Recursive Modules for Programming

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

Jacques Garrigue
Graduate School of Mathematics,
Nagoya University
garrigue@math.nagoya-u.ac.jp

Abstract

The ML module system is useful for building large-scale programs. The programmer can factor programs into nested and parameterized modules, and can control abstraction with signatures. Yet ML prohibits recursion between modules. As a result of this constraint, the programmer may have to consolidate conceptually separate components into a single module, intruding on modular programming. Introducing recursive modules is a natural way out of this predicament. Existing proposals, however, vary in expressiveness and verbosity. In this paper, we propose a type system for recursive modules, which can infer their signatures. Opaque signatures can also be given explicitly, to provide type abstraction either inside or outside the recursion. The type system is decidable, and is sound for a call-by-value semantics. We also present a solution to the expression problem, in support of our design choices.

Categories and Subject Descriptors D.3.1 [*PROGRAMMING LANGUAGES*]: Formal Definitions and Theory; D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Recursion, Modules

General Terms Languages, Theory, Design

Keywords Type systems, type inference, recursive modules, applicative functors, the expression problem

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 [15], provides a powerful module system [16, 14]. 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 [9, 12].

In spite of this flexibility, the ML module language prohibits recursion between modules. This is a major disadvantage of ML, when compared to object-oriented languages, like Java. These languages have supported recursive definitions across class boundaries from the beginning, and this feature is heavily used in practice.

We, ML programmers, enjoy strong type safety. Yet, due to the lack of recursive modules, we may have to consolidate conceptually separate components into a single module, intruding on modular programming [23]. If we had both recursive modules and this flexible module language, we could enjoy a strongly type safe programming language with an equally strong expressive power.

Recently, much work has been devoted to investigating extensions with recursion of the ML module system. Two important issues involved are type checking and initialization. Crary, Harper and Puri [3], Russo [23], and Dreyer [5] have given type theoretic accounts for recursive modules. Boudol [1], Hirschowitz and Leroy [11], and Dreyer [4] have investigated type systems which guarantee well-formedness of recursive modules, ensuring that initialization of recursive modules will not attempt to access not-yet-evaluated values.

To some extent, ML programmers can already use recursive modules in everyday programming. Several languages of the ML family support recursive modules [15, 22], allowing practical programming, or, at least, a flavor of it.

In this paper, we first review two examples. In the first one, two recursive modules `Tree` and `Forest` respect each other's privacy: we seal them with opaque signatures individually. Thus type abstraction is enforced inside the recursion. In the second, `Tree` and `Forest` are intimate: they know each other's exact implementations, and we seal them with an opaque signature as a whole. Thus type abstraction is enforced outside the recursion.

Both privacy and intimacy will be important for practical uses of recursive modules. Existing proposals, however, vary in their way to handle them. We may be denied privacy. We may have to write two different signatures for the same module; one of them solely for assisting the type checker, while the other gives the resulting signature for the module.

Our goal is to develop a type system for recursive modules, which is practical and useful from the programmer's perspective; we want to use them easily in everyday programming, possibly combining with other constructs of the core and module languages.

With this goal in mind, we propose a type system for recursive modules, in which modules can have privacy or intimacy depending on the situation they are in. The type system does not require additional signature annotations. Thus the programmer can either omit writing signatures or give signatures explicitly to control abstraction. Moreover, he can rely on type inference during development; all previous proposals by others do not support type inference for recursive modules.

In this paper, we also present an advanced example of recursive modules, by giving a concise and type safe solution to the expression problem [26]. In the example, we use recursive modules, applicative functors [13] and private row types [8] together. The example confirms that by combining recursive modules with other

```
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
 end : sig type t val max : t → int 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
  end : sig type t val max : t → int end)
end
```

Figure 1. Modules for trees and forests

language constructions we can indeed enjoy a highly expressive power in a type safe and modular way.

Our contributions are summarized as follows.

- We examine two typical uses of recursive modules by giving concrete examples. These examples are useful for understanding basic uses of recursive modules.
- We propose a new type system for recursive modules with first-order applicative functors. The type system supports type inference for recursive modules, and is decidable and sound for a call-by-value semantics.

  All examples we present in this paper are type checked in this type system, without requiring additional signature annotations.
- We give a type safe and concise solution to the expression problem, in order to demonstrate that recursive modules give us high expressive power in a modular way when combined with other language constructions.

The rest of the paper is organized as follows. In the next section, we review two examples of recursive modules and present the main features of our calculus, *Traviata*, used for our formal development. Section 3 gives the concrete syntax of *Traviata*. Section 4 and 5 explain the type system and present a soundness result. In Section 6, we give a solution to the expression problem. In Section 7, we examine the double vision problem [6]. Section 9 examines related work and Section 10 concludes.

2. Examples

In this section, we review two examples to illustrate two possible uses of recursive modules and to informally present *Traviata*[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.

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.

To enable forward references, we extend structures and signatures with implicitly typed declarations of *self variables*. Components of structures and signatures can refer to each other recursively using the self variables. For instance, `TreeForest` declares 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. They are sealed with signatures individually, enforcing type abstraction inside the recursion.

```
module TreeForest =
 functor(X : sig type t val compare : t → t → bool end) →
  (struct (TF)
    module S = MakeSet(X)
    module Tree = struct
     module F = TF.Forest
     type 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.singleton i
        | Node (i, f) ⇒ TF.S.add i (F.labels f)
    end
    module Forest = struct
     module T = TF.Tree
     type t = T.t list
     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 = λf.λt.let l1 = labels f in
        let l2 = T.labels t in
        if TF.S.diff l1 l2 != TF.S.empty then (t :: f) else f
    end
   end : 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 : t → Z.Tree.t → t
    end
   end)
```

Figure 2. Intimate modules for trees and forests

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 constructors `Leaf` and `Node` are parameterized. The module abbreviation `module F = TF.Forest` inside `Tree` allows 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`.

In this second example, `Tree` and `Forest` are intimate: the functions `Tree.split` and `Forest.sweep` know the underlying implementations of the types `Forest.t` and `Tree.t` of the others, thus can 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 a single signature. The signature only exposes functions `split`, `sweep`, and `incr`, which augments a given forest only if a given tree contains original labels that are not contained in the forest, but hides functions `Tree.labels` and `Forest.labels`, which are utility functions for `incr`. The signature also enforces type abstraction by

[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.

hiding implementations of the types `Tree.t` and `Forest.t`, thus it protects privacy of the two modules 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. We think both uses are natural and would become common in practice.

Comparison with existing type systems The two examples presented are type checked in our type system without requiring additional annotations. Below, we examine the ways existing type systems handle these examples.

To avoid presenting too much annotations, we remove the module abbreviation `module F = TF.Forest` from `Tree` in Figure 2. Yet, although we can dispense with abbreviations by replacing them with their definitions altogether, they are useful in practice [24].

In Russo's system [23] there is no obvious way to type check the first example, keeping type abstraction between `Tree` and `Forest`. A suggested solution, which is found in his paper, is to annotate the self variable `TF` of `TreeForest` with a *recursive signature* [2] [3] [23]:

```
sig (Z : sig module Tree : sig type t end
             module Forest : sig type t = Tree.t list end
         end)
 module Tree : sig
  datatype t = Leaf of int | Node of int * Z.Forest.t end
 module Forest : sig
  type t = Tree.t list val max : t → int end
end
```

This annotation for `TF`, however, would break type abstraction between `Tree` and `Forest`, exposing underlying implementations of types `Tree.t` and `Forest.t` to each other.

In Dreyer's system [5], the sealing signatures for `Tree` and `Forest` must be given in advance. That is, the programmer has to write both signatures before defining either of the two modules, as opposed to Figure 1, where the signatures are written in a module-wise way.

O'Caml [15] type checks Figure 1 without modifications.

Next, we examine the second example.

In Russo's system, the programmer must annotate `TF` with a recursive signature:

```
sig (Z : sig module Tree : sig type t end
             module Forest : sig type t = Tree.t list end
         end)
 module Tree : sig
  datatype t = Leaf of X.t | Node of X.t * Z.Forest.t end
 module Forest : sig
  type t = Tree.t list val labels : t → MakeSet(X).t end
end
```

Note that this signature is solely for assisting the type checker. We have already given in Figure 2 the eventual signatures that `Tree` and `Forest` should have; these signatures do not reveal the underlying implementations of types `Tree.t` and `Forest.t` or the function `Forest.labels`.

To type check Figure 2 in Dreyer's system and O'Caml, the programmer must write fully manifesting signatures of `Tree` and `Forest` in advance, where the signatures declare every component of the modules. The type checker first type checks the two modules assisted by these manifest signatures. Once this succeeds, type abstraction is enforced using the sealing signature given in Figure 2. Thus the programmer has to write annotations yet more verbose than in Russo's system.

[2] This recursive signature does not exactly follow his syntax, *e.g.* we have to use the keyword `structure` instead of `module` in his system.

[3] We note that by permuting the definition order of `Tree` and `Forest` the amount of required annotations can be reduced to some extent in this case. However permutation does not always work.

