsto (id, q_2)$, and this will result in a call to $\text{sgnlz}(P, \pi, q_2.ap)$, with $q_2.ap$ is located source form, and we conclude by induction hypothesis.

Theorem 4. [completeness] If P and p are in located source form, \overline{P} is safe, and $\overline{P} \vdash p \downarrow q$, then $\text{sgnlz}(P, \emptyset, p) \neq \text{error}$.

Proof. By theorem 1, $\overline{P} \vdash p \downarrow q$ implies that there is no infinite reduction from p using $\text{Rules}_{\overline{P}} \cup \text{Errors}_{\overline{P}}$.

We show by functional induction that $\text{sgnlz}(P, \pi, p) = \text{error}$ (or $\text{sgnlz}(P, \pi, p.it) = \text{error}$ if p points inside an access structure) implies the existence of such an infinite reduction starting from either p or an abbreviation of \overline{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 $q_i^* \xrightarrow{*} C_i'[q_i]$ (where q_i^* is a path of \overline{P}) and $q_i \xrightarrow{*} C_i[\theta_i(p)]$. In case of repeated attempt 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 \overline{P} , we also keep a reduction $q_0 \xrightarrow{*} C_0[p]$, q_0 being either the original p , or the last abbreviation of \overline{P} we are currently normalizing. This last reduction will be used either for errors (producing an infinite reduction with rules of $Errors_{\overline{P}}$), or to produce the reduction $q'_i \xrightarrow{*} C'_i[q_i]$ needed when we normalize q_i for the first time. 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 $p = \epsilon$, then p does not point to an access structure, and $\text{sgnlz}(P, \pi, p) = p \neq \text{error}$.

If $p = p_1.m$ or $p = p_1(p_2)$, or p points inside an access structure and $p_1 = p$, then we first try to evaluate $\text{sgnlz}(P, \pi, p_1)$. 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 $\text{sgnlz}(P, \pi, p_1) = p'_1$, then we call $\text{expand}(P, \pi, p')$, with p' either $p'_1.m$ or $p'_1(p_2)$, or $\text{expand}(P, \pi, p'_1.it)$ with $p' = p'_1$, and since by correctness we have a reduction $p_1 \xrightarrow{*} p'_1$, we have also 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 $ap \notin \text{sig}_P(x)$, and we can use a reduction (1) of $Errors_{\overline{P}}$ 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 $\text{lookup}(P, p')$ (or $\text{lookup}(P, p'.it)$). This cannot fail for $p'.it$, as such a path may only come from the call in `sig_subs` on line 21, where p' was built from a path p_0 pointing to the top of access structure, and an access path ap in $\text{sig}_P(x)$, and access structures are supposed to contain all access paths in all the signatures of P . So if this fails, we are in the former case, and we can use either a reduction (3) or (4) of $Errors_{\overline{P}}$ to create an infinite reduction. Otherwise, we obtain θ and e .

Then we call $\text{vp_subs}(P, \pi, \theta)$. 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 $p_j(p'_j) \rightarrow \text{snd}(\text{chk}(p'_j.ap_{j1}, \dots, p_2.ap_{jn_j}, p_j!(p'_j)))$ inside the reduction leading to p' , allowing us to check all the required subpaths of the argument p'_j . This is also what $\text{sig_subs}(P, \pi, x_j, p'_j, \text{sig}_P(x_j))$ does. If p'_j is a variable path $x.ap$, then it checks for each $ap' \in \text{sig}_P(x_j)$ whether $ap.ap' \in \text{sig}_P(x)$. 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 $ap' \in \text{sig}_P(x_j)$, we call $\text{sgnlz}(P, \pi, p'_j.ap'.it)$. 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 $\text{vp_subs}(P, \pi, \theta)$ 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 $p' = \theta(q_i)$ (both for normal and access structure paths), and we have a reduction $q_i \xrightarrow{*} C_i[\theta_i(p')] = C_i[\theta_i(\theta(q_i))]$, so that we can build an infinite reduction by repeatedly appending it to the reduction from $q_0 \xrightarrow{*} C'_i[q_i]$, i.e. $q_0 \xrightarrow{*} C'_i[q_i] \xrightarrow{*} C'_i[C_i[\theta_i(\theta(q_i))]] \xrightarrow{*} C'_i[C_i[\theta_i(\theta(C_i[\theta_i(\theta(q_i))])]]] \xrightarrow{*} \dots$

