

Path Resolution for Nested Recursive Modules

Keiko Nakata
*CNAM**

Jacques Garrigue
Graduate School of Mathematics, Nagoya University†

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 definitely adds the expressivity of the module system. However it also brings out non-trivial problems that non-recursive modules do not have.

In this paper, we address one of such problems, namely resolution of path references. Paths are a mechanism of ML to refer to modules. Without recursion, well-typedness guarantees termination of path resolution; in other words we can statically determine the module that a path refers to. This will not hold with a recursive modules extension, since the module system then can encode the 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 still undecidable only with nested structures and access to sub-modules of functor arguments but without higher-order functors through an encoding of the Turing machine. Motivated by this result and by technique in ground term rewriting, we develop a terminating algorithm for path resolution by restricting ourselves to first-order functors which do not access sub-modules of arguments.

Keywords:

1. Introduction

Modularity is an important factor in the smooth development and maintenance of large programs. Many modern programming languages have mechanisms to support program structuring. Among such mechanisms, the ML module system is well-known (Milner et al., 1997; Leroy, 2000). Two of its important features are nested structures and functors. Firstly ML modules are nestable, that is, a module can contain sub-modules, as well as type and value definitions; indeed nesting is a simple but powerful way to organize program codes and namespace hierarchically. Secondly ML support functions on modules, so-called functors, which facilitate code reuse in a modular way.

Despite this flexibility, ML prohibits recursion between modules. In other words, recursive type or function definitions may not cross mod-

* 292 rue Saint-Martin F-75141 Paris Cédex 03, France

† Chikusa-ku, NAGOYA 464-8602, Japan

ule 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 could hinder extensible program development (Nakata and Garrigue, 2006).

Introducing recursive modules is a natural way out of this predicament. As recursion is a powerful language construct, its addition definitely 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 the decidable type checking or the error-free module initialization. Indeed extensions with recursion of the module system poses non-trivial problems and much work has been devoted to investigate such extensions that are not too constrained but retain desirable safety guarantees (Crary et al., 1999; Russo, 2001; Nakata and Garrigue, 2006; Boudol, 2004; Hirschowitz and Leroy, 2002; Dreyer, 2004).

One such problem that a recursive modules extension raises is resolution of path references. Paths, also known as qualified identifiers, are a mechanism of ML to refer to modules. In the absence of recursive modules, type checking can guarantee termination of path resolution, that is, we are certain that the reference of a path is not dangling and that the runtime can find the module that the path refers to without diverging. Technically this is same as that well-typedness guarantees strong normalization of lambda calculus. However the property does not hold anymore with a recursive modules extension, since the extension allows the module language to encode lambda calculus with recursion, whose termination is undecidable regardless of well-typedness. The possibility of divergence at the module language level is unfortunate, since the module language is not for performing computation but for organizing program codes; non-terminating path resolution can be described that program execution diverges in search of program codes to execute.

In this paper, we examine and address this problem of path resolution. We formalize path resolution by defining a rewrite system on paths (Section 2 and 3). Then we prove that termination of path resolution is still undecidable only with first-order functors and access to sub-modules of functor arguments but without higher-order functors, through an encoding of the Turing machine (Section 4). The result is interesting since it attests to the expressivity of nesting structure, which is a distinguishing feature of modules but is often paid less attention. The result also justifies a restriction on sub-module access of functor arguments, not only on higher-order functors, for designing path resolution algorithm. Indeed, in Section 5 we present a terminating

<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 1. Syntax

algorithm for path resolution by restricting functors not to take functors as arguments or access sub-modules of parameters.

As well as the two technical results, namely proof of the undecidability and the path resolution algorithm, identifying the problem in a rewriting context is a contribution of the paper. In particular, we hope that our work serves as a first-step towards developing a more sophisticated algorithm for path resolution by benefiting from well-investigated technique in term rewriting theory.

2. Syntax

In Figure 1 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 definitions, or a record of expressions e_i labeled with field 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) 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 definition ordering. Thus paths introduce recursion to the calculus. We may call a definition $m = p$ in a structure an *abbreviation definition*.

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 that the toplevel structure contains a special field named `main`, where evaluation of the program starts.

We assume the following two conventions: 1) any sequence of definitions in a structure does not bind the same field 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 subject of the paper, namely path resolution. We use the same vocabulary with the ML module system as a reminiscence of the correspondence to it.

