Path resolution for recursive modules

Keiko Nakata
Research Institute for Mathematical Sciences, Kyoto University
Jacques Garrigue
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 is one of the main subjects of study. Since recursion is an essential mechanism, one is to face several non-trivial issues to be considered for designing a practical type system.

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 type system for recursive modules, which is practical and useful from the programmer's perspective.

In this paper, we present a decidable type system which can reconstruct the necessary type information during type checking of recursive modules. In particular, we develop algorithms for resolving forward references in recursive modules, by confining ourselves to first-order functors. The type system is provably sound for a call-by-value operational semantics.

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[20] and Objective Caml [18], provides a powerful module system [19, 17]. 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 [11, 15]. 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 [25].

Recently, much work has been devoted to investigating extensions with recursion of the ML module system. 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 of 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 [3], Hirschowitz and Leroy [14], and Dreyer [5] 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 proposals [4, 25, 24, 6, 18] rely on annotations from programmers; 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, enforcing type abstraction inside them, or they do but the rest of the program does not, enforcing type abstraction towards the outside. In all proposals, one has to write two different signatures for the same module to enforce abstraction towards the outside; one of the signatures is solely for assisting the type checker and does not affect the resulting signature of the module. Moreover, due to the annotation requirement programmers cannot rely on type inference during development. This is 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 following 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 [16]. 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`. 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 syntactically identical objects, but rather 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 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 harder to decide.

1.3 Potential existence of cyclic type definitions

Another subtle issue is how to account for the potential existence 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. Hence the modules `M` and `N` need not be accused of defining cyclic types. At least, one could argue that potential cycles in type definitions 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 re-

cursive 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 [6] 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 a 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 [10] and a list to represent trees and forests, respectively. (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 can make this program type checked in his system. Polymorphic variants, however, have their own merits that datatypes do not have.

The path Objective Caml [18] chose is a more liberal one. It does not care whether cyclic type definitions are hidden by opaque signatures or not, as long as the signatures themselves do not contain cycles. The type checker complains when recursive signatures specify cyclic types whenever it terminates. (Recall that applicative functors make it difficult to detect cycles in a terminating way.) Objective Caml already has a very rich core language, including structural types such as objects [23] and polymorphic variants. Moreover, the path it chose keeps flexibility in using these types and in abstracting them away by opaque signatures. This provides for an extremely expressive language, which only lacks a formal proof of soundness. We conjecture that this cannot be proven easily by a translation into an explicit type-passing system. When we make opaque signatures transparent, we may expose 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 Objective 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 stage, we confine ourselves to first-order functors. We defer it to future developments to accommodate higher-order functors by presumably adapting existing approaches.

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

To make the presentation accessible, this paper focuses on the first two of our extensions, that is, inference and detection of cycles. For a formal study, we design a calculus, called *Remonade*, for recursive modules with first-order applicative functors but without type abstraction. We present a decidable and sound type system for *Remonade*, by developing "expansion algorithms" so as to type check recursive modules without relying on signature annotations.

The expansion algorithms are the main contribution of this paper. They either resolve forward references in recursive modules or raise an error if a reference is dangling, by tracing module, type and value abbreviations in a call-by-value strategy. Although not complete with respect to a call-by-name strategy, the algorithms are provably terminating whether or not recursive modules are eventually well-typed. Using the algorithms, we design a type system for *Remonade* by extending Leroy's applicative functor calculus [16] in a straightforward way.

We defer the last part of our extension to another paper [21], in which we extend *Remonade* with type abstraction by introducing opaque signature ascription, and prove soundness of the type system. The type system allows opaque signatures to hide the potential existence of cyclic type definitions, hence programmers can use structural types liberally. In that paper, we also present an example which solves a variation on the expression problem [27], in support of our choice of applicative functors.

The rest of this paper is organized as follows. In the next section, we present the main features of *Remonade* in the context of a concrete example. In Section 3, we give the concrete syntax of *Remonade*. In Sections 4 to 6 we describe the type system. We present a soundness result in Section 7. In Section 8, we give a brief overview of an extension of *Remonade* with type abstraction. In Section 9, we examine related work. In Section 10, we conclude and give a brief overview of ongoing and future work.

2 Example

In this section, we explain difficulties involved in type checking recursive modules and informally present *Remonade*, using an example given in Figure 1.

The top-level module `TreeForest` contains three modules `S`, `Tree` and `Forest`: `S` is an abbreviation for the module `IntegerSet`, which we assume to be given in a library for making sets of integers; `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. In the example, 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.

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.labels` and `Forest.labels`. These functions calculate the set of integers a tree and a forest contain, respectively.

To enable forward references, we extend structures with *self variables*. Components of the structure can refer to each other recursively, using the self variable. For instance `TreeForest` declares a self variable `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 the function `Tree.labels` 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.

The function `Tree.labels` calls the function `Forest.labels` using the *path* `F.labels`. To type check `Tree.labels`, a type system needs to know the type of `F.labels`, although `Forest.labels` is not yet defined at the point where `Tree.labels` is defined. Yet, if the type system knew that the path `F.labels` has the type `Forest.t` $\rightarrow$ `S.t`, then it can type check `Tree.labels` in a standard way, by putting the binding `F.labels : Forest.t` $\rightarrow$ `S.t` into the type environment.

The critical part of the example lies in the definition of the function `Tree.split`, which cuts off the root node of a given tree and returns the resulting forest. To type check the function, a type system needs to know that the constructor `Node` contains a pair `(i, f)` of an integer `i` and a list of trees `f` so as to ensure that `(Leaf i) :: f` is a well-typed expression. The definition of the type `Tree.t` describes that `Node` contains a pair of an integer and a forest of type `F.t`. The type system should easily find out that the type `F.t` is an abbreviation for the type `TF.Forest.t`, by tracing backward references. According to the definition order, however, the underlying implementation of the type `TF.Forest.t` is not known at the point where the type

`Tree.t` and the function `Tree.split` are defined. Hence, to type check `Tree.split`, the type system has to foresee part of the definition of the module `Forest` so as to find out the underlying implementation of the type `TF.Forest.t`. Permutation of the definitions does not work. To type check the function `Forest.sweep`, the type system needs to know the underlying implementation of the type `Tree.t`. The function `sweep` gathers the leaves from a given forest.

2.1 Existing proposals

To type check the module `TreeForest`, existing type systems require programmers to write signature annotations, as we will examine below.

To avoid presenting too much annotations, we remove the module abbreviation `module F = TF.Forest` inside `Tree`. Yet, although we can dispense with abbreviations by replacing them with their definitions, abbreviations are useful in practical programs [26].

To type check `TreeForest` in Dreyer's system [6] or Objective Caml [18], it must come with fully transparent signature annotations of the modules `Tree` and `Forest`, that is, one has to present the type system with the following signatures:

```
module Tree : sig
 datatype t = Leaf of int | Node of int * Forest.t
 val labels : t → S.t
 val split : t → Forest.t
end
```

and

```
module Forest : sig
 type t = Tree.t list
 val labels : t → S.t
 val incr : Tree.t → t → t
 val sweep : t → t
end
```

In Russo's system [25], the self variable `TF` of `TreeForest` must be annotated with the recursive signature below. In his system, a recursive signature contains a typed declaration of a self variable to support forward references in the 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 int | Node of int * Z.Forest.t end
  module Forest : sig
   type t = Tree.t list val labels : t → S.t end
end
```

Signature annotations are indispensable in existing proposals and must be given before type checking `TreeForest`. We note that when `Tree` contains the module abbreviation `module F = TF.Forest` as in Figure 1, the required annotations must expose the signature of `Forest` twice; once for the abbreviation and once for the module `Forest` itself.

2.2 Our approach

The type system presented in this paper can type check `TreeForest` without relying on signature annotations from programmers.

The idea of our type system is simple. We develop algorithms for resolving forward references in recursive modules. Precisely, we develop "expansion algorithms" to determine the component that a path refers to. Then type checking of `TreeForest` is straightforward. Moreover, the module abbreviation `module F = TF.Forest` can be used inside `Tree` with no harm. For instance, using the expansion algorithms, the type system finds out that the path `F.t`, used inside the definition of the type `Tree.t`, refers to the type `Tree.t list`. Thus the difficulty involved in type checking `Tree.split` is resolved.

We devote Sections 4 to 6 to explain the type system. The typing rules are not our novelty; we develop them by extending Leroy's applicative functor calculus [16] in a straightforward way. Our novelty is the expansion algorithms, which are provably terminating and coincide with the intuitive expansion, defined in Section 7, when recursive modules are well-typed. We give detailed explanations on these algorithms in this paper.

2.3 Contribution of our type system

The strength of our type system is that it can reconstruct the necessary type information by itself during type checking, instead of relying on annotations from programmers. We do not intend to argue that signatures are useless. Signatures

are useful for many reasons; they give means of controlling type abstraction, and serve as documentation. We actually use them in our system to provide flexible type abstraction in the presence of recursive modules.

Suppose that one wants to give the module `TreeForest` the following opaque signature, by hiding implementations of the types `Tree.t` and `Forest.t` and the functions `Tree.labels` and `Forest.labels`.

```
sig
 module Tree : sig type t val split : t → Forest.t end
 module Forest : sig
  type t val incr : Tree.t → t → t val sweep : t → t end
end
```

This opaque signature would have a perfect sense for programmers, but might not be informative enough to assist a type checker. Indeed, in existing type systems, one has to write this opaque signature in addition to either the fully transparent ones or the annotation on the self variable which we examined in Section 2.1. The opaque signature is useful for programmers. But the transparent ones and the annotation on the self variable are redundant; they serve only in assisting the type checker and do not affect the resulting signature of `TreeForest`.

Unlike existing type systems, our type system does not need the assistance of annotations. Hence the above opaque signature is sufficient to type check `TreeForest` and to enforce type abstraction. In short, our type system can check that `TreeForest` inhabits the signature, using the result of type reconstruction in a straightforward way. Since type abstraction is not in the scope of this paper, we do not give further details of how the type system type checks `TreeForest` with the opaque signature given. We refer interested readers to another paper [21].

The example of this section does not cover the issue of detecting cyclic type specifications in signatures. Since we do not consider type abstraction in this paper nor require programmers to write signatures of recursive modules, we want to reject programs which contain cyclic type definitions in structures. Indeed, we first developed the expansion algorithms for detecting cyclic types, since we wanted to define a decidable judgment for type equality, and cyclic types may make it undecidable. We later noticed that applying the same idea, we can enable type inference for recursive modules. We will revisit the issue of detecting cyclic types later in Sections 5 and 6.

