Recursive modules for programming

(Extended abstract)

Keiko Nakata[1] Jacques Garrigue[2]
[1] Kyoto University Research Institute for Mathematical Sciences
[2] Graduate School of Mathematics, Nagoya University

Abstract

The ML module system enables flexible development of large software systems by its support of nested structures, functors and signatures. In spite of this flexibility, however, recursion between modules is prohibited, since dependencies between modules must accord with the order of definitions. As a result of this constraint, programmers may have to consolidate conceptually separate components into a single module, intruding on modular programming. Recently much work has been devoted to extending the module system with recursion, and developing a type system for recursive modules has been one of the main subjects of study. Since recursion is an essential mechanism, one faces several issues to be considered for designing a practical type system.

Our goal is to make recursive modules an ordinary construct for ML programmers. We want to use them easily in everyday programming, possibly combining with other constructs of the core and the module languages. With this goal, we are to develop a type system for recursive modules, which is practical and useful from the programmer's perspective. We recognize difficulties involved in type checking recursive modules and address these difficulties by confining ourselves to first-order functors. The type system is provably decidable and sound for a call-by-value operational semantics. The technical development of the type system and a soundness proof are somewhat involved. In this paper, we present our design of a language for recursive modules and give several examples; one of the examples expresses a variation on the expression problem in support of our design choice.

1 Introduction

When building a large software system, it is useful 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 ML programming languages, which includes SML[18] and Objective Caml [16], provides a powerful module system [17, 15]. 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 [10, 13]. Despite the flexibility of the module language, however, mutual recursion between modules is prohibited, since dependencies between modules must accord with the order of definitions. As a result of this constraint, programmers may have to consolidate conceptually separate components into a single module, intruding on modular programming.

Much work has been devoted to extending the ML module system with recursion. There are at least two important issues involved in recursive modules, namely initialization and type checking.

Initialization: Suppose that we can freely refer to value components in structures forward and backward. Then we might carelessly define value components cyclically like `val l = m val m = l`. Initialization of modules having such cyclic value definitions would either raise runtime errors or cause meaningless infinite computation. Boudol [1], Hirschowitz and Leroy [12], and Dreyer [3] examined type systems which ensure safe initialization of recursive modules. Their type systems ensure that the initialization does not attempt to access undefined recursive variables. The above cyclic definitions will be rejected because initialization of the value component `l` requires an access to itself. This path is not the main focus of this paper.

Type checking: Designing a type system for recursive modules is another important and non-trivial issue; this is the main focus of this paper. Suppose that we can layout modules in any order regardless of their dependencies. Then, it might happen that a function returns a value whose type is not yet defined at the point where the function is defined. To type check the function, a type system should somehow know about the type, which is going to be defined in the following part of the program.

1.1 Type checking recursive modules

To type check recursive modules, existing type systems [2, 25, 24, 5, 4, 16] rely on annotations; programmers have to assist the type checker by writing enough type information by themselves so that recursive modules do not burden the type checker with forward references.

The amount of required annotations varies in each proposal and depends on whether type abstraction is enforced inside the recursion or outside, that is, whether recursive modules do not know exact implementations of each other, or they do but the rest of the program does not. In all proposals, one has to write signatures twice for the same module to enforce type abstraction outside; one of the signatures is solely for assisting the type checker and does not affect the resulting signature of the module. Moreover, the annotation requirement disables programmers from using type inference of the core language in the presence of recursive modules. This would be unfortunate since a lot of useful inference algorithms have been and will be developed to support smooth development of

programs.

Even if we write annotations for recursive modules, this still leaves two subtle issues to be considered.

1.2 Cyclic type specifications in signatures

To annotate recursive modules with signatures, existing type systems provide recursive signatures, in which components of signatures can refer to each other recursively. To develop a practical algorithm for judging type equality, one may want to ensure that transparent type specifications in recursive signatures do not declare cyclic types. For instance, one may want to forbid programmers from writing the recursive signature `sig type t = s type s = t end`.