3. Semantics

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

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

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

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 alias 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 a 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 a 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 to the above example, we extract the following four rewrite rules from it, by collecting abbreviation definitions:

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

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

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

which reflects the previous informal explanation of path resolution, describing every steps of the program evaluation.

3.1. TERMINOLOGY

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

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

Figure 2. Path rewrite rules of a program

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

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

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

$$\theta(\epsilon) = \epsilon \qquad \theta(x) = \begin{cases} x & \text{when } x \notin dom(\theta) \\ p & \text{when } x \in dom(\theta) \text{ and } \theta(x) = p \end{cases}$$

$$\theta(p.m) = \theta(p).m \qquad \theta(p_1(p_2)) = \theta(p_1)(\theta(p_2))$$

Path contexts, ranged over by $C[]$, are define 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' 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 when there is no q such that $p \rightarrow_R q$. A path p is a normal form of q with respect to R when $p \xrightarrow{*}_R q$ and q is in normal form with respect to R . A path rewrite system R is *well-founded* if there is no infinite sequence $\{p_i\}_{i=1}^{\infty}$ such that, for all i in $1, 2, \dots$, $p_i \rightarrow_R p_{i+1}$.

3.2. PROGRAM EVALUATION

Evaluation of a program P consists of two phases. First the path $\epsilon.\text{main}$ 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

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

Figure 3. Lookup

form by making certain that it does not contain dangling references. Below we formalize these steps in turn.

3.2.1. Path rewrite systems of programs

In Figure 2, we define a function *Rules* for building the corresponding path rewrite system to a program. The functionality of *Rules* is straightforward; it traverses the toplevel structure of a program collecting abbreviation definitions, as we have informally described above. The first argument, a path, keeps track of the location of the second argument, an expression which *Rules* is currently examining. The path is extended with *m* when *Rules* taking 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 definition $m = p$ 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 . Note that owing to the convention mentioned in Section 2, $Rules_P$ does not contain overlapping rules for any P , thus the rewriting is deterministic. This is natural as we are considering a deterministic programming language ML, but is seen as a rewriting system; the absence of overlap makes the problem much tractable.

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 $Rules_P$.

3.2.2. 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 $\epsilon.m_2.n_4$ or $\epsilon.m_1.n_2.n$; the former is dangling and the latter need 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 Fig 3. The judgment $P \vdash p \mapsto (\theta, e)$ means that, with respect to the program P , the path p refers to the expression e , where variables x appearing e are bound to $\theta(x)$. The inference rules are

as expected. 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 exist.

Finally in terms of the lookup relation we define the source form. Intuitively 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. Either p is a variable or there exist a substitution θ and an expression e other than a path such that $P \vdash p \mapsto (\theta, e)$ hold.
2. For any q in $\text{args}(p)$, 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 .

The follow lemma states that the source form is invariant of substitution; this lemma is useful in Section 5.

Definition 3. A path p is in source form with respect to a program P if, for all x in $\text{dom}(\theta)$, $\theta(x)$ is in source form with respect to P .

Lemma 1. For any program P , path p and substitution θ , if p and θ are in source forms with respect to P , then $\theta(p)$ is in source form with respect to P .

Proof. By induction on the structure of p .

We conclude this section with a formal definition of program evaluation.

Definition 4. A program P evaluates into p if $\epsilon.\text{main}$ rewrites into a normal form p and p is in source form with respect to P .

4. Undecidability

We are interested in a safe evaluation strategy of programs. In other words, we want to evaluate a given program P in a terminating way so that the evaluation returns a path p if P evaluates into p and signals

<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 4. A first-order fragment

an error when either the evaluation is diverging or the path $\epsilon.main$ to be resolved is dangling.

Having both higher-order functors and recursion, whether or not the evaluation is terminating is undecidable in general; the problem amounts to determining the termination of normalization of 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 sub-module access, but without higher-order functors.

Our proof is by encoding any Turing machine into a first-order fragment of the calculus. To preclude the potential use of higher-order functors during path rewriting, it is enough to ensure the following two conditions.

1. The path rewrite system of a program P does not yield paths of the forms $x(p)$ or $x.m(p)$. Thus variables or their sub-modules cannot be applied.
2. Only the toplevel structure can define functors λxe , where e must not a functor. Thus functors cannot return functors.

We enforce these conditions by confining ourselves to a fragment of the calculus defined in Figure 4. 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. Any structure other than the toplevel can only contains abbreviation definitions. In particular, it does not contain functor definitions.