```
moduleTreeForest= struct(TF)
 module S = IntegerSet
 module Tree = struct
  module F = TF.Forest
  datatype t = Leaf of int | Node of int * F.t
  val labels = fun x → case x of
       Leaf i ⇒ TF.S.singlton i
     | Node(i, f) ⇒ TF.S.add i (F.labels f)
  val split = fun x → case x of
       Leaf i ⇒ [Leaf i]
     | Node(i, f) ⇒ (Leaf i) ::  f
 end
 module Forest = struct
  module T = TF.Tree
  type t = T.t list
  val labels = fun x → case x of [] ⇒ TF.S.empty
     | hd ::  tl ⇒ TF.S.union (T.labels hd) (labels tl)
  val incr = fun x → fun y →
     let l = T.labels x in
     let l' = labels y in
     if TF.S.subset l' l then (x ::  y) else y
  val sweep= fun x → case x of [] ⇒ []
     | (T.Leafy) ::tl   ⇒ (T.Leaf y) ::  (sweep tl)
     | (T.Nodey) ::  tl ⇒ sweep tl
 end
end
```

Figure 1: Modules for trees and forests

Module expr.	E	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	*structure*
		|	$\texttt{functor}(X : S) \to E$	*functor*
		|	*mid*	*module ident.*
		|	X	*module var.*
Definitions	D	::=	$\texttt{module}\ M = E$	*module def.*
		|	$\texttt{val}\ l = e$	*value def.*
		|	$\texttt{datatype}\ t = T$	*datatype def.*
		|	$\texttt{type}\ t = \tau$	*type abbrev.*
Signature	S	::=	$\texttt{sig}\ B_1 \ldots B_n\ \texttt{end}$	
Specifications	B	::=	$\texttt{type}\ t = \tau$	*manifest type spec.*
		|	$\texttt{type}\ t$	*opaque type spec.*
		|	$\texttt{val}\ l : \tau$	*value spec.*
Recursive ident.	*rid*	::=	$Z \mid rid.M$	
Module ident.	*mid*	::=	$rid \mid mid_1(mid_2) \mid mid(X)$	
Extended ident.	*ext_id*	::=	$Z \mid ext_id.M \mid ext_id_1(ext_id_2) \mid ext_id(X)$	
Module paths	p, q, r	::=	$ext_id \mid X$	
Program	P	::=	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	

Figure 2: Syntax for the module language

3 Syntax

We give the syntax for our module language in Figure 2. It is based on Leroy's module calculus with manifest types [15]. 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 with implicitly typed declarations of self variables to support recursive references between modules. In the construct $\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$, the self variable Z is bound in $D_1 \ldots D_n$, and Z itself is bound to $\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$.

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, 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.

Core types	τ	::=	$\mathtt{1} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid p.t$
Datatype definition	T	::=	$c\ \mathtt{of}\ \tau$
Core expressions	e	::=	$x \mid () \mid (\mathtt{fun}\ x \to e : \tau) \mid (e_1, e_2) \mid \pi_i(e)$
		$\mid$	$e_1(e_2) \mid rid.c\ e \mid \mathtt{case}\ e\ \mathtt{of}\ ms \mid rid.l \mid X.l$
Matching clause	ms	::=	$rid.c\ x \Rightarrow\ e$

Figure 3: Syntax for the core language

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

To support applicative functors [16], we define a slightly extended class of identifiers, named *module paths* in Figure2, which can liberally include functor applications. Core types defined in Figure 3 may use module paths. Applicative functors give us more flexibility in expressing type sharing constraint between recursive modules. In [21], we give a practical example which uses both recursive modules and applicative functors in support of our design choice. Note that module paths include the syntactic objects *rid* and *mid*. We may use the metavariables p, q and r to mean these objects together.

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 arguments, 2) nor to access inner modules of arguments. The first condition means that our functors are first-order, and the second implies that one has 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 expressions excludes those of the forms $X.M$ and $X(mid)$, and that signatures do not contain module specifications. This is due to the restriction.

The module language does not support means of type abstraction, which is one of the critical features of the ML module system. As we mentioned in Section 1, this paper focuses on type reconstruction for recursive modules. In Section 8, we give a brief overview of an extension of *Remonade* with type abstraction. From a technical point of view, the extension is orthogonal to the development in this paper.

We gives the syntax for our core language in Figure 3. We use x as a metavariable for ranging over program variables (variables, for short), and c for ranging over value constructor names.

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 a soundness result, as the syntax of paths is kept uniform, that is, every path is prefixed by either a self variable or a module variable. Moreover there are situations where this convention is useful for practical programming [21].

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 declares a self variable; 3) a path is always prefixed by either 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 of the core language. The original program may still require some type annotations, to avoid running into the polymorphic recursion problem. In Section 10, we discuss the details of this inference algorithm. The elaboration phase also infers omitted self variables, to complete implicit backward references.

We assume that the following three conditions hold: 1) module variables and self variables in a program differ from each other; 2) a program does not contain free module variables nor self variables; 3) any sequence of module definitions, type abbreviations, datatype definitions, value definitions, transparent and opaque type specifications, and value specifications does not contain duplicate definitions nor specifications for the same name.

4 Expanding module paths

In this section, we present a module path expansion algorithm for determining the module that a module path refers to. We use the algorithm to obtain the necessary type information about module paths during type checking and to define a type expansion algorithm and a type reconstruction algorithm in Sections 5 and 6, respectively.

In the rest of the paper, all judgments and predicates are relative to a complete source program P, which is omitted in notations. All proofs are valid for any P.

We make the following three assumptions.

1. All occurrences of module expressions and signatures in the program P are labeled with distinct integers. We write E^i and S^j when the module expression E and the signature S are labeled with the integers i and j, respectively. One may think of the integer label i of E^i as the location of E in P. We use Σ as a metavariable for ranging over sets of integers, and write Σ_P to mean the set of integer labels appearing in P. Note that Σ_P is finite.

   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 $(\texttt{functor(X : sig type } t \texttt{ end}^1) \rightarrow \texttt{ X}^2)^3$.

2. There is a global mapping Δ, which sends i) a self variable Z to the structure to which Z is bound, and ii) a module variable X to the signature specified for X. We could avoid this assumption of a global mapping by including this information in the look-up judgment defined in Figure 4. Yet, this assumption makes the presentation concise.

3. Each self variable Z is superscripted 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 occurrence of a self variable in the program P is implicitly superscripted with the identity substitution *id*. That is, we regard Z as an abbreviation for Z^{id}. We use θ as a metavariable ranging over module variable environments.

The module path expansion algorithm either reduces a module path into a *located form* or raise an error if this cannot be done.

A located form is a module path which refers to a structure or a functor in the program P. To give the formal definition, we define *look-up judgment* in Figure 4.

[**lk-sf**]

$$\vdash Z^{\theta} \mapsto (\theta, \Delta(Z))$$

[**lk-dot**]

$$\frac{\vdash p \mapsto (\theta, \texttt{struct} \ldots \texttt{module}\ M = E^{j} \ldots \texttt{end}^{i})}{\vdash p.M \mapsto (\theta, E^{j})}$$

[**lk-app**]

$$\frac{\vdash p_1 \mapsto (\theta, (\texttt{functor}(X\ :\ S^{j}) \rightarrow E^{k})^{i})}{\vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], E^{k})}$$

Figure 4: Look-up

```
struct (Z)
 module M1 = (functor(X : sig type t end^3) →
  struct
   module M11 = struct end^5
   module M12 = X^6
  end^4)^2
 module M2 = struct type t = int end^7
 module M3 = Z.M1(Z.M2)^8
end^1
```

Figure 5: A program P_1

The judgment $\vdash p \mapsto (\theta, E^{i})$ is read that the module path p refers to the module expression E labeled with the integer i, where each module variable X is bound to $\theta(X)$. Note that the judgment implicitly depends on P.

Let us examine each rule of the look-up. For a self variable, the judgment consults the global mapping Δ([**lk-sf**]). A path $p.M$ refers to the module component M in the structure that p refers to([**lk-dot**]). A path $p_1(p_2)$ refers to the body of the functor that p_1 refers to, where the module variable environment is augmented with the new binding $[X \mapsto p_2]$([**lk-app**]). Note that our assumption of the absence of free module variables means that when $\vdash p \mapsto (\theta, q^{i})$, then $MVars(q) \subseteq dom(\theta)$. For a module variable environment θ, $dom(\theta)$ denotes the domain of θ.

Then located forms are defined as follows.

Definition 1 *A module path p is in located form if and only if either of the following two conditions holds.*

1. *p is a module variable.*

2. *The following two conditions hold.*

   - $\vdash p \mapsto (\theta, E^i)$ *where E is neither a module identifier nor a module variable.*
   - *For all q in $args(p)$, q is in located form.*

For a module path p, $args(p)$ denotes the set of module paths to which p is applied, or:

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

We say that a module variable environment θ is in located form if and only if, for all X in $dom(\theta)$, $\theta(X)$ is in located form.

For instance, consider the program P_1 in Figure 5. The module path $\texttt{Z.M}_1\texttt{(Z.M}_2\texttt{).M}_{11}$ is in located form, since $\vdash \texttt{Z.M}_1\texttt{(Z.M}_2\texttt{).M}_{11} \mapsto (\texttt{[X} \mapsto \texttt{Z.M}_2\texttt{]}, \texttt{struct end}^5)$ holds, but $\texttt{Z.M}_3\texttt{.M}_{11}$ is not. (We need to expand $\texttt{Z.M}_3$ first to make $\texttt{Z.M}_3\texttt{.M}_{11}$ located form.)

The basic idea of the module path expansion algorithm is straightforward; the algorithm traces module abbreviations until it meets a structure or a functor. To keep the algorithm terminating, we have to be careful about the potential existence of cyclic module definitions. Below we give two pathological examples which contain cyclic definitions.

To reduce notational burden, we may omit, in examples here and elsewhere, preceding self variables even for forward references, when no ambiguity arises. Moreover, we may omit the top-level `struct` and `end`.

The first example is:

```
module F = functor(X : sig end) → X
module L = F(L)
```

Through the identity functor `F`, the definition of `L` makes a cycle. The second one is:

```
module M = M.N
```

The second example is more annoying for us than the first one, since the unrolling of `M`'s definition could result in the following infinite rewriting sequence, yielding module paths of arbitrary long length.

```
M → M.N → M.N.N → M.N.N.N → ...
```

We design the module path expansion algorithm separately from the type system. The type system, to be defined in Section 6, is based on Leroy's applicative functor calculus [16]. Essentially, it differs from Leroy's in that it uses the algorithm to obtain the necessary type information about module paths instead of consulting

$$\frac{\vdash p \rightsquigarrow_g q}{\vdash p \rightsquigarrow_s \eta(q)}$$

Figure 6: Semi-ground normalization

a type environment, in order to reason about forward references. To keep the type system decidable, we design the algorithm to be terminating for any program whether or not the program is eventually well-typed.

In the rest of this section, we detail the algorithm for expanding module paths.

4.1 Semi-ground normalization

To expand module paths, we define *semi-ground normalization*, which appears in Figure 6. The semi-ground normalization is defined by composing two normalizations, namely *ground normalization* and *variable normalization*. The inference rule for the semi-ground normalization denotes that if the ground normalization reduces p into q, written $\vdash p \rightsquigarrow_g q$, then the variable normal form of q, written $\eta(q)$, is a located form of p. We say that q is a located form of p when $\vdash p \rightsquigarrow_s q$.

The ground normalization and the variable normalization are defined below. They are provably terminating. As a result, the semi-ground normalization is terminating.

4.2 Ground normalization

The ground normalization is a normalization which is essentially ground, that is, it does not depend on functor arguments. It either reduces a module path into a *pre-located form* or raises an error when this cannot be done.

4.2.1 Pre-located forms

We first define pre-located forms, the central idea for defining the terminating ground normalization.

Definition 2 *A module path p is in pre-located form if and only if either of the following two conditions holds.*

1. *p is a module variable.*
2. *The following two conditions hold.*

- $\vdash p \mapsto (\theta, E^i)$ *and* E *is not a module identifier. (Hence* E *can be a module variable.)*
- *For all* q *in* $args(p)$, q *is in pre-located form.*

The locution "pre-located form" indicates that we can turn a pre-located form into a located form by substituting functor arguments, as we formally state in Lemma 4 in Section 4.3.

We say that a module variable environment θ is in pre-located form if and only if, for all X in $dom(\theta)$, $\theta(X)$ is in pre-located form.

The important feature of pre-located forms is that they satisfy a substitution property, as stated in Lemma 1 below. Here we first define length of module paths as follows:

$$|X| = 1 \qquad |p.M| = 1 + |p| \qquad |p(q)| = |p| + |q|$$

$$|Z^{\theta}| = 1 + |\theta(X_1)| + |\theta(X_2)| + \ldots |\theta(X_n)| \text{ where } dom(\theta) = \{X_1, X_2, \ldots, X_n\}$$

For a module path p and a module variable environment θ with $MVars(p) \subseteq dom(\theta)$, we write $\theta(p)$ to denote the module path obtained by applying the substitution θ to p, or:

$$\theta(p.M) = \theta(p).M \quad \theta(p_1(p_2)) = \theta(p_1)(\theta(p_2)) \quad \theta(Z^{\theta'}) = Z^{\theta \circ \theta'}$$

We write $\theta \circ \theta'$ to denote the composition of the two substitutions θ and θ'.

Lemma 1 (Substitution property) *Let* p *and* θ *be in pre-located form and* $MVars(p) \subseteq dom(\theta)$. *Then* $\theta(p)$ *is in pre-located form.*

Proof. By induction on the length of p. □

We also use the following lemma to define the ground normalization.

Lemma 2 *Let* p *be in pre-located form. If* $\vdash p \mapsto (\theta, E^i)$, *then* θ *is in pre-located form.*

Proof. By induction on the derivation of $\vdash p \mapsto (\theta, E^i)$. □

It is an important observation that Lemma 1 holds due to the fist-order structure restriction. If functors took nested arguments, then the module path `[X ↦ L]X.M` would not be in pre-located form in the program:

```
module F = functor(X : sig module M : sig end end) →
  struct module M = X.M end
module L = struct module N = struct end module M = N end
```

Note that the module variable environment `[X ↦ L]` is in pre-located form, but the module path `L.M` is not (because `L.M` refers to a module identifier).

[gnlz-mv]

$$\frac{}{\Sigma \vdash X \rightsquigarrow_g X}$$

[gnlz-sf]

$$\frac{}{\Sigma \vdash Z^{\theta} \rightsquigarrow_g Z^{\theta}}$$

[gnlz-exp1]

$$\frac{\Sigma \vdash p \rightsquigarrow_g p' \quad \vdash p'.M \mapsto (\theta, E^i) \quad E \notin mid}{\Sigma \vdash p.M \rightsquigarrow_g p'.M}$$

[gnlz-pth1]

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

[gnlz-exp2]

$$\frac{\Sigma \vdash p_1 \rightsquigarrow_g p_1' \quad \Sigma \vdash p_2 \rightsquigarrow_g p_2' \quad \vdash p_1'(p_2') \mapsto (\theta, E^i) \quad E \notin mid}{\Sigma \vdash p_1(p_2) \rightsquigarrow_g p_1'(p_2')}$$

[gnlz-pth2]

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

Figure 7: Ground-normalization

4.2.2 Ground normalization

We present inference rules for the ground normalization in Figure 7. The judgment $\Sigma \vdash p \rightsquigarrow_g q$ means that the ground normalization expands p into q with Σ locked. We may say that q is a pre-located form of p when $\Sigma \vdash p \rightsquigarrow_g q$ holds for some Σ. The notation $\Sigma \uplus i$ means $\Sigma \cup \{i\}$ whenever $i \notin \Sigma$.

We regard the ground normalization as an algorithm which takes a module path p and a lock Σ as inputs, then either returns a pre-located form or raises an error if there is no applicable rule. Note that derivations of the ground normalization are deterministic. We may write $\vdash p \rightsquigarrow_g q$ to mean $\emptyset \vdash p \rightsquigarrow_g q$.

Let us examine each rule. The first two rules [**gnlz-mv**] and [**gnlz-sf**] are straightforward. For a path of the form $p.M$, the ground normalization first expands the prefix p ([**gnlz-exp1**][**gnlz-pth1**]). Suppose that p' is a pre-located form of p. Then there are two cases depending on whether $p'.M$ refers to a module identifier or not. When $p'.M$ refers to a module expression other than a module identifier ([**gnlz-exp1**]), then $p'.M$ is in pre-located form and the ground normalization terminates. When $p'.M$ refers to a module identifier q ([**gnlz-pth1**]), then the ground normalization traces the abbreviation q. This is the key rule, hence we explain it in detail.

As a simple case, suppose that q is in pre-located form. Then $\Sigma \uplus i \vdash q \rightsquigarrow_g q$ holds immediately whenever i is not in Σ (see Lemma 6), and the ground normalization returns $\theta(q)$. By Lemma 1 and 2, we are sure that $\theta(q)$ is in pre-located form. In general, q is not necessarily in pre-located form. Hence, the ground normalization expands q first to obtain its pre-located form in the premise $\Sigma \uplus i \vdash q \rightsquigarrow_g r$, then

apply the substitution θ to r.

This explains the idea of the ground normalization. It additionally holds a lock Σ during the expansion for termination. In short, when the ground normalization holds a lock Σ, then it is in the middle of expansion of the module paths labeled with the integers in Σ. The rules [**gnlz-pth1**] and [**gnlz-pth2**] have the side condition $i \notin \Sigma$ implicitly; thanks to the condition, the ground normalization can avoid tracing the same module abbreviation cyclically.

The rules [**gnlz-exp2**] and [**gnlz-pth2**] for paths of the form $p_1(p_2)$ are similar to [**gnlz-exp1**] and [**gnlz-pth1**], respectively.

To understand the ground normalization, it may be useful to think of it as a kind of partial evaluation. Indeed, in the premise $\Sigma \uplus i \vdash q \rightsquigarrow_g r$, the rules [**gnlz-pth1**] and [**gnlz-pth2**] expand q as far as they can without using functor arguments. Once this "partial expansion" is done, the rules apply substitution to replace formal parameters of functors with corresponding actual arguments. (And, by Lemma 1 and 2, the substitution produces a pre-located form, hence the ground normalization need not continue the expansion after the substitution.)

4.2.3 Well-definedness and termination

Here we show that the ground normalization does reduce module paths into pre-located forms unless it raises an error and that it defines an algorithm which is terminating.

We first define a sanity condition which we assume to hold for all input module paths to the ground normalization.

Definition 3 *A module path p has pre-located variables if and only if all the self variables contained in p are in pre-located form.*

Note that Z^{id} is in pre-located form. Hence, all the module paths appearing in the program P are appropriate for the input to the ground normalization.

Proposition 1 (Well-definedness of the ground normalization) *Let p have pre-located variables. If $\Sigma \vdash p \rightsquigarrow_g q$, then q is in pre-located form.*

Proof. By induction on the derivation of $\Sigma \vdash p \rightsquigarrow_g q$ and by case on the last rule used. Use Lemma 1 and 2. □

Now we show termination of the ground normalization. Our proof proceeds by defining well-founded relations.

$$
\begin{array}{lcl}
\eta(Z^\theta) & = & Z^{\eta\circ\theta} \\
\eta(X) & = & \mathrm{X} \\
\eta(p.M) & = & \zeta(\eta(p).M) \\
\eta(p_1(p_2)) & = & \zeta(\eta(p_1)(\eta(p_2)))
\end{array}
\qquad
\zeta(p) \;=\; \begin{cases} \theta(X) & \text{when } \vdash p \mapsto (\theta, X^i) \\ p & \text{otherwise} \end{cases}
$$

Figure 8: Variable normalization

Definition 4 *A binary relation $\mathcal{R}$ on any set is well-founded if and only if there is no infinitely descending sequence in $\mathcal{R}$, that is, there is no sequence $\{r_i\}_{i=1}^{\infty}$ such that, for all i in $1, 2, \ldots$, $r_i \;\mathcal{R}\; r_{i+1}$ holds.*

Proposition 2 (Termination of the ground normalization) *For any module path p and lock Σ, proof search for $\Sigma \vdash p \rightsquigarrow_g _$ will terminate.*

Proof. Below, we define a well-founded relation $>_g$ on pairs (p, Σ) of a module path p and a lock Σ. It is easy to check that if $\Sigma_2 \vdash p_2 \rightsquigarrow_g _$ is a premise of $\Sigma_1 \vdash p_1 \rightsquigarrow_g _$, then $(p_1, \Sigma_1) >_g (p_2, \Sigma_2)$. Thus, if there is an infinitely deep derivation tree of the ground normalization, then there is an infinitely descending sequence in $>_g$. This contradicts well-foundedness of $>_g$. By Köning's lemma on finitely branching trees, we obtain the proposition.

$(p_1, \Sigma_1) >_g (p_2, \Sigma_2)$ holds if and only if either of the following three conditions holds.

1. $p_1 = p_1'.M$ and $p_2 = p_1'$ and $\Sigma_1 = \Sigma_2$.
2. $p_1 = p_{11}(p_{12})$ and $p_2 = p_{1i}$ and $\Sigma_1 = \Sigma_2$.
3. i is not in Σ_1 and $\Sigma_2 = \Sigma_1 \cup \{i\} \subseteq \Sigma_P$

Well-foundedness of $>_g$ is shown by the finiteness of Σ_P. □

4.3 Variable normalization

The variable normalization turns pre-located forms into located forms. We define the variable normalization using functions η and ζ on pre-located forms, which are found in Figure 8. We write $\eta \circ \theta$ to denote a module variable environment θ' such that $dom(\theta) = dom(\theta')$ and, for all X in $dom(\theta)$, $\eta(\theta(X)) = \theta'(X)$.

Given a module path p in pre-located form, functions η and ζ recursively replace each module path q contained in p with the corresponding functor argument if q refers to a module variable.

Lemma 3 below is proven by easy induction. Lemma 4 indicates that by combining the ground normalization and the variable normalization, we obtain located forms.

Lemma 3 *Let p be in located form. If $\vdash p \mapsto (\theta, E^i)$, then θ is in located form.*

Lemma 4 *Let p be in pre-located form. Then the computation of $\eta(p)$ terminates returning a module path in located form.*

Proof. By induction on the length of p. Use Lemma 3. □

Returning to the example in Figure 5, the ground normalization reduces $\mathtt{Z.M_3.M_{12}}$ into $\mathtt{Z.M_1(Z.M_2).M_{12}}$. Then the variable normalization reduces $\mathtt{Z.M_1(Z.M_2).M_{12}}$ into $\mathtt{Z.M_2}$. As a whole, we have $\vdash \mathtt{Z.M_3.M_{12}} \rightsquigarrow_s \mathtt{Z.M_2}$.

What is good for us about the stratification of the semi-ground normalization into the ground normalization and the variable normalization is that we can enforce the first-order structure restriction without relying on the type system. For instance, in Figure 5, the ground normalization fails in expanding the module path $\mathtt{Z.M_3.M_{12}.M}$. The fist-order structure restriction, in turn, enables us to define the terminating ground normalization. Once we are certain that the semi-ground normalization is terminating, we can safely use it in the type system. Moreover, thanks to the restriction, the semi-ground normalization coincides with the intuitive normalization, which is defined in Section 7, when the program P is well-typed. This result appears in Proposition 10.

4.4 Termination and well-definedness of the semi-ground normalization

In this section, we show that the semi-ground normalization defines an algorithm which is terminating, and that it does reduces module paths into located forms unless the ground normalization raises an error. We also present some useful lemmas that are used later in the paper.

Proposition 3 (Termination of the semi-ground normalization) *For any module path p having pre-located variables, proof search for $\vdash p \rightsquigarrow_s _$ will terminate.*

Proof. The proposition is an immediate consequence of Proposition 2 and Lemma 4. □

Proposition 4 (Well-definedness of the semi-ground normalization) *Let p have pre-located variables. If $\vdash p \rightsquigarrow_s q$, then q is in located form.*

Proof. By hypothesis, we have $\vdash p \rightsquigarrow_g p'$ and $\eta(p') = q$. By Proposition 1, p' is in pre-located form. By Lemma 4, q is in located form. □

The following lemmas are shown by easy induction.

Lemma 5 *Let p and θ be in located form. Then $\theta(p)$ is in located form.*

Lemma 6 *Let p be in pre-located form. Then $\Sigma \vdash p \rightsquigarrow_g p$ for any Σ.*

Lemma 7 *Let p be in located form. Then $\eta(p) = p$.*

Lemma 8 *Let p be in located form. Then $\vdash p \rightsquigarrow_s p$.*

Proof. By Lemma 6 and 7. □

It is a useful observation that located forms are invariant of the semi-ground normalization, the ground normalization and the variable normalization, and that pre-located forms are invariant of the ground normalization.

5 Expanding types

In this section, we present an algorithm for expanding types. The aim of the type expansion is to reduce types into canonical forms so that we can define a type equivalence relation in a syntactic way. For instance, in Figure 1, the algorithm reduces the type `F.t` used inside `Tree` into `TF.Tree.t list`.

The type expansion algorithm reduces types into *located types*. Located types are defined in terms of *simple located types*.

Definition 5 *A simple located type is either* `1`*, or* $p.t$ *where either of the following two conditions holds.*

1. $p = X$ *and* $\Delta(X) = \texttt{sig}\ \ldots\ \texttt{type}\ t\ \ldots\texttt{end}^i$.
2. p *is in located form and* $\vdash p \mapsto (\theta, \texttt{struct}\ldots\texttt{datatype}\ t = c\ \texttt{of}\ \tau\ldots\texttt{end}^i)$.

Then, located types are composed of simple located types.

Definition 6 *A located type is a type* τ *where each type* τ' *in* $cmpnt(\tau)$ *is a simple located type.*

For a type τ, $cmpnt(\tau)$ denotes the set of types from which τ is constructed. Precisely,

$$cmpnt(\tau) = \begin{cases} cmpnt(\tau_1) \cup cmpnt(\tau_2) & \text{when } \tau = \tau_1 \to \tau_2 \text{ or } \tau = \tau_1 * \tau_2 \\ \{\tau\} & \text{otherwise} \end{cases}$$

We develop the type expansion algorithm separately from the type system, as we did for the semi-ground normalization. The separation is useful for 1) having intuitive typing rules (although the expansion algorithm is somewhat involved, the typing rules to be defined are straightforward) and for 2) accommodating a possible extension of the algorithm, that is, when we come up with a cleverer algorithm, we can replace the current one with the new one without changing the rest of the type system.

One may think that we can apply the same idea as the semi-ground normalization for developing a type expansion algorithm. Recall the first-order structure restriction we imposed on functors. If we are going to put a similar restriction, the restriction would require functors not to access type components of functor arguments. This seems too restrictive. Hence we define another algorithm on top of the semi-ground normalization to expand types.

$$\textbf{[t-uni]}\ \frac{}{\Omega \vdash 1 \downarrow 1} \qquad \textbf{[t-arr]}\ \frac{\Omega \vdash \tau_1 \downarrow \tau_1' \quad \Omega \vdash \tau_2 \downarrow \tau_2'}{\Omega \vdash \tau_1 \to \tau_2 \downarrow \tau_1' \to \tau_2'} \qquad \textbf{[t-pair]}\ \frac{\Omega \vdash \tau_1 \downarrow \tau_1' \quad \Omega \vdash \tau_2 \downarrow \tau_2'}{\Omega \vdash \tau_1 * \tau_2 \downarrow \tau_1' * \tau_2'}$$

$$\textbf{[t-dtyp]}\ \frac{\vdash p \rightsquigarrow_s p' \quad \vdash p' \mapsto (\theta, \texttt{struct} \ldots \texttt{datatype}\ t = c\ \texttt{of}\ \tau \ldots \texttt{end}^i)}{\Omega \vdash p.t \downarrow p'.t}$$

$$\textbf{[t-typ]}\ \frac{\vdash p \rightsquigarrow_s p' \quad \vdash p' \mapsto (\theta, \texttt{struct} \ldots\ \texttt{type}\ t = \tau_1 \ldots\ \texttt{end}^i) \quad \Omega \uplus (i,t) \vdash \tau_1 \downarrow \tau_2 \quad \Omega \vdash \theta(\tau_2) \downarrow \tau}{\Omega \vdash p.t \downarrow \tau}$$

$$\textbf{[t-opq]}\ \frac{\vdash p \rightsquigarrow_s X \quad \Delta(X) = \texttt{sig} \ldots\ \texttt{type}\ t\ \ldots \texttt{end}^i}{\Omega \vdash p.t \downarrow X.t}$$

$$\textbf{[t-tran]}\ \frac{\vdash p \rightsquigarrow_s X \quad \Delta(X) = \texttt{sig} \ldots\ \texttt{type}\ t = \tau_1 \ldots\ \texttt{end}^i \quad \Omega \uplus (i,t) \vdash \tau_1 \downarrow \tau}{\Omega \vdash p.t \downarrow \tau}$$

Figure 9: Type expansion

5.1 Type expansion

We present inference rules for the type expansion algorithm in Figure 9. The judgment $\Omega \vdash \tau \downarrow \tau'$ is read that the algorithm expands τ into τ' with Ω locked. We use Ω as a metavariable for ranging over sets of pairs (i, t) of an integer i and a type name t. Note that the derivations of the type expansion are deterministic. The algorithm takes a lock Ω and a type τ as inputs, then either returns a located type τ' or raises an error when there is no applicable rule. We may write $\vdash \tau \downarrow \tau'$ to mean $\emptyset \vdash \tau \downarrow \tau'$.

Let us examine each rule. The first three rules **[t-uni]**, **[t-arr]** and **[t-pair]** are straightforward. For the type $p.t$, the algorithm first expands its prefix p using the semi-ground normalization, in order to determine the module that p refers to (**[t-dtyp]**[**t-typ**][**t-opq**][**t-tran**]). Suppose that p' is a located form of p and is not a module variable. Then there are two cases depending on the definition of the type $p'.t$. When $p.'t$ is a datatype (**[t-dtyp]**), the algorithm terminates and returns $p'.t$, which is a located type. When $p'.t$ is an abbreviation for another type τ_1 (**[t-typ]**), the algorithm needs to compute the expansion of $\theta(\tau_1)$. This rule is most important. The rule says that, to obtain a located type of $\theta(\tau_1)$, 1) check that (i, t) is not in the

current lock; 2) then expand τ_1 under the augmented lock $\Omega \uplus (i, t)$, without applying the substitution θ to τ_1; 3) apply the substitution θ to the newly obtained type τ_2, then expand $\theta(\tau_2)$ under the original lock Ω. Compare the rule [**t-typ**] to the rule [**gnlz-pth1**] of the ground normalization. Both handle abbreviations and have similar premises except that the type expansion continues after applying the substitution θ to the newly obtained type τ_2, while the ground normalization terminates immediately after applying the substitution θ to the newly obtained path r. Since located types do not satisfy a substitution property like module paths in located form do, it does not necessarily hold that applying a substitution θ in located form to a located type produces a located type. Due to this difference, the type expansion appears to be more involved than the ground normalization. We first examine a simple case in detail below, to give an intuition of the type expansion. Then we review key cases by giving concrete examples in Example 1 and 2.

The rules [**t-opq**] and [**t-tran**] are similar to, but simpler than the rules [**t-dtyp**] and [**t-typ**], respectively.

First, we present two useful lemmas about the type expansion algorithm. Lemma 9 gives a weakened substitution property simple located types satisfy; Lemma 10 shows that located types are invariant of the type expansion algorithm.

Lemma 9 (Weak substitution property) *Let τ be a simple located type and θ be in located form and $MVars(\tau) \subseteq dom(\theta)$. Then either of the following two conditions holds.*

1. *$\theta(\tau)$ is a simple located type.*
2. *There is a module variable X in $dom(\theta)$ such that $\theta(\tau) = \theta(X).t$ for some type name t.*

Proof. By definition of simple located types. Use Lemma 5. □

Lemma 10 *Let τ be a located type, then $\Omega \vdash \tau \downarrow \tau$ for any Ω.*

Proof. By induction on the structure of τ. Use Lemma 8. □

Now let us examine a simple case. Suppose that every type abbreviation in the program P abbreviates a simple located type and that every transparent type specification in P specifies a simple located type. That is, we suppose that, for all `type` $t = \tau$ appearing in P, τ is a simple located type.

To expand a type $p.t$, the algorithm first expands p into a located form p' to find the definition of the type $p.t$. Let us assume $\vdash p' \mapsto (\theta_1, \texttt{struct} \ \ldots \texttt{type} \ t =$

$\tau \ldots$ `end`^{i}) holds. Since τ is a simple located type, $\Omega \vdash \tau \downarrow \tau$ holds by Lemma 10. Hence, by Lemma 9, $\theta_1(\tau)$ is either a simple located type or else $\theta_1(X_1).t_1$ for some X_1 in $dom(\theta_1)$ and some type name t_1. When $\theta_1(\tau)$ is a simple located type, the algorithm terminates. When it is not, the algorithm continues expanding $\theta_1(X_1).t_1$. Since $\theta_1(X_1)$ is in located form (Lemma 3) and located forms are invariant of the semi-ground normalization (Lemma 8), we have $\vdash \theta(X_1) \rightsquigarrow_s \theta(X_1)$. Thus the only possible case where the algorithm further continues is where $\vdash \theta_1(X_1) \mapsto (\theta_2, \texttt{struct} \ \ldots \ \texttt{type} \ t_1 = \tau_1 \ldots \ \texttt{end}^j)$ holds. Again, by Lemma 9, $\theta_2(\tau_1)$ is either a simple located type or else $\theta_2(X_2).t_2$ for some X_2 in $dom(\theta_2)$ and t_2. Here, one should notice that $\theta_2(X_2)$ is structurally smaller than $\theta_1(X_1)$, since $\theta_2(X_2)$ appears syntactically inside $\theta_1(X_1)$. This implies that the algorithm eventually terminates, since $\theta_1(X_1)$ is structurally finite.

In general, type abbreviations may contain more complex types than simple located types, and so do transparent type specifications. Yet, if the algorithm knows all the type abbreviations and specifications that are looked up during the expansion of a type τ and if it has expanded these types in advance, it can expand τ in a similar way to the above simple case. In other words, the algorithm expands types in an appropriate order so that a type τ is expanded only after all those types that are looked up during the expansion of τ have been expanded. The algorithm simultaneously searches such an order and expands types along the order. Locks Ω are used to ensure that the order does not contain cycles.

The following two examples are good exercises to understand how the algorithm works in more complex cases. When designing the algorithm, we were careful to distinguish between the two; the former should be disallowed, while the latter allowed.

Example 1 *Consider the functor* `F` *defied as*