Module expression

E	::=	E_d^i	

Module expression descriptions

E_d	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	*structure*
	|	$\texttt{functor}\ (X : A) \to E$	*functor*
	|	$(E : S)$	*sealing*
	|	mid	*module identifier*
	|	X	*module variable*

Definitions

D	::=	$\texttt{module}\ M = E$	*module def.*
	|	$\texttt{datatype}\ t = c\ \texttt{of}\ \tau$	*datatype def.*
	|	$\texttt{type}\ t = \tau$	*type abbreviation*
	|	$\texttt{val}\ l = e$	*value def.*

Signature

S	::=	S_d^i	

Signature descriptions

S_d	::=	$\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}$	*structure type*
	|	$\texttt{functor}(X : A) \to S$	*functor type*

Module variable signature

A	::=	A_d^i	

Module variable signature description

A_d	::=	$\texttt{sig}\ B_1 \ldots B_n\ \texttt{end}$	

Specifications

B	::=	$\texttt{module}\ M : S$	*module spec.*
	|	$\texttt{datatype}\ t = c\ \texttt{of}\ \tau$	*datatype spec.*
	|	$\texttt{type}\ t = \tau$	*manifest type spec.*
	|	$\texttt{type}\ t$	*abstract type spec.*
	|	$\texttt{val}\ l : \tau$	*value spec.*

Recursive identifiers

rid	::=	$Z \mid rid.M$

Module identifiers

mid	::=	$rid \mid mid(mid) \mid mid(X)$

Extended module identifiers

ext_mid	::=	$Z \mid ext\_mid.M$
	|	$ext\_mid(ext\_mid) \mid ext\_mid(X)$

Module paths

p, q, r	::=	$ext\_mid \mid X$

Program

P	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}^i$

Figure 3. The module language of *Traviata*

We believe that both privacy and intimacy are important for practical uses of recursive modules. Existing type systems, however, do not handle them equally. These type systems may deny privacy. They may require additional annotations that are used only for helping the type checker, but do not affect resulting signatures of modules. Even if we assume that these annotations provide some useful information, our experience with type inference in ML is that one often writes a module without its signature, and then eventually writes a signature by editing the result of type inference. This technique is not available for recursive modules in these type systems.

3. Syntax

Figure 3 gives the module language of *Traviata*, which is based on Leroy's applicative functor calculus [13]. We use M as a metavariable for module names, X for module variables and Z for self variables. For simplicity, we distinguish them syntactically, however the context could tell them apart without this distinction. We also use t for type names and l for (core) value names.

For the purpose of both defining type equality and designing a decidable type system, we label module expressions, signatures and *module variable signatures* with integers. For instance, a module expression E is a module expression description E_d labeled with an integer i, where E_d is either a structure, a functor, a sealing,

Core types	τ	::=	$\mathbf{1} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid p.t$
Core expr.	e	::=	$x \mid () \mid (\lambda x.e : \tau) \mid (e_1, e_2) \mid \pi_i(e) \mid e_1(e_2)$
		$\mid$	$rid.c\, e \mid X.c\, e \mid \texttt{case}\ e\ \texttt{of}\ ms \mid rid.l \mid X.l$
Matching	*ms*	::=	$rid.c\, x \Rightarrow e \mid X.c\, x \Rightarrow e$

Figure 4. The core language of *Traviata*

a module identifier or a module variable [4]. One can think of the integer label i of E_d^i as the location of E_d in the source program. For the interest of brevity, we may omit integer labels when they are not used. For the interest of clarity, we may write additional parentheses, for instance $(\texttt{functor(X : sig type t end}^2) \to \texttt{X}^3)^1$. We use metavariables i, j for integers.

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

For simplicity, we provide different syntax for signatures and module variable signatures; the latter are used to specify signatures of functor arguments and do not declare self variables. In a practical system, we can unify their syntax for the programmer's benefit.

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 sub-modules 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, 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 contain bound self variables, and that self variables of sealed modules are unbound outside them. Otherwise type abstraction could be broken.

For the sake of simplicity, functor applications only contain module identifiers and module variables.

To support applicative functors [13], we define a slightly extended class of identifiers, named *module paths* in Figure 3, which can liberally include functor applications. Core types defined in Figure 4 may use module paths. Applicative functors give us more flexibility in expressing type sharing constraints between recursive modules. In Section 6, we give a practical example which uses recursive modules and applicative functors together in support of our design choices. It will be useful to note that $Z \subseteq rid \subseteq mid \subseteq ext\_mid \subseteq p$ holds.

A program is a top-level structure which contains a bunch of recursive modules. In this paper, we only consider recursive modules, but not ordinary ones.

To obtain a decidable type system, we impose a *first-order structure restriction* that requires functors 1) not to take functors as argument, 2) nor to access sub-modules of arguments. The first condition means that our functors are first-order, and the second implies that the programmer has to pass sub-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 expression descriptions excludes those of the forms $X.M$ and $X(mid)$. This is consistent with this restriction.

Figure 4 gives the our core language of *Traviata*. We use x as a metavariable for program variables (variables, for short), and c for value constructor names.

[4] Note that *Traviata* does not have separate notions for opaque signatures and transparent ones.

The core language describes a simple functional language extended with *value paths* $X.l$ and $rid.l$, and *type paths* $p.t$. Value paths $X.l$ and $rid.l$ refer to the value components l in the structures referred to by X and *rid*, respectively. A type path $p.t$ refers to the type component t in the structure that p refers to.

We may say paths to mean module, type and value paths as a whole.

An unusual convention is that a module variable is bound inside its own signature. For instance,

```
functor(X : sig type t val l : X.t end) → X
```

is a legal expression, which should be understood as

```
functor(X : sig type t val l : t end) → X
```

This convention is convenient when proving type soundness, as the syntax of paths is kept uniform, that is, every path is prefixed by either a self variable or a module variable. In Section 6, we give examples where this this convention is useful.

We write $MVars(p)$ to denote the set of module variables contained in the module path p. We also write $MVars(\tau)$, $MVars(e)$ and the likes with obvious meanings.

In the formalization, 1) function definitions are explicitly type annotated; 2) every structure and structure type declares a self variable; 3) a path is always prefixed by a self variable or a module variable. Our examples do not stick to 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 on the core language. The original program may still require some type annotations, to avoid running into the polymorphic recursion problem. In Section 8, we discuss the details of this inference algorithm. The elaboration phase also infers omitted self variables, to complete implicit backward references.

We assume the following five conventions: 1) a program does not contain free module variables or free self variables; 2) all binding occurrences of module or self variables use distinct names; 3) any sequence of module definitions, type abbreviations, datatype definitions, value definitions, module specifications, manifest and opaque type specifications, datatype specifications and value specifications does not contain duplicate definitions or specifications for the same name; 4) all occurrences of module expressions, signatures and module variable signatures in a program are labeled with distinct integers; 5) module variable signatures do not contain module specifications.

4. Reconstruction

The type system is composed of two parts, namely a type reconstruction part and a type-correctness check part. Concretely, we type check a program P in two steps: 1) reconstruct a *lazy program type* of P; at this point, we do not require the reconstructed type to be correct; 2) check type-correctness of P by type checking P in the intuitive way, using the reconstructed type as type environment; once this second step is completed, we are certain both that P is type-correct and that the reconstruction was correct.

In this section we describe the reconstruction part; the next section explains the type-correctness check part.

The rest of this section is organized as follows. In Section 4.1, we define lazy program types, which are output by the reconstruction algorithm. In Section 4.2, we define a *look-up judgment* for using programs and lazy program types as lookup tables. In Section 4.3, we introduce "resolution algorithms", the key for enabling the reconstruction. Finally, in Section 4.4, we present an algorithm for reconstructing lazy program types from programs.

In the rest of the paper, we assume that each self variables Z is annotated with a *module variable environment* θ, written Z^θ. A module variable environment is a substitution of module paths for module variables. Correspondingly, we assume that each occur-

Lazy signature

T ::= T_d^i

Lazy signature descriptions

T_d ::= $\texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end}$ *lazy structure type*

| $\texttt{functor}(X : A) \to T$ *lazy functor type*

| $(T_1 : T_2)$ *lazy sealing type*

| p

Lazy specifications

C ::= $\texttt{module}\ M : T$

| $\texttt{datatype}\ t = c\ \texttt{of}\ \tau$

| $\texttt{type}\ t = \tau$

| $\texttt{type}\ t$

| $\texttt{val}\ l : \tau$

Lazy program type

U ::= $\texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end}^i$

Figure 5. Lazy module types

Top-levels	O	::=	$P \mid U$
Module descriptions	K	::=	$E_d \mid S_d \mid A_d \mid T_d$
	:=	::=	$= \mid :$
	ss	::=	$\texttt{struct} \mid \texttt{sig}$

Figure 6. Notation convention

$$\frac{}{O \vdash Z^\theta \mapsto (\theta, \rho_O(Z))}\ (1) \qquad \frac{}{O \vdash X \mapsto (id, \rho_O(X))}\ (2)$$

$$\frac{O \vdash p \mapsto (\theta, \texttt{ss}\ \ldots \texttt{module}\ M := K^j \ldots \texttt{end}^i) \quad K \neq (K_1^{j_1} : K_2^{j_2})}{O \vdash p.M \mapsto (\theta, K^j)}\ (3)$$

$$\frac{O \vdash p \mapsto (\theta, \texttt{ss}\ \ldots \texttt{module}\ M := K^j \ldots \texttt{end}^i) \quad K = (K_1^{j_1} : K_2^{j_2})}{O \vdash p.M \mapsto (\theta, K_2^{j_2})}\ (4)$$

$$\frac{O \vdash p_1 \mapsto (\theta, (\texttt{functor}(X : A) \to K^j)^i) \quad K \neq (K_1^{j_1} : K_2^{j_2})}{O \vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], K^j)}\ (5)$$

$$\frac{O \vdash p_1 \mapsto (\theta, (\texttt{functor}(X : A) \to K^j)^i) \quad K = (K_1^{j_1} : K_2^{j_2})}{O \vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], K_2^{j_2})}\ (6)$$

Figure 7. Look-up

rence of a self variable in a program P is implicitly annotated with an identity substitution *id*. That is, we regard Z as an abbreviation for Z^{id}. We use θ as a metavariable for module variable environments.

4.1 Lazy module types

Figure 5 gives the syntax for lazy module types, which we use as signatures of modules during type checking. The syntax for lazy signature descriptions extends that for signature descriptions with the sealing construction $(T_1 : T_2)$ and module paths. We use the sealing construction $(T_1 : T_2)$ to check type-correctness of the sealing construction $(E : S)$ of module expression descriptions ((25) in Figure 15). We use module paths to instantiate signatures lazily ((55) in Figure 18). In the construct $\texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end}$, the self variable Z is bound in $C_1 \ldots C_n$. A lazy program type is a top-level lazy structure type labeled with an integer. Note that lazy signatures include signatures.

