A Syntactic Type System for Recursive Modules

Hyeonseung Im
Pohang University of Science and Technology (POSTECH), Korea
genilhs@postech.ac.kr

Keiko Nakata
Institute of Cybernetics at Tallinn University of Technology, Estonia
keiko@cs.ioc.ee

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

Sungwoo Park
Pohang University of Science and Technology (POSTECH), Korea
gla@postech.ac.kr

Abstract

A practical type system for ML-style recursive modules should address at least two technical challenges. First, it needs to solve the double vision problem, which refers to an inconsistency between external and internal views of recursive modules. Second, it needs to overcome the tension between practical decidability and expressivity which arises from the potential presence of cyclic type definitions caused by recursion between modules. Although type systems in previous proposals solve the double vision problem and are also decidable, they fail to typecheck common patterns of recursive modules, such as functor fixpoints, that are essential to the expressivity of the module system and the modular development of recursive modules.

This paper proposes a novel type system for recursive modules that solves the double vision problem and typechecks common patterns of recursive modules including functor fixpoints. First, we design a type system with a type equivalence based on weak bisimilarity, which does not lend itself to practical implementation in general, but accommodates a broad range of cyclic type definitions. Then, we identify a practically implementable fragment using a type equivalence based on type normalization, which is expressive enough to typecheck typical uses of recursive modules. Our approach is purely syntactic and the definition of the type system is ready for use in an actual implementation.

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

General Terms Languages, Theory, Design

Keywords Type systems, modules, recursion, abstract types, weak bisimulations

1. Introduction

Modular programming is considered the key enabling technology for large-scale software development and maintenance. Consequently, a modern programming language usually provides its own module system to facilitate modular programming. The ML module system [22], among others, provides powerful support for modular programming and data abstraction through nested modules, higher-order functors (functions from modules to modules), and abstract types.

While regarded as a powerful mechanism for modular programming, the ML module system does not traditionally support recursive modules. This is in sharp contrast to typed object-oriented languages such as Java, which put almost no restriction on recursion between components. One could argue that recursive modules are harder to support in ML precisely because of the strengths of the language, such as its type system, which supports polymorphic type inference, or the absence of default values such as null pointers, which makes more difficult defining a semantics for recursive module initialization. Due to the lack of recursive modules, ML programmers often have to consolidate conceptually separate mutually recursive definitions into a single module, thus compromising modular programming. In response, recently, several authors have proposed recursive module extensions to ML [6–9, 21, 26, 28, 34], some of which are successfully implemented in Moscow ML [2] and OCaml [3].

Despite such abundant literature and practical implementations, however, the number of examples of using recursive modules reported in the literature is still rather limited: expression/binding recursive modules [6], Okasaki's bootstrapped heaps [31], polymorphic recursion [21], tree/forest recursive data structures [28], and the expression problem [13]. While they give us some idea of how recursive modules can be useful in practice, their scarcity may also demonstrate that existing systems are either too cumbersome or not expressive enough to accommodate other interesting examples.

Specifically, to reach its full power, a type system for recursive modules needs to address *the double vision problem* [9] and *cyclic type definitions*. The double vision problem refers to the inability to identify the external name of an abstract type with its internal name inside the recursive module where the type is defined. It has proven difficult to solve and only Dreyer [8] and Montagu and Rémy [26] successfully solve it in a type-theoretic way.

Independently of recursive type definitions, recursive modules allow us to indirectly introduce recursion in types that span across module boundaries. The potential presence of arbitrary cyclic types would require equi-recursive type constructors of higher-kind, for which no practical algorithm for type equivalence is known [6], and non-contractive types, for which the equational theory is not known. Moreover, one has to decide what to do with type cycles hidden through opaque sealings. Most previous proposals choose to reject type cycles, whether or not they are hidden through opaque sealings, in a conservative way at the expense of reducing the flexibility and expressive power of recursive modules. Here, we use the term "conservatively" because it is generally impossible to accurately detect cycles without breaking abstraction. For example, Dreyer [8] supports only a restricted form of functor fixpoints, and Montagu and Rémy [26] do not support functor fixpoints returning structures at all. As we will see in Sections 2 and 3, however, there are important examples that are rejected by these approaches.

