Path resolution for recursive nested modules is undecidable

Keiko Nakata[1],[3] and Jacques Garrigue[2]

[1] Kyoto University
[2] Nagoya University
[3] Cedric

Abstract. The ML module system supports 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. To keep type normalization terminating in such an extension, thus to keep type checking decidable, path references must be resolved in a terminating way. Here paths are a mechanism to refer to components of modules. In this paper, we show that the termination of path resolution is undecidable for a ML-like module system with recursive modules and first-order applicative functors, by encoding any Turing machine. This demonstrates the need for some restriction.

1 Introduction

The ML module system provides strong support for the modular development of programs [7, 5]. A programmer can decompose a large program hierarchically using *nested structures*. *Functors*, which are functions over modules, ease code reuse. Moreover, the programmer can control abstraction of programs with *signatures*, which represent types of modules. To increase its flexibility, much work has been devoted to extending the module system with recursion, which is currently prohibited in ML [1, 8, 3, 6].

In our previous work [6], we have proposed a type system for a module system extended with recursion and fully applicative functors. We had to be careful about the potential existence of cyclic type definitions. In a programming language with recursive modules and applicative functors, a programmer might carelessly write pathologically cyclic type definitions, for which the naïve implementation of type normalization can diverge and thus for which type checking can diverge. To keep type checking decidable, we designed a terminating type normalization by requiring functors not to take functors as arguments or to access sub-modules of arguments.

The first restriction sounds reasonable: type normalization in the presence of higher-order applicative functors and recursive modules amounts to normalization of a lambda calculus with recursion, which is clearly undecidable. While it may seem too strong, the second restriction is also critical; we prove in this paper that termination of type normalization is still undecidable only with first-order functors and sub-module access. Our proof works by encoding any Turing machine into a small calculus featuring *paths*, where paths are a mechanism to refer to modules components. To normalize types in ML, we need to resolve path references (i.e. to find the module that the path refers to). Undecidability of path resolution hence implies that of type normalization.

The result of this paper is important for us since it justifies the need for a restriction on nested arguments. Moreover, the encoding itself exposes the underlying nature of type normalization, which will be useful to find a more relaxed restriction, hence to make our type system more flexible.

This work is initially motivated by a desire to define a decidable type system for recursive modules. Yet the problem we consider is general; we examine termination of an untyped tiny calculus with recursion and labeled records. We also believe that our work is potentially useful for guaranteeing safe evaluation of recursive modules, where we want to ensure the absence of cyclic aliases between modules.

2 Syntax and Semantics

Below, we define a calculus for our formal study, where m and x are metavariables for field names and variables, respectively.

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

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, that is, a record of expressions e_i labeled with field names m_i. A functor $\lambda x.e$ represents a function over expressions; x is the name of the formal parameter and e is the body.

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 "$p.m$", meaning access to the field named m of the structure that p refers to; 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, paths 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. 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 any sequence of definitions in a structure does not bind the same field name twice and that a program does not contain free variables.

For instance, consider the program:

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

The path $\epsilon.\mathtt{m}_1.\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$. Hence, the path $\epsilon.\mathtt{m}_1.\mathtt{n}_2$, which is an alias for $\epsilon.\mathtt{m}_1.\mathtt{n}_1$, refers to the field $\mathtt{n}_1$ of the structure $\mathtt{m}_1$, too. A path can contain functor applications. For instance, the path $\epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1$ refers to the field $\mathtt{n}_1$ of the body of the functor $\mathtt{m}_2$.

Resolution of path references may require more complex computation. For instance, $\epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2$ resolves to $\epsilon.\mathtt{m}_1.\mathtt{n}_1$; by reducing the functor application, we obtain $\epsilon.\mathtt{m}_1.\mathtt{n}_2$, which resolves to $\epsilon.\mathtt{m}_1.\mathtt{n}_1$, as we have explained above.

2.1 Path rewriting

A program defines a set of rewrite rules on paths. For instance, the previous example gives rewrite rules:

$$\{\quad \epsilon.\mathtt{m}_1.\mathtt{n}_2 \to \epsilon.\mathtt{m}_1.\mathtt{n}_1, \quad \epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_2 \to \mathtt{x}.\mathtt{n}_2,$$
$$\epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_3 \to \epsilon.\mathtt{m}_2(\mathtt{x}).\mathtt{n}_1, \quad \epsilon.\mathtt{m}_3 \to \epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2\ \}$$

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

$$\epsilon.\mathtt{m}_3 \to \epsilon.\mathtt{m}_2(\epsilon.\mathtt{m}_1).\mathtt{n}_2 \to \epsilon.\mathtt{m}_1.\mathtt{n}_2 \to \epsilon.\mathtt{m}_1.\mathtt{n}_1$$

which reflects our previous explanation of path resolution.