```
module F =
  (functor(X : sig type t end²) → struct type t = F(F(X)).t end³)¹
```

The functor `F` *contains a dangling type component named* `t`*. The type expansion algorithm raises an error for the input* `F(F(X)).t`*, since it attempts to lock* `(3.t)` *under the lock* `{(3.t)}` *during the expansion.*

We note that if the algorithm expanded the type `F(F(X)).t` *without locks, it would not terminate, but yield an infinite rewriting sequence:*

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

Example 2 *Consider the following program:*

```
module F = (functor(X : sig type t end[2]) →
 struct module L = X[4] type t = L.t * int end[3])[1]
module M = struct type s = int type t = s end[5]
module N = struct type t = F(F(M)).t end[6]
```

The type component `t` *of the module* `N` *has a valid reference, and the type expansion algorithm successfully expands the type* `F(F(M)).t` *into* `int * int * int`.

Observe that the algorithm expands the type `L.t * int` *into* `X.t * int` *before expanding* `F(F(M)).t`*, since the expansion of* `F(F(M)).t` *looks up the type* `t` *of the functor* `F`.

5.2 Well-definedness and termination

Here we show that the type expansion algorithm does reduce types into located types unless it raises an error and that it is terminating.

We first define a sanity condition on input types to the algorithm.

Definition 7 *A module path p has located variables if and only if all the self variables contained in p are in located form.*

Definition 8 *A type τ has located variables if and only if all the module paths contained in τ have located variables.*

All the types that appear in the program P have located variables, hence they are appropriate for the input to the algorithm.

Proposition 5 (Well-definedness of the type expansion) *Let τ have located variables. If $\Omega \vdash \tau \downarrow \tau'$, then τ' is a located type.*

Proof. By induction on the derivation of $\Omega \vdash \tau \downarrow \tau'$ and by case on the last rule used. We show the main case.
[t-typ] We have $\tau = p.t$ and $\vdash p \rightsquigarrow_s p'$ and $\vdash p' \mapsto (\theta, \texttt{struct} \ \ldots \ \texttt{type}\ t = \tau_1 \ldots \ \texttt{end}^i)$ and $\Omega \uplus (i,t) \vdash \tau_1 \downarrow \tau_2$ and $\Omega \vdash \theta(\tau_2) \downarrow \tau'$. By Proposition 4, p' is in located form. By Lemma 3, θ is in located form. By induction hypothesis, τ_2 is a located type. By Lemma 5, $\theta(\tau_2)$ has located variables. By induction hypothesis, τ' is a located type. □

Proposition 6 (Termination of the type expansion) *For any lock Ω and a type τ having located variables, proof search for $\Omega \vdash \tau \downarrow _$ will terminate.*

Proof. Below, we define a well-founded relation $>_t$ on pairs (τ, Ω) of a type τ and a lock Ω. Using Lemma 9 and Proposition 5, it can be easily checked that if there is an infinitely deep derivation tree of the type expansion, then one can construct an infinitely descending sequence in $>_t$ from the tree. This contradicts well-foundedness of $>_t$. By Köning's lemma on finitely branching trees, we obtain the proposition.

$(\tau_1, \Omega_1) >_t (\tau_2, \Omega_2)$ holds if and only if either of the following four conditions holds. We write $Tnames_P$ to denote the set of type names appearing in P.

1. $\Omega_1 = \Omega_2$ and $\tau_1 = \tau_{11} * \tau_{12}$ and $\tau_2 = \tau_{1i}$.
2. $\Omega_1 = \Omega_2$ and $\tau_1 = \tau_{11} \to \tau_{12}$ and $\tau_2 = \tau_{1i}$.
3. All the following three conditions hold.
   - $\Omega_1 = \Omega_2$.
   - $\tau_1 = p.t$ and $\vdash p \rightsquigarrow_s p_1$ and $\vdash p_1 \mapsto (\theta, \texttt{struct}\ \ldots \texttt{type}\ t = \tau' \ldots\ \texttt{end}^i)$.
   - For all τ in $cmpnt(\tau_2)$, τ is either a simple located type or else $\theta(X).t_1$ for some module variable X in $dom(\theta)$ and some type name t_1.
4. $(i, t) \notin \Omega_1$ and $\Omega_2 = \Omega_1 \cup \{(i, t)\} \subseteq \{(i, t) \mid i \in \Sigma_P,\ t \in Tnames_P\}$.

To prove well-foundedness of $>_t$, we define a well-founded relation $>_\tau$ on types. Then we show that well-foundedness of $>_\tau$ implies that of $>_t$.

$\tau_1 >_\tau \tau_2$ holds if and only if either of the following three conditions holds.

1. $\tau_1 = \tau_{11} \to \tau_{12}$, and $\tau_2 = \tau_{1i}$.
2. $\tau_1 = \tau_{11} * \tau_{12}$, and $\tau_2 = \tau_{1i}$.
3. The following two conditions hold.
   - $\tau_1 = p.t$ and $\vdash p \mapsto (\theta, \texttt{struct}\ \ldots \texttt{type}\ t = \tau' \ldots\ \texttt{end}^i)$.
   - For all τ in $cmpnt(\tau_2)$, τ is either a simple located type or else $\theta(X).t_1$ for some module variable X in $dom(\theta)$ and some type name t_1.

Note the slight but crucial difference between the second condition of the rule 3. of $>_t$ and the first condition of the rule 3. of $>_\tau$; in the latter, we do not expand p.

First we show well-foundedness of $>_\tau$. Suppose that there is an infinitely descending sequence $\{\tau_i\}_{i=1}^{\infty}$ in $>_\tau$. Such sequence can only be constructed using the rule 3. of $>_\tau$ infinitely often. Hence there is an infinite sequence $\{p_i.t_i\}_{i=1}^{\infty}$

such that, for all i in $1, 2, \ldots$, p_{i+1} is in $args(p_i)$. Since the length of p_1 is finite, this is a contradiction. (Note that if a type $p.t$ is a simple located type, then $\vdash p \mapsto (\theta, \texttt{struct } \ldots \texttt{type } t = \tau' \ldots \texttt{ end}^i)$ cannot hold.)

Now we show the well-foundedness of $>_t$. Suppose that there is an infinitely descending sequence in $>_t$. Since $\{(i, t) \mid i \in \Sigma_P,\ t \in Tnames_P\}$ is finite, there is a lock Ω_0 such that there is an infinitely descending sequence $\{(\tau_i, \Omega_0)\}_{i=1}^{\infty}$ in $>_t$. Let j be an integer such that $(\tau_j, \Omega_0) >_t (\tau_{j+1}, \Omega_0)$ holds due to the rule 3. of $>_t$. (It is easy to check that such j exists.) Let $\tau_j = p.t$. We have $\vdash p \rightsquigarrow_s p_1$ and $\vdash p_1 \mapsto (\theta, \texttt{struct } \ldots \texttt{type } t = \tau' \ldots \texttt{ end}^{i_1})$. By Proposition 4, p_1 is in located form. By Lemma 3, for all X in $dom(\theta)$, $\theta(X)$ is in located form. Since module paths in located form are invariant for the semi-ground normalization (Lemma 8), it holds that, for all $k > j$, if $(\tau_k, \Omega_0) >_t (\tau_{k+1}, \Omega_0)$ holds due to the rule 3. of $>_t$ and $\tau_k = p'.t'$ for some p' and t', then $\vdash p' \rightsquigarrow_s p'$. Thus, $\{\tau_i\}_{i=j+1}^{\infty}$ is a descending sequence in $>_\tau$. This contradicts well-foundedness of $>_\tau$. $\square$

$$\frac{}{\vdash \mathsf{1} \equiv \mathsf{1}} \qquad \frac{\vdash \tau_1 \equiv \tau_1' \quad \vdash \tau_2 \equiv \tau_2'}{\vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\vdash \tau_1 \equiv \tau_1' \quad \vdash \tau_2 \equiv \tau_2'}{\vdash \tau_1 * \tau_2 \equiv \tau_1' * \tau_2'} \qquad \frac{\vdash p_1 \equiv p_2}{\vdash p_1.t \equiv p_2.t}$$

Figure 10: Type equivalence

$$\frac{}{\vdash X \equiv X}$$

$$\frac{\begin{array}{c}\vdash p_1 \mapsto (\theta_1, E_1^{i_1}) \quad \vdash p_2 \mapsto (\theta_2, E_2^{i_2}) \quad i_1 = i_2 \\ \forall X \in dom(\theta_1), \ \vdash \theta_1(X) \equiv \theta_2(X)\end{array}}{\vdash p_1 \equiv p_2}$$

Figure 11: Path equivalence

6 Type system

In this section, we present the overall typing rules. Having defined algorithms for expanding module paths and types, the remaining part of the type system is straightforward.

6.1 Type equality

We define a type equivalence relation in Figure 10, with an auxiliary judgment in Figure 11.

The judgment $\vdash \tau_1 \equiv \tau_2$ is read that the types τ_1 and τ_2 are equivalent. The type equivalence relation judges equivalence of located types. Hence, to check equivalence between types which are not necessarily located types, the type system first expands them into located types, then appeals to the judgment.

Let us examine each rule. The first three rules are straightforward. The last rule judges whether two abstract types are equivalent. The types $p_1.t_1$ and $p_2.t_2$ are equivalent if and only if 1) their type names t_1 and t_2 are identical; and 2) their prefixes p_1 and p_2 are equivalent module paths.

In Figure 11, we present inference rules for judging equivalence of module paths. Two module paths p_1 and p_2 are equivalent if and only if either 1) p_1 and p_2 are the same module variable or else 2) they refer to module expressions at the same location and their functor arguments are equivalent.

[v-var]

$$\frac{}{\Psi;\Gamma \vdash x :: \Gamma(x)}$$

[v-uni]

$$\frac{}{\Psi;\Gamma \vdash () :: \mathtt{1}}$$

[v-prd]

$$\frac{\Psi;\Gamma \vdash e_1 :: \tau_1 \quad \Psi;\Gamma \vdash e_2 :: \tau_2}{\Psi;\Gamma \vdash (e_1, e_2) :: \tau_1 * \tau_2}$$

[v-prj]

$$\frac{\Psi;\Gamma \vdash e :: \tau_1 * \tau_2}{\Psi;\Gamma \vdash \pi_i(e) :: \tau_i}$$

[v-app]

$$\frac{\Psi;\Gamma \vdash e_1 :: \tau' \to \tau}{\Psi;\Gamma \vdash e_1(e_2) :: \tau}$$

[v-fun]

$$\frac{\vdash \tau' \downarrow \tau}{\Psi;\Gamma \vdash (\mathtt{fun}\ x \to e : \tau') :: \tau}$$

[v-cnstr]

$$\frac{\vdash p \rightsquigarrow_s p' \quad \gamma(p', c) = (t, \tau)}{\Psi;\Gamma \vdash p.c\ e :: p'.t}$$

[v-case]

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

[v-vpth1]

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

[v-vpth2]

$$\frac{\vdash p \rightsquigarrow_s X \quad \Delta(X) = \mathtt{sig} \ldots \mathtt{val}\ l : \tau' \ldots \mathtt{end}^i \quad \vdash \tau' \downarrow \tau}{\Psi;\Gamma \vdash p.l :: \tau}$$

Figure 12: Type reconstruction

$\gamma(p, c) = (t, \tau)$ when
$\vdash p \mapsto (\theta, \texttt{struct} \ldots \texttt{datatype}\ t\ =\ c\ \texttt{of}\ \tau'\ \ldots \texttt{end})$ and $\vdash \theta(\tau') \downarrow \tau$

Figure 13: Datatype look-up

6.2 Type reconstruction

