Recursion for structured modules

Keiko Nakata
Research Institute for Mathematical Sciences, Kyoto University
keiko@kurims.kyoto-u.ac.jp

Abstract

This paper proposes an extension of the ML module system with recursion that keeps a flexible use of nested structures and functors. For the purpose of formal study, we design a calculus, called *PathCal*, which has a module system with nested structures and simple functors along with recursion. It is important for this calculus to guarantee resolvability of recursive references between modules. We present a decidable type system to ensure this resolvability.

1 Introduction

When building a large software system, it is indispensable to decompose the system into smaller parts and to reuse them in different contexts. Module systems play an important role in facilitating such factoring of programs. Many modern programming languages provide some forms of module systems.

The family of the ML programming languages such as SML and Objective Caml provides a powerful module system [10, 1, 9]. Nested structures of modules allow hierarchical decomposition of programs. Functors can be used to express advanced forms of parameterization, which ease code reuse. Abstraction can be controlled by signatures with transparent, opaque, or translucent types [8]. However, despite the flexibility of the module language, mutual recursion between modules is prohibited as dependencies between modules must accord with the order of definitions.

There has been much work on recursive module extensions of the module system in recent years [3, 2, 6, 5, 14]. Their concern for side-effects imposes a severe restriction on access to inner components of such recursive modules, resulting in a less flexible use of nested structures and functors. There are many situations where their restriction seems unnecessarily strict, in particular when modules have no side-effects.

In this paper, we propose an extension of the module system with lenient recursion. We evaluate effectless modules in a free order to retrieve the flexible use of nested structures and functors in the presence of recursion. In such a system, it is important to guarantee well-definedness of modules, as recursion might introduce cyclic or dangling references that end up with uninitializable modules.

We design a calculus, called *PathCal*, to investigate this problem. *PathCal* has a module system, which supports nested structures and simple functors along with recursion. A key feature of our calculus is a flexible referencing mechanism given by *path*s. Paths can refer to any modules at any levels of nested structures regardless of the order of definitions. Moreover, simple cases of functor applications are allowed in paths, where the functor and its arguments themselves are paths. We expand paths, *i.e.* resolve the references of paths based on a "lazy evaluation mechanism" of the module language. This laziness allows liberal layout of mutually recursive modules into hierarchies, but introduces possibilities of defining ill-defined modules that have unexpandable paths, *i.e.* paths that have cyclic or dangling references. We propose a decidable type system that guarantees the well-definedness of modules, ensuring that the expansion of paths always successfully terminates. The type system also designates a right order for evaluating modules, which is of practical importance.

Interestingly enough, what we discuss in this paper is also important for introducing the ML module system into object-oriented languages. The recursion extension is highly desirable in such systems as mutual recursion is intrinsic to class definitions. Then we meet a similar situation when we try to ensure the absence of cyclic inheritance. Our previous work [11] exploited the usefulness of the combination of a module system and object-oriented mechanisms, and proposed a decidable type system that ensures the absence of cyclic inheritance. This paper also serves as a generalization of our previous work.

The remainder of this paper is organized as follows. In the next section, we present the formal definition for *PathCal*. In Section 3, we give examples of *PathCal*. Section 4 discusses well-definedness of modules. Section 5 and Section 6 present our approach to the well-definedness. A soundness result of our calculus appears in Section 7. In Section 8, we review related

E	::=		(*module expression*)
		`struct` L `end`	(*structure*)
	$\mid$	`functor` $(X)E$	(*functor*)
	$\mid$	p	(*path*)
	$\mid$	X	(*module variable*)
L	::=	$\epsilon \mid DL$	
D	::=	`module` $M = E$	(*module definition*)
	$\mid$	`val` $l = e$	(*term definition*)
p	::=	$\epsilon \mid p.M \mid p(p) \mid p(X)$	(*path*)
e	::=	$(e, e) \mid p.l \mid X.l \mid i$	(*expression*)
i	::=	$1 \mid 2 \mid \ldots$	(*integer*)
P	::=	`struct` L `end`	(*program*)

Figure 1: Syntax

```
struct
  module Even = struct
    val compare = function ...
    val add = function x y −>
        if x% 2 = 0 then add_impl x y
    val add_impl = EvenSet.add
  end
  module EvenSet = MakeSet(Even)
  module MakeSet = functor (X) struct
    val compare = X.compare
    val add = function ...compare ...
    val remove = function ...
  end
end
```

Figure 2: Modules for even numbers

work. Section 9 concludes.

2 Syntax

The syntax for *PathCal* is given in Figure 1.

We reserve M and N for metavariables ranging over module names, l for a metavariable over value names, X for a metavariable over module variables. We let *MNames* and *VNames* be the sets of module names and value names, respectively.