Detection of cyclic type specifications is not a trivial task when the module language supports both recursive signatures and applicative functors [14]. Applicative functors give us more flexibility in expressing type sharing constraint between modules; at the same time, it is possible to specify cyclic types in such a way that a straightforward check cannot detect, by combining applicative functors and recursive signatures. One pathological example of cyclic type specifications is:

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

Compare the above recursive signature to the recursive signature below.

```
module type G = functor(X : sig type t end) → sig type t = G(X).t end
```

On the one hand, a type system would easily detect the latter cycle, since the unrolling of the type `G(X).t` would be `G(X).t` itself. On the other hand, it might not be easy to detect the former cycle, since the unrolling of the type `F(F(X)).t` could yield the following infinite rewriting sequence.

```
F(F(X)).t →F(F(F(X))).t →F(F(F(F(X)))).t →...
```

Observe that this sequence does not contain the syntactically same object, but produces arbitrary large objects.

The situation could become harder, when we want to allow the recursive signature:

```
module type H = functor(X : sig type t type s end) →
    sig type t = H'(H'(X)).t type s = X.s → X.s end
and H' = functor(X : sig type t type s end) →
    sig type t = X.t * X.t type s = H(H(X)).s end
```

Although `H` and `H'` may seem to define more complex types than `F`, this last recursive signature does not contain cycles.

The three recursive signatures we have seen here are simple. Hence one may easily distinguish between them, judging that only the last one is legal. When recursive signatures specify more complex types, however, this issue becomes subtle to address.

1.3 Potential existence of cyclic type definitions

Another subtle issue is how to account for the potential of cyclic type definitions in structures, when opaque signatures hide their implementations. For instance, should the type checker reject the program below?

```
module M = struct type t = N.t end : sig type t end
and N = struct type t = M.t end : sig type t end
```

On the one hand, one could argue that this is unacceptable since the underlying implementations of the types `M.t` and `N.t` make a cycle. On the other hand, one could argue that this is acceptable since, according to their signatures, the types `M.t` and `N.t` are nothing more than abstract types. Thus the modules `M` and `N` need not be accused of defining cyclic types. At least, one could argue that potential cycles are acceptable, if the type system is sound and decidable and this choice has merits over the other choice.

Existing type systems take different stands on this issue.

In Russo's system [25], programmers have to write forward declarations for recursive modules, in which implementations of types other than datatypes cannot be hidden. Thus there is no potential that cyclic type definitions are hidden by opaque signatures. At the same time, programmers cannot enforce type abstraction inside recursive modules.

Dreyer's work [4] focuses on type abstraction inside recursive modules. He requires the absence of cyclic type definitions whether or not they are hidden by opaque signatures. To ensure the absence of cycles without peeking inside signatures, he puts restriction on types whose implementation can be hidden. As a consequence, the use of structural types is restricted. For instance, his type system would reject the following program, which uses polymorphic variants [6] and a list to represent trees and forests. (Here we use polymorphic variants, which are supported only in the Objective Caml variant of ML, since the core language we want to support is that of O'Caml. Yet, similar restrictions could arise in the context of SML, when one attempts to use records to represent trees.)

```
module Tree = (struct
  type t = [‘Leaf of int | ‘Node of int * Forest.t ]
end : sig type t end)
and Forest = (struct type t = Tree.t list end : sig type t end)
```

By replacing polymorphic variants with usual datatypes, we could make this program typed in his system. Polymorphic variants, however, have their own merits that datatypes do not have.

The path Objective Caml [16] chose is a more liberal one. It does not care whether cyclic type definitions are hidden inside opaque signatures or not, as long as the signatures themselves do not specify cycles. The type checker complains when recursive signatures contain cyclic type specifications whenever it terminates. (Recall that applicative functors make it difficult to detect cycles in a terminating way.) On the one hand, O'Caml supports a highly expressive core language, including structural types such as objects and polymorphic variants. On the other hand, the path it chose keeps flexibility in using these types and

in abstracting them away by opaque signatures. Only that there is no formal proof for the soundness of this path and that the soundness might not be proven by a translation into an explicit type-passing system. When we make opaque signatures transparent, we may reveal cyclic type definitions which were hidden inside signatures. If a type is defined cyclically, there is no concrete type to be passed explicitly.

1.4 Our proposal of a type system for recursive modules

Our goal is to make recursive modules an ordinary construct of the module language for ML programmers. We want to use them easily in everyday programming, possibly combining with other constructs of the core and the module languages. With this goal, we are to develop a practical type system for recursive modules, which overcomes as much of the difficulties discussed above as possible. Concretely, we follow the path O'Caml chose but are to extend it by 1) enabling type inference; 2) ensuring that signatures of recursive modules do not specify cyclic types, while keeping applicative functors; 3) proving soundness of the type system formally, but allowing potential cyclic type definitions to be hidden by opaque signatures, thus keeping flexibility in using structural types. At the current development, we confine ourselves to first-order functors. We defer it to future stage to accommodate higher-order functors by presumably adapting existing approaches.