The reconstruction algorithm infers types of expressions, but does not check that the inferred types are correct. For instance, to reconstruct a type of an application $e_1(e_2)$, it only reconstructs a type of e_1, which must be in the form $\tau' \to \tau$, then returns the result type τ. We defer ensuring that e_2 does have a type equivalent to τ' to *well-typedness judgment* of the form $\Gamma \vdash e : \tau$, which is defined later in Figure 14.

We present inference rules for the type reconstruction algorithm in Figure 12, with an auxiliary judgment in Figure 13. The judgment $\Psi; \Gamma \vdash e :: \tau$ is read that the algorithm reconstructs the type τ for the expression e under the type environment Γ with Ψ locked. We use Ψ as a metavariable for ranging over pairs (i, l) of an integer i and a value name l. We use Γ as a metavariable for ranging over type environments, which assign located types to variables. The inference rules define an algorithm; their derivations are deterministic. The algorithm takes an expression e, a type environment Γ and a lock Ψ as inputs, then either returns a located type or raises an error when there is no applicable rule. We may write $\vdash e :: \tau$ to mean $\emptyset; \emptyset \vdash e :: \tau$.

In the same way as the type expansion algorithm does, the reconstruction algorithm holds a lock Ψ so as to avoid tracing value paths cyclically. For instance, it does not attempt to reconstruct a type of the value component `l` in the program below, but raises an error.

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

The inference rules of the reconstruction are mostly straightforward. Here, we explain the key rules [**v-vpth1**] and [**v-vpth2**]. These two rules infer a type of a value path $p.l$. When the located form of p is a module variable X([**v-vpth2**]), then the type of $X.l$ should be found in the signature of X. The rule [**v-vpth1**] handles the case where the located form of p refers to a structure which contains a value definition $\texttt{val}\ l = e$ with a module variable environment θ. In this case, the algorithm first reconstructs a type of e without applying the substitution θ to e, then returns the located type of $\theta(\tau')$ where τ' is the reconstructed type of e. Hence, when $p.l$ refers to a value component in the body of a functor, the algorithm first reconstructs a type of the component in the functor, then instantiates the reconstructed type.

Observe that the third premise of the rule [**v-vpth1**] has an empty type environment. Hence the algorithm always reconstructs the same type for the same value path under whatever type environment, unless it raises an error. This implies that we can memorize results of the reconstruction for efficiency in a practical system.

Proposition 7 (Termination of the type reconstruction) *For any expression e, type environment Γ and lock Ψ, proof search for $\Psi; \Gamma \vdash e :: _$ will terminate.*

Proof. Below we define a well-founded relation $>_v$ on pairs (e, Ψ) of an expression e and a lock Ψ. The proposition can be proven by induction on $>_v$. We write $Vnames_P$ to denote the set of value names appearing in the program P.

$(e_1, \Psi_1) >_v (e_2, \Psi_2)$ holds if and only if either of the following two conditions holds.

1. e_2 is structurally smaller than e_1 and $\Psi_1 = \Psi_2$.
2. $(i, l) \notin \Psi_1$ and $\Psi_2 = \Psi_1 \cup \{(i, l)\} \subseteq \{(i, l) \mid i \in \Sigma_P,\ l \in Vnames_P\}$.

The well-foundedness of $>_v$ can be proven by the finiteness of $\{(i, l) \mid i \in \Sigma_P,\ l \in Vnames_P\}$. □

6.3 Typing rules

Finally, we present the rest of the typing rules in Figure 14, with auxiliary judgments in Figure 15 and 16.

The judgment $\vdash D \diamond$ is read that the definition D is well-typed. The judgment $\Gamma \vdash e : \tau$ is read that the core expression e has the type τ under the type environment Γ. The other judgments are read similarly. We may write $\vdash e : \tau$ to mean $\emptyset \vdash e : \tau$.

The typing rules are straightforward. Note only that, for type checking a core expression $p.l$, the type system uses the type reconstruction. While the value path $p.l$ may be a forward reference, the reconstruction algorithm can resolve both forward and backward references using semi-ground normalization.

In Figure 15, we present inference rules for judging well-formedness of modules paths; the type system uses the judgment to check well-typedness of module paths. The judgment $\vdash p$ wf is read that the module path p is well-formed. The inference rules check that the references of module paths are not dangling and that functor applications contained in the paths are well-typed.

In Figure 16, we define *realization judgment* $\vdash p \triangleright B$ for checking that the module path p refers to a module which has a component satisfying the specification B. The inference rules for the judgment are straightforward.

Definitions

$$\frac{\vdash E \diamond}{\vdash \texttt{module}\ M = E \diamond} \qquad \frac{\vdash \tau \diamond}{\vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau \diamond} \qquad \frac{\vdash \tau \diamond}{\vdash \texttt{type}\ t = \tau \diamond} \qquad \frac{\vdash e : \tau}{\vdash \texttt{val}\ l = e \diamond}$$

Module expressions

$$\frac{\vdash D_1\ \diamond\ \ldots\ \vdash D_n \diamond}{\vdash \texttt{struct}\ (Z)\ D_1\ \ldots\ D_n\ \texttt{end} \diamond} \qquad \frac{\vdash S \diamond \quad \vdash E \diamond}{\vdash \texttt{functor}(X : S) \to E \diamond} \qquad \frac{\vdash p\ \text{wf}}{\vdash p \diamond}$$

Signatures

$$\frac{\vdash B_1\ \diamond \ldots \vdash B_n \diamond}{\vdash \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} \diamond}$$

Specifications

$$\frac{\vdash S \diamond}{\vdash \texttt{module}\ M : S \diamond} \qquad \frac{\vdash \tau \diamond}{\vdash \texttt{type}\ t = \tau \diamond} \qquad \frac{}{\vdash \texttt{type}\ t \diamond} \qquad \frac{\vdash \tau \diamond}{\vdash \texttt{val}\ l : \tau \diamond}$$

Core types

$$\frac{}{\vdash \texttt{1} \diamond} \qquad \frac{\vdash \tau_1 \diamond \quad \vdash \tau_2 \diamond}{\vdash \tau_1 \to \tau_2 \diamond} \qquad \frac{\vdash \tau_1 \diamond \quad \vdash \tau_2 \diamond}{\vdash \tau_1 * \tau_2 \diamond} \qquad \frac{\vdash p\ \text{wf} \quad \vdash p.t \downarrow \tau}{\vdash p.t \diamond}$$

Core expressions

$$\frac{}{\Gamma \vdash () : \texttt{1}} \qquad \frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\vdash \tau \diamond \quad \vdash \tau \downarrow \tau_1 \to \tau_2 \quad \Gamma, x : \tau_1 \vdash e : \tau_3 \quad \vdash \tau_2 \equiv \tau_3}{\Gamma \vdash (\lambda x.e : \tau) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_i(e) : \tau_i} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash \tau_2 \equiv \tau_1}{\Gamma \vdash e_1\ (e_2) : \tau}$$

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

$$\frac{\vdash p\ \text{wf} \quad \vdash p \rightsquigarrow_s p' \quad \gamma(p', c) = (t, \tau_1) \quad \Gamma \vdash e : \tau_2 \quad \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash p.c\ e : p'.t} \qquad \frac{\vdash p\ \text{wf} \quad \vdash p.l :: \tau}{\Gamma \vdash p.l : \tau}$$

Figure 14: Typing rules

$$\frac{}{\vdash X\ \text{wf}} \qquad \frac{}{\vdash Z^{id}\ \text{wf}} \qquad \frac{\vdash p\ \text{wf} \quad \vdash p.M \rightsquigarrow_s q}{\vdash p.M\ \text{wf}}$$

$$\frac{\begin{array}{c}\vdash p_1\ \text{wf} \quad \vdash p_2\ \text{wf} \quad \vdash p_1 \rightsquigarrow_s p_1' \quad \vdash p_2 \rightsquigarrow_s p_2' \quad \vdash p_1(p_2) \rightsquigarrow_s q \\ \forall i \in \{1, \ldots, n\},\ \vdash p_2' \rhd \theta[X \mapsto p_2']\ B_i \\ \vdash p_1' \mapsto (\theta, (\texttt{functor}(X : \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^j) \to E^k)^i)\end{array}}{\vdash p_1(p_2)\ \text{wf}}$$

Figure 15: Well-formed module paths

$$\frac{\vdash p.t \downarrow \tau}{\vdash p \triangleright \texttt{type}\ t} \qquad \frac{\vdash p.t \downarrow \tau_1 \quad \vdash \tau \downarrow \tau_2 \quad \vdash \tau_1 \equiv \tau_2}{\vdash p \triangleright \texttt{type}\ t = \tau} \qquad \frac{\vdash p.l :: \tau_1 \quad \vdash \tau \downarrow \tau_2 \quad \vdash \tau_1 \equiv \tau_2}{\vdash p \triangleright \texttt{val}\ l : \tau}$$

Figure 16: Realization

Definition 9 *The program P is well-typed if and only if $\vdash P \diamond$ holds.*

Proposition 8 below is an immediate consequence of the termination of the semi-ground normalization (Proposition 3), the type expansion (Proposition 6) and the type reconstruction (Proposition 7).

Proposition 8 (Decidability of the type system) *It is decidable whether $\vdash P \diamond$ holds or not.*

Here, we add two important observations of the type system.

1. The type reconstruction algorithm is developed separately from the typing rules given in Figure 14. In other words, the algorithm can reconstruct types in a separate phase from type checking. Hence, in a practical system, it would be natural to complete the reconstruction in advance, and to use the result of the reconstruction during type checking to infer types of value paths. The current presentation of the type system makes the soundness proof concise.

2. The type system checks that every type abbreviation and transparent type specification in the program P only contains expandable types. Hence, well-typed programs do not contain cyclic types.

6.3.1 Detecting structural types forming non-regular trees

To support flexible development of programs, it is useful to extend the core language with structural recursive types, such as objects [23] and polymorphic variants [10]. For developing a practical type system, we want to forbid programmers to specify structural types which form non-regular trees, since there is little hope that a practical algorithm for judging equality between non-regular types is found.

There are known algorithms in the core language for detecting non-regular types defined by programmers, for instance the one implemented in Objective Caml [18]. However, the algorisms may not be available for detecting non-regular types defined by combining recursive modules and applicative functors.

For instance, consider the following module `G`, which defines a polymorphic variant type.