We use the notation convention in Figure 6. In particular, we use O as a metavariable for top-levels, which are either programs or lazy program types, and K for *module descriptions*, which are either module expression descriptions, signature descriptions, module variable signature descriptions or lazy signature descriptions.

```
struct (Z)
  module M1 = (functor(X : sig type t end^3) →
    struct module M11 = struct end^5 end^4)^2
  module M2 = struct type t = int end^6
  module M3 = Z.M1(Z.M2)^7
end^1
```

Figure 8. A program P_1

4.2 Look-up

Next, we define a look-up judgment for finding module descriptions and their integer labels from a top-level. During the reconstruction we use the judgment against programs; during the type-correctness check, we use the judgment against lazy program types.

We assume that, for a top-level O, there is a global mapping ρ_O which sends i) a self variable Z to the structure or the (lazy) structure type to which Z is *ascribed* in O, and ii) a module variable X to the module variable signature specified for X in O. We say that in the construct $\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}^i$ the self variable Z is ascribed to $\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}^i$. Similarly, in the constructs $\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}^i$ and $\texttt{sig}\ (Z')\ C_1 \ldots C_m\ \texttt{end}^j$, Z and Z' are ascribed to $\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}^i$ and $\texttt{sig}\ (Z')\ C_1 \ldots C_m\ \texttt{end}^j$, respectively. The use of ρ_O makes the presentation concise [5].

We present inference rules for the look-up judgment in Figure 7. The judgment $O \vdash p \mapsto (\theta, K^i)$ means that the module path p refers to the module description K labeled with the integer i in the top-level O, where each module variable X is bound to $\theta(X)$.

Let us examine each rule. For self variables and module variables, the judgment consults the global mapping ρ_O. Next two rules (3) and (4) handle module paths of the form $p.M$. A module path $p.M$ refers to the sub-module named M in the module that p refers to. Hence p must refer to either a structure or a (lazy) structure type. The rules (3) and (4) distinguish whether M is bound to a sealing construction $(K_1^{j_1} : K_2^{j_2})^j$ or not; when it is, then $p.M$ resolves to the sealing part $K_2^{j_2}$. Thus, the judgment prevents peeking inside of sealed modules from outside them. The last two rules (5) and (6) handle module paths of the form $p_1(p_2)$. When p_1 refers to either a functor or a (lazy) functor type, then $p_1(p_2)$ resolves to the body of the functor, where the module variable environment is augmented with the new binding $[X \mapsto p_2]$. Again the rules (5) and (6) distinguish whether the body is a sealing construction or not.

The look-up judgment does not hold for arbitrary module paths. For instance, consider Figure 8. We have $\mathrm{P}_1 \vdash \texttt{Z.M}_1(\texttt{Z.M}_2).\texttt{M}_{11} \mapsto ([\texttt{X} \mapsto \texttt{Z.M}_2], \texttt{struct end}^5)$. But, the judgment does not hold for the module path $\texttt{Z.M}_3.\texttt{M}_{11}$.

Recall that we have assumed the absence of free module variables. This means that when $O \vdash p \mapsto (\theta, q^i)$, then $MVars(q) \subseteq dom(\theta)$. For a module variable environment θ, $dom(\theta)$ denotes the domain of θ.

4.3 Resolution algorithms

Our type system differs from others in that it can resolve path references. Concretely, we developed a terminating procedure for determining the component that a path refers to, where the path may contain forward references. The motivation of this procedure was to define a decidable judgment for type equality. In a language with recursive modules and applicative functors, there is the potential that

[5] We could avoid this assumption of a global mapping by annotating each self variable with the source program location of the structure or structure type to which the self variable is ascribed. Since the source program can be regarded as a finite tree, we can represent every node of the tree by a finite representation (i.e., we need not use file names or line numbers.)

$$\frac{O \vdash p \rightsquigarrow_n p'}{O \vdash p.M \rightsquigarrow_n p'.M} \qquad \frac{O \vdash p \rightsquigarrow_n p'}{O \vdash p(q) \rightsquigarrow_n p'(q)}$$
$$\frac{O \vdash q \rightsquigarrow_n q'}{O \vdash p(q) \rightsquigarrow_n p(q')} \qquad \frac{O \vdash p \mapsto (\theta, q^i)}{O \vdash p \rightsquigarrow_n \theta(q)}$$

Figure 9. Normalization of module paths with respect to O

a program contains pathologically cyclic type abbreviations which may cause type equality check to diverge. We later noticed that a similar procedure enables type inference for recursive modules. Note that we cannot use the well-typedness of the source program when resolving path references, since we already need type equality to ensure this well-typedness.

We implement the procedure for path resolution as three algorithms, namely, a module path expansion algorithm *PathExp*, a type expansion algorithm *TyExp* and a core type reconstruction algorithm *CtyReconst*. These algorithms use termination criteria based on ground term rewriting and recursive path ordering; the criteria do not rely on the well-typedness of the source program, and still allow flexible handling of module and type abbreviations.

For lack of space, we do not explain all these algorithms; we only give their specifications, needed to present the rest of the type system. We give definitions of *PathExp* and *CtyReconst* in Appendix A. In [19], the reader can find detailed explanations.

Located types We define a canonical form of types, called *located types*. The type system checks equality between two arbitrary types by reducing them into located types using *TyExp*.

A located type is a type composed of *simple located types* and $\mathbf{1}$ using $\rightarrow$ and $*$. Intuitively, a simple located type is an abstract type which is obtained by expanding all type and module abbreviations.

We first define *located forms*, a canonical form of module paths. A module path p is in located form if and only if p does not contain a module path which resolves to a module abbreviation.

DEFINITION 1. *A module path p is in located form with respect to a top-level O if and only if the following two conditions hold.*

- *$O \vdash p \mapsto (\theta, K^i)$ where K is not a module path.*
- *For all q in $args(p)$, q is in located form.*

For a module path p, $args(p)$ denotes the set of module paths that p contains as functor arguments, or:

$$args(Z^\theta) = \bigcup_{X \in dom(\theta)} \{\theta(X)\}$$
$$args(p.M) = args(p) \qquad args(p_1(p_2)) = args(p_1) \cup \{p_2\}$$

A simple located type is an abstract type whose prefix is a located form.

DEFINITION 2. *A simple located type with respect to a top-level O is a type path $p.t$ where p is in located form with respect to O and either $O \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end}^i)$ or $O \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t \ldots \mathtt{end}^i)$ holds.*

Now located types are defined below.

DEFINITION 3. *A located type with respect to a top-level O is a type τ where each type τ' in $typaths(\tau)$ is a simple located type with respect to O.*

For a type τ, $typaths(\tau)$ denotes the set of type paths that τ contains. Precisely,

$$typaths(\tau) = \begin{cases} typaths(\tau_1) \cup typaths(\tau_2) & \text{when } \tau = \tau_1 \rightarrow \tau_2 \text{ or } \tau = \tau_1 * \tau_2 \\ \{p.t\} & \text{when } \tau = p.t \\ \emptyset & \text{when } \tau = \mathbf{1} \end{cases}$$

$$\frac{}{O, \Omega \vdash \mathbf{1} \downarrow \mathbf{1}}$$
$$\frac{O, \Omega \vdash \tau_1 \downarrow \tau_1' \quad O, \Omega \vdash \tau_2 \downarrow \tau_2'}{O, \Omega \vdash \tau_1 \rightarrow \tau_2 \downarrow \tau_1' \rightarrow \tau_2'} \qquad \frac{O, \Omega \vdash \tau_1 \downarrow \tau_1' \quad O, \Omega \vdash \tau_2 \downarrow \tau_2'}{O, \Omega \vdash \tau_1 * \tau_2 \downarrow \tau_1' * \tau_2'}$$
$$\frac{PathExp(O, p) = p' \quad O \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t \ldots \mathtt{end}^i)}{O, \Omega \vdash p.t \downarrow p'.t}$$
$$\frac{PathExp(O, p) = p' \quad O \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end}^i)}{O, \Omega \vdash p.t \downarrow p'.t}$$
$$\frac{PathExp(O, p) = p' \quad O \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t = \tau_1 \ldots \mathtt{end}^i) \quad O, \Omega \uplus (i, t) \vdash \tau_1 \downarrow \tau_2 \quad O, \Omega \vdash \theta(\tau_2) \downarrow \tau}{O, \Omega \vdash p.t \downarrow \tau}$$

Figure 10. Type expansion with respect to O

module path expansion To specify the module path expansion algorithm *PathExp*, we define the *normalization* of module paths in Figure 9. The judgment $O \vdash p \rightsquigarrow_n q$ means that p reduces into q in one step, with respect to the top-level O. The normalization traces module abbreviations in the intuitive way. We write $O \vdash p \rightsquigarrow_{n^*} q$ to mean that p reduces into q in more than or equal to zero step.

The proposition below states that *PathExp* is terminating and that, when it succeeds, it coincides with normalization. *PathExp* raises an error when it cannot prove that the input module path does not contain cyclic or dangling references.

PROPOSITION 1.

1. *For any top-level O and module path p, PathExp(O, p) either returns a module path in located form with respect to O, or else raises an error.*
2. *When $PathExp(O, p) = q$, then $O \vdash p \rightsquigarrow_{n^*} q$.*

Type expansion We define the type expansion algorithm in Figure 10. The judgment $O, \Omega \vdash \tau_1 \downarrow \tau_2$ means that the algorithm expands the type τ_1 into the type τ_2 where Ω is locked, with respect to the top-level O. We use Ω as a metavariable for pairs (i, t) of an integer i and a type name t. The notation $\Omega \uplus (i, t)$ means $\Sigma \cup \{(i, t)\}$ whenever $(i, t) \notin \Omega$. Note that derivations of the type expansion are deterministic. In particular, it is an error state when there are no applicable rules.

Then we define *TyExp* such that it takes as argument a top-level O and a type τ, then either returns a type τ' when $O, \emptyset \vdash \tau \downarrow \tau'$ holds or else raises an error when no rule applies, *i.e.* it cannot prove that the input type does not contain cyclic or dangling references.