In this paper, we start our analysis by providing three practical examples illustrating basic uses of recursive modules: tree and forest recursive data structures adapted from [28]; encodings of classes and objects using functors and functor fixpoints; and an implementation of a compiler backend. These examples demonstrate that, when sufficiently expressive, recursive modules have a wide range of applications and help us improve modularity. We then address the double vision problem and decidability of typechecking separately. To solve the double vision problem, we introduce a notion of *path substitutions*, which are mappings from external names to internal names of abstract types. Locally enriching the typing context with path substitutions proves sufficient to solve the double vision problem. To avoid restricting expressivity, we allow the type system to account for arbitrary cyclic type definitions. Specifically, we use weak bisimilarity [23] on a labeled transition system on types [28] to build our type equivalence relation. Using weak bisimilarity, we generalize extensional equality, or bisimilarity, on infinite trees to account for type abbreviations. This type system is proven sound but may be difficult to implement in general. We then propose an algorithmic type system by rejecting *transparent type cycles* (as defined in Section 3.2). The algorithmic type system is sound and complete with respect to the one based on weak bisimilarity if the program does not contain transparent type cycles, and can still typecheck all our motivating examples.

The main motivation of this work is to design a type system that typechecks flexible uses of recursive modules already supported by OCaml, and possibly more. (OCaml 3.12.0, the most recent version, does not completely solve the double vision problem.) Hence, we focus only on typing issues and do not investigate a logical account of recursive modules in contrast to Dreyer [7, 8] and Montagu and Rémy [26], who give logical interpretations of recursive type generativity and abstract types, respectively.

Our contributions are summarized as follows:

- We provide three practical examples that are useful for understanding basic uses of recursive modules.
- We propose a novel type system that solves the double vision problem and rejects only transparent type cycles, which is sufficient to keep typechecking practically implementable and does not impede expressivity. We also define a light-weight call-by-value operational semantics and show that the type system is sound with respect to it.
- Our type system is purely syntactic and may thus act as a reference for the implementation of an external programmable module language. We believe that our type system is simpler than previous proposals.

The rest of the paper is organized as follows. Section 2 gives three practical examples of using recursive modules. Section 3 presents technical challenges, namely the double vision problem and cyclic type definitions. It also discusses previous approaches to these challenges and the key ideas of our approach. Section 4 gives the concrete syntax of our language. Section 5 explains the type system, and Section 6 shows the operational semantics and a type soundness result. Section 7 presents an algorithmic type equivalence relation. Section 8 discusses separate compilation of mutually recursive modules and type reconstruction for our language. Section 9 discusses related work and Section 10 concludes.

2. Examples of Using Recursive Modules

This section illustrates typical uses of recursive modules and gives us a baseline for what a practical type system for recursive modules should handle. All the examples typecheck in OCaml 3.12.0 (with Figures 3 and 4 being given appropriate definitions for the omitted part).

```
module rec Tree : sig
  type t
  val max : t -> int
end = struct
  type t = Leaf of int | Node of int * Forest.f
  let max x = match x with Leaf i -> i
    | Node (i, f) -> let j = Forest.max f in
        if i > j then i else j
end
and Forest : sig
  type f
  val max : f -> int
end = struct
  type f = Tree.t list
  let rec max x = match x with [] -> 0
    | hd :: tl -> let i = Tree.max hd in
        let j = max tl in if i > j then i else j
end
```

Figure 1. `Tree` and `Forest` mutually recursive modules

2.1 Basic Terminology

Basic modular programming constructs in the ML module system are structures, signatures, and functors. A structure, or a module, is a collection of related declarations such as definitions of datatypes and associated operations. A functor, or a parameterized module, is a function from structures to structures. A signature and a functor signature, which are also called module types, specify an interface for a structure and a functor, respectively. An opaque signature sealing allows one to hide the implementation details of a module behind the signature; hence, one may define abstract types via sealings. The notion of modules extends to nested modules which allow modules as components, and the notion of functors extends to higher-order functors which take functors as arguments.

2.2 `Tree` and `Forest` Mutually Recursive Modules

The first example is `Tree` and `Forest` mutually recursive modules in Figure 1, which is adapted from [28]. `Tree` is a module for trees whose leaves and nodes are labeled with integers while `Forest` is a module for a list of those integer trees. Components of `Tree` and `Forest` refer to each other in a mutually recursive way: type components `Tree.t` and `Forest.f` refer to each other and similarly value components `Tree.max` and `Forest.max`, which find the maximum integer that a given tree and forest contain, respectively. In this example, we enforce type abstraction inside the recursion by sealing each module with a signature specifying an abstract type individually.

The main benefit of using recursive modules in this example is the modularity between `Tree` and `Forest`. Without recursive modules, the programmer has to consolidate two conceptually separate modules, each of which may usually include dozens of function definitions, into a single module. This merges two separate namespaces into a single one, thereby reducing the modularity and readability of the program. For example, similar functions in `Tree` and `Forest` with the same name such as `max` should now be renamed `max_tree` and `max_forest` to avoid a name collision.