We say that a program P is *well-founded* if the rewrite rules that P defines do not induce infinite reduction steps. The reader will find formal definitions in the appendix.

Example 1. The program:

$$\{\mathtt{m}_1 = \epsilon.\mathtt{m}_2.\mathtt{m}_1 \ \ \mathtt{m}_2 = \epsilon.\mathtt{m}_1\}$$

is not well-founded, since it induces the infinite reduction:

$$\epsilon.\mathtt{m}_1 \to \epsilon.\mathtt{m}_2.\mathtt{m}_1 \to \epsilon.\mathtt{m}_1.\mathtt{m}_1 \to \epsilon.\mathtt{m}_2.\mathtt{m}_1.\mathtt{m}_1 \to \cdots$$

Example 2. The program:

$$\{\mathtt{m}_1 = \lambda\mathtt{x}.\mathtt{x} \ \ \mathtt{m}_2 = \epsilon.\mathtt{m}_1(\epsilon.\mathtt{m}_2)\}$$

is not well-founded, since it induces infinite reduction:

$$\epsilon.\mathtt{m}_2 \to \epsilon.\mathtt{m}_1(\epsilon.\mathtt{m}_2) \to \epsilon.\mathtt{m}_2 \to \cdots$$

The keen reader may have noticed that when a program does not contain functors at all, the problem of well-foundedness is reduced to termination of head-reduction of a string rewriting system, which is known to be decidable [2]. Yet for programs with first-order functors, well-foundedness is undecidable, as we shall show in the next section.

3 Translation of the Turing Machine

We encode any Turing machine into a first-order fragment of the calculus, defined by the syntax:

$$\begin{array}{lrl}
\textit{Expressions} & e & ::= \{m_1 = e_1 \cdots m_n = e_n\ \} \mid \lambda x.e \mid p \\
\textit{Paths} & p & ::= \epsilon \mid x \mid \epsilon.m(p) \mid p.m \\
\textit{Toplevel expression} & te & ::= \lambda x.\{m_1 = e_1 \cdots m_n = e_n\ \} \\
\textit{Program} & P & ::= \{m_1 = te_1 \cdots m_n = te_n\}
\end{array}$$

The new syntax is restricted in the following two ways to syntactically preclude higher-order functors. 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. Observe that, under these two restrictions, the rewrite rules of a program cannot yield paths of the forms $x(p)$ or $x.m(p)$.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, b, F)$ be a Turing machine[4], 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 $a_1a_2 \cdots a_{i-1}qa_ia_{i+1} \cdots a_n$ of a Turing machine is represented by a path

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

The special symbol $\hat{b}$ is not contained in Q or Γ. The intuition is that we encode the right hand side of the tape with the dots and the left side with functor applications. The head part $\epsilon.q$ of the path represents the current state. 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.

Given a Turing machine M, we construct a set of rewrite rules 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. $\{\epsilon.a(x).q \rightarrow \epsilon.q(x).a \mid a \in \Gamma, a \in Q\}$

Below we observe 1) that we can construct a program P_M with R_M as the corresponding set of rewrite rules and 2) that the rewrite rules R_M encode the Turing machine M.

It is easy to see the first condition hold by considering that the left-hand side of every rewrite rule in R_M is of the form $\epsilon.q(x).a$ and that a program $\{q = \lambda x.\{a = p\}\}$ has the set $\{\epsilon.q(x).a \rightarrow p\}$ as the corresponding rewrite rule. Note also that R_M does not contain overlapping rules; this is important to avoid a structure containing duplicate definitions for the same name like $\{q = \lambda x.\{a = p_1 \ a = p_2\}\}$, which breaks the syntactic convention we mentioned in Section 2.

For instance, the rules from 5 require the toplevel structure of P_M to contain a definition $a = \lambda x.\{q_1 = \epsilon.q_1(x).a \ \cdots \ q_n = \epsilon.q_n(x).a\}$ when a is a tape symbol of the Turing machine M and $\{q_1, \cdots, q_n\}$ is the set of states.

Let's verify that the second condition holds. The first two sets of rules encode transitions of M. The rules from third and fourth sets allow us to elongate the tape, moving the edge by adding a blank symbol to the left or right on demand.

[4] We borrow the notations and terminology from [4], to which the reader is referred for a complete definition.

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.