A module expression E is either a *structure*, a *functor*, a *path*, or a module variable. A structure is a sequence of module definitions and term definitions. A functor is a module expression parameterized by a module variable. Functors can be seen as functions over module expressions. A path is a reference to another module. A flexible referencing mechanism given by paths is a key feature of our calculus. Paths can locate any modules at any levels of nested structures by specifying their locations relative to the top-level structure. Moreover, simple cases of functor applications are allowed in paths, where the functor and its arguments themselves are paths. Formally, paths comprise four constructs: ϵ denotes the top-level structure; $p.M$ expresses access to inner module M of the module referred to by p; $p(p)$ and $p(X)$ are functor applications. We usually omit the leading ϵ. when writing paths.

To obtain a decidable type system, we impose a restriction on functor arguments that forbids 1) accessing their inner modules, and 2) applying them to other modules. Hence, we do not include paths of the forms $X(p)$ and $X.M$ in the syntax. This restriction means that our functors are first-order and that we have to pass inner modules as independent parameters for functors instead of passing a module which contains all of them.

An expression is either a pair (e, e), or aliases $p.l$ and $X.l$, which denote value l in the module referred to by p and X respectively, or an integer.

We call a top-level structure a program, and reserve P for a metavariable ranging over programs. "$P \equiv$ `struct` L `end`" means that a program P is a top-level structure defined by `struct` L `end`.

We consider this leanest calculus as a minimal core of the ML module system for theoretical study.

We assume the following two conditions: 1) any sequence of module definitions and term definitions that defines a structure does not contain duplicate definitions for module names and value names; 2) a program does not contain free module variables, and all bound module variables differ from each other.

3 Example

In this section, we overview our calculus concentrating on use of nested structures, functors with recursion.

We start with the example given in Figure 2. To make the example more familiar one, we assume that we have extended our calculus with a construct `function` for function definition.

The program in Figure 2 consists of structures `Even`, `EvenSet`, and a functor `MakeSet`. `Even` and `EvenSet` mutually refer to each other. On the one hand, `Even` contains a function `add_impl`, defined as `EvenSet.add`, which is an alias for the function `add` contained in `EvenSet`. On the other hand, `EvenSet`, which is defined by applying the `MakeSet` functor to `Even`, contains a function `compare` as an alias for the function `compare` in `Even`.

Here we define aliases when using functions defined in other modules. The intension is to make explicit linking of module components required by initializa-

```
struct
  module Number = struct
    val compare = function ...
    module Even = struct
      val compare = Number.compare
      val add = function x y − >
        if x% 2 = 0 then add_impl ...
      val add_impl = NumberSet.EvenSet.add
    end
    module Odd = struct
      val compare = Number.compare
      val add = function x y − >
        if x% 2= 1 then add_impl...
      val add_impl = NumberSet.OddSet.add
    end
  end
  module NumberSet = struct
    module EvenSet = MakeSet(Number.Even)
    module OddSet = MakeSet(Number.Odd)
  end
  module MakeSet = functor (X) struct
    val compare = X.compare
    val add = function ...  compare ...
  end
end
```

Figure 3: Modules for even and odd numbers

tion of recursive modules.

Now, we flesh out the first example with modules `Odd` and `OddSet` as shown in Figure 3. This time, we pack two modules `Even` and `Odd` into `Number`, and `EvenSet` and `OddSet` into `NumberSet`, using nested structures of modules.

Here we used a bit more involved paths. For example, the function `add_impl` in `Even` is defined as `NumberSet.EvenSet.add`, which refers to the function `add` contained in `EvenSet`, which in turn contained in `NumberSet`.

Actually, a program can be thought of a bunch of recursive modules, wherein paths offer a recursive referencing mechanism. It might be helpful getting intuition behind paths to compare our calculus without functors with the Unix file system. Then, paths in *PathCal* correspond to absolute paths, with which you can locate any directories and files in the file system. Functors add to the flexibility of our modules, yet raise some technical difficulties discussed later.

Returning to the example, module `Even` refers to the module `EvenSet`, having a function `add_impl` aliased for `add` in `EvenSet`. At the same time, `EvenSet` refers to this `Even`, having a function `compare` as an alias for `compare` in `Even`. Similarly, modules `Odd` and `OddSet` mutually refer to each other. As such you can have liberal layout of mutually recursive modules into hierarchies, using the referencing mechanism offered by paths.

4 Well-definedness

While we can enjoy liberal recursion along with a flexible use of nested structures and functors, we should be careful not to introduce ill-defined programs.

Continuing to the example in Figure 3, we consider adding functions `plus` into modules `Even` and `Odd`. We might carelessly implement them as follows.

```
module Even = struct
  val plus = Number.Odd.plus
  val compare = ...
  ...
end
module Odd = struct
  val plus = Number.Even.plus
  val compare = ...
  ...
end
```

The function `plus` in `Even` is defined as an alias for the function `plus` in `Odd`, while `plus` in `Odd` is defined as an alias for `plus` in `Even`. As the two aliases make a cycle, they cannot be resolved, meaning that linking of these functions would result in a meaningless cycle.

The motivation of our work is to statically reject such ill-defined programs and to ensure that aliases are always resolved.

To achieve our purpose, we have to be careful about nested structures and functors. We observe in Example 1 and 2 typical cases which give rise to technical difficulties.