If $i \notin \pi$, we call $\text{sgnlz}(P, \{i\} \cup \pi, q)$, which must fail. First we need to extend our reductions. If p' points inside an access structure, then we are currently processing a path generated by sig_subs , on line 21. There is a decomposition $p' = p_0.ap_0$, with p_0 pointing to the top of this access structure, so that $\bar{P} \vdash p_0 \mapsto (\theta, e_0)$, with $e = e_0.ap_0$. If p_0 is in access source form, then $\theta = \text{id}$, and we have a reduction $p_0 \xrightarrow{*} e_0$, which gives $p' = p_0.ap_0 \xrightarrow{*} e_0.ap_0 = e$, so that we can update our various reductions: $q_j \xrightarrow{*} C_j[\theta_j(e)]$ ($j \in \pi$) and $q_0 \xrightarrow{*} C_0[e]$. Since $e = (q_i)^i$ (same i 's), we also add new reductions $q'_i = q_0 \xrightarrow{*} C_0[q_i]$ and $q_i \xrightarrow{*} q_i$. If p_0 is not in access source form, by lemma 7 the call to sig_subs must succeed, so the normalization of e cannot fail, and we do not need to provide updated reductions here. If p' does not point inside an access structure, then we directly have $\bar{P} \vdash p \mapsto (\theta, e)$, and $p' \xrightarrow{*} \theta(e)$. We can combine this reduction with our other reductions to obtain $q_j \xrightarrow{*} C_j[\theta_j(\theta(e))]$ ($j \in \pi$). Since q_i is an abbreviation of \bar{P} , we update q_0 to q_i , with reductions $q'_i = q_i \xrightarrow{*} q_i$, $q_i \xrightarrow{*} q_i$, and $q_0 = q_i \xrightarrow{*} q_i$. So by induction hypothesis, in case of error we have an infinite reduction starting from one of the q'_i or q_0 .

Combining correctness and completeness, sgnlz provides a decidable check for the safety of programs: for each abbreviation (binding) of \bar{P} , we check in turn if its right hand side is normalizable by running sgnlz . If all the abbreviations in \bar{P} are normalizable then \bar{P} is safe, otherwise \bar{P} is unsafe. Since sgnlz will end up normalizing dependent abbreviations repeatedly, we can greatly improve the performance of this process by caching each normalized abbreviation.

6.5. TOWARDS A PRACTICAL LANGUAGE

As we mentioned before, 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 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 absolute paths, starting from the root ϵ . This provides a simple approach to expressing recursion, but in a practical language recursive references should be supported through recursion variables, as in Moscow ML (Russo, 2000) and in our previous

work (Nakata and Garrigue, 2006). The use of recursion variables is crucial for 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 signature of the whole program. From a technical point of view, the switch from the single root ϵ to multiple recursion variables is a trivial change. However, with a single root and the convention on bound variables (*i.e.* their names must be distinct), we were able to formalize and prove Lemma 2 (substitution) in Coq⁵, and this in a reasonably short time. Our experience with proofs for lambda-calculi suggests that a syntax with binders would have required much more efforts to prove it.

Our technical machinery to recover decidability should not impose any burden on the programmer. In Section 6.4 we require that programs should be in *access source form*, but also have given transformations for converting any program to access source form. Hence this does not incur a restriction on the programs we are able to check. We expect a practical language to require usual signatures on functor arguments, as this is already the case for ML programs; note however that we do not require annotations on recursion variables. Then, a direct way to apply our algorithm would be: (1) through preprocessing, extract the access signatures of each argument, (2) convert the program first to *rewritable form*, then to *located source form*, (3) finally apply the semi-ground normalization to check safety. Once the safety check is done, path resolution is guaranteed to succeed, so that we can apply a type checking algorithm without concern. From the programmer's point of view, the only restrictions would be that functors must be first-order, and that signatures on functor arguments should be explicit enough that we can extract access signatures from them.