Our technical developments and proofs are somewhat involved. We obtain ideas from term rewriting theory for enabling type inference and detecting cyclic type specifications. We use a technique in labeled transition systems for the soundness proof.

In this paper, we do not give the formalization of our developments nor proofs. It is hard for us to describe then in this limitation of the space. There are papers [20, 19] devoted to their explanations. In this paper, we present our design of a language, named *Remonade*, for recursive modules, and give several examples; one of the examples expresses a variation on the expression problem [26] in support of our design choice.

2 Example

In this section, we review two examples to illustrate two possible uses of recursive modules and to present the main features of *Remonade*[1].

The first example appears in Figure 1. The top-level module `TreeForest` contains two modules `Tree` and `Forest`: `Tree` represents a module for trees whose leaves and nodes are labeled with integers; `Forest` represents a module for unordered sets of those integer trees.

[1] In examples, we shall allow ourselves to use some usual core language constructions, such as let and if expressions and list constructors, even though they are not part of the formal development given in Section 3.

```
module TreeForest = struct (TF)
 module Tree = (struct
  datatype t = Leaf of int | Node of int * TF.Forest.t
  val max = λx.case x of Leaf i ⇒ i
      | Node (i, f) ⇒ let j = TF.Forest.max f in if i > j then i else j
  val mk_tree = λx.let i = TF.Forest.max x in Node(i, x)
 end : sig type t val max : t → int val mk_tree : TF.Forest.t → t end)
 module Forest = (struct
  type t = TF.Tree.t list
  val max = λx.case x of [] ⇒ 0
      | hd :: tl ⇒ let i = TF.Tree.max hd in let j = max tl in
        if i > j then i else j
  val combine = λx.λy.TF.Tree.mk_tree ([x;y] : t :: TF.Forest.t)
  end : sig
    type t val max : t → int
    val combine : TF.Tree.t → TF.Tree.t → TF.Tree.t end)
end
```

Figure 1: Modules for trees and forest

The modules `Tree` and `Forest` refer to each other in a mutually recursive way. Their type components `Tree.t` and `Forest.t` refer to each other, as do their value components `Tree.max` and `Forest.max`. These functions calculate the maximum integers a tree and a forest contain, respectively.

We defer explanations of functions `Tree.mk_tree` and `Forest.combine` to Section 3.

To enable forward references, we extend structures and signatures with *self variables*. Components of structures and signatures can refer to each other recursively using the self variables. For instance, `TreeForest` has a self variable named `TF`, which is used inside `Tree` and `Forest` to refer to each other recursively. We keep the usual ML scoping rules for backward references. Thus `Tree.max` can refer to the `Leaf` and `Node` constructors without going through a self variable. `Tree` might also be used without prefix inside `Forest`, but the explicit notation seems clearer.

This first example illustrates a possible use of recursive modules, where they respect each other's privacy, that is, they are sealed with opaque signatures individually, enforcing type abstraction inside the recursion.

The second example appears in Figure 2. Now `TreeForest` is a functor, parameterized by the type of labels of trees. We assume that an applicative functor `MakeSet` is given in a library for making sets of comparable elements.

The modules `Tree` and `Forest` define the same recursive types as the first example, except that the argument types of the value constructors `Leaf` and `Node` are parameterized. The module abbreviation `module F = TF.Forest` inside `Tree` enables us to use an abbreviation `F` for `TF.Forest` inside `Tree`. Similarly, the type `s` in `Tree` is an abbreviation which expands into `TF.Forest.t`. Although we can dispense with abbreviations by replacing them with their definitions altogether, they are useful in practice [22].

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

Figure 2: Intimate modules for trees and forests

In the second example, `Tree` and `Forest` have intimate relations: the functions `Tree.split` and `Forest.sweep` need to know the underlying implementations of the types `Forest.t` and `Tree.t` of the others to construct and deconstruct values of those types. Given a tree, `split` cuts off the root node of the tree and returns the resulting forest. The function `sweep` gathers the leaves from a given forest.