PROPOSITION 2. *For any top-level O and type τ, $TyExp(O, \tau)$ either returns a located type with respect to O or else raises an error.*

Core type reconstruction The type reconstruction algorithm *CtyReconst* propagates type annotations on functions. It does not check that these annotations are correct. For instance, for an expression $e_1(e_2)$, *CtyReconst* only reconstructs a type of e_1, which must be of the form $\tau_1 \rightarrow \tau_2$, then returns the result type τ_2. In the type-correctness check part, the type system checks that e_2 has a type equivalent to τ_1.

At this time, we only require *CtyReconst* to be terminating.

PROPOSITION 3. *For any program P and core expression e, $CtyReconst(P, e)$ either returns a located type with respect to P or else raises an error.*

4.4 Lazy program type reconstruction algorithm

Figure 11 presents inference rules for the lazy program type reconstruction algorithm with respect to a program P. The algorithm is a straightforward application of *CtyReconst* (see (10)), hence it

Definitions

$$\frac{P \vdash E \triangleright T}{P \vdash \texttt{module}\ M = E \triangleright \texttt{module}\ M : T} \quad (7)$$

$$\frac{}{P \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau \triangleright \texttt{datatype}\ t = c\ \texttt{of}\ \tau} \quad (8)$$

$$\frac{}{P \vdash \texttt{type}\ t = \tau \triangleright \texttt{type}\ t = \tau} \quad (9) \qquad \frac{\mathit{CtyReconst}(P, e) = \tau}{P \vdash \texttt{val}\ l = e \triangleright \texttt{val}\ l : \tau} \quad (10)$$

Module expression

$$\frac{P \vdash E_d \triangleright T_d}{P \vdash E_d^i \triangleright T_d^i} \quad (11)$$

Module expression descriptions

$$\frac{P \vdash D_1 \triangleright C_1\ \ldots\ P \vdash D_n \triangleright C_n}{P \vdash \texttt{struct}\ (Z)\ D_1\ \ldots\ D_n\ \texttt{end} \triangleright \texttt{sig}\ (Z)\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (12)$$

$$\frac{P \vdash E \triangleright T}{P \vdash \texttt{functor}(X : A) \to E \triangleright \texttt{functor}(X : A) \to T} \quad (13)$$

$$\frac{P \vdash E \triangleright T}{P \vdash (E : S) \triangleright (T : S)} \quad (14) \qquad \frac{}{P \vdash p \triangleright p} \quad (15)$$

Figure 11. Lazy program type reconstruction with respect to P

does not ensure type-correctness of the program P. It either returns a lazy program type that P would have when P is type-correct, or else raises an error when it cannot prove that P does not contain cyclic or dangling references. Note that it does not check type-correctness of functor applications (see (15)).

Then we define *ReconstP* such that it takes a program P as argument, then either returns U when $P \vdash P \triangleright U$ holds or else raises an error when this cannot be done.

PROPOSITION 4. *For any program P, $\mathit{ReconstP}(P)$ either returns a lazy program type or raises an error.*

5. Type-correctness check

One of the main difficulties in type checking recursive modules is how to reason about forward references. Usually, a type checker consults a type environment for the necessary type information about paths. When paths only contain backward references, it is sufficient to accumulate in the type environment signatures of previously type checked modules. When modules are defined recursively, however, paths may contain forward references. Then the type checker may attempt to ask the type environment for a signature of a module which is not yet type checked.

To circumvent difficulties arising from forward references, other type systems rely on signature annotations. As we examined in Section 2, this requirement compels the programmer to write two different signatures for the same module to enforce type abstraction outside the recursion. Moreover, the programmer cannot rely on type inference during development due to it. This is unfortunate since a lot of useful inference algorithms have been and will be developed to support smooth development of programs.

We have a reconstruction algorithm, hence we do not need the assistance of signature annotations. That is, we use the result of reconstruction as type environment instead of using programmer-supplied annotations.

There are three tasks to be completed in this type-correctness check part: 1) to check type-correctness of core expressions. (Recall that *CtyReconst* does not ensure type-correctness of expressions that it reconstructs types for.); 2) to check well-formedness of module paths, that is, to check that functor applications contained in the paths are type-correct and that references of the paths are not cyclic or dangling; 3) to check that, for every sealing construction $(E : S)$, the module expression E inhabits the signature S.

$$\frac{U \vdash \mathit{TyExp}(U, \tau_1) \equiv_\tau \mathit{TyExp}(U, \tau_2)}{U \vdash \tau_1 \equiv \tau_2} \quad (16)$$

Figure 12. Type equivalence with respect to U

$$\frac{}{U \vdash \mathbf{1} \equiv_\tau \mathbf{1}} \quad (17) \qquad \frac{U \vdash \tau_1 \equiv_\tau \tau_1' \quad U \vdash \tau_2 \equiv_\tau \tau_2'}{U \vdash \tau_1 \to \tau_2 \equiv_\tau \tau_1' \to \tau_2'} \quad (18)$$

$$\frac{U \vdash \tau_1 \equiv_\tau \tau_1' \quad U \vdash \tau_2 \equiv_\tau \tau_2'}{U \vdash \tau_1 * \tau_2 \equiv_\tau \tau_1' * \tau_2'} \quad (19) \qquad \frac{U \vdash p_1 \equiv_p p_2}{U \vdash p_1.t \equiv_\tau p_2.t} \quad (20)$$

Figure 13. Equivalence on located types with respect to U

$$\frac{U \vdash p_1 \mapsto (\theta_1, T_{d1}^{i_1}) \quad U \vdash p_2 \mapsto (\theta_2, T_{d2}^{i_2}) \quad i_1 = i_2 \quad \forall X \in \mathit{dom}(\theta_1),\ U \vdash \theta_1(X) \equiv_p \theta_2(X)}{U \vdash p_1 \equiv_p p_2} \quad (21)$$

Figure 14. Equivalence on located forms with respect to U

5.1 Type equality

We define a type equivalence judgment in Figure 12, with auxiliaries in Figure 13 and 14. The judgment $U \vdash \tau_1 \equiv \tau_2$ means that two types τ_1 and τ_2 are equivalent with respect to the lazy program type U. We check equivalence between two arbitrary types by reducing them into located types.

Figure 13 defines equivalence on located types. The first three rules are straightforward. The last rule judges whether two abstract types are equivalent. Two types $p_1.t$ and $p_2.t$ are equivalent if and only their prefixes p_1 and p_2 are equivalent module paths. (Note that since $p_1.t$ is a located type, p_1 is in located form.)

Figure 14 defines a judgment for equivalence on module paths in located form. Two located forms p_1 and p_2 are equivalent if and only if they refer to module descriptions at the same location (i.e., labeled with the same integer) and their functor arguments are equivalent. Take a look at the look-up judgment (Figure 7) again. The module variable environment θ_1 collects all module paths contained in p_1 as functor arguments.

5.2 Typing rules

In Figure 15, we present typing rules for the type-correctness check, with auxiliaries in Figure 16, 17, 18 and 19.

The judgment $U \vdash E : T$ means that the module expression E of lazy signature T is type-correct with respect to the lazy program type U. The judgment $U; \Gamma \vdash e : \tau$ means that the core expression e of type τ is type-correct under the type environment Γ with respect to U. A type environment assigns a located type to a variable. Other judgments are read similarly.

The typing rules in Figure 15 are mostly straightforward. Here we only explain the rule for sealing.

The rule (25) checks that the sealing construction $(E : S)$ is type-correct. In particular, the third premise is for ensuring that the module expression E inhabits the signature S; it checks that the lazy signature T_1 of E_1 is a subtype of $\mathit{Subst}(T_1, S)$.

The subtyping relation, to be given below, follows that of Leroy's applicative functor calculus. In particular, for two manifest type specifications $\texttt{type}\ t : \tau_1$ and $\texttt{type}\ t : \tau_2$ to be in subtyping relations, τ_1 and τ_2 must be equivalent. To check type equivalence, the type system expand types using *TyExp*; here is the reason that we define the function *Subst*, which is found in Figure 17.

The function *Subst* performs explicit substitution for self variables declared inside sealing signatures. For instance, consider Figure 2. The reconstruction algorithm infers that the function `split` in `Forest` has a type `TF.Tree.t` $\to$ `TF.Tree.t list` (For clarity, we add omitted self variables.). The sealing signature specifies

Module expression

$$\frac{U \vdash E_d : T_d}{U \vdash E_d^i : T_d^i} \quad (22)$$

Module expression descriptions

$$\frac{U \vdash D_1 : C_1 \ \ldots \ U \vdash D_n : C_n}{U \vdash \texttt{struct}\ (Z)\ D_1\ \ldots\ D_n\ \texttt{end} : \texttt{sig}\ (Z)\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (23)$$

$$\frac{U \vdash A : A' \quad U \vdash E : T}{U \vdash \texttt{functor}(X : A) \to E : \texttt{functor}(X : A') \to T} \quad (24)$$

$$\frac{U \vdash E : T_1 \quad U \vdash S : T_2 \quad U \vdash T_1 <: \mathit{Subst}(T_1, S)}{U \vdash (E : S) : (T_1 : T_2)} \quad (25) \qquad \frac{U \vdash p\ \text{wf}}{U \vdash p : p} \quad (26)$$

Definitions and Specifications

$$\frac{U \vdash E : T}{U \vdash \texttt{module}\ M = E : \texttt{module}\ M : T} \quad (27) \qquad \frac{U; \emptyset \vdash e : \tau}{U \vdash \texttt{val}\ l = e : \texttt{val}\ l : \tau} \quad (28)$$

$$\frac{U \vdash \tau \diamond}{U \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau : \texttt{datatype}\ t = c\ \texttt{of}\ \tau} \quad (29)$$

$$\frac{U \vdash \tau \diamond}{U \vdash \texttt{type}\ t = \tau : \texttt{type}\ t = \tau} \quad (30) \qquad \frac{U \vdash S : T}{U \vdash \texttt{module}\ M : S : \texttt{module}\ M : T} \quad (31)$$

$$\frac{}{U \vdash \texttt{type}\ t : \texttt{type}\ t} \quad (32) \qquad \frac{U \vdash \tau \diamond}{U \vdash \texttt{val}\ l : \tau : \texttt{val}\ l : \tau} \quad (33)$$