Example 1 *The alias* $\mathrm{M}_1.\mathrm{l}$ *cannot be resolved in the following program.*

```
struct
  module M1 = M2.M3
  module M2 = M1
end
```

To resolve $\mathrm{M}_1.\mathrm{l}$*, we first have to know the module referred to by* M_1*. The expansion of the path* M_1 *gives rise to the following infinite sequence.*

$$\mathrm{M}_1 \to \mathrm{M}_2.\mathrm{M}_3 \to \mathrm{M}_1.\mathrm{M}_3 \to \mathrm{M}_2.\mathrm{M}_3.\mathrm{M}_3 \to \ldots$$

To expand M_1*, we first have to expand* $\mathrm{M}_2.\mathrm{M}_3$*, which is the definition of* M_1*. As* M_2 *is an alias for* M_1*, we reduce* $\mathrm{M}_2.\mathrm{M}_3$ *into* $\mathrm{M}_1.\mathrm{M}_3$*. Then we replace* M_1 *with* $\mathrm{M}_2.\mathrm{M}_3$ *again, following the definition of* M_1*. Now it is*

```
struct
 module M1 = struct
  val l11 = M2.l22
  val l12 = 3
 end
 module M2 = struct
  val l21 = M1.l12
  val l22 = 4
 end
end
```

Figure 4: A program with safe mutually recursive modules

obvious that we are wandering into an infinite rewriting procedure.

Example 2 *The alias* $\mathrm{M}_1(\mathrm{M}_2).\mathrm{l}$*, which defines* l *in the module* M_2*, cannot be resolved in the following program.*

```
struct
 module M1 = functor (X) X
 module M2 = struct val l = M1(M2).l end
end
```

To resolve $\mathrm{M}_1(\mathrm{M}_2).\mathrm{l}$*, we first have to expand the path* $\mathrm{M}_1(\mathrm{M}_2)$*. As* M_1 *is the identity functor,* $\mathrm{M}_1(\mathrm{M}_2)$ *expands to* M_2*. Thus,* $\mathrm{M}_1(\mathrm{M}_2).\mathrm{l}$ *refers to* $\mathrm{M}_2.\mathrm{l}$*, which is defined as* $\mathrm{M}_1(\mathrm{M}_2).\mathrm{l}$*, making a cycle.*

We would like to emphasize that, while programs in Example 1 and 2 should be rejected, we do not want to give up mutually recursive modules such as in Figure 4. This program is harmless. Under our lazy evaluation mechanism of modules, we can resolve aliases $\mathrm{M}_2.\mathrm{l}_{22}$ and $\mathrm{M}_1.\mathrm{l}_{12}$ into 4 and 3, respectively.

Our overall approach to ensure the well-definedness consists of the following two steps.

- We first check that a program P has *finite path dependencies* by considering well-foundedness of a binary relation on paths constructed from P.
- Then, the type system checks that the program are well-defined, ensuring resolvability of aliases. The finite path dependencies ensure the decidability of the type checking.

5 Finite path dependency

Let P be a program. We extract a *path dependency relation* from P by conservatively approximating the dependencies between modules.

$$
\begin{array}{lcl}
dp(p, \texttt{struct module } M_1 = E_1 \ldots \texttt{module } M_n = E_n & & \\
\quad \texttt{val } l_1 = e_1 \texttt{ val } l_m = e_m \texttt{ end}) & & \\
\qquad = \bigcup_{i=1}^{n} dp(p.M_i, E_i) & & \\
dp(p, \texttt{functor } (X)E \texttt{ end}) & = & dp(p, E) \\
dp(p, q) & = & \{(p, r) \mid r \in \mathit{flatsSet}(q)\} \\
dp(p, X) & = & \emptyset
\end{array}
$$

Figure 5: Extraction of the base relation

$$
\begin{array}{lcl}
\mathit{flatsSet}(p) & = & \{\mathit{flat}(p)\} \cup \bigcup_{q \in \mathit{args}(p)} \mathit{flatsSet}(q) \\
\mathit{flat}(\epsilon) & = & \epsilon \\
\mathit{flat}(p.M) & = & \mathit{flat}(p).M \\
\mathit{flat}(p(p')) & = & \mathit{flat}(p) \\
\mathit{flat}(p(X)) & = & \mathit{flat}(p) \\
\mathit{args}(\epsilon) & = & \emptyset \\
\mathit{args}(p.M) & = & \mathit{args}(p) \\
\mathit{args}(p(q)) & = & \{q\} \cup \mathit{args}(p) \\
\mathit{args}(p(X)) & = & \mathit{args}(p)
\end{array}
$$

Figure 6: Auxiliary functions

The path dependency relation of P is a binary relation on flat paths, where a flat path is a path containing no functor application. The construction of this relation takes two steps: 1) extract a base relation from P; 2) expand the base relation in order to take into account the dependencies that do not explicitly appear in P.