Since the two modules are intimate, we do not seal `Tree` and `Forest` individually here. Instead, we seal them as a whole with an opaque signature. The signature only exposes functions `split`, `sweep`, and `incr`, which augments a given forest only if the set of labels in a given tree subsumes the set of labels in the forest, but hides functions `Tree.labels` and `Forest.labels`, which are utility functions for `incr`. The signature also enforces type abstraction, protecting their privacy from the outside.

The two examples we have seen so far illustrate two possible uses of recursive modules. They may have privacy, enforcing type abstraction inside the recursion. They may have intimacy, enforcing type abstraction outside the recursion. Both uses would be useful and would become common in practice.

Module expr.	E	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	*structure*
		|	$\texttt{functor}(X : S) \to E$	*functor*
		|	$(E : S)$	*sealing*
		|	p	*module path*
Definitions	D	::=	$\texttt{module}\ M = E$	*module definition*
		|	$\texttt{val}\ l = e$	*value definition*
		|	$\texttt{datatype}\ t = T$	*datatype definition*
		|	$\texttt{type}\ t = \tau$	*type abbreviation*
Signature expr.	S	::=	$\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}$	*signature type*
		|	$\texttt{functor}(X : S) \to S$	*functor type*
Specifications	B	::=	$\texttt{module}\ M : S$	*module specification*
		|	$\texttt{type}\ t = \tau$	*manifest type specification*
		|	$\texttt{type}\ t$	*abstract type specification*
		|	$\texttt{val}\ l : \tau$	*value specification*
Recursive ident.	*rid*	::=	$Z \mid rid.M$	
Module ident.	*mid*	::=	$rid \mid mid(mid) \mid mid(X)$	
Module path	p	::=	$mid \mid X$	
Program	P	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	

Figure 3: The module language of *Remonade*

3 Syntax

We give the syntax for our module language in Figure 3, which is based on Leroy's module calculus with manifest types [13]. We use M as a metavariable for ranging over module names, X for ranging over module variables, and Z for ranging over self variables. For simplicity, we distinguish them syntactically, however the context could tell them apart without this distinction.

As explained in the previous section, we extend structures and signatures with implicitly typed declarations of self variables to support recursive references. In the constructs $\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$ and $\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}$, the self variable Z is bound in $D_1 \ldots D_n$, and $B_1 \ldots B_n$, respectively.

An unusual convention of bound module variables is that we let a module variable X bound in the signature specified for X. That is, in the constructions $\texttt{functor}(X : S) \to E$ and $\texttt{functor}(X : S) \to S'$, we let X bound in S [2]. There are situations where this convention is useful, as we see in the next section.

The construct which enables recursive references is *recursive identifiers*. A recursive identifier is constructed from a self variable and the dot notation ".M", which represents access to the module component M of a structure. A recursive identifier may begin from any bound self variable, and may refer to a module at any level of nesting within the recursive structure or signature, regardless of component ordering. For instance, through the self variable of the top-level structure, one can refer to any module named in that structure except for those hidden within sealed sub-structures. It is important that recursive identifiers can only use bound self variables, and that self variables of sealed modules are unbound outside them. Otherwise type abstraction could be broken.

[2] This implies that the module expressions `functor(X : sig (Z) type t type s = Z.t end) → X` and `functor(X : sig type t type s = X.t end) → X` denote the same functor.

Types	τ	::=	$\mathbf{1} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid ext\_rid.t \mid X.t$
Datatype definition	T	::=	c `of` τ
Core expressions	e	::=	$x \mid () \mid (\lambda x.e : \tau) \mid (e_1, e_2) \mid \pi_i(e) \mid e_1(e_2)$
		$\mid$	$rid.c\ e \mid$ `case` e `of` $ms \mid rid.l \mid X.l \mid (e : \tau_1 :: \tau_2)$
Matching clause	ms	::=	$rid.c\ x \Rightarrow\ e$
Extended recursive ident.	ext_rid	::=	$Z \mid ext\_rid.M \mid ext\_rid(ext\_rid) \mid ext\_rid(X)$

Figure 4: The core language of *Remonade*

For the sake of simplicity, we assume that functor applications only contain module paths. In a practical system, however, we can weaken this restriction following Leroy's proposal [15].

To obtain a decidable type system, we impose a *first-order structure restriction* that requires functors 1) not to have functors as arguments, 2) nor to access inner modules of arguments. The first condition means that our functors are first-order, and the second implies that we have to pass inner modules as independent parameters for functors instead of passing a module which contains all of them. One might have noticed that the syntax of module paths excludes those of the forms $X.M$ and $X(p)$. This is due to the restriction.