Signature

$$\frac{U \vdash S_d : T_d}{U \vdash S_d^i : T_d^i} \quad (34)$$

Signature descriptions

$$\frac{U \vdash B_1 : C_1 \ldots U \vdash B_n : C_n}{U \vdash \texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end}} \quad (35)$$

$$\frac{U \vdash A : A' \quad U \vdash S : T}{U \vdash \texttt{functor}(X : A) \to S : \texttt{functor}(X : A') \to T} \quad (36)$$

Module variable signature

$$\frac{U \vdash A_{d1} : A_{d2}}{U \vdash A_{d1}^i : A_{d2}^i} \quad (37)$$

Module variable signature description

$$\frac{U \vdash B_1 : B_1' \ldots U \vdash B_n : B_n'}{U \vdash \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ B_1' \ldots B_n'\ \texttt{end}} \quad (38)$$

Core types

$$\frac{}{U \vdash \mathbf{1} \diamond} \quad (39) \qquad \frac{U \vdash \tau_1 \diamond \quad U \vdash \tau_2 \diamond}{U \vdash \tau_1 \to \tau_2 \diamond} \quad (40) \qquad \frac{U \vdash \tau_1 \diamond \quad U \vdash \tau_2 \diamond}{U \vdash \tau_1 * \tau_2 \diamond} \quad (41)$$

$$\frac{U \vdash p\ \text{wf} \quad \mathit{TyExp}(U, p.t) = \tau}{U \vdash p.t \diamond} \quad (42)$$

Core expressions

$$\frac{}{U; \Gamma \vdash () : \mathbf{1}} \quad (43) \qquad \frac{x \in \mathit{dom}(\Gamma)}{U; \Gamma \vdash x : \Gamma(x)} \quad (44)$$

$$\frac{U; \Gamma \vdash e_1 : \tau_1 \quad U; \Gamma \vdash e_2 : \tau_2}{U; \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \quad (45) \qquad \frac{U; \Gamma \vdash e : \tau_1 * \tau_2}{U; \Gamma \vdash \pi_i(e) : \tau_1} \quad (46)$$

$$\frac{U \vdash \tau \diamond \quad \mathit{TyExp}(U, \tau) = \tau_1 \to \tau_2 \quad U; \Gamma, x : \tau_1 \vdash e : \tau_3 \quad U \vdash \tau_2 \equiv \tau_3}{U; \Gamma \vdash (\lambda x.e : \tau) : \tau_1 \to \tau_2} \quad (47)$$

$$\frac{U; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad U; \Gamma \vdash e_2 : \tau_3 \quad U \vdash \tau_1 \equiv \tau_3}{U; \Gamma \vdash e_1\ (e_2) : \tau_2} \quad (48)$$

$$\frac{U \vdash p\ \text{wf} \quad \mathit{PathExp}(U, p) = p' \quad \gamma(U, p', c) = (t, \tau_1) \quad U; \Gamma \vdash e : \tau_2 \quad U \vdash \tau_1 \equiv \tau_2}{U; \Gamma \vdash p.c\ e : p'.t} \quad (49)$$

$$\frac{U; \Gamma \vdash e_1 : \tau_1 \quad U \vdash p\ \text{wf} \quad \mathit{PathExp}(U, p) = p' \quad \gamma(U, p', c) = (t, \tau_2) \quad U \vdash \tau_1 \equiv p'.t \quad U; \Gamma, x : \tau_2 \vdash e_2 : \tau}{U; \Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ p.c\ x \Rightarrow e_2 : \tau} \quad (50)$$

$$\frac{U \vdash p\ \text{wf} \quad \mathit{PathExp}(U, p) = p' \quad U \vdash p' \mapsto (\theta, \texttt{sig} \ldots \texttt{val}\ l : \tau' \ldots \texttt{end}^i) \quad \mathit{TyExp}(U, \theta(\tau')) = \tau}{U; \Gamma \vdash p.l : \tau} \quad (51)$$

Figure 15. Typing rules with respect to U

$\gamma(U, p, c) = (t, \tau)$ when
$U \vdash p \mapsto (\theta, \texttt{sig} \ldots \texttt{datatype}\ t = c\ \texttt{of}\ \tau' \ \ldots \texttt{end}^i)$
and $\mathit{TyExp}(U, \theta(\tau')) = \tau$

Figure 16. Datatype look-up with respect to U

$\mathit{Subst}(T_d^i, S_d^j) = \mathit{subst1}(T_d, S_d)^j$

$\mathit{subst1}(p, \texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end})$
$= (\texttt{sig}\ (Z)\ \mathit{subst2}(p, [Z \mapsto p]B_1)\ \ldots\ \mathit{subst2}(p, [Z \mapsto p]B_n)\ \texttt{end})$

$\mathit{subst1}(p, \texttt{functor}(X : A) \to S_d^i)$
$= \texttt{functor}(X : A) \to \mathit{subst1}(p(X), S_d)^i$

$\mathit{subst1}(\texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end}, \texttt{sig}\ (Z')\ B_1 \ldots B_m\ \texttt{end})$
$= \texttt{sig}\ (Z')$
$\quad \mathit{subst3}(C_{\sigma(1)}, [Z' \mapsto Z]B_1) \ldots \mathit{subst3}(C_{\sigma(m)}, [Z' \mapsto Z]B_m)\ \texttt{end}$

$\mathit{subst1}(\texttt{sig} \ldots \texttt{end}, \texttt{functor}(X : A) \to S) = \text{raise Error}$

$\mathit{subst1}(\texttt{functor}(X : A) \to T_d^i, \texttt{functor}(X' : A') \to S_d^j)$
$= \texttt{functor}(X' : A') \to \mathit{subst1}([X \mapsto X']T_d, S_d)^j$

$\mathit{subst1}(\texttt{functor}(X : A) \to T, \texttt{sig} \ldots \texttt{end}) = \text{raise Error}$

$\mathit{subst1}((T : T_d^i), S_d) = \mathit{subst1}(T_d, S_d)$

$\mathit{subst2}(p, \texttt{module}\ M : S_d^i) = \texttt{module}\ M : \mathit{subst1}(p.M, S_d)^i$

$\mathit{subst2}(p, B) = B \quad$ where B is not a module specification

$\mathit{subst3}(\texttt{module}\ M : T_d^i, \texttt{module}\ M : S_d^j) = \texttt{module}\ M : \mathit{subst1}(T_d, S_d)^j$

$\mathit{subst3}(C, B) = B \quad$ where B is not a module specification

Figure 17. Substitution

$$\frac{U \vdash T_d <: S_d}{U \vdash T_d^i <: S_d^j} \quad (52) \qquad \frac{U \vdash T_d <: A_d}{U \vdash T_d^i <: A_d^j} \quad (53)$$

$$\frac{U \vdash T_d <: S_d}{U \vdash (T : T_d^i) <: S_d} \quad (54)$$

$$\frac{\mathit{PathExp}(U, p) = p' \quad U \vdash p' \mapsto (\theta, T_d^i) \quad U \vdash \theta(T_d) <: S_d}{U \vdash p <: S_d} \quad (55)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\},\ U \vdash C_{\sigma(i)} <: B_i}{U \vdash \texttt{sig}\ (Z)\ C_1 \ldots C_n\ \texttt{end} <: \texttt{sig}\ (Z')\ B_1 \ldots B_m\ \texttt{end}} \quad (56)$$

$$\frac{U \vdash A' <: [X \mapsto X']A \quad U \vdash [X \mapsto X']T <: S}{U \vdash \texttt{functor}(X : A) \to T <: \texttt{functor}(X' : A') \to S} \quad (57)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\},\ U \vdash C_{\sigma(i)} <: B_i}{U \vdash \texttt{sig}\ C_1 \ldots C_n\ \texttt{end} <: \texttt{sig}\ B_1 \ldots B_m\ \texttt{end}} \quad (58)$$

$$\frac{}{U \vdash \texttt{type}\ t <: \texttt{type}\ t} \quad (59) \qquad \frac{}{U \vdash \texttt{type}\ t = \tau <: \texttt{type}\ t} \quad (60)$$

$$\frac{}{U \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau <: \texttt{type}\ t} \quad (61)$$

$$\frac{U \vdash \tau_1 \equiv \tau_2}{U \vdash \texttt{type}\ t = \tau_1 <: \texttt{type}\ t = \tau_2} \quad (62)$$

$$\frac{U \vdash \tau_1 \equiv \tau_2}{U \vdash \texttt{val}\ l : \tau_1 <: \texttt{val}\ l : \tau_2} \quad (63)$$

$$\frac{U \vdash \tau_1 \equiv_\tau \tau_2}{U \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau_1 <: \texttt{datatype}\ t = c\ \texttt{of}\ \tau_2} \quad (64)$$

$$\frac{U \vdash T <: S}{U \vdash \texttt{module}\ M : T <: \texttt{module}\ M : S} \quad (65)$$

Figure 18. Subtyping with respect to U

$$\frac{}{U \vdash X\ \text{wf}} \qquad \frac{}{U \vdash Z^{id}\ \text{wf}} \qquad \frac{U \vdash p\ \text{wf} \quad \mathit{PathExp}(U, p.M) = q}{U \vdash p.M\ \text{wf}}$$

$$\frac{\begin{array}{c} U \vdash p_1\ \text{wf} \quad U \vdash p_2\ \text{wf} \\ \mathit{PathExp}(U, p_1) = p_1' \quad \mathit{PathExp}(U, p_2) = p_2' \quad \mathit{PathExp}(U, p_1(p_2)) = q \\ U \vdash p_1' \mapsto (\theta, (\texttt{functor}\ (X : A_d^j) \to T)^i) \quad U \vdash p_2' <: \theta[X \mapsto p_2'](A_d) \end{array}}{U \vdash p_1(p_2)\ \text{wf}}$$

Figure 19. Well-formed module paths with respect to U

that `split` has a type `Z.Tree.t` $\rightarrow$ `Z.Forest.t`. Both the reconstructed type and the specified type are located types, but they are not equivalent according to the type equivalence judgment. In fact, for `Forest` to inhabit the sealing signature, the reconstructed type `TF.Tree.t` $\rightarrow$ `TF.Tree.t list` should be equivalent to `TF.Tree.t` $\rightarrow$`TF.Forest.t`, which is the type obtained by substituting `TF` for `Z` in the specified type `Z.Tree.t` $\rightarrow$ `Z.Forest.t`. Indeed, this is satisfied since the type `TF.Forest.t` expands into `TF.Tree.t list`.