```
module G = functor(X : sig type t end) →
 struct
  type t = X.t * X.t
  type u = [ ‘A of t | ‘B of G(G(X)).u ]
 end
```

An instantiation of `G` would yield a non-regular type. Compare the module `G` to the module `F` defined as:

```
module F = (X : sig type t end) →
 struct
  type t = X.t * X.t
  type s = [ ‘C of t | ‘D of F(X).s ]
 end
```

An instantiation of `F` will yield a regular type, thus there is no reason to disable `F`.

In these simple examples, it is easy to distinguish between the modules `G` and `F`. Since the type of the constructor ‘`B`, namely `G(G(X)).u`, refers to the defining type itself but changes the functor's parameter form `X` to `G(X)`, the type definition of `u` should be illegal. Since the type of the constructor ‘`D`, namely `F(X).s`, refers to the defining type without changing the functor's parameter, the definition of `s` should be legal.

Observe that we have implicitly assumed that we have a type expansion algorithm; in the general case, we would need to expand the types of the constructors ‘`B` and ‘`D`. We believe that our type expansion algorithm is useful to support structural recursive types and recursive modules together, while keeping the type system decidable and practical.

[nlz-sf]

$$\frac{}{\vdash Z^\theta \rightsquigarrow Z^\theta}$$

[nlz-exp1]

$$\frac{\vdash p \rightsquigarrow p' \quad \vdash p'.M \mapsto (\theta, E^i) \quad E \neq q}{\vdash p.M \rightsquigarrow p'.M}$$

[nlz-pth1]

$$\frac{\vdash p \rightsquigarrow p' \quad \vdash p'.M \mapsto (\theta, q^i) \quad \vdash \theta(q) \rightsquigarrow r}{\vdash p.M \rightsquigarrow r}$$

[nlz-exp2]

$$\frac{\vdash p_1 \rightsquigarrow p_1' \quad \vdash p_2 \rightsquigarrow p_2' \quad \vdash p_1'(p_2') \mapsto (\theta, E^i) \quad E \neq q}{\vdash p_1(p_2) \rightsquigarrow p_1'(p_2')}$$

[nlz-pth2]

$$\frac{\vdash p_1 \rightsquigarrow p_1' \quad \vdash p_2 \rightsquigarrow p_2' \quad \vdash p_1'(p_2') \mapsto (\theta, q^i) \quad \vdash \theta(q) \rightsquigarrow r}{\vdash p_1(p_2) \rightsquigarrow r}$$

Figure 17: Normalization

7 Soundness

In this section, we give a call-by-value operational semantics and present a soundness result.

Before presenting our small step reductions, we present *normalization* of module paths, which we use in reductions to resolve references of module paths. The normalization expands module paths by tracing module abbreviations in the intuitive way.

In Figure 17, we present inference rules for the normalization. The normalization acts similarly to the ground normalization, but it does not defer substitution of module variables. The judgment $\vdash p \rightsquigarrow q$ is read that the normalization expands p into q. The normalization is defined only for module paths containing no module variables.

Values v and evaluation contexts E are:

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

A small step reduction is either:

$$\pi_i(v_1, v_2) \overset{\texttt{prj}}{\longrightarrow} v_i \quad (\texttt{fun}\ x \to e : \tau)(v) \overset{\texttt{fun}}{\longrightarrow} [x \mapsto v]e$$

$$\texttt{case}\ p.c\ v\ \texttt{of}\ q.c\ x \Rightarrow e \overset{\texttt{case}}{\longrightarrow} [x \mapsto v]e$$

$p.l \overset{\texttt{vpth}}{\longrightarrow} \theta(e)$ when $\vdash p \rightsquigarrow q$ and $\vdash q \mapsto (\theta, \texttt{struct} \ldots\ \texttt{val}\ l = e \ldots\ \texttt{end}^i)$

or an inner reduction obtained by induction:

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

For an expression e, $[x \mapsto v]e$ denotes the expression obtained by applying the substitution $[x \mapsto v]$ to e, and $\theta(e)$ denotes the expression obtained by applying the substitution θ to e.

When deconstructing a value through the case expression $\texttt{case}\ p.c\ v\ \texttt{of}\ q.c\ x \Rightarrow e$, we do not explicitly check that p and q refer to the same module. Our type system already ensures that p and q expands into equivalent module paths.

We could define a small step normalization of module paths. The big step one is convenient for us to prove a soundness result, since the ground-normalization is big step.

Proposition 9 (Soundness) *Let the program P be well-typed. Then, the following two results hold.*

1. *Suppose $\vdash e_1 : \tau$ and $e_1 \to e_2$, then $\vdash e_2 : \tau'$ with $\vdash \tau \equiv \tau'$.*
2. *Suppose $\vdash e : \tau$, then either e is a value or else there is some e' with $e \to e'$.*

7.1 Proofs of the soundness

The soundness result can be proven in a standard way for the most part. The only difficulty in the proof is about the reduction rule $\overset{\texttt{vpth}}{\to}$. Below we show progress and type preservation properties for this rule.

We have already shown decidability of the type system in Proposition 8. Locks Σ, Ω and Ψ are useful only for the decidability result. In the soundness proof, we are interested in a derivation tree which proves $\vdash P \diamond$, but not in how we can construct the tree. Hence, in the proofs below, we use judgments of the ground normalization, the type expansion and the type reconstruction that do not hold locks. For instance, we may say that "$\vdash p \rightsquigarrow_g q$ holds", when $\vdash p \rightsquigarrow_g q$ can be proven by the inference rules that are completely same as the rules in Figurere 7 but do not have locks. (It is clear that whether or not the inference rules use locks does not affect outputs of the ground normalization; the ground normalization without locks may diverge and the ground normalization with locks may raise more errors than without.)

In the rest of this section, we assume that the program P is well-typed.

We first show in Proposition 10 that the semi-ground normalization coincides with the normalization for well-typed module paths. The proof proceeds in two steps: 1) we prove in Lemma 15 that the ground normalization coincides with the

[ugnlz-mv]

$$\frac{}{\vdash X \rightsquigarrow_G X}$$

[ugnlz-sf]

$$\frac{}{\vdash Z^\theta \rightsquigarrow_G Z^\theta}$$

[ugnlz-exp1]

$$\frac{\vdash p \rightsquigarrow_G p' \quad \vdash p'.M \mapsto (\theta, E^i) \quad E \notin mid}{\vdash p.M \rightsquigarrow_G p'.M}$$

[ugnlz-pth1]

$$\frac{\vdash p \rightsquigarrow_G p' \quad \vdash p'.M \mapsto (\theta, q^i) \quad q \neq X \quad \vdash \theta(q) \rightsquigarrow_G r}{\vdash p.M \rightsquigarrow_G r}$$

[ugnlz-exp2]

$$\frac{\vdash p_1 \rightsquigarrow_G p_1' \quad \vdash p_2 \rightsquigarrow_G p_2' \quad \vdash p_1'(p_2') \mapsto (\theta, E^i) \quad E \notin mid}{\vdash p_1(p_2) \rightsquigarrow_G p_1'(p_2')}$$

[ugnlz-pth2]

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

Figure 18: Unsafe ground-normalization

unsafe ground normalization defined in Figure 18; 2) then we prove in Lemma 20 that the composition of the unsafe one and the variable normalization coincides with the normalization. For the unsafe ground normalization, we use judgments of the form $\vdash p \rightsquigarrow_G q$. In rules **[ugnlz-pth1]** and **[ugnlz-pth2]**, the unsafe one applies the substitution θ to q, before expanding q.

The following three lemmas can be shown by easy induction. In particular, we use Lemma 11 implicitly throughout the proofs below.

Lemma 11 *Let $MVars(p) \subseteq dom(\theta)$. If $\vdash p \mapsto (\theta_1, E^i)$, then $\vdash \theta(p) \mapsto (\theta \circ \theta_1, E^i)$.*

Lemma 12 *Let p have pre-located variables. If $\vdash p \rightsquigarrow_G q$ then q is in pre-located form.*

Lemma 13 *Let p be in pre-located form. Then $\vdash p \rightsquigarrow_G p$.*

Lemma 14 *Let p have pre-located variables and θ be in pre-located form and $MVars(p) \subseteq dom(\theta)$. If $\vdash p \rightsquigarrow_G q$, then $\vdash \theta(p) \rightsquigarrow_G \theta(q)$.*

Proof. By induction on the derivation of $\vdash p \rightsquigarrow_G q$ and by case on the last rule used. Use Lemma 12 and 13. □

Lemma 15 *Let p have pre-located variables. If $\vdash p \rightsquigarrow_g q$, then $\vdash p \rightsquigarrow_G q$.*

Proof. By induction on the derivation of $\vdash p \leadsto_g q$ and by case on the last rule used. We show the main case.
[gnlz-pth1] We have $p = p_1.M$ and $\vdash p_1 \leadsto_g p_1'$ and $\vdash p_1'.M \mapsto (\theta, r^i)$ and $r \neq X$ and $\vdash r \leadsto_g q_1$ and $q = \theta(q_1)$. By induction hypothesis we have $\vdash p_1 \leadsto_G p_1'$ and $\vdash r \leadsto_G q_1$. By Proposition 1 and Lemma 2, θ is in pre-located form. By Lemma 14, $\vdash \theta(r) \leadsto_G \theta(q_1)$. □

To make the normalization account for module paths having pre-located variables, we modify the inference rules of the normalization by replacing the rule **[nlz-sf]** with the rule:

$$\frac{dom(\theta) = dom(\theta_1) \quad \forall X \in dom(\theta),\ \vdash \theta(X) \leadsto \theta_1(X)}{\vdash Z^{\theta} \leadsto Z^{\theta_1}}$$

The operational semantics uses the normalization only with module paths having located variables. This new rule and the original one have the same effect for these paths.

The following two lemmas are shown by easy induction.

Lemma 16 *Let p have pre-located variables and contain no module variables. If $\vdash p \leadsto q$ then q is in located form.*

Lemma 17 *Let p be in located form containing no module variables. Then $\vdash p \leadsto p$.*

Lemma 18 *Let p be in pre-located form and contain no module variables. Then $\vdash p \leadsto \eta(p)$.*

Proof. By induction on the length of p and by case on the structure of p. Use Lemma 16 and 17, and show that when p is in pre-located form and $\vdash p \mapsto (\theta, E^i)$ with $E \neq X$ then $\vdash \eta(p) \mapsto (\eta \circ \theta, E^i)$. □

For a module variable environment θ, we write $MVars(\theta)$ to mean $\bigcup_{X \in dom(\theta)} \theta(X)$.

Lemma 19 *Let θ be in pre-located form, and p have pre-located variables, and $MVars(p) \subseteq dom(\theta)$, and $MVars(\theta) = \emptyset$. If $\vdash \theta(p) \leadsto q$, then $\vdash (\eta \circ \theta)(p) \leadsto q$.*

Proof. By induction on the structure of p. For the case where p is a module variable, use Lemma 4, 17 and 18 . □

Lemma 20 *Let p have pre-located variables and contain no module variables. If $\vdash p \leadsto_G q$, then $\vdash p \leadsto \eta(q)$.*

Proof. By induction on the derivation of $\vdash p \leadsto_G q$ and by case on the last rule used. We show the main case.
[ugnlz-pth1] We have $p = p_1.M$ and $\vdash p_1 \leadsto_G p_1'$ and $\vdash p_1'.M \mapsto (\theta, r^i)$ and $r \neq X$ and $\vdash \theta(r) \leadsto_G q$. By induction hypothesis, $\vdash p_1 \leadsto \eta(p_1')$ and $\vdash \theta(r) \leadsto \eta(q)$. We have $\vdash \eta(p_1').M \mapsto (\eta \circ \theta, r^i)$. By Lemma 19, $\vdash (\eta \circ \theta)(r) \leadsto \eta(q)$. □

Lemma 21 *Let θ be in located form and p have located variables and $MVars(p) \subseteq dom(\theta)$. If $\vdash p \leadsto_s q$, then $\vdash \theta(p) \leadsto_s \theta(q)$.*

Proof. Show that if θ is in located form and p has pre-located variables and $\vdash p \leadsto_g q$, then $\vdash \theta(p) \leadsto_g \theta(q)$, by induction on the derivation of $\vdash p \leadsto_g q$. Show also that if θ and p are in pre-located form, then $\eta(\theta(p)) = (\eta \circ \theta)(\eta(p))$, by induction on the length of p. Then, by Lemma 7, we obtain the lemma. □

Lemma 22 *Let p have located variables. If $\vdash p \diamond$, then $\vdash p \leadsto_s q$.*

Proof. By case on the structure of p. Use Lemma 21. □

Proposition 10 *Let p have located variables and contain no module variables, and $\vdash p \diamond$. Then $\vdash p \leadsto_s q$ if and only if $\vdash p \leadsto q$.*

Proof. By Lemma 22, we have $\vdash p \leadsto_g p'$ and $\eta(p') = q$. By Lemma 15, $\vdash p \leadsto_G p'$. By Lemma 20, $\vdash p \leadsto \eta(p')$. Since derivations of the normalization are deterministic, if $\vdash p \leadsto q_1$ and $\vdash p \leadsto q_2$ then q_1 and q_2 are identical. Thus we have the proposition. □

Now we show a progress property of the reduction $\overset{\texttt{vpth}}{\rightarrow}$.