We give the syntax for our core language in Figure 4. We use x as a metavariable for program variables and c for value constructor names.

The core language describes a simple functional language extended with the following three constructions.

- *Value paths* $X.l$ and $rid.l$ refer to the value components l in the modules referred to by X and *rid*, respectively.

- *Type paths* $X.t$ and $ext\_rid.t$ refer to the type components t in the modules referred to by X and *ext_rid*, respectively. For type paths, we slightly extend the class of paths so as to liberally include functor applications, motivated by Leroy's proposal [14]. This gives us more flexibility in expressing type sharing constraint between modules.

- *Type conversion* $(e : \tau :: \tau)$ is for turning values of concrete types into values of abstract types, and vice versa.

  When a module is sealed with a signature, we distinguish the module defined inside the signature and the module which inhabits the signature. In other words, a sealed module has different appearances depending on whether the module is referred to inside the sealing or outside. For instance, inside the module `Forest` in Figure 1, we do not equate the type `t` and the type `TF.Forest.t`; the former is an internal type referring to `Forest`'s type `t` inside the sealing, but the latter is an external type referring to `Forest`'s type `t` outside.

  This design choice reflects our stand that an abstract type generates a new type that is not equivalent to any types but itself, and keeps semantics of type equality simple. Nevertheless, this choice might be occasionally

inconvenient, when one wants to build a value of an external type inside the sealing. This is where the type conversion can be used.

Now, we shall explain functions `Tree.mk_tree` and `Forest.combine`; the former builds a tree from a given forest and the latter builds a tree from given two trees. The type system infers that the expression `[x;y]` in the body of `combine` has the type `TF.Tree.t list`. The internal type `t` of `Forest` is equivalent to `TF.Tree.t list`, but the external type is not. Hence, to apply `Tree.mk_tree` to `[x;y]`, one needs to turn the type of `[x;y]` into the external type `t` of `Forest`, namely `TF.Forest.t`, using the type conversion as we did in Figure 1. Note that `Tree.mk_tree` expects an argument of type `TF.Forest.t`.

Inside `Forest`, it is also possible to turn, for instance, a variable `x` of the external type `t` of `Forest` into that of the internal type, with the construction `(x : TF.Forest.t :: t)`. Yet, it is not possible, inside `Tree`, to produce a null list `[]` of type `TF.Forest.t` using the conversion, since the type abstraction of `Forest` towards `Tree` is enforced. (Observe that the internal type `t` of `Forest` can be only seen inside `Forest`.)

We note that the type conversion construct does not add to expressiveness of *Remonade* essentially. That is, one can produce the same effect without the construct, but using nested structures. We include it in the language for programmer's convenience.

From a technical point of view, it might be possible to infer the necessary type conversions to some extent, and to insert the conversions automatically instead of requiring programmers to write them explicitly. But we think that the explicit conversions are useful for programmers to avoid exporting values of internal types unintentionally.

Note that when sealing is not involved, type conversions are unnecessary at all. For instance, inside the module `Forest` in Figure 2, the types `t`, `TF.Forest.t` and `TF.Tree.t list` are equivalent.

In our formalization, 1) function definitions are explicitly type annotated; 2) each structure and signature type declares a self variable; 3) a path is always prefixed by a self variable or a module variable. Our running examples do not obey these rules. Instead, we have assumed that there is an elaboration phase, prior to type checking, that adds type annotations for functions by running a type inference algorithm of the core language. The original program may still require some type annotations, to avoid running into the polymorphic recursion problem. In [20], we discuss the details of this inference algorithm. The elaboration phase also infers omitted self variables, to complete implicit backward references.

```
module type E =
 sig type exp val eval : exp → int val simp : exp → exp end
module PF = functor(X : E with type exp = private [> PF(X).exp ] ) →
 struct
   type exp = [‘Num of int | ‘Plus of X.exp * X.exp]
   val eval : exp → int = λx.case x of ‘Num n ⇒ n
      | ‘Plus (e1, e2) ⇒ X.eval e1 + X.eval e2
   val simp : exp → X.exp = λx.case x of ‘Num n ⇒ ‘Num n
      |‘Plus(e1, e2) ⇒ case (X.simp e1, X.simp e2) of
           (‘Num m, ‘Num n) ⇒ ‘Num(m+n)
        | e12 ⇒ ‘Plus e12
 end
module Plus = (PF(Plus) : E with type exp = PF(Plus).exp)
```

Figure 5: A first language

4 The expression problem

In this section, we present an example which expresses a variation on the expression problem [26]. The expression problem is one of the most fundamental issues one faces during the development of extensible software. A typical example of this problem is to enrich progressively a small expression language with new constructors and provides operations on this language. Giving a concise and type safe solution to this problem is notoriously difficult.