One may think that it would be more natural to identify signatures related by α-renaming rather than to perform explicit substitution. Yet implicit renaming makes it complex to use the type expansion algorithm, which is developed separately from the typing rules.

In Figure 18, we define subtyping relations between a lazy signature and a signature ((52)), between a lazy signature and a module variable signature ((53)), between a lazy signature description and a signature description ((54) to (57)), between a lazy signature description and a module variable description ((58)) and between a lazy specification and a specification ((59) to (65)). The rules are mostly intuitive. The reader should only look at the rule (55). A lazy signature description can be a module path p. To check that p is a subtype of a signature description S_d, we instantiate the lazy signature of the module that p refers to; we use the module path expansion algorithm *PathExp* to resolve p's reference. For the decidability of the subtyping relations, it is important that only lazy signature descriptions can be module paths, but ordinary signature descriptions cannot.

The judgment $U \vdash p$ wf, defined in Figure 19, checks that the module path p is well-formed. For instance, the rule (26) in Figure 15 uses this judgment for checking type-correctness of module paths. The judgment ensures that the module path p does not contain cyclic or dangling references and that functor applications contained in p are type-correct.

DEFINITION 4. *A program P is well-typed if and only if $ReconstP(P) = U$ and $U \vdash P : U$ holds.*

PROPOSITION 5. *For any program P, it is decidable whether P is well-typed or not.*

Soundness

Here we give a call-by-value operational semantics and state its type soundness.

Values v and evaluation contexts E are:

$$\begin{array}{lcl} v & ::= & () \mid (v_1, v_2) \mid p.c\ v \mid (\lambda x.e : \tau) \\ E & ::= & \{\} \mid (E, e) \mid (v, E) \mid \pi_i(E) \mid E\ (e) \\ & \mid & v\ (E) \mid p.c\ E \mid \texttt{case}\ E\ \texttt{of}\ ms \end{array}$$

where p does not contain module variables.

Then a small step reduction is either:

$$p.l \xrightarrow{\texttt{mp}} p'.l \ \text{ when } P \vdash p \rightsquigarrow_n p' \qquad \pi_i(v_1, v_2) \xrightarrow{\texttt{prj}} v_i$$

$$(\lambda x.e : \tau)\ v \xrightarrow{\texttt{fun}} [x \mapsto v]e \qquad \texttt{case}\ p.c\ v\ \texttt{of}\ q.c\ x \Rightarrow e \xrightarrow{\texttt{case}} [x \mapsto v]e$$

$$p.l \xrightarrow{\texttt{vpth}} \theta(e) \ \text{ when } P \vdash p \mapsto (\theta, \texttt{struct} \ldots \texttt{val}\ l = e \ldots \texttt{end}^i)$$

or an inner reduction obtained by induction:

$$\frac{e_1 \rightarrow e_2 \quad E \neq \{\}}{E\{e_1\} \rightarrow E\{e_2\}}$$

Again, these reductions are defined with respect to a program P.

When deconstructing a value through the case expression `case` $p.c\ v$ `of` $q.c\ x$, we do not explicitly check that p and q are equivalent. The type system already ensures that p and q expand into equivalent module paths.

During the reduction, we would like to look up actual implementations of modules instead of their signatures from the program P. For this purpose, we assume that once P is type checked, all sealing signatures in P are erased. (Then, the rules (2), (4) and (6) in Figure 7 are not used any more.) After the erasure, the look-up judgment always looks up actual implementations during the normalization of module paths and in the reduction step $\xrightarrow{\texttt{vpth}}$.

We assume that the top-level structure of every program P contains a value component named `main`. The evaluation of P begins by reducing the defining expression of `main`.

```
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 20. A first expression language

PROPOSITION 6 (Soundness). *Let a program P be well-typed. Then the evaluation of P either returns a value or else gives rise to an infinite reduction sequence.*

We cannot state a subject reduction lemma in the context of *Traviata*. For the decidability result, the type system of *Traviata* rejects cyclically defined types. Yet, for proving subject reduction, we want to establish type equality which can handle these cycles. In proofs, we define another type system, called *TraviataX*, which may not be decidable, but can reason about cyclically defined types. We prove that *TraviataX* is sound for the operational semantics, by proving subject reduction and progress properties. Then, Proposition 6 is obtained by proving that if a program P is well-typed in *Traviata*, then so is in *TraviataX*.

6. The expression problem

In this section, we present an advanced example of recursive modules, by giving a solution to the expression problem [26].

The expression problem, named by Phil Wadler, dates back to Cook [2]. It is one of the most fundamental problems one faces during the development of extensible software. Here, we paraphrase a typical example of this problem in the following way: suppose that we have a small expression language, composed of a recursively defined datatype and processors which operate on this datatype; then we want to extend the expression language in two dimensions, that is, to extend the datatype with new constructors and to add new processors. That a programming language can solve this problem in a type safe and concise way has been regarded as a measure of the expressive power of the language. Many researchers have addressed this problem, using different programming languages [21, 27, 25].

Our aim here is not to draw a conclusion that our solution is better than others. Instead, we aim to give a useful example of recursive modules, in order to show that by combining recursive modules with other constructs of the core and the module languages we can obtain more expressive power in a modular way.

The example we use here extends the one in [8]. It is a variation on the expression problem, where we only insist on the addition of new constructors. Adding new processors is easy in this setting.

We shall assume that we have extended *Traviata* with polymorphic variants [7], private row types [8] and some usual module lan-

```
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 21. A second expression language

```
module TreeForest = struct (TF)
 module Tree = (struct
  datatype t = Leaf of int | Node of int * TF.Forest.t
  val max = ...
  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 (F)
  type t = TF.Tree.t list
  val max = ...
  val combine = λx.λy.TF.Tree.mk_tree [x;y]
 end : sig (FS)
  type t val max : t → int
  val combine : TF.Tree.t → TF.Tree.t → TF.Tree.t end)
end
```

Figure 22. Modules for trees and forests(2)

guage constructions. Adding polymorphic variants and private row types is straightforward. We add typing rules for them to our language. Allowing structures to contain module type definitions may not be easy, but having module type definitions in the top-level is easy.

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

We define our first expression language in Figure 20, using the functor `PF`. The type `exp` defined in the body of `PF` indicates that the first language supports expressions composed of integers and addition. The function `eval` is for evaluating expressions into integers. The function `simp` is for simplifying expressions, by reducing the `‘Plus` constructor into the `‘Num` constructor when possible.

To keep the first language extensible, we leave recursion open in `PF`; the polymorphic variant type `exp` and functions `eval` and `simp` recur through `PF`'s parameter `X`.

The intuition of the example is that `PF` takes as argument an expression language which is built by extending the addition language that `PF` defines. This is exactly what the signature of `X` expresses; here is the key of the example. The type specification `type t = private [> PF(X).exp]` specifies an abstract type into which the type `PF(X).exp` can be coerced, or, informally, an abstract type which is a supertype of `PF(X).exp`. The type `PF(X).exp` refers to the type `exp` defined inside `PF`'s body. Hence `X`'s signature specifies that `PF` can only be applied to a module whose defining expression language supports both integers and addition. This recursive use of `PF(X).exp` to constrain `PF`'s argument is the main difference with the solution in [8]. By avoiding the need to define types outside of the functor, it allows for a more concise and scalable solution. Observe that if it were not for all of applicative functors, private row types and flexible path references, we could not write `X`'s signature in this way.

The use of polymorphic variants, which are structural types unlike usual nominal datatypes, is important also for defining the function `simp`. The function `simp` has the type `exp → X.exp`. Since the type `X.exp` structurally contains the type `exp`, as specified in the `X`'s signature, all of `‘Num n`, `‘Num(m+n)` and `‘Plus e12`, which are the results of the case branches, are of type `X.exp`.

The module `Plus` instantiates the addition language, by closing `PF`'s open recursion. Observe that both the type and the value level recursion are closed simultaneously, that is, by taking the fix-point of `PF`, the forwardings `X.exp`, `X.eval` and `X.simp` are connected to `exp`, `simp` and `eval` themselves, thus yielding self-contained recursive type `exp` and recursive functions `eval` and `simp`.

Now we can perform addition on the first language. For instance,

```
val e1 = Plus.eval (‘Plus(‘Num 3, ‘Num 4))
```

Next, we define our second expression language using the functor `MF` in Figure 21. The second language supports expressions composed of multiplication and addition on integers.

We use the exactly same idiom as the first language to define this second language. In particular, the type `MF(X).exp` appearing in `X`'s signature refers to the type `exp` defined in the body of `MF`.

Note that we instantiate the first addition language inside `MF`, and use it in functions `eval` and `simp` to delegate known cases by variant dispatch. Thus we avoid duplication of program codes.

The module `Mult` instantiates the second language, by closing `MF`'s open recursion. Now we can do arithmetic on the second language. For instance,

```
val e2 = Mult.eval (‘Plus(‘Mult(‘Num 3, ‘Num 4), ‘Num 5))
```

Having seen examples here and in Section 2, we confirm that recursive modules are useful in several situations. Moreover, when combined with other language constructions, they give us the highly expressive power in a modular way. We believe that they are a promising candidate for supporting robust extensible software.