The base relation of P is extracted by the function dp given in Figure 5 with auxiliary functions in Figure 6. Given a flat path p and a module expression E, dp calculates dependencies assuming that p depends on E. When E is of the form `struct module` $M_1 = E_1 \ldots$ `module` $M_n = E_n$ `val` $l_1 = e_1$ `val` $l_m = e_m$ `end`, dp recursively calculates dependencies assuming that $p.M_i$ depends on E_i. Note that, instead of regarding p as depending on E_i, it employs more precise dependencies. Although this makes the dependencies more complex, it gives more freedom for recursion between modules. When E is of the form `functor` $(X)E$, p depends on E. If E is a path q, dp approximates functor applications in q by making p depend on all flat paths appearing in q. The function *flatsSet* returns the set of flat paths appearing in a path. For example, $\mathit{flatsSet}(\mathrm{M}_1.\mathrm{M}_2(\mathrm{M}_3(\mathrm{M}_4.\mathrm{M}_5)(\mathrm{X})).\mathrm{M}_6) = \{\mathrm{M}_1.\mathrm{M}_2.\mathrm{M}_6,\ \mathrm{M}_3,\ \mathrm{M}_4.\mathrm{M}_5\}$. Finally, if E is a module variable, dp returns the empty set.

Definition 1 *The path dependency relation of a program* P *is the postfix and transitive closure of* $dp(\epsilon, P)$.

$$\frac{P \equiv \texttt{struct}\ L\ \texttt{end}}{\vdash \epsilon \mapsto (\texttt{id}, \texttt{struct}\ L\ \texttt{end})} \qquad \frac{\vdash p \mapsto (\theta, \texttt{struct}\ \ldots, \texttt{module}\ M = E, \ldots \texttt{end})}{\vdash p.M \mapsto (\theta, E)}$$

$$\frac{\vdash p \mapsto (\theta, \texttt{functor}\ (X)E)}{\vdash p(q) \mapsto (\theta[X \mapsto q], E)} \qquad \frac{\vdash p \mapsto (\theta, \texttt{functor}\ (X')E)}{\vdash p(X) \mapsto (\theta[X' \mapsto X], E)}$$

Figure 7: Source form

Definition 2 *Let D be a binary relation on flat paths. The postfix and transitive closure of D, denoted as $\tilde{D}$, is the smallest transitive relation which contains D and meets the postfix condition that if (p, q) is in $\tilde{D}$ and M in MNames, then $(p.M, q.M)$ is also in $\tilde{D}$.*

We call postfix closure of D the smallest relation that contains D and satisfies the postfix condition.

Definition 3 *Let D be a binary relation on flat paths. D is well-founded if and only if D does not contain an infinite descending sequence,* i.e. *there is no infinite sequence $\{p_i\}_{i=1}^{\infty}$ such that, for all natural number i, (p_i, p_{i+1}) is in D.*

Definition 4 *A program P has finite path dependencies if and only if the path dependency relation of P is well-founded.*

Proposition 1 *It is decidable whether a program P has finite path dependencies or not.*

Example 3 *Consider the following program* P.

```
struct
  module M1 = struct
    module M11 = struct ... end
    module M12 = M1.M13.N
    module M13 = M2.M21
  end
  module M2 = struct
    module M21 = struct
      module N = struct ... end
    end
    module M22 = M1.M11
  end
end
```

The base relation of P *is:*
$\{(\mathrm{M_1.M_{12}},\ \mathrm{M_1.M_{13}.N}), (\mathrm{M_1.M_{13}},\ \mathrm{M_2.M_{21}}),$
$(\mathrm{M_2.M_{22}},\ \mathrm{M_1.M_{11}})\}$.
Then the path dependency relation is the postfix closure of the following set:
$\{(\mathrm{M_1.M_{12}},\ \mathrm{M_1.M_{13}.N}), (\mathrm{M_1.M_{13}},\ \mathrm{M_2.M_{21}}),$
$(\mathrm{M_2.M_{22}},\ \mathrm{M_1.M_{11}}), (\mathrm{M_1.M_{13}.N},\ \mathrm{M_2.M_{21}.N}),$
$(\mathrm{M_1.M_{12}},\ \mathrm{M_2.M_{21}.N})\}$.

In the following sections, we fix a program P having finite path dependencies.

```
struct
  module M1 = functor(X1) functor(X2)
   struct
    module M11 = struct val l = X1.l end
    module M12 = X2
   end
  module M2 = struct val l = 2 end
  module M3 = M1(M2)(M2)
end
```

Figure 8: Program $\mathrm{P_1}$

6 Type System

The type system checks that the program P is well-defined by checking resolvability of aliases. It resolves aliases through normalization of paths. Normalization reduces paths into *source forms*.

A path p is of source form if it allows us to look up its definition from P.

Definition 5 *A path p is of source form if and only if $\vdash p \mapsto (\theta, E)$ holds for some θ and some E other than a path.*

The judgment $\vdash p \mapsto (\theta, E)$ is defined in Figure 7, where θ is a metavariable ranging over substitutions of module variables for paths.[1] $\vdash p \mapsto (\theta, E)$ says that p is defined by module expression E with module variables X in E bound to $\theta(X)$.