Objective Caml [16] was probably the first language to solve the problem in a type safe way, using either polymorphic variants [7] or classes [23]. Later, Garrigue refined the solution by combining recursive modules of Objective Caml and private row types [8]. Here we further refine his solution by using recursive modules of our proposal instead of those of O'Caml.

The example we use here is originally given in [9]. It is a variation on the expression problem, where we only insist on the addition of new constructors; adding new operations is trivial in this setting.

We shall assume that we have extended *Remonade* with polymorphic variants, private row types and some usual module language constructions. Adding polymorphic variants and private row types is straightforward. We add typing rules for them to our language. Our choice of applicative functors is essential for having private row types. Allowing structures to have module type components may not be easy, but having module type definitions in the top-level is easy.

To reduce notational burden, we omit preceding self variables even for forward references, since no ambiguity seems to arise. We also omit the top-level `struct` and `end`.

The functor `PF` in Figure 5 defines a first expression language. The type `exp` in `PF` indicates that the first language supports expressions composed of integer constants and addition. The function `eval` evaluates the expressions into integers. The function `simp` simplifies the expressions, by reducing the `‘Plus` constructor into `‘Num` when possible.

```
module MF = functor(X : E with type exp = private [> MF(X).exp ]) →
  struct
    module Plus = PF(X)
    type exp = [Plus.exp | ‘Mult of X.exp * X.exp ]
    val eval : exp → int = λx.case x of
        #Plus.exp as e ⇒ Plus.eval e
       |‘Mult(e1, e2) ⇒ X.eval e1 * X.eval e2
    val simp : exp → X.exp = λx.case x of
        #Plus.exp as e ⇒ Plus.simp e
       |‘Mult(e1, e2) ⇒ case (X.simp e1, X.simp e2) of
           (‘Num m, ‘Num n) ⇒ ‘Num(m*n)
         | e12 ⇒ ‘Mult e12
  end
module Mult = (MF(Mult) : E with type exp = MF(Mult).exp)
```

Figure 6: A second language

To keep the first language extensible, we have the variant type and operations recur through the parameter `X` of the functor `PF`, leaving `PF` open recursion.

We use a private row type in the specification for `PF`'s argument type. Informally, the specification `type exp = private [> PF(X).exp]` denotes an abstract type into which `PF(X).exp` can be coerced. For more details on private row types, we refer readers to [8]. Recall our convention on bound module variables described in Section 3; in the signature of `PF`'s argument, `X` is bound.

The module `Plus` closes the recursion. Observe how it is easy; we take the fixed-point of the functor `PF`.

In this small example, we used private row types, applicative functors and recursive modules in a crucial way. In particular, if it were not for them, we cannot specify the signature of `PF`'s argument; the signature contains the type which shall be obtained by instantiating `PF` itself.

Next, we define a second expression language using the functor `MF`, which appears in Figure 6. The second language supports expressions composed of multiplication and addition on integer constants. Inside `MF`, we instantiate the first addition language, and use it in operations `eval` and `simp` to delegate known cases by variant dispatch.

Our solution here to this variation on the expression problem refines the previous solution [9] in that we close the recursion at the type and the operation levels simultaneously. In particular, it seems useful for programmers that one needs not define recursive types explicitly so as to close the type level recursion separately from the operation level.

5 Related work

Much work has been devoted to investigating type systems for the ML module system with recursive module extensions. To the best of our knowledge, how-

ever, no formal work has been examined a type system for recursive modules with applicative functors, except for the experimental implementation in Objective Caml, nor has proposed type inference for recursive modules whether functors are applicative or generative. Among others, only our type system can handle the example on the expression problem in Section 4.

Below we give a short review on other work.

Crary, Harper and Puri [2] gave a foundational, type-theoretic account for recursive modules. They analyzed recursive modules in the context of a phase-distinction formalism [11], by introducing a fixed-point operator for modules and *recursively dependent signatures*. Their type system requires fully transparent signature annotations for recursive modules, where all components of the modules must be made public. This means that one cannot enforce type abstraction inside recursive modules, hence Figure 1 from Section 2 is not typed.