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. In particular, the arguments of $\delta(q, a)$ are a state q and a tape symbol a . 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.

The rest of the section presents our encoding in the following three steps.

1. Firstly we explain how to construct a set of rewrite rules R_M from any Turing machine M .
2. Then, we show that R_M indeed 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 enable us to elongate the tape, moving the edge by adding a blank symbol to the left or right on demand. Finally, the rules from the last set allow commutation between state and tape symbol. 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 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 P whose path rewrite system $Rules_P$ is R_M . Firstly 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 $\{q_1, \dots, q_n\} = Q$.

This construction, together with our experience in examining the problem, has posed one question: is the path resolution problem still undecidable if we put a restriction on access to sub-sub-modules of functor parameters while allowing access to sub-modules? In other words, we have an open problem of whether the path resolution problem is undecidable if we consider the same fragmentary calculus where paths are defined by the following syntax:

$$\begin{array}{ll} \textit{Pre paths} & pp ::= \epsilon \mid \epsilon.m(p) \mid pp.m \\ \textit{Paths} & p ::= pp \mid x \mid x.m \end{array}$$

The restriction may look strange from the theoretical point of view, but can be acceptable from the programmer point of view; indeed our experience is that many programmers do not often use higher-order functors or deeply nested modules. Finding the exact restriction which lets the problem decidable is one direction for future work.

5. Terminating path resolution

On the one hand, the undecidability result of the previous section justifies a restriction on access to sub-modules of functor parameters to resolve path references. On the other hand, we can observe that path resolution is decidable for programs without functors. Indeed if a program P does not contain functors at all, every path rewrite rule in $Rules_P$ is of the form $\epsilon.m_{11}.m_{12}.\cdots.m_{1i} \rightarrow \epsilon.m_{21}.m_{22}.\cdots.m_{2j}$. Then the problem of path resolution is reduced to the problem of head-reduction of a string rewrite system, whose termination is known to be decidable (Dauchet and Tison, 1990).

In this section, we present a terminating path resolution algorithm, named *semi-ground normalization*, whose design is motivated by the above observation.

5.1. SEMI-GROUND NORMALIZATION

In Fig. 5, we define the semi-ground normalization, on which we will expand in the rest of this section.

Firstly in Fig. 6, we elaborate the syntax of the calculus by labeling expressions with integers. 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

```

1:  sgnlz( $P, \pi, p$ ) =
2:    match  $p$  with
3:    |  $x \Rightarrow x$  |  $\epsilon \Rightarrow \epsilon$ 
4:    |  $p_1.m \Rightarrow$ 
5:      let  $p'_1 = \text{sgnlz}(P, \pi, p_1)$  in
6:      let  $(\theta, i, e) = \text{lookup}(P, p'_1.m)$  in
7:      match  $e$  with
8:      |  $\{m_1 = e_1 \cdots m_n = e_n\} \Rightarrow p'_1.m$  |  $\lambda(x)e' \Rightarrow p'_1.m$ 
9:      |  $q \Rightarrow$ 
10:       if  $i \in \pi$  then error else let  $q_2 = \text{sgnlz}(P, \{i\} \cup \pi, q)$  in  $\theta(q_2)$ 
11:    |  $p_1(p_2) =>$ 
12:      let  $p'_1 = \text{sgnlz}(P, \pi, p_1)$  in
13:      let  $p'_2 = \text{sgnlz}(P, \pi, p_2)$  in
14:      let  $(\theta, i, e) = \text{lookup}(P, p'_1(p'_2))$  in
15:      match  $e$  with
16:      |  $\{m_1 = e_1 \cdots m_n = e_n\} \Rightarrow p'_1(p'_2)$  |  $\lambda(x)e' \Rightarrow p'_1(p'_2)$ 
17:      |  $q \Rightarrow$ 
18:       if  $i \in \pi$  then error else let  $q_2 = \text{sgnlz}(P, \{i\} \cup \pi, q)$  in  $\theta(q_2)$ 

```

Figure 5. Semi-ground normalization

Expression $e ::= d^i$
Expression descriptions $d ::= \{m_1 = e_1 \cdots m_n = e_n\} \mid \lambda(x)e \mid p$

Figure 6. Expressions with integer labels

for any program P , $\text{Labels}(P)$ is finite and we will exploit this finiteness to keep the semi-ground normalization terminating.