7. The double vision problem

Here we examine the double vision problem [6], a typing difficulty involved in recursive modules, in the context of *Traviata*. Detailed examinations of this problem are found in [6, 3].

The situation we want to deal with When a module is sealed with a signature, the type system distinguishes the module defined inside the signature and the module which inhabits the signature. For instance, consider Figure 1. Inside `Forest`, the type `t` and the type `TF.Forest.t` are not equivalent; the former is an internal type, which refers to `Forest`'s type `t` inside the sealing, but the latter is an external type, which refers to `Forest`'s type `t` outside.

This design choice of type equivalence keeps the type equivalence judgment simple. Yet, it might be occasionally inconvenient, for instance, when the programmer wants to build a value of an external type inside sealing.

To see a concrete situation, consider Figure 22. This is the same program as in Figure 1, but here `Tree` and `Forest` contain new functions `mk_tree` and `combine`, respectively; the former is for building a tree from a given forest and the latter for building a tree from given two trees.

Our type system cannot type check the defining expression of `combine`. For the expression `[x;y]` inside the body of `combine`, the core type reconstruction algorithm infers that the expression has a type `TF.Tree.t list`; the function `TF.Tree.mk_tree` takes an argument of type `TF.Forest.t`, which is specified in `Tree`'s sealing signature. According to our type equivalence judgment, however, the types `TF.Forest.t` and `TF.Tree.t list` are not equiv-

alent, since `TF.Forest.t` is an abstract type thus is not equivalent to any other types than itself.

This kind of situation typically occurs when the programmer attempts to cyclically import, inside a sealed module, a value that is exported by the same module as a value of an abstract type. Note that such reimportation is only possible with recursive modules, but not with ordinary modules.

Type coercion Currently we provide a core language construction, called *type coercion*, that allows the programmer to coerce types of expressions from internal types to external types and vice versa, in an explicit way. The type coercion construction is of the form $(e : \tau ::> \tau')$, which informally reads as "to coerce the type τ of the expression e into τ'". For instance, the programmer can define a type-correct `combine` as

```
val combine =
  λx.λy.TF.Tree.mk_tree ([x;y] : t ::> TF.Forest.t)
```

(Observe that the internal type `t` of `Forest` is only visible inside `Forest`.)

For lack of space, we refer the reader to [20] for a typing rule for type coercion. In short, for the construction $(e : \tau ::> \tau')$, the type system checks type equality between τ and τ' in a way more sensitive to sealing but without using *TyExp*. We also note that there is a (somewhat verbose) workaround to define a type-safe `combine` without using type coercions; the programmer can define his own functions which perform type coercion.

8. Type inference for the core language

We implemented a type inference algorithm for the core language by determining an inference order using the module path expansion algorithm, then running a standard inference algorithm along this order. Concretely, using *PathExp*, we build a call graph of functions (represented by a directed graph), which expresses how functions in modules depend on each other: the strongly connected components of the graph indicate sets of value components whose type should be inferred simultaneously, referring to each other monomorphically; by topologically sorting the connected components, we generalize types in a connected component before moving on to typing the next one. For instance in Figure 2, we build an inference order:

`Tree.split` → {`Tree.labels, Forest.labels`}
→ {`Forest.sweep`} → `Forest.incr`

where braces indicate strongly connected component. The inference order we build for Figure 1 is

{`Tree.map`} → {`Forest.map`}

For the purpose of type inference, we do not consider that `Tree.map` and `Forest.map` are mutually recursive, since the signatures of `Tree` and `Forest` specify exported types for these functions.

We must also check for well-formedness of types, as module variables should not escape their scope during unification. This is checked after the inference. Note that when an abstract type depends on a functor argument, then the argument explicitly appears inside the type. For instance, in Figure 2, the type `Tree.t` is internally represented as $\texttt{TF}^{[\texttt{X}\mapsto\texttt{X}]}\texttt{.Tree.t}$.

Explicit type annotations can be used to break dependencies in the call graph, and allow polymorphic recursion. Annotations cannot be completely avoided, as type inference for polymorphic recursion is known to be undecidable.

9. Related work

Much work has been devoted to investigating recursive module extensions of the ML module system. Notably, type systems and initialization of recursive modules pose non-trivial issues, and have been the main subjects of study.

9.1 Type systems

To the best of our knowledge, no work has proposed a type system for recursive modules with applicative functors, except for the experimental implementation in Objective Caml [15], or examined type inference for recursive modules whether functors are applicative or generative. Among other proposals, only *Traviata* can type the examples on the expression problem in Section 6.

The experimental implementation of recursive modules in Objective Caml is most related to our work. Indeed, we followed it in large part when designing *Traviata*. O'Caml supports a highly expressive core language and a strong type inference algorithm, which are one of our motivations for the effort to enable type inference. O'Caml also supports recursive signatures, with a rather concise syntax. However, it allows to write problematic modules whose type checking diverges.The potential for divergence when typing O'Caml modules is well-known, but is assumed to be a rare phenomenon in practice. Recursive signatures seem to make the problem much more acute. This is one of our motivations in insisting on decidable type checking for *Traviata*. Of course we obtain it through restrictions, and a less expressive signature language. Yet, this may be the price for safety. Since we have similar typing rules, we hope that our approach can apply to O'Caml with little change.

Crary, Harper and Puri [3] gave a foundational type theoretic analysis of recursive modules in the context of a phase-distinction formalism [10].

Russo [23, 22] proposed a type system for recursive modules, which we examined in Section 2.

Dreyer [5] gave a theoretical account for type abstraction inside recursive modules. In particular, he investigated generative functors in the context of recursive modules, by proposing a "destination passing" interpretation of type generativity. There is a critical difference in design choices between us, with respects to type abstraction inside recursive modules. For instance, consider the two programs:

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

and

```
module M = (struct type t = N.t list end : sig type t end)
module N = (struct type t = M.t * M.t end : sig type t end)
```

Dreyer prohibits both programs, whereas we accept both. A motivation of our design choice is that we want to keep liberal uses of polymorphic variants and objects, which are useful constructs supported in O'Caml; prohibiting the latter program may result in restriction in using these constructs and recursive modules together.

9.2 Initialization

Boudol [1], Hirschowitz and Leroy [11], and Dreyer [4] have proposed type systems which ensure that initialization of recursive modules does not try to access components of modules that are not yet evaluated. They are interested in the safety of initialization, hence their modules do not have type components.

Their type systems judge the two modules:

```
module M = struct (Z) val l = Z.m val m = Z.l end
```

and

```
module N = struct (Z)
  val l = λx → x + Z.m val m = Z.l(3) end
```

to be ill-typed. In both cases, evaluation of the component `m` cyclically requires evaluation of itself. Our type system, in particular the core type reconstruction algorithm, can reject the cycle for the former, but not for the latter.

10. Conclusion

In this paper, we presented a type system for recursive modules by extending Leroy's applicative functor calculus. The type system is

decidable and sound for a call-by-value operational semantics. It supports type inference for recursive modules, hence type abstraction both inside and outside the recursion is handled equally; the programmer does not need to write two different signatures for the same module to assist the type checker.

We examined three examples. The first two presented typical uses of recursive modules with different choices of where to enforce type abstraction. The last one gave a solution to the expression problem and demonstrated how recursive modules add to the expressive power of the programming language when combined with other language constructions.

Here we give a brief overview of future work.

Separate type checking Although we have not discussed, *Travi-ata* is already prepared for separate type checking. In short, we only have to extend the look-up judgment (Figure 7) so that the judgment informs the type system of signatures of modules which are type checked separately(i.e., to replace concrete module expressions with their signatures).

Lazy modules with eager value components The operational semantics presented in this paper uses lazy evaluation for both modules and their value components in the sense that only components of modules that are accessed are evaluated, and the evaluation is triggered at access time. This semantics simplifies the soundness statement and its proof. For a practical system, however, we are investigating lazy modules with eager value components, that is, to keep modules lazy but evaluate all the value components (but not module components) of a module at once, triggered by the first access to some component of the module. Lazy semantics of modules would allow flexible uses of recursive modules; eager semantics of value components would give the programmer a way to initialize recursive modules. We need more investigation on this topic.

The double vision problem It is desirable to solve the double vision problem without requiring type coercion annotations from the programmer. The current type system always passes to *TyExp* the whole lazy program type *ReconstP* constructed. This seems too naïve. Given that *TyExp* terminates for whatever input, we think it is safe to pass *TyExp* different signature information depending on whether it is used inside sealing or not. For instance in Figure 22, we should make *TyExp* interpret the sealing signature of `Forest` transparently during type checking inside `Forest`.

Acknowledgements

We thank Masahito Hasegawa for useful discussions on the soundness proof and for comments on this paper. We thank anonymous reviewers for their detailed comments, which were most helpful.

References

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

[2] W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In *Proc. REX Workshop*, volume 489 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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.

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

[17] R. Milner. *Communicating and Mobile Systems: the pi-Calculus*. Cambridge University Press, 1999.

[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. Path resolution for recursive modules. Technical Report 1545, Kyoto University Research Institute for Mathematical Sciences, 2006.

[20] K. Nakata and J. Garrigue. Recursive modules for programming. Technical Report 1546, Kyoto University Research Institute for Mathematical Sciences, 2006.

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

[22] 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`.

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

[24] C. Stone. Type definitions. In *Advanced Topics in Types and Programming Languages*, chapter 9. The MIT Press, 2004.

[25] M. Torgersen. The Expression Problem Revisited. In *European Conference on Object-Oriented Programming:LN CS*, volume 3086. Springer-Verlag, 2004.

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

[27] M. Zenger and M. Odersky. Independently Extensible Solutions to the Expression Problem. In *Proc. FOOL 12*, 2005.

Appendices

A. Path resolution algorithms

Here we define *PathExp* and *CtyReconst*.

A.1 Module path expansion algorithm

We define *PathExp* by composing *ground normalization* and *variable normalization*, which are defined below.