In real programs, not all modules need to be recursive. So we need to 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

Paths are present in many programming languages and as soon as a programming language supports type path abbreviations, some form of path resolution must be performed during type checking. In a language

⁵ The proof script is available at <http://www.math.nagoya-u.ac.jp/~garrigue/papers/path-subst-coq.zip>.

with a weaker module system, however, path resolution is trivially decidable: we have shown in the paper that it only becomes undecidable when all of nesting, abstraction and recursion are present. As a consequence the problem of the decidability of path resolution has only been rarely acknowledged and the related works are few. Below we overview approaches to path resolution and cyclic type definitions, in previous works on module systems having all these three features.

The most closely related work is (Cremet et al., 2006), which formalized a minimal core calculus for the Scala programming language (Programming Methods Laboratory, EPFL, 2007) and gave a decidable type checking for this calculus. A key concept of this calculus is path dependent types, which are also present in Scala and its theoretical foundation νObj (Odersky et al., 2003). Since they also have a concept of type path abbreviations, and abstract type members in classes which can be instantiated in a way similar to functor application, they end up with a path resolution similar to ours, which they need to solve statically. They do not impose a syntactic restriction on paths or 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 6.4. Here is the encoding of our example in Scala:

```
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 2.7.7), but here is the result of accessing `n1.res`.

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

It would be interesting to see whether our algorithm can help improve this situation.

We have also previously studied the question of path resolution in the context of an object system with type parameters and type members (Nakata et al., 2005). This gave us our first insights in the problem of path resolution, and how to rule out cycles while keeping expressiveness.

OCaml supports a recursive module extension atop a ML module system with applicative functors (Leroy, 2003), which is our starting point. We have discussed in Section 2 deficiencies of OCaml's applicative functors: the type system cannot deduce some desired type equalities and may diverge while resolving path references. Our previous work (Nakata and Garrigue, 2006) addressed these deficiencies by enriching the signature language with module abbreviations, allowing

for strengthened module path equality, and by developing a decidable path resolution. To achieve decidability in (Nakata and Garrigue, 2006) we prohibited any access to submodules of functor parameters. In this paper we lifted this restriction, which is a major improvement.

With the exception of OCaml, previous works on recursive module extensions of the ML module system consider generative functors (Crary et al., 1999; Russo, 2001; Dreyer, 2005; Dreyer, 2007b). The calculus of (Crary et al., 1999) permits equi-recursive type constructors of higher kind, accepting type cycles which we reject. For instance, $\{m_1 = \{m_2 = \epsilon.m_1.m_2\}\}$, which we reject as a illegal type cycle, is permitted in their calculus. In this respect their calculus is stronger than ours. At the same time, it is not known if equality for equi-recursive type constructors at higher kinds is decidable. The other works rule out the potential of cyclic type definitions more conservatively than ours, making path resolution trivially terminating. In Russo’s system (Russo, 2001) forward references must appear only under datatype definitions, and may not be used with type abbreviations. Dreyer (Dreyer, 2005; Dreyer, 2007b) requires that the internal definitions of abstract types in a sealed module and the type components of the argument module in functor application shall not depend on type variables bound as *undefined* in the typing context. A type variable is said to be undefined, roughly, its definition is not yet known to the type checker. These restrictions are motivated so that type cycles will not be present even if all uses of opaque sealing are stripped away. In (Dreyer, 2007b) Dreyer argues that the more permissive approach taken in OCaml and our previous work, namely accepting type cycles when the cycles are broken by opaque sealing, cannot handle the double vision problem, *i.e.* that external and internal views of modules become incoherent. We disagree about it. Indeed OCaml 3.11.2 keeps external and internal views coherent, and we believe OCaml’s solution is applicable to our system too.