The semi-ground normalization uses an auxiliary function `lookup`. The function `lookup` is an algorithmic version of the lookup relation introduced in Section 3 and takes account of integer labels. Precisely,

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

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

Taking a broad view, the semi-ground normalization traces abbreviation definitions until it encounters a structure or a functor to resolve path references. It may signal an `error` when paths contain cyclic or dangling references or may return paths not in source forms when it

needs to rewrite paths in a non-ground way, that is, when it needs to apply substitution to rewrite further.

Below we highlight the two key factors about the semi-ground normalization.

Semi-groundness The algorithm is semi-ground in the sense that it does not rewrite paths after applying substitution to variables. See lines 9 to 10 and 17 to 18, where it applies substitution hence does not recursively call `sgnlz` on the result of the substitution. In this way we avoid difficulties in ensuring termination of non-ground term rewriting, while retaining the first-order functionality of functors. At the same time, the semi-groundness manifests the incompleteness of the algorithm with respect to the higher-order functionality of functors; the algorithm cannot handle functors which takes functors as arguments or which access sub-modules of arguments. One such example is

$$\{\mathbf{m}_1 = \lambda(\mathbf{x})\mathbf{x}.\mathbf{m}_2 \quad \mathbf{m}_2 = \epsilon.\mathbf{m}_1 \quad \mathbf{main} = \epsilon.\mathbf{m}_1(\epsilon)\}$$

The algorithm cannot successfully rewrite the path $\epsilon.\mathbf{main}$ into $\epsilon.\mathbf{m}_1$, since such rewriting interleaves substitution as illustrated below:

$$\epsilon.\mathbf{main} \rightarrow \epsilon.\mathbf{m}_1(\epsilon) \rightarrow [\mathbf{x} \mapsto \epsilon]\mathbf{x}.\mathbf{m}_2 \rightarrow \epsilon.\mathbf{m}_2 \rightarrow \epsilon.\mathbf{m}_1$$

The idea behind the semi-groundness is that the algorithm only rewrites paths that appear in the source program or that are contained in the input path to the algorithm. When the algorithm is successful, both the substitution θ and the argument q_2 to which θ is applied in lines 10 and 18 are in source forms and Lemma ?? ensures that the resulting path $\theta(q_2)$ is in source form too.

Eager normalization With our restrictive handling of functors, the only possible scenario where the semi-ground normalization could diverge is to trace the same module abbreviation definition infinitely often. Such scenario is boiled down to the following two cases:

$$\{\mathbf{m}_1 = \epsilon.\mathbf{m}_2 \quad \mathbf{m}_2 = \epsilon.\mathbf{m}_2.\mathbf{m}_3 \quad \mathbf{main} = \epsilon.\mathbf{m}_2\}$$

and

$$\{\mathbf{m}_4 = \lambda(\mathbf{x})\mathbf{x} \quad \mathbf{m}_5 = \epsilon.\mathbf{m}_4(\epsilon.\mathbf{m}_5) \quad \mathbf{main} = \epsilon.\mathbf{m}_5\}$$

The former induces an infinite rewriting:

$$\begin{aligned} \epsilon.\mathbf{main} &\rightarrow \epsilon.\mathbf{m}_1 \rightarrow \epsilon.\mathbf{m}_2 \rightarrow \epsilon.\mathbf{m}_2.\mathbf{m}_3 \rightarrow \epsilon.\mathbf{m}_2.\mathbf{m}_3.\mathbf{m}_3 \rightarrow \\ &\epsilon.\mathbf{m}_2.\mathbf{m}_3.\mathbf{m}_3.\mathbf{m}_3 \rightarrow \dots \end{aligned}$$

and the latter does:

$$\epsilon.\mathbf{main} \rightarrow \epsilon.\mathbf{m}_5 \rightarrow \epsilon.\mathbf{m}_4(\epsilon.\mathbf{m}_5) \rightarrow \epsilon.\mathbf{m}_5 \rightarrow \epsilon.\mathbf{m}_4(\epsilon.\mathbf{m}_5) \rightarrow \dots$$

Fortunately these are not difficult to detect; rewriting the right-hand sides of the abbreviation definitions $m_2 = \epsilon.m_2.m_3$ in the former and $m_5 = \epsilon.m_4(\epsilon.m_5)$ in the latter cyclically come back to the same definitions respectively, which clearly causes non-termination thus we must avoid.