We define the ground normalization in Figure 23. The judgment $O, \Sigma \vdash p \rightsquigarrow_g q$ means that the ground normalization expands p into q where Σ is locked, with respect to O. We use Σ as a metavariable for sets of integers.

$$\frac{}{O, \Sigma \vdash X \rightsquigarrow_g X} \qquad \frac{}{O, \Sigma \vdash Z^\theta \rightsquigarrow_g Z^\theta}$$

$$\frac{O, \Sigma \vdash p \rightsquigarrow_g p' \quad O \vdash p'.M \mapsto (\theta, K^i) \quad K \notin ext\_mid}{O, \Sigma \vdash p.M \rightsquigarrow_g p'.M} \qquad \frac{O, \Sigma \vdash p \rightsquigarrow_g p' \quad O \vdash p'.M \mapsto (\theta, q^i) \quad q \neq X \quad O, \Sigma \uplus i \vdash q \rightsquigarrow_g r}{O, \Sigma \vdash p.M \rightsquigarrow_g \theta(r)}$$

$$\frac{O, \Sigma \vdash p_1 \rightsquigarrow_g p_1' \quad O, \Sigma \vdash p_2 \rightsquigarrow_g p_2' \quad O \vdash p_1'(p_2') \mapsto (\theta, K^i) \quad K \notin ext\_mid}{O, \Sigma \vdash p_1(p_2) \rightsquigarrow_g p_1'(p_2')}$$

$$\frac{O, \Sigma \vdash p_1 \rightsquigarrow_g p_1' \quad O, \Sigma \vdash p_2 \rightsquigarrow_g p_2' \quad O \vdash p_1'(p_2') \mapsto (\theta, q^i) \quad q \neq X \quad O, \Sigma \uplus i \vdash q \rightsquigarrow_g r}{O, \Sigma \vdash p_1(p_2) \rightsquigarrow_g \theta(r)}$$

Figure 23. Ground-normalization with respect to O

$$\begin{array}{lcl}
\eta_O(Z^\theta) & = & Z^{\theta'} \\
\multicolumn{3}{l}{\text{where } dom(\theta) = dom(\theta'),} \\
\multicolumn{3}{l}{\text{and, for all } X \in dom(\theta),\ \theta'(X) = \eta_O(\theta(X))} \\
\eta_O(X) & = & \mathrm{X} \\
\eta_O(p.M) & = & \zeta_O(\eta_O(p).M) \\
\eta_O(p_1(p_2)) & = & \zeta_O(\eta_O(p_1)(\eta_O(p_2))) \\
\zeta_O(p) & = & \begin{cases} \theta(X) & \text{when } O \vdash p \mapsto (\theta, X^i) \\ p & \text{otherwise} \end{cases}
\end{array}$$

Figure 24. Variable normalization with respect to O

$$\frac{}{P; \Sigma; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{P; \Sigma; \Gamma \vdash () : \mathbf{1}} \qquad \frac{P; \Sigma; \Gamma \vdash e_1 : \tau_1 \quad P; \Sigma; \Gamma \vdash e_2 : \tau_2}{P; \Sigma; \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{P; \Sigma; \Gamma \vdash e : \tau_1 * \tau_2}{P; \Sigma; \Gamma \vdash \pi_i(e) : \tau_i} \qquad \frac{P; \Sigma; \Gamma \vdash e_1 : \tau' \to \tau}{P; \Sigma; \Gamma \vdash e_1(e_2) : \tau} \qquad \frac{TyExp(P, \tau') = \tau}{P; \Sigma; \Gamma \vdash (\lambda x.e : \tau') : \tau}$$

$$\frac{PathExp(P, p) = p' \quad \gamma(U, p', c) = (t, \tau_1)}{P; \Sigma; \Gamma \vdash p.c\ e : p'.t}$$

$$\frac{PathExp(P, p) = p' \quad \gamma(U, p', c) = (t, \tau_1) \quad P; \Sigma; \Gamma, x : \tau_1 \vdash e_2 : \tau}{P; \Sigma; \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ p.c\ x \Rightarrow\ e_2 : \tau}$$

$$\frac{PathExp(P, p) = p' \quad P \vdash p' \mapsto (\theta, \mathtt{struct} \ldots \mathtt{val}\ l = e \ldots \mathtt{end}^i) \quad P; \Psi \uplus (i, l); \emptyset \vdash e : \tau_1 \quad TyExp(P, \theta(\tau_1)) = \tau}{P; \Sigma; \Gamma \vdash p.l : \tau}$$

$$\frac{PathExp(P, p) = p' \quad P \vdash p' \mapsto (\theta, \mathtt{sig} \ldots \mathtt{val}\ l : \tau' \ldots \mathtt{end}^i) \quad TyExp(P, \theta(\tau')) = \tau}{P; \Sigma; \Gamma \vdash p.l : \tau}$$

Figure 25. Core type reconstruction with respect to P

We define the variable normalization with respect to a top-level O using functions η_O and ζ_O, found in Figure 24.

Then we define the module path expansion algorithm *PathExp* such that it takes as argument a top-level O and a module path p, then either returns a module path q when $O, \emptyset \vdash p \rightsquigarrow_g p'$ and $\eta_O(p') = q$ hold or else raises an error when this cannot be done.

A.2 Core type reconstruction algorithm

We define the core type reconstruction algorithm in Figure 25. The judgment $P; \Sigma; \Gamma \vdash e : \tau$ means that the algorithm reconstructs the type τ for the expression e where Ψ is locked, with respect to the program P. We use Ψ as a metavariable for pairs (i, l) of an integer i and a value name l.

Then we define *CtyReconst* such that it takes as argument a program P and a core expression e, then either returns a type τ when $P; \emptyset; \emptyset \vdash e : \tau$ holds or else raises an error when this cannot be done.

B. Proof sketch of the type soundness

Here, we present our central idea for proving Proposition 6. For details, see [20].

$$\frac{U \vdash p \xrightarrow{\tau} p'}{U \vdash p.M \xrightarrow{\tau} p'.M} \qquad \frac{U \vdash p \xrightarrow{\tau} p'}{U \vdash p(q) \xrightarrow{\tau} p'(q)}$$

$$\frac{U \vdash q \xrightarrow{\tau} q'}{U \vdash p(q) \xrightarrow{\tau} p(q')} \qquad \frac{U \vdash p \mapsto (\theta, q^i)}{U \vdash p \xrightarrow{\tau} \theta(q)}$$

$$\frac{dom(\theta) = dom(\theta') \quad \exists X \in dom(\theta),\ U \vdash \theta(X) \xrightarrow{\tau} \theta'(X) \quad \forall X' \in dom(\theta) \backslash \{X\},\ \theta(X') = \theta'(X')}{U \vdash Z^\theta \xrightarrow{\tau} Z^{\theta'}}$$

Figure 26. Transition rules for module paths with respect to U

$$U \vdash \mathbf{1} \xrightarrow{\mathbf{1}} 0 \qquad U \vdash \tau_1 \to \tau_2 \xrightarrow{\mathrm{ar}_i} \tau_i \qquad U \vdash \tau_1 * \tau_2 \xrightarrow{\mathrm{prd}_i} \tau_i$$

$$\frac{U \vdash p \xrightarrow{\tau} p'}{U \vdash p.t \xrightarrow{\tau} p'.t} \qquad \frac{U \vdash X \mapsto (\theta, \mathtt{sig} \ldots \mathtt{type}\ t \ldots \mathtt{end}^i)}{U \vdash X.t \xrightarrow{X.t} 0}$$

$$\frac{U \vdash p \mapsto (\theta, \mathtt{sig} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end}^i)}{U \vdash p.t \xrightarrow{c} \theta(\tau)}$$

$$\frac{U \vdash p \mapsto (\theta, \mathtt{sig} \ldots \mathtt{type}\ t = \tau \ldots \mathtt{end}^i)}{U \vdash p.t \xrightarrow{\tau} \theta(\tau)}$$

Figure 27. Transition rules for types with respect to U

$$\frac{U \vdash_X E : T_1 \quad U \vdash_X S : T_2 \quad U \vdash_X T_1 < S}{U \vdash_X (E : S) : (T_1 : T_2)}$$

Figure 28. A typing rule for sealing with respect to U in *TraviataX*

Our proof proceeds in the following two steps.

1. We define a type system *TraviataX*, whose type equivalence relation is defined by the weak bisimulation relation on a labeled transition system on types. We establish a soundness result for *TraviataX*, by proving subject reduction and progress properties.
2. We prove that if a program P is type-correct in *Traviata*, then P is also type-correct in *TraviataX*.

We define labeled transition systems on module paths and types in Figure 26 and 27, respectively.

Typing rules in *TraviataX* are same as those in *Traviata*, except for the rule for sealing, which is given in Figure 28, and for that a type equivalence relation in *TraviataX* is defined by the largest weak bisimulation relation [17] on the labeled transition system on types. For typing rules in *TraviataX*, we use the subscript X.

To prove subject reduction in *TraviataX*, we break type abstraction in the lazy program type U that *ReconstP* reconstructed. That is, we build a lazy program type $U^\sharp$ from U by making all abstract type specifications in U manifest, except for type specifications appearing in module variable signatures. This is consistent with the typing rule in Figure 28, since, once type abstraction is broken, we do not need substitution of self variables. In [20], the reader can find how to build $U^\sharp$ from U.

PROPOSITION 7. *Assume $ReconstP(P) = U$ and $U^\sharp \vdash_X P : U^\sharp$. We have the following two results.*

- *If $U^\sharp, \emptyset \vdash_X e : \tau$ and $e \to e'$, then $U^\sharp, \emptyset \vdash_X e' : \tau$.*
- *If $U^\sharp, \emptyset \vdash_X e : \tau$ then either e is a value or else there is some e' with $e \to e'$.*

PROPOSITION 8. *Assume $ReconstP(P) = U$ and $U \vdash P : U$, then $U^\sharp \vdash_X P : U^\sharp$.*