For example, consider a program $\mathrm{P_1}$ given in Figure 8. $\mathrm{M_1(M_2)(M_2).M_{11}}$ is of source form, since $\vdash \mathrm{M_1(M_2)(M_2).M_{11}} \mapsto ([\mathrm{X_1} \mapsto \mathrm{M_2}; \mathrm{X_2} \mapsto \mathrm{M_2}],\ \texttt{struct val l = X1.l end})$ holds, but $\mathrm{M_3.M_2}$ is not, as there is no θ and E such that $\vdash \mathrm{M_3.M_2} \mapsto (\theta, E)$ holds.

Figure 9 gives an axiom and inference rules for the normalization. The judgment "$\vdash p \triangleright q$" denotes that q is a source form of p.

[1] This judgment (and other judgment which we are to define) should also take the program we are considering as a parameter, but we omit it throughout this paper, supposing a fixed program having finite path dependencies.

[nlz-root]
$\vdash \epsilon \triangleright \epsilon$

[nlz-dot]
$$\frac{\vdash p \triangleright p' \quad \vdash p'.M \mapsto (\theta, E) \quad E \not\equiv q}{\vdash p.M \triangleright p'.M}$$

[nlz-dot-path]
$$\frac{\vdash p \triangleright p' \quad \vdash p'.M \mapsto (\theta, q) \quad \vdash \theta(q) \triangleright r}{\vdash p.M \triangleright r}$$

[nlz-app]
$$\frac{\vdash p_1 \triangleright p_1' \quad \vdash p_1'(p_2) \mapsto (\theta, E) \quad E \not\equiv q}{\vdash p_1(p_2) \triangleright p_1'(p_2)}$$

[nlz-app-path]
$$\frac{\vdash p_1 \triangleright p_1' \quad \vdash p_1'(p_2) \mapsto (\theta, q) \quad \vdash \theta(q) \triangleright r}{\vdash p_1(p_2) \triangleright r}$$

[nlz-vapl]
$$\frac{\vdash p_1 \triangleright p_1' \quad \vdash p_1'(X) \mapsto (\theta, E) \quad E \not\equiv q}{\vdash p_1(X) \triangleright p_1'(X)}$$

[nlz-vapl-path]
$$\frac{\vdash p_1 \triangleright p_1' \quad \vdash p_1'(X) \mapsto (\theta, q) \quad \vdash \theta(q) \triangleright r}{\vdash p_1(X) \triangleright r}$$

Figure 9: Normalization of paths

[T-int]
$\Delta \vdash i$ ok

[T-pair]
$$\frac{\Delta \vdash e_1 \text{ ok} \quad \Delta \vdash e_2 \text{ ok}}{\Delta \vdash (e_1, e_2) \text{ ok}}$$

[T-var]
$$\frac{\vdash p \triangleright p_1 \quad \vdash p_1 \mapsto (\theta, X) \quad \Delta \vdash \theta(X).l \text{ ok}}{\Delta \vdash p.l \text{ ok}}$$

[T-def]
$$\frac{\vdash p \triangleright p_1 \quad \vdash p_1 \mapsto (\theta, \texttt{struct } \ldots, l = e, \ldots \texttt{end}) \quad p_1.l \not\prec \Delta \quad \Delta, p_1.l \vdash \theta(e) \text{ ok}}{\Delta \vdash p.l \text{ ok}}$$

Figure 10: Typing rules

ϵ is of source form([**nlz-root**]). [**nlz-dot**] says that $p'.M$ is a source form of $p.M$, if p' is a source form of p, and the definition of $p'.M$ is not a path. [**nlz-dot-path**] says that r is a source form of $p.M$, if p' is a source form of p, and $p'.M$ is defined by a path q, and r is a source form of $\theta(q)$. Normalization for paths of the forms $p_1(p_2)$ $p(X)$ is defined in the similar way. Following these rules, $\mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{11}$ is a source form of $\mathrm{M}_3.\mathrm{M}_{11}$ in P_1.

Note that we consider $p.M$ to be of source form if the definition of $p.M$ is a module variable. Hence, taking P_1 as an example, the normalization reduces $\mathrm{M}_3.\mathrm{M}_{12}$ into $\mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{12}$, not into M_2.

As mentioned in Section 2, we impose a restriction on functor arguments that forbids accessing their inner modules and applying them to other modules. Our calculus does not explicitly include paths of the forms $X(p)$ and $X.M$, this is not, however, enough to enforce our restriction due to the liberal aliases. For example, in P_1, we would not like to normalize $\mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{12}.\mathrm{N}$ into $\mathrm{M}_2.\mathrm{N}$ even if module N is defined in M_2. For that purpose, we regards $\mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{12}$ as of source form. Since there is no rules that are applicable to $\mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{12}.\mathrm{N}$, we cannot deduce $\vdash \mathrm{M}_1(\mathrm{M}_2)(\mathrm{M}_2).\mathrm{M}_{12}.\mathrm{N} \triangleright \mathrm{M}_2.\mathrm{N}$.