Rewriting might encounter a cyclic abbreviation definition on the way of rewriting another path. For instance in the former example, we start with the path $\epsilon.main$ then encounter the cyclic abbreviation definition $m_2 = \epsilon.m_2.m_3$. Hence we want to check the absence of cycles on demand; this is where the eagerness is involved (Lines 9&10 and 17&18). On encountering an abbreviation definition $m = q^i$ during rewriting a path p , the algorithm leaves off p and takes up q for rewriting, while remembering the integer label i to avoid re-taking up the same abbreviation definition cyclically. When the algorithm successfully finds a normal form q_2 of q then it returns $\theta(q_2)$ as a result of the semi-ground normalization of p_2 . As we already explained above, $\theta(q_1)$ is already in source form when successful.

The above two key factors being on our mind, the semi-ground normalization is easily explained. It takes three arguments, a program P , a lock π , which is a subset of $Labels(P)$, and a path p to be resolved with respect to P .

Variables x and the root path ϵ are immediately returned (line 3). When the input path is of the form $p_1.m$, the semi-ground normalization first resolves the prefix p_1 . If the prefix is successfully resolved into p'_1 , then `lookup` is consulted for the expression that p'_1 refers to. Now there are two cases to consider. When p'_1 refers to either a structure or a functor (line 8), then the semi-ground normalization returns $\theta(p'_1).m$, which cannot be rewritten further. When p'_1 refers to a path q , then q is resolved without applying the substitution θ . To avoid non-termination due to the potential cyclic abbreviation definitions, the recursive call to `sgnlz` augments the lock π with the integer label i of q ; in particular, when i is already recorded in π , `error` is signaled and the semi-ground normalization gives up the resolution. If q is successfully resolved into q_2 , then $\theta(q_2)$ is returned as the result of semi-ground normalizing $p_1.m$. The case where the input path is of the form $p_1(p_2)$ is similar.

Below we prove termination and correctness of the semi-ground normalization.

Lemma 2. For any program P , lock π with $\pi \subset Labels(P)$ and path p , `sgnlz`(P, π, 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 paths and locks, 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 $\pi_1 \subset \pi_2$.
- $p_1 \prec_p p_2$ if p_2 is either $p_1.m$ for some field name m , or else $p'_2(p_1)$ or $p'_2(p_1)$ for some path p'_2 .

Both \prec_π and \prec_p are obviously well-founded, thus its lexicographic combination \prec is well-founded too.

As a corollary of the lemma, we obtain the following theorem.

Theorem 1. (Termination) For any program P , $\text{sgnlz}(P, \emptyset, \epsilon.\text{main})$ is terminating.

As we mentioned above, the semi-ground normalization will not return source forms when it need to rewrite paths in non-ground way; in such cases it returns an intermediate result of rewriting. The following lemma clarifies this fact.

Lemma 3. For any program P , lock π with $\pi \subset \text{Labels}(P)$ and path p , if $\text{sgnlz}(P, \pi, p) = q$ then $p \xrightarrow{*}_{\text{Rules}_P} q$.

Proof. By induction on \prec .

The correctness statement checks that the result of the semi-ground normalization is indeed in source form; we recall that we can algorithmically decide it. The following theorem is obtained as a corollary to Lemma 2.

Theorem 2. (correctness) For any program P , if $\text{sgnlz}(P, \emptyset, \epsilon.\text{main}) = p$ and p is in source form with respect to P , then P evaluates into p .

6. Related work

To the best of our knowledge, little work has been done in the programming language community to address the problem of path resolution. We are motivated to consider the problem on the way to design a recursive module extension of the ML module system; recursive modules

are intensively studied recently and we guess this is one reason why the problem is not well-examined so far.

In a recursive modules context, terminating path resolution is potentially useful for both type checking and runtime evaluation. Our previous work (Nakata and Garrigue, 2006) proposed a type system for a recursive module extension of the ML module system with applicative functors. We needed to statically resolve path references to keep type checking decidable and to support type inference for recursively defined modules, and used a similar path resolution algorithm to the one to be presented in the paper. How to evaluate recursively defined modules is still a moot and unsettled question. In particular, the classical eager evaluation strategy of ML modules does not easily fit recursion, posing difficult problems to be addressed (Hirschowitz and Leroy, 2002; Dreyer, 2004). Introducing a lazy evaluation strategy to recursive modules is another design possibility (Nakata, 2008); this is essentially equivalent to interpret modules as nestable records with lazy fields. Introduction of laziness would increase flexibility in using recursive modules compared to the classical eager evaluation strategy. At the same time, the flexibility would come at the cost of statically ensured runtime safety due to the potential existence of cyclically defined modules, which will be only detected at runtime. The presence of path resolution algorithm will ease the cost, statically detecting, thus rejecting, those cyclic definitions.