Some techniques underlying the semi-ground normalization were inspired by ground term rewriting (Dauchet and Tison, 1990), yet we are not aware of any related work on this precise topic in the term rewriting community. On the one hand, we believe that the way we apply these techniques to the problem of path resolution is new. On the other hand, we hope that there are stronger techniques, in the areas of ground or non-ground term rewriting, applicable to our problem. Exploring these possibilities is one direction for future work.

8. Conclusion

In this paper we have examined the problem of resolving path references, motivated by recent works on recursive extensions to the ML module system. We have 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* (Nakata and Garrigue, 2006).

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 for 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 at integrating 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.

References

- Boudol, G.: 2004, ‘The Recursive Record Semantics of Objects Revisited’. *Journal of Functional Programming* **14**, 263–315.
- Cardelli, L.: 1997, ‘Program Fragments, Linking, and Modularization’. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 266–277.
- Crary, K., R. Harper, and S. Puri: 1999, ‘What is a Recursive Module?’. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 50–63.
- Cremet, V., F. Garillot, S. Lenglet, and M. Odersky: 2006, ‘A Core Calculus for Scala Type Checking’. In: *Int. Symposium on Mathematical Foundations of Computer Science*.
- Dauchet, M. and S. Tison: 1990, ‘The Theory of Ground Rewrite Systems is Decidable’. In: *IEEE Symposium on Logic in Computer Science*.
- Dreyer, D.: 2004, ‘A Type System for Well-Founded Recursion’. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
- Dreyer, D.: 2005, ‘Recursive Type Generativity’. In: *ACM SIGPLAN International Conference on Functional Programming*.
- Dreyer, D.: 2007a. Post to the Caml Mailing List, <http://caml.inria.fr/pub/ml-archives/caml-list/2007/03/73e1ea81e35002046fdce6f14c1d8848.en.html>.

- Dreyer, D.: 2007b, ‘A Type System for Recursive Modules’. In: *ACM SIGPLAN International Conference on Functional Programming*.
- Dreyer, D., K. Crary, and R. Harper: 2003, ‘A type system for higher-order modules’. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 236–249.
- Harper, R., J. C. Mitchell, and E. Moggi: 1990, ‘Higher-Order Modules and the Phase Distinction’. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 341–354.
- Hirschowitz, T. and X. Leroy: 2002, ‘Mixin Modules in a Call-by-value Setting’. In: *European Symposium on Programming*, Vol. 2305 of *Springer LNCS*. pp. 6–20.
- Hopcroft, J., R. Motwani, and J. Ullman: 2001, *Introduction to Automata Theory, Languages, and Computation*, Chapt. 8. Addison-Wesley.
- Leroy, X.: 1995, ‘Applicative Functors and Fully Transparent Higher-order Modules’. In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 142–153, ACM Press.
- Leroy, X.: 1996, ‘A Syntactic Theory of Type Generativity and Sharing’. *Journal of Functional Programming* **6**(5), 667–698.
- Leroy, X.: 2000, ‘A Modular Module System’. *Journal of Functional Programming* **10**(3), 269–303.
- Leroy, X.: 2003, ‘A proposal for recursive modules in Objective Caml’. Available at http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf.
- Leroy, X., D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon: 2008, ‘The Objective Caml system, release 3.11’. Software and documentation available on the Web, <http://caml.inria.fr/>.
- Milner, R., M. Tofte, R. Harper, and D. MacQueen: 1997a, *The Definition of Standard ML - Revised*. The MIT Press.
- Milner, R., M. Tofte, R. Harper, and D. MacQueen: 1997b, *The Definition of Standard ML - Revised*. The MIT Press.
- Nakata, K.: 2005. OCaml bug report. <http://caml.inria.fr/mantis/view.php?id=3674>.
- Nakata, K. and J. Garrigue: 2006, ‘Recursive Modules for Programming’. In: *ACM SIGPLAN International Conference on Functional Programming*. ACM Press.
- Nakata, K., A. Ito, and J. Garrigue: 2005, ‘Recursive Object-Oriented Modules’. In: *Int. Workshop on Foundations of Object-Oriented Languages*.
- Odersky, M., V. Cremet, C. Röckl, and M. Zenger: 2003, ‘A Nominal Theory of Objects with Dependent Types’. In: *European Conference on Object-Oriented Programming*.
- Programming Methods Laboratory, EPFL: 2007, ‘The Scala Programming Language’. Software and documentation available on the Web, <http://www.scala-lang.org/>.
- Russo, C.: 2000, ‘First-class Structures for Standard ML’. In: *European Symposium on Programming*, Vol. 1782 of *Springer LNCS*.
- Russo, C.: 2001, ‘Recursive Structures for Standard ML’. In: *ACM SIGPLAN International Conference on Functional Programming*. pp. 50–61, ACM Press.
- Russo, C. V.: 1999, ‘Non-dependent Types for Standard ML Modules’. In: *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, Vol. 1702 of *Springer LNCS*. pp. 80–97.