The type system is given in Figure 10. Δ is a finite set of *value access path*s, where a value access path is a path followed by $.l$ for some l in *VNames*. The judgment $\Delta \vdash e$ ok denotes that e is well-defined with a lock on Δ. We use Δ to keep the type system decidable as detailed later.

An integer is well-defined with a lock on any Δ ([**T-int**]). A pair (e_1, e_2) is well-defined with a lock on Δ, if both of e_1 and e_2 are well-defined with a lock on Δ([**T-pair**]). The rule [**T-var**] says that $p.l$ is well-defined, if p_1 is a source form of p, and the definition of p_1 is a module variable X, and $\theta(X).l$ is well-defined with a lock onΔ. This rule takes care of substitution of paths for module variables on behalf of normalization, ensuring that the substitution is required for accessing a value component of a functor argument, not a module component. The rule [**T-def**] says that $p.l$ is well-defined with a lock on Δ, if p_1 is a source form of p, and the definition of p_1 is a structure in which l is defined by an expression e, and $p_1.l$ *is not locked in* Δ, and $\theta(e)$ is well-defined under $\Delta, p_1.l$. We say a value access path $p.l$ is not locked in Δ, denoted $p.l \not\prec \Delta$, if

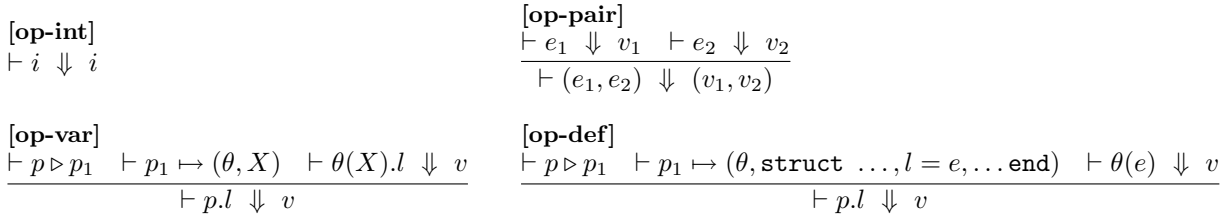

Figure 11: Evaluation of expressions

and only if Δ does not contain value access path $p'.l$ such that $\mathit{flat}(p) = \mathit{flat}(p')$ holds. The intention of this locking is to disable the type system from looking up the exactly same term definition twice during type checking. For example, the type system does not judge M.l well-defined in the following program

```
struct module M = struct val l = M.l end end
```

owing to the locking, while $\mathrm{M.l_1}$ in the following program

```
struct
  module M = struct
    val l1 = M.l2
    val l2 = 2
  end
end
```

is well-defined.

Proposition 2 *If the program P has finite path dependencies, then, for any expression e, it is decidable whether $\vdash e$* ok *holds or not.*

7 Soundness

We introduce an operational semantics for *PathCal* which reduces an expression into a value.

A value v is either an integer or a pair of values.

$$v ::= i \mid (v, v)$$

The judgment $\vdash e \ \Downarrow \ v$ denotes that e is reduced into v. An axiom and inference rules for the judgment are given in Figure 11, which mimic typing rules except that they do not lock value access paths.

The rule [**op-int**] is obvious. A pair (e_1, e_2) is reduced into (v_1, v_2), if e_1 and e_2 are reduced into v_1 and v_2, respectively([**op-pair**]). The rule [**op-var**] says that $p.l$ is reduced into v, if p_1 is a source form of p, and the definition of p_1 is a module variable X, and $\theta(X).l$ is reduced into v. The rule [**op-def**] says that $p.l$ is reduced into v, if p_1 is a source form of p, and the definition of p_1 is a structure in which l is defined by an expression e, and $\theta(e)$ is reduced into v. Following these rules, we can judge $\vdash \mathrm{M_3.M_{11}.l} \Downarrow 2$ and $\vdash \mathrm{M_3.M_{12}.l} \Downarrow 2$ in the program given in Figure 8.

The following proposition says that we can statically ensure resolvability of aliases.

Proposition 3 *If the program P has finite path dependencies and $\vdash e$* ok *holds, then e is reduced into a value,* i.e *we have an algorithm calculating v such that $\vdash e \ \Downarrow \ v$ holds.*

We can give another view that this proposition concerns a safety of initialization of programs. By checking resolvability of all value definitions in a program, we can be sure that linking of aliases does not cause cyclic or dangling links.

8 Related work

Recursive module extensions of the ML module system are investigated in [14, 2, 6, 5]. Boudol [2], Hirschowitz & Leroy [6], and Dreyer [5] also proposed type systems to ensure a safety of the recursion. Their type systems ensure the safety property that recursively defined variables are not dereferenced before their contents are completely evaluated. On the one hand, their type systems accept the following program (suppose we have extended our calculus with some constructs)

```
struct
  module M = functor (X)
    struct
      val f x = ...; X.f (x − 1)
    end
  module N = M(N)
end
```

which we reject, since the path dependency relation of this program contains (N, N), which makes a cycle. Their approach relies on the knowledge that M is *nonstrict*, *i.e.* it does not dereference its argument when applied. We could accept this program, if we can build path dependency relations using this information of

the non-strictness. However, such non-strictness analysis itself is generally hard in our system. It would require normalization of paths, whose termination in turn relies on path dependency relations, while we need the non-strictness analysis to build this relations. On the other hand, the program given in Figure 4 is rejected by their systems, since one cannot complete the evaluation of M_1 nor M_2 without dereferencing itself.