A similar difficulty of resolving path references arises in the context of sophisticated object systems, like ν Obj (Odersky et al., 2003), the theoretical underpinning of the Scala programming language (Programming Methods Laboratory, EPFL, 2007). To statically ensure the absence of cyclic inheritance between classes, one needs to resolve path references. Having generic classes with type members, ν Obj is at least as expressive as recursive modules as far as path references are concerned; indeed this is one reason that the type checking of ν Obj is undecidable. V. Crement et al. formalized decidable type checking for a fragment of ν Obj (?). ν Obj supports abstraction via abstract members, whereas ML does via functors. This difference makes it less obvious to compare their algorithm to ours. Our understanding is that their algorithm is non-ground but not eager in terms of our terminology used in Section 5. Thus two algorithms should have distinct weakness and strength; future work include a more thorough comparison, which would be useful to design a clearer algorithm.

We have also examined an object system with generics and type members, and admitted the need of path resolution for the decidability of type checking (Nakata et al., 2005).

Whilst the technique we use in the paper are inspired by ground term rewriting, we are not aware of related work in the term rewriting community. On the one hand we believe that the way we adopt the technique to this problem of path resolution is new. On the other hand, we hope that there are better ways to address the problem by exploiting the well-developed technique in both ground and non-ground term rewriting. This is a direction for the future work.

7. Conclusion

In this paper we have examined the problem of resolving path references, motivated by recent work on extensions with recursion of the ML module system. We have formalized the problem by defining a rewrite system on paths and proved that the problem is undecidable only with first-order functors and sub-module access to functor arguments, but without higher-order functors. This result and technique in ground term rewriting led us to design a terminating path resolution algorithm by restricting its availability to first-order functors that do not access sub-modules of arguments.

References

- Boudol, G.: 2004, ‘The recursive record semantics of objects revisited’. *Journal of Functional Programming* **14**, 263–315.
- Crary, K., R. Harper, and S. Puri: 1999, ‘What is a Recursive Module?’. In: *Proc. PLDI*. pp. 50–63.
- Dauchet, M. and S. Tison: 1990, ‘The theory of ground rewrite systems is decidable’. In: *Proc. LICS’90*.
- Dreyer, D.: 2004, ‘A Type System for Well-Founded Recursion’. In: *Proc. POPL*.
- Hirschowitz, T. and X. Leroy: 2002, ‘Mixin modules in a call-by-value setting’. In: *Proc. ESOP*, Vol. 2305. pp. 6–20, Springer.
- Hopcroft, J., R. Motwani, and J. Ullman: 2001, *Introduction to Automata Theory, Languages, and Computation*, Chapt. 8. Addison-Wesley.
- Leroy, X.: 2000, ‘A modular module system’. *Journal of Functional Programming* **10**(3), 269–303.
- Milner, R., M. Tofte, R. Harper, and D. MacQueen: 1997, *The Definition of Standard ML - Revised*. The MIT Press.
- Nakata, K.: 2008, ‘Frozen Modules: A lazy evaluation strategy for more recursive initialization patterns’. a draft paper available at <http://pauillac.inria.fr/~nakata/>.
- Nakata, K. and J. Garrigue: 2006, ‘Recursive Modules for Programming’. In: *Proc. ICFP*. ACM Press.
- Nakata, K., A. Ito, and J. Garrigue: 2005, ‘Recursive Object-Oriented Modules’. In: *Proc. FOOL’05*.

- Odersky, M., V. Cremet, C. Röckl, and M. Zenger: 2003, 'A Nominal Theory of Objects with Dependent Types'. In: *Proc. ECOOP'03*.
- Programming Methods Laboratory, EPFL: 2007, 'The Scala Programming Language'. Software and documentation available on the Web, <http://www.scala-lang.org/>.
- Russo, C.: 2001, 'Recursive Structures for Standard ML'. In: *Proc. ICFP'01*. pp. 50–61, ACM Press.