For instance, suppose $\delta(q, a_i) = (q', a'_i, L)$; i.e., we have a move

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

Then we can induce the corresponding reduction of paths by rules $\epsilon.q(x).a_i \to x.q'.a'_i$ from 2, and $\epsilon.a_i(x).q' \to \epsilon.q'(x).a_i$ from 5:

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

4 Conclusion

We have shown that termination of path resolution for first-order nested recursive modules is undecidable by an encoding of the Turing machine. While the result justifies a restriction on nested functor arguments, we think that the current restriction that prohibits all accesses to sub-modules of arguments is stronger than necessary.

Since path rewrite rules are derived from programs, they are already restricted: 1) there are no overlapping rules; 2) every rule is left-linear; 3) functor-application positions in the left-hand side of any rewrite rule must be module variables (*e.g.* a path like $\epsilon.m_1(\epsilon.m_2)$ cannot be in the left-hand side, but $\epsilon.m_1(x)$ can). Besides, in ML, functor parameters are explicitly typed, which means that we can statically know the possible nesting-depth of functor arguments.

A direction for future work would be to exploit these properties to find a relaxed restriction, thus making our type system stronger.

References

1. K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. PLDI'99*, pages 50–63, 1999.
2. M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. LICS'90*, 1990.
3. D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, 2005.
4. J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, chapter 8. Addison-Wesley, 2001.
5. X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
6. K. Nakata and J. Garrigue. Recursive Modules for Programming. In *Proc. ICFP'06*. ACM Press, 2006.
7. R.Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
8. C. Russo. Recursive Structures for Standard ML. In *Proc. ICFP'01*, pages 50–61. ACM Press, 2001.

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

Fig. 1. Path rewrite rules of a program

Appendix

A Definitions

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

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

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

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 $p \rightarrow p'$. Let $R = \{p_1 \rightarrow p'_1, \dots, p_n \rightarrow p'_n\}$ be a set of path rewrite rules. A path p *rewrites into* p' with respect to R if there is a substitution σ, a path context $C[]$ and a rewrite rule $p_i \rightarrow p'_i \in R$ such that $p = C[\sigma(p_i)]$ and $p' = C[\sigma(p'_i)]$. We write $p \rightarrow_R p'$ when p rewrites into p' with respect to R.

Definition 1. *A set of path rewrite rules R is well-founded if there is no infinite sequence $\{p_i\}_{i=1}^{\infty}$ such that, for all i in $1, 2, \dots$, $p_i \rightarrow_R p_{i+1}$.*

Fig. 1 defines a function *Rules* for extracting a set of path rewrite rules from a program. The first argument of *Rules*, which is a path, records the location of the second argument, which is an expression, with respect to the toplevel structure. *Rules* traverses a given program and builds a rewrite rule $p.m = p'$ from a definition $m = p'$ with p being the location of the definition.

Definition 2. *A program P is well-founded if $\mathit{Rules}(\epsilon, P)$ is well-founded.*

B Correctness of the encoding

By construction, our encoding of any Turing Machine does not introduce overlapping rules. Here we see correspondences between moves of a Turing machine and reductions in the encoding.
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
$$\begin{array}{ll} & \epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \epsilon.a_{i-1}(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).q'.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \\ \to & \epsilon.q'(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_{i-1}.a'_i.a_{i+1}.\cdots.a_n.\hat{b} \end{array}$$
2. When $i = 1$, then we have a move:
$$q a_1 a_2 \cdots a_n \vdash q' b a'_1 a_2 \cdots a_n$$
We have reductions
$$\begin{array}{ll} & \epsilon.q(\epsilon.\hat{b}(\epsilon)).a_1.a_2\cdots.a_n.\hat{b} \\ \to & \epsilon.\hat{b}(\epsilon).q'.a'_1.a_2\cdots.a_n.\hat{b} \\ \to & \epsilon.q'(\epsilon.\hat{b}(\epsilon)).b.a'_1.a_2\cdots.a_n.\hat{b} \end{array}$$
3. When $i = n$ and $a'_i = b$, then we have a move:
$$a_1 a_2 \cdots a_{n-1} q a_n \vdash a_1 a_2 \cdots a_{n-2} q' a_{n-1}$$
We have reductions
$$\begin{array}{ll} & \epsilon.q(\epsilon.a_{i-1}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).a_n.\hat{b} \\ \to & \epsilon.a_{i-1}(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).q'.b.\hat{b} \\ \to & \epsilon.q'(\epsilon.a_{i-2}(\cdots(\epsilon.a_1(\epsilon.\hat{b}(\epsilon)))\cdots)).b.\hat{b} \end{array}$$

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