Objective Caml [16] supports recursive modules. The design of *Remonade* is motivated by it in large part. To type check Figure 2 in O'Caml, one has to write duplicate signatures for the modules `Forest` and `Tree`.

Russo [25] proposed a type system for recursive modules, which is implemented in Moscow ML [24]. In Russo's system, self variables must be annotated with forward declarations, in which implementations of types other than datatypes cannot be hidden. Hence, one can not enforce type abstraction inside recursive modules.

Dreyer [4] gave a theoretical account for recursive modules with generative functors, by proposing a "destination passing" interpretation for the modules. The type system requires duplicate signature annotations for Figure 2.

6 Conclusion

With the goal to use recursive modules easily in every day programming, we have been developing a practical type system for the ML module system with a recursive module extension. In this paper, we first reviewed important issues we have faced during the development of the type system, then described our design choice by giving several examples. In particular, the last example shall support our design choice by presenting a concise and type safe solution to the expression problem.

A Overview of proofs

Here, we give a very brief summary of our development and proofs.

To enable type inference and to detect cyclic type specifications in signatures, we develop "expansion algorithms" for resolving forward references in recursive modules. The development of the algorithms are somewhat involved. The idea comes from our previous work [21]; the first-order restriction allows us to use technique in ground term rewriting theory, where termination conditions are well-investigated. Once we developed the expansion algorithms, we can

design the type system as a straightforward extension of Leroy's applicative functor calculus [14]. We obtain decidability of the type system as an immediate consequence of the termination of the expansion algorithms.

Our proof of the soundness takes the following two steps.

1. We define a type system, named *RemonadeX*, whose type equality is defined by the weak bisimulation on a labeled transition system on types. *RemonadeX* is not necessarily decidable but can account for cyclic type definitions in recursive modules. We prove progress and type preservation lemmas in *RemonadeX*.

2. We show that when a program P is typed in *Remonade*, then the program obtained from P by making opaque signatures transparent is typed in *RemonadeX*.

References

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

[2] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. PLDI'99*, pages 50–63, 1999.

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

[4] D. Dreyer. Recursive Type Generativity. In *Proc. ICFP'05*, 2005.

[5] D. Dreyer. *Understanding and Evolving the ML Module System.* PhD thesis, Carnegie Mellon University, 2005.

[6] J. Garrigue. Programming with polymorphic variants. In *In Proc. ML workshop'98*, 1998.

[7] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, 2000.

[8] J. Garrigue. Private rows: abstracting the unnamed. `http://www.math.nagoya-u.ac.jp/~garrigue/papers/privaterows.pdf`, 2005.

[9] J. Garrigue. Private rows: abstracting the unnamed. Slides. `http://www.math.nagoya-u.ac.jp/~garrigue/papers/private-show.pdf`, 2005.

[10] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. POPL'94*, 1994.

[11] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Porc. of POPL'90*, pages 341–354, 1990.

[12] T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In *Proc. ESOP'02*, pages 6–20, 2002.

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

[14] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. POPL'95*, pages 142–153. ACM Press, 1995.

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

[16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.09. Software and documentation available on the Web, `http://caml.inria.fr/`, 2005.

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

[18] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[19] K. Nakata and J. Garrigue. Recursive modules for programming. `http://www.kurims.kyoto-u.ac.jp/~keiko/`, 2005.

[20] K. Nakata and J. Garrigue. Type inference for recursive modules. `http://www.kurims.kyoto-u.ac.jp/~keiko/`, 2005.

[21] K. Nakata, A. Ito, and J. Garrigue. Recursive Object-Oriented Modules. In *Proc. FOOL'05*, 2005.

[22] B. Pierce, editor. *Advanced Topics in Types and Programming Languages*, chapter 9. The MIT Press, 2004.

[23] D. Rémy and J. Garrigue. On the expression problem. `http://pauillac.inria.fr/~remy/work/expr/`, 2004.

[24] S. Romanenko, C. Russo, N. Kokholm, and P. Sestoft. Moscow ML, 2004. Software and documentation available on the Web, `http://www.dina.dk/~sestoft/mosml.html`.

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

[26] P. Wadler. The expression problem. Java Genericity mailing list, 1998. `http://www.cse.ohio-state.edu/~gb/cis888.07g/java-genericity/20`.