Proposition 11 (Progress for the reduction $\overset{\texttt{vpth}}{\rightarrow}$) *Let p have located variables and contain no module variables. If $\vdash p.l : \tau$, then $\vdash p \leadsto q$ and $\vdash q \mapsto (\theta, \texttt{struct} \ \ldots \ \texttt{val}\ l = e \ldots \texttt{end}^i)$.*

Proof. By hypothesis, we have $\vdash p \diamond$ and $\vdash p \leadsto_s p_1$ and $\vdash p_1 \mapsto (\theta', \texttt{struct} \ \ldots \ \texttt{val}\ l = e' \ \ldots \ \texttt{end}^j)$. By Proposition 10, $\vdash p \leadsto p_1$. □

Before showing a type preservation property of the reduction $\overset{\texttt{vpth}}{\rightarrow}$, we show in Proposition 12 that the semi-normalization preserves well-formedness of module paths. To show the proposition, we strengthen the inference rules of well-formedness of module paths (Figure 15), by replacing the rule for self variables with the rule:

$$\frac{\vdash \theta \text{ wf}}{\vdash Z^\theta \text{ wf}}$$

where $\vdash \theta$ wf is defined as follows.

Definition 10 *A module variable environment θ is well-formed, written $\vdash \theta$ wf, if and only if, for all X in $dom(\theta)$, the following two conditions hold.*

1. $\vdash \theta(X)$ wf.

2. *When* $\Delta(X) = \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^i$, *then* $\forall i \in \{1, \ldots, n\}, \vdash \theta(X)\ \triangleright\ \theta(B_i)$.

All the self variables appearing in the program P are superscripted with the identity substitution, and $\vdash Z^{id}$ wf holds with this new rule. Hence this new rule does not affect the type system in practice.

For types τ_1 and τ_2, we write $\vdash \tau_1 \equiv_\tau \tau_2$ to mean that τ_1 and τ_2 expand into equivalent types, that is, that there are types τ_1' and τ_2' such that $\vdash \tau_1 \downarrow \tau_1'$ and $\vdash \tau_2 \downarrow \tau_2'$ and $\vdash \tau_1' \equiv \tau_2'$. It is easy to check that 1) the relation is transitive, that is, if $\vdash \tau_1 \equiv_\tau \tau_2$ and $\vdash \tau_2 \equiv_\tau \tau_3$ then $\vdash \tau_1 \equiv_\tau \tau_3$, and that 2) if $\vdash \tau \equiv_\tau \tau'$ and both τ and τ' are located types, then $\vdash \tau \equiv \tau'$.

Lemma 23 *Let θ be in located form and $MVars(\tau) \subseteq dom(\theta)$. If $\vdash \tau \downarrow \tau'$ and $\vdash \theta$ wf, then $\vdash \theta(\tau) \equiv_\tau \theta(\tau')$.*

Proof. By induction on the derivation of $\vdash \tau \downarrow \tau'$ and by case on the last rule used. We show main cases.
[**t-typ**]We have $\tau = p.t$ and $\vdash p \rightsquigarrow_s p'$ and $\vdash p' \mapsto (\theta_1, \texttt{struct} \ldots \texttt{type}\ t = \tau_1 \ldots \texttt{end}^i)$ and $\vdash \tau_1 \downarrow \tau_1'$ and $\vdash \theta_1(\tau_1') \downarrow \tau'$. By Lemma 21, $\vdash \theta(p) \rightsquigarrow_s \theta(p')$. We have $\vdash \theta(p') \mapsto (\theta \circ \theta_1, \texttt{struct} \ldots \texttt{type}\ t = \tau_1 \ldots \texttt{end}^i)$. By induction hypothesis, $\vdash \theta \circ \theta_1(\tau_1') \equiv_\tau \theta(\tau')$.
[**t-tran**]We have $\tau = p.t$ and $\vdash p \rightsquigarrow_s X$ and $\Delta(X) = \texttt{sig} \ldots \texttt{type}\ t = \tau_1 \ldots \texttt{end}^i$ and $\vdash \tau_1 \downarrow \tau'$. By Lemma 21, $\vdash \theta(p) \rightsquigarrow_s \theta(X)$. By the well-formedness of θ, $\vdash \theta(X).t \equiv_\tau \theta(\tau_1)$. By induction hypothesis, $\vdash \theta(\tau_1) \equiv_\tau \theta(\tau')$. □

Lemma 24 *Let τ and τ' be located types and θ be in located form and $MVars(\tau) \cup MVars(\tau') \subseteq dom(\theta)$. If $\vdash \theta$ wf and $\vdash \tau \equiv \tau'$, then $\vdash \theta(\tau) \equiv_\tau \theta(\tau')$.*

Proof. By induction on the derivation of $\vdash \tau \equiv \tau'$. □

We say that a type environment Γ is in located form if and only if , for all x in $dom(\Gamma)$, $\Gamma(x)$ is a located type. For a type environment Γ, $dom(\Gamma)$ denotes the domain of Γ.

Lemma 25 *Let Γ and θ be in located form, and $MVars(e) \cup MVars(\tau) \subseteq dom(\theta)$, and Γ_1 be a type environment in located form such that $dom(\Gamma) = dom(\Gamma_1)$ and, for all $x \in dom(\Gamma)$, $\vdash \theta(\Gamma(x)) \equiv_\tau \Gamma_1(x)$. If $\Gamma \vdash e :: \tau$ and $\vdash \theta$ wf, then $\Gamma_1 \vdash \theta(e) :: \tau'$ with $\vdash \theta(\tau) \equiv_\tau \tau'$.*

Proof. By induction on the derivation of $\Gamma \vdash e :: \tau$ and by case on the last rule used. Use Lemma 23. □

Lemma 26 *Let θ be in located form and p have located variables and $MVars(p) \cup MVars(B) \subseteq dom(\theta)$. If $\vdash p \triangleright B$ and $\vdash \theta$ wf, then $\vdash \theta(p) \triangleright \theta(B)$.*

Proof. We show the main case. Suppose $\vdash p \triangleright \texttt{val}\ l : \tau$. We have $\vdash p.l :: \tau_1$ and $\vdash \tau \downarrow \tau_2$ and $\vdash \tau_1 \equiv \tau_2$. By Lemma 25, $\vdash \theta(p.l) :: \tau_3$ and $\vdash \tau_3 \equiv_\tau \theta(\tau_1)$. By Lemma 23, $\vdash \theta(\tau) \equiv_\tau \theta(\tau_2)$. By Lemma 24, $\vdash \theta(\tau_1) \equiv_\tau \theta(\tau_2)$. □

Lemma 27 *Let p have located variables and θ in located form and $MVars(p) \subseteq dom(\theta)$. If $\vdash p$ wf and $\vdash \theta$ wf, then $\vdash \theta(p)$ wf.*

Proof. By induction on the derivation of $\vdash p$ wf and by case on the last rule used. We show the main case. Suppose $p = p_1(p_2)$. We have $\vdash p_1$ wf and $\vdash p_2$ wf and $\vdash p_1 \rightsquigarrow_s p_1'$ and $\vdash p_2 \rightsquigarrow_s p_2'$ and $\vdash p_1' \mapsto (\theta_1, (\texttt{functor}(X : \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^j) \to E^k)^i)$ and, for all i in $\{1 \ldots n\}$, $\vdash p_2' \triangleright \theta_1[X \mapsto p_2']B_i$. By induction hypothesis, $\vdash \theta(p_1)$ wf and $\vdash \theta(p_2)$ wf. By Lemma 21, $\vdash \theta(p_1) \rightsquigarrow_s \theta(p_1')$ and $\vdash \theta(p_2) \rightsquigarrow_s \theta(p_2')$. We have $\vdash \theta(p_1') \mapsto (\theta \circ \theta_1, (\texttt{functor}(X : \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^j) \to E^k)^i)$. By Lemma 26, for all i in $\{1 \ldots n\}$, $\vdash \theta(p_2') \triangleright \theta \circ \theta_1[X \mapsto \theta(p_2')]B_i$.

□

Lemma 28 *Let p be in pre-located form. If $\vdash p$ wf, then $\vdash \eta(p)$ wf.*

Proof. By induction on the length of p. □

Lemma 29 *Let p have pre-located variables. If $\vdash p$ wf and $\vdash p \rightsquigarrow_G q$, then $\vdash q$ wf.*

Proof. By induction on the derivation of $\vdash p \rightsquigarrow_G q$ and by case on the last rule used. Use Lemma 27 and 28. □

Proposition 12 *Let p have located variables. If $\vdash p$ wf and $\vdash p \rightsquigarrow_s q$, then $\vdash q$ wf.*

Proof. By hypothesis, we have $\vdash p \rightsquigarrow_g p'$ and $\eta(p') = q$. By Lemma 15, we have $\vdash p \rightsquigarrow_G p'$. By Lemma 29 and 28, $\vdash q$ wf. □

Finally, we show a type preservation property of $\overset{\texttt{vpth}}{\longrightarrow}$ in Proposition 13.

Lemma 30 *Let τ have located variables and θ be in located form and $MVars(\tau) \subseteq dom(\theta)$. If $\vdash \tau \diamond$ and $\vdash \theta$ wf, then $\vdash \theta(\tau) \diamond$.*

Proof. By induction on the derivation of $\vdash \tau \diamond$ and by case on the last rule used. Use Lemma 23 and 27. □

Lemma 31 *Let τ have located variables. If $\vdash \tau \diamond$ and $\vdash \tau \downarrow \tau'$, then $\vdash \tau' \diamond$.*

Proof. By induction on the derivation of $\vdash \tau \downarrow \tau'$ and by case on the last rule used. We show the main case.
[**t-typ**]We have $\tau = p.t$ and $\vdash p \rightsquigarrow_s p'$ and $\vdash p' \mapsto (\theta, \texttt{struct} \ \ldots \texttt{type}\ t = \tau_1 \ldots\ \texttt{end}^i)$ and $\vdash \tau_1 \downarrow \tau_2$ and $\vdash \theta(\tau_2) \downarrow \tau'$. By Proposition 12, $\vdash p'$ wf, hence $\vdash \theta$ wf. By well-typedness of the program P and by induction hypothesis, we have $\vdash \tau_2 \diamond$. By Lemma 30, $\vdash \theta(\tau_2) \diamond$. By induction hypothesis, $\vdash \tau' \diamond$. □

We say that a type environment Γ is well-formed, written $\vdash \Gamma$ wf, if and only if, for all x in $dom(\Gamma)$, $\vdash \Gamma(x) \diamond$. For an expression e, we say that e has located variables if and only if all the self variables contained in e are in located form.

Lemma 32 *Let θ and Γ be in located form, and e have located variables, and $MVars(e) \cup MVars(\Gamma) \subseteq dom(\theta)$, and Γ_1 be a type environment in located form such that $dom(\Gamma) = dom(\Gamma_1)$ and, for all $x \in dom(\Gamma)$, $\vdash \theta(\Gamma(x)) \equiv_\tau \Gamma_1(x)$. If $\vdash \theta$ wf and $\vdash \Gamma$ wf and $\Gamma \vdash e : \tau$, then $\Gamma_1 \vdash \theta(e) : \tau'$ with $\vdash \tau' \equiv_\tau \theta(\tau)$.*

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$ and by case on the last rule used. We show the main case.
Suppose $e = (\texttt{fun}\ x \to e_1 : \tau_1)$. We have $\vdash \tau_1 \diamond$ and $\vdash \tau_1 \downarrow \tau_2 \to \tau_3$ and $\Gamma, x : \tau_2 \vdash e_1 : \tau_4$ and $\vdash \tau_4 \equiv \tau_3$. By Lemma 30, $\vdash \theta(\tau_1) \diamond$. By Lemma 23, $\vdash \theta(\tau_1) \downarrow \tau_5 \to \tau_6$ with $\vdash \tau_5 \equiv_\tau \theta(\tau_2)$ and $\vdash \tau_6 \equiv_\tau \theta(\tau_3)$. By Lemma 31, $\vdash \tau_2 \diamond$. By induction hypothesis, $\Gamma_1, x : \tau_5 \vdash \theta(e_1) : \tau_7$ with $\vdash \tau_7 \equiv_\tau \theta(\tau_4)$. By Lemma 24, $\vdash \theta(\tau_4) \equiv_\tau \theta(\tau_3)$, hence we have $\vdash \tau_7 \equiv_\tau \tau_6$. Since both of τ_7 and τ_6 are located types, we have $\vdash \tau_7 \equiv \tau_6$. Hence we have $\Gamma_1 \vdash \theta(\texttt{fun}\ x \to e_1 : \tau_1) : \tau_5 \to \tau_6$. □

Proposition 13 (Type preservation for the reduction $\overset{\texttt{vpth}}{\to}$) *Let p have located variables and contain no module variables. If $\vdash p.l : \tau$ and $\vdash p \rightsquigarrow p_1$ and $\vdash p_1 \mapsto (\theta, \texttt{struct}\ \ldots \texttt{val}\ l = e \ldots\ \texttt{end}^i)$, then $\vdash \theta(e) : \tau'$ with $\vdash \tau \equiv \tau'$.*

Proof. By Proposition 10, $\vdash p \rightsquigarrow_s p_1$. By Proposition 12, $\vdash p_1$ wf. By hypothesis, we have $\vdash p.l :: \tau$, hence we have $\vdash e :: \tau_1$ and $\vdash \theta(\tau_1) \downarrow \tau$. By Lemma 32, we have $\vdash \theta(e) : \tau_2$ with $\vdash \theta(\tau_1) \equiv_\tau \tau_2$. Since both τ and τ_2 are located types, $\vdash \tau \equiv \tau_2$. □

Module expr.	E	::=	$\ldots$	
		$\mid$	$(E : S)$	*sealing*
Signature expr.	S	::=	$\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}$	*signature type*
		$\mid$	$\texttt{functor}(X : S) \rightarrow S$	*functor type*
Specifications	S	::=	$\ldots$	
		$\mid$	$\texttt{module}\ M : S$	*module specification*

Figure 19: Syntax for the module language with type abstraction

8 Towards type abstraction

In this section, we informally present an extension of *Remonade* with type abstraction, by giving concrete syntax. A formal account for the extension, including a type system and a soundness proof, is found in [21].

We present the syntax for the module language with type abstraction in Figurere 19, which only contains constructions that differ from those given in Figurere 2.

Now a module expression can be a *sealing* of the form $(E : S)$, which seals the module expression E with the signature S. To seal functors with signatures, we introduce functor types; to seal nested structures, we allow signature types to contain module specifications. Note that signature types are extended with declarations of self variables. Using the self variables, components of signatures can refer to each other recursively.

Returning to the example in Figurere 1, we can now seal the module `TreeForest` with the opaque signature:

```
sig  (Z)
 module Tree : sig type t val split : t → Z.Forest.t end
 module Forest : sig
  type t val incr : Z.Tree.t → t → t val sweep : t → t end
end
```

We reiterate that this opaque signature is sufficient to type checking `TreeForest` and to enforce type abstraction; our type system does not require programmers to write the transparent signature of `TreeForest` in addition to the opaque one, like other type systems do.

Having given the concrete syntax, it would be clear that the type expansion algorithm presented in the paper can detect cyclic type specifications in signatures, in the same way that it detects cyclic type definitions in modules.

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. Here, we first examine existing work on these issues, then give an overview of work on *mixin modules*, another proposal for introducing recursion to ML-like module systems.

9.1 Type systems

To the best of our knowledge, no formal work has been examined a type system for recursive modules with applicative functors, except for the experimental implementation in Objective Caml [18], nor proposed type inference for recursive modules whether functors are applicative or generative. Type abstraction inside recursive modules is not in the scope of this paper. Hence, we do not give a detailed comparison in that respect, but defer it to [21].

The experimental implementation of recursive modules in Objective Caml is most related to our work. Indeed, we followed it in large part when designing *Remonade*. 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. (We think the "-i" option of O'Caml compiler, which infers signatures of modules, is useful during the development of programs. 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.)

The type checker of O'Caml does not terminate for the program:

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

This is the same functor we examined in Example 1 from Section 5. The potential for divergence when typing O'Caml modules is well-known, but is assumed to be a rare phenomenon in practice. Recursive modules seem to make the problem much more acute. Moreover, the type checker accepts the following program, which we examined in Section 6.3.1.

```
module G : functor(X : sig type t end) →
 sig
  type t = X.t * X.t
  type u = [‘B of t | ‘C of G(G(X)).u ]
 end
= functor(X : sig type t end) →
 struct
  type t = X.t * X.t
  type u = [‘B of t | ‘C of G(G(X)).u ]
 end
module M = G(struct type t = int end)
```

A use of the type `M.t` can cause divergence when the type checker needs to reason about the type. Since O'Caml does not support type inference for recursive modules, one may have to write duplicate signatures for the same module as we examined in Section 2.

Crary, Harper and Puri [4] (revisited later in [7]) gave a foundational type theoretic account of recursive modules. They analyzed recursive modules in the context of a phase-distinction formalism [12], 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. Due to this requirement, one cannot enforce type abstraction inside recursive modules.

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 as we described in Section 2. One can not enforce type abstraction inside recursive modules in his system.

Dreyer [6] 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. As we discussed in Section 1, his system restricts uses of structural types. Programmers have to write duplicate signatures for the same module as in the other systems.

9.2 Initialization

Boudol [3], Hirschowitz and Leroy [14], and Dreyer [5] 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 programs:

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

and

```
struct (Z) val l = fun x → x + Z.m val m = Z.l(3) end
```

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

From a technical point of view, the reconstruction algorithm and their type systems are orthogonal. Hence, we think that it is possible to combine both to obtain a stronger type system. This looks like a promising avenue for future work.

9.3 Mixin modules

Mixin modules have been investigated as a new construct for module languages, where recursive linking is primal and hierarchical linking is special.

Duggan and Sourelis [8, 9] proposed mixin modules specifically for the Standard ML language. Their mixin modules can split individual definitions of a datatype and a function into separate mixins: constructors of a datatype can be defined in several mixins; a function defined by cases on a datatype can be defined in several mixins, each mixin defining only certain cases. An operator for linking mixins is provided, to stitch together these constructors and cases to form a single datatype definition and a single function definition. Although we share the same motivation in principle, the ways we address them are rather different.

Ancona and Zucca [2] developed a theory for mixin modules in a call-by-name setting. The work focuses on value level recursion of mixin modules, and is closely related to work on initialization of recursive modules.

Odersky et al. designed a calculus, called νObj [22], for objects and classes, which is implemented in Scala [1]. Although the concrete syntax is rather different, νObj supports most mechanisms of ML-modules, including higher-order functors and nested structures with type components. It also allows recursion between modules. *Remonade* and νObj is closely related in that the path expansion plays a crucial role in defining type equality. The type system of νObj is undecidable. It traces type abbreviations in the intuitive way; this is a reason for the undecidability since there is the potential of cyclic type definitions.

10 Conclusion and Future work

In this paper, we presented a type system for recursive modules, which can reconstruct the necessary type information during type checking, instead of relying on signature annotations from programmers. The type system is provably decidable and sound for a call-by-value operational semantics.

The main contribution of the paper is the expansion algorithms, which are provably terminating and can resolve path references. In addition to that the algorithms enable type inference, they can also detect cyclic type definitions in recursive modules. Using the algorithms, we designed a decidable judgment for type equivalence.

The technical development of the paper constitutes the basis of the type system we develop in [21], in which we extend *Remonade* with type abstraction.

There is still a lot of work to be done to obtain a fully practical system. Here we give a brief overview of ongoing and future work.

10.1 Type inference for the core language

We can define a type inference algorithm for the core language by combining a standard type inference algorithm and an equivalent of the type reconstruction algorithm. We have to be careful about polymorphism and well-formedness. To obtain as much polymorphism as possible, we need to determine the best order for type inference. The equivalent of the reconstruction algorithm returns, instead of a type, the order along which it looks up value components in recursive modules during the reconstruction. Using this order, we can build a call graph of functions (represented by a directed graph), which expresses how functions in modules depend on each other. This graph gives us useful information to control inference: 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 can generalize types in a connected component before moving on to typing the next one. We must also check for well-formedness of types, as module variables should not escape their scope. Explicit type annotations can be used to break dependencies in the call graph, and allow polymorphic recursion. Note that these annotations cannot be completely avoided, as type inference for polymorphic recursion is known to be undecidable [13].

10.2 Separate type checking

Although we have not discussed this in the paper, the type system allows separate type checking. In short, we only have to extend the look-up judgment (Figure 4) so that the judgment informs the type system of signatures of modules which are type checked separately.

10.3 Taking fixed-points of functors

The call-by-value strategy of the ground normalization disables programmers from taking fixed-points of functors. For instance, assume the functor `F` defined as:

```
module F = functor(X : sig type t val f : t → int end) →
 struct
  type t = A | B of X.t
  let f = fun x → case x of A ⇒ 0 | B y ⇒ 1 + X.f y
 end
```

The type system of the paper cannot type the following module definition, since the ground normalization cannot safely expand the module path `F(M)`. `module M = F(M)`

When we extend *Remonade* with type abstraction as explained in Section 8, the extended type system can type the definition below.

```
module N = (F(N) : sig type t val f : t → int end)
```

In short, by writing signatures explicitly, one can break possible cycles in type definitions that may arise from connecting the result of the instantiation of `F` to the argument.

This style of programming is useful for the development of extensible programs. In [21], we give a concise and type safe solution to a variation on the expression problem [27] using this style; we define open recursion using functors, and close the recursion by taking fixed-points of these functors. This solution also requires applicative functors, which comes in support of our design choice.

10.4 Lazy modules

The operational semantics presented in the 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. It might not be natural

for practical programming, however. Currently 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 programmers a way to initialize recursive modules. Moreover, this semantics seems to give us a uniform way to handle statically and dynamically loaded modules, that is, we can trigger initialization of a module by accessing its components whether the module is loaded statically or dynamically. We believe that our expansion algorithms are useful for efficient and safe implementation of lazy recursive modules. We need more investigation on this topic.

References

[1] P. Altherr, E. Burak, N. Mihaylov, M. Odersky, M. Schinz, and M. Zenger. The Scala Programming Language, version 2.0. Software and documentation available on the Web, `http://scala.epfl.ch/`, 2006.

[2] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[3] G. Boudol. The recursive record semantics of objects revisited. *Journal of Functional Programming*, 14(3):263–315, 2004.

[4] K. Crary, R. Harper, and S. Puri. What is a Recursive Module? In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1999.

[5] D. Dreyer. A Type System for Well-Founded Recursion. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 2004.

[6] D. Dreyer. Recursive Type Generativity. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2005.

[7] D. Dreyer, R. Harper, and K. Crary. Toward a Practical Type Theory for Recursive Modules. Technical report, Carnegie Mellon University, 2001.

[8] D. Duggan and C. Sourelis. Mixin modules. In *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1996.

[9] D. Duggan and C. Sourelis. Parameterized Modules, Recursive Modules and Mixin modules. In *ACM SIGPLAN Workshop on ML*, 1998.

[10] J. Garrigue. Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*, 1998.

[11] R. Harper and M. Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[12] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[13] F. Henglein. Type Inference with Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[14] T. Hirschowitz and X. Leroy. Mixin modules in a Call-by-Value Setting. In *European Symposium on Programming:LNCS*, volume 2305, pages 6–20. Springer-Verlag, 2002.

[15] X. Leroy. Manifest types, modules, and separate compilation. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 1994.

[16] X. Leroy. Applicative functors and fully transparent higher-order modules. In *ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, 1995.

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

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

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

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

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

[22] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *European Conference on Object-Oriented Programming:LNCS*. Springer-Verlag, 2003.

[23] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.

[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 *ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2001.

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

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