νObj [12] is a calculus for objects and classes. It supports recursion and nesting, and provides an operator, called "merging", which can serve as functors. νObj allows to define modules as fix points of functors, but does not to define modules as in Figure 4.

Objective Caml [7] and Moscow ML [13] are real languages, that support recursion between modules. As their type systems do not guarantee the safety of the recursion, run-time errors might occur due to access to uninitialized values.

In our previous work [11] we designed a calculus, called *Room*, which unifies modules (with nested structures and first-order functors) with classes. *Room* allows recursion between modules and ensures well-definedness of modules by requiring functor arguments to have the exactly same inner modules as their upper type bounds. As types are modules in [11], this requirement means that we have no way to access to inner modules that are specific to actual functor arguments. From the perspective of *PathCal*, *Room* can be viewed as a calculus unifying a module language with a term language and forbidding access to both of module and value components of functor arguments. Contrary to *Room*, *PathCal* separates a module language from a term language and relaxes the restriction to allow accessing to value components of the arguments. This was made possible by a stratified approach to decidability of the type system. In *PathCal*, we first ensure termination of the normalization of paths by constructing path dependency relations. Then, decidability of the type system is obtained using locking of value access paths in addition to reliance on the termination of the normalization. We think such stratification is a promising approach to have more flexible functors.

9 Conclusions

In this paper, we proposed a lenient recursion extension of the ML module system to allow flexible use of nested structures and functors in the presence of recursion. We designed a calculus, called *PathCal*, and investigated in detail a safety property that ensures resolvability of recursive references. This safety is ensured in *PathCal* by first checking that a program has finite path dependencies, then type checking it. Our approach is provably decidable.

We have at least two directions for future work. One is to make it possible to define modules as fix points of functors, *i.e.* to allow module definitions like `module` M = N(M), for some functor N. With such an extension, we need to be careful about the way in which M is used inside N to ensure termination of the normalization of N(M). Possible approaches are to construct more involved path dependency relations, or, in an easier way, to introduce another construct like box(M), prohibiting dereference of M during the normalization. The other direction for future work is relaxing the restriction on functor arguments to support higher-order functors. As our decidability result essentially relies on this restriction, this is more challenging.

We are also considering separate compilation and modules with side-effects. More work seems left to be done for their support. To support separate compilation, we would need to introduce signatures enriched with information about module dependencies so as to ensure that linking of independently compiled modules does not produce ill-defined programs. We are seeking such appropriate signatures that unnecessitate recheck on well-definedness of the overall program. For effectful modules, we first have to decide how to resolve aliases if they are aliases for effectful expressions, *i.e.* should we evaluate the effects before the resolution of aliases or not? This is an important design issue we have not yet resolved.

Acknowledgment

Many thanks to Jacques Garrigue for his useful suggestions through many fruitful discussions. I thank Masahito Hasegawa, Susumu Nishimura, and Sinya Katumata for their valuable comments.

References

[1] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Proc. the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528, pages 1–13. Springer Verlag, 1991.

[2] Gerard Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14:263–315, 2004.

[3] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Proc. PLDI'99*, pages 50–63, 1999.

[4] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. LICS'90*, 1990.

[5] Derek Dreyer. A type system for well-founded recursion. In *Proc. POPL'04*, 2004.

[6] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *Proc. ESOP'02*, number 2305, pages 6–20, 2002.

[7] X. Leroy, D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system. Software and documentation available on the Web, `http://caml.inria.fr` /.

[8] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. POPL'94*, pages 109–122. ACM Press, 1994.

[9] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[10] David B. MacQueen. Modules for Standard ML. In *Proc. the 1984 ACM Conference on LISP and Functional Programming*, pages 198–207. ACM Press, 1984.

[11] Keiko Nakata, Akira Ito, and Jacques Garrigue. Recursive Object-Oriented Modules. In *Proc. FOOL'05*, 2005.

[12] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, 2003.

[13] S. Romanenko, C. Russo, N. Kokholm, and P. Sestoft. Moscow ML. Software and documentation available on the Web, `http://www.dina.dk/~sestoft/mosml.html`.

[14] Claudio V. Russo. Recursive Structures for Standard ML. In *Proc. ICFP'01*, pages 50–61. ACM Press, 2001.

A Proof

This section briefs the proof. We reserve u for a metavariable ranging over flat paths, and let *FPaths* be the set of flat paths.

Sketch of proof.(Proposition 1) This proposition can be reduced into Lemma 1. □

Lemma 1 *Let $\mathcal{A}$ be a finite set of alphabets, and $\mathcal{R}$ be a finite set of binary relations on strings of $\mathcal{A}$. It is decidable whether the binary relation $\rightarrow$ is well-founded or not, where $s \rightarrow t$ holds if and only if there exists strings s', t', u such that (s', t') is in $\mathcal{R}$, and $s = s'u$ and $t = t'u$ hold.*

Proof. This Lemma has been proved [4]. □

To prove Proposition 2, we first define a well-founded relation $\succ_2$ on paths. The relation $\succ_2$ relies on two well-founded relations $\succ_3$ and $\succ_4$. We will present these relations in order.

In the following, we fix a program P having finite path dependencies. By definition, the postfix and transitive closure of $dp(\epsilon, P)$ is well-founded.

Definition 6 *The relation $\succ_3$ on flat paths is the smallest transitive relation that contains 1) the postfix and transitive closure of $dp(\epsilon, P)$ and 2) $\{(s.M, s) \mid s \in FPaths, M \in MNames\}$.*

Lemma 2 *$\succ_3$ is well-founded.*

Definition 7 *A path tree t is defined as follows.*

$$\begin{array}{lcl} t & ::= & u \mid u(nodes) \\ nodes & ::= & t \mid t, nodes \end{array}$$

Definition 8 *The relation $\succ_4$ on path trees is the smallest transitive relation satisfying the condition that $u([t_i]_{i=1}^{n}) \succ_4 u'([t'_i]_{i=1}^{n'})$ holds if either of the following conditions holds.*

1. - $u \succ_3 u'$
   - *For all i in $\{1 \ldots n'\}$, either of the followings holds.*
     - $u([t_i]_{i=1}^{n}) \succ_4 t'_i$
     - *there exists j such that $t_j = t'_i$ or $t_j \succ_4 t'_i$ holds.*
2. - $u = u'$
   - *There exist i, j such that $t_i \succ_4 t'_j$ holds, and, for all k in $\{1, \ldots, n'\} \backslash \{j\}$, there exists l such that either of $t_l \succ_4 t'_k$ or $t_l = t'_k$ holds.*
3. *There exists j such that $t_j = u'([t'_i]_{i=1}^{n'})$ holds.*
4. *$u = u'$ and, there exists j such that $\{t_1, \cdots, t_n\} \backslash \{t_j\} = \{t'_1, \cdots, t'_{n'}\}$ holds.*

Lemma 3 *$\succ_4$ is well-founded.*

Sketch of proof. $\succ_4$ is a variant of recursive path ordering with precedence $\succ_3$. The proof of this lemma is similar to that of the well-foundedness of recursive path ordering. □

Now we define the relation $\succ_2$ on paths. Given a path p, we construct a path tree $\mathcal{T}(p)$ as follows,

- $\mathcal{T}(p) = p$ where p is in *FPaths*

- $\mathcal{T}(p) = u(\mathcal{T}(p_1), \ldots, \mathcal{T}(p_n))$ where $\mathit{flat}(p) = u$ and $\mathit{args}(p) = \{p_1, \ldots, p_n\}$

Then $p \succ_2 p'$ holds iff $\mathcal{T}(p) \succ_4 \mathcal{T}(p')$ holds. By Lemma 3, $\succ_2$ is well-founded.

The following two lemmas can be shown by induction on $\succ_2$.

Lemma 4 *If $\vdash p \triangleright p'$ and $p \not\equiv p'$, then $p \succ_2 p'$.*

Lemma 5 *It is decidable, for any path p, whether there exists a path p' such that $\vdash p \triangleright p'$ holds. And we have an algorithm calculating such p', if it exists.*

Sketch of proof.(Proposition 2) To show the proposition, we construct a well-founded relation on pairs (Δ, e) of a set of value access paths Δ and an expression e. In the following, we will assume any Δ meets the two conditions that: 1) if $p.l$ is in Δ, then p is of source form; 2) if $p.l$ and $p'.l$ are in Δ, and $\mathit{flat}(p) = \mathit{flat}(p')$, then $p \equiv p'$. Note that this condition is maintained throughout valid type judgments due to the rule [**T-def**].

$(\Delta_1, e_1) \succ_1 (\Delta_2, e_2)$ holds iff either of the following conditions holds.

- $\Delta_1 = \Delta_2$, $e_1 \equiv p_1.l$, $e_1 \equiv p_2.l$, and $p_1 \succ_2 p_2$
- Δ_1 is a proper subset of Δ_2.
- $\Delta_1 = \Delta_2$, $e_1 \equiv (e_{11}, e_{12})$, and $e_2 \equiv e_{1i}$ with $i \in \{1, 2\}$

The well-foundedness of $\succ_1$ is obtained by the following two facts.

- $\succ_2$ is well-founded
- there does not exists an infinite sequence $\{\Delta_i\}_{i=1}^{\infty}$ such that, for all natural number i, $\Delta_i \subset \Delta_{i+1}$

Then, the proposition can be shown by induction on $\succ_1$. □

Sketch of proof.(Proposition 3) This proposition can be checked by induction on the structure of the deduction of $\vdash e$ ok. □