`Tree`/`Forest`-style recursive modules are indeed used in a real world project such as Amthing [1] which is a multi-threaded GUI library for OCaml. For example, recursive modules `Focus`/`Focusee` and `Signal`/`Event` are a variant of the `Tree`/`Forest` pattern.

```
module type Point = sig
  val get_x : unit -> int
  val move : int -> unit
end
module PointF (X : sig end) = struct
  let x = ref 0
  let get_x () = !x
  let move d = x := !x + d
end

let new_point () =
  (module PointF(struct end) : Point)

module type Show = sig
  val show : unit -> string
end
module ShowF (P : Point) = struct
  let show () =
      Printf.sprintf "x = %i" (P.get_x ())
end

module type ShowPoint = sig
  include Point
  include Show
end
module ShowPointF (P : ShowPoint) = struct
  include PointF(P)
  include ShowF(P)
end

(* Create a new showpoint, by tying the knot *)
let new_showpoint () =
  let module M = struct
    module rec SP : ShowPoint = ShowPointF(SP)
  end in (module M.SP : ShowPoint)

let sp = new_showpoint ()
let a = let module SP = (val sp : ShowPoint) in
    SP.move 2; SP.show ()
let sp_as_p =
    (module (val sp : ShowPoint) : Point)
```

Figure 2. Encodings of classes and objects

2.3 Encodings of Classes and Objects

The second example is an encoding of classes and objects using functors and functor fixpoints. The basic idea is to turn classes (or traits [35]) into functors receiving the object to be constructed and returning a list of methods. Figure 2 demonstrates this idea using recursive modules and first-class modules. First-class modules, as provided by Moscow ML and OCaml, allow us to pack a module as a value of the core language, which can later be dynamically unpacked with the same signature. While the type system we will present does not support first-class modules, they can easily be added. A more powerful encoding would represent object state as a separate abstract type, avoiding the need for first-class modules and permitting more advanced operations such as binary methods, but at the cost of more complexity.

Figure 2 first defines a `Point` interface and the corresponding class, defined as a functor `PointF`. In this basic case, where methods do not call one another, a functor is sufficient to allow our points to have some state. We then define a constructor `new_point` by applying `PointF` to an arbitrary argument, and wrapping the result in a first-class module using the packing construct (**module** ... : `Point`).

We now see how traits and inheritance can be encoded. We first define an interface `Show` containing a single `show` method. The corresponding trait is a functor `ShowF` which receives a module satisfying the signature `Point` and uses its method to define an implementation for `show`. The next step, corresponding to inheritance, is to merge this trait into the `PointF` class, which is done by defining a functor taking an argument satisfying the signature `ShowPoint` (the union of `Show` and `Point`) and returning the union of `PointF` and `ShowF`. The last step is to define the constructor `new_showpoint`. Since methods may now call one another, we need to tie the recursive knot by taking the fixpoint of `ShowPointF`. For syntactic reasons, this has to be done inside a local module, before returning this fixpoint `M.SP` as a first-class module.

Once we create such an object, we can use it as a normal one. Syntactically, we first need to unpack it as (**val** sp : `ShowPoint`) to be able to call its methods. Module subtyping also provides for object subtyping, and can be used by unpacking and repacking.

Using recursive modules and first-class modules, we can similarly encode other features of object-oriented languages such as method overriding, virtual methods, or private methods.

2.4 An Implementation of a Compiler Back-end

The third example is an implementation of a compiler back-end[1] in Figures 3 and 4, where we omit many type and function definitions. Basically, it implements a compilation pass for translating a machine independent intermediate language into machine dependent ones, with different instructions.

The basic idea is similar to class and object encodings in Section 2.3: we define a functor which takes the target language and machine-specific translation functions and returns

```
type id = ...                    (* identifier *)
type reg = ...             (* register location *)
and reg_s = reg list
type env = (id * reg_s) list

(* The source language *)
type exp = Cint of int | Cadd of exp * exp
         | Cif of exp * exp * exp | ...
type test = Leq | Lne | ...

module type S = sig
(* The target language *)
  type spec_op
  type instr = Ladd | ... | Lspec of spec_op
  and instr_s = instr list
      ⋮            (* a bunch of declarations *)
  val sel_cond : exp -> test * exp * exp
  val sel_add : env -> exp -> exp -> instr_s
              -> instr * reg_s * instr_s
end

module FSelectgen (X : S) = struct
(* emit_exp : env -> exp -> X.instr_s  *)
(*            -> reg_s * X.instr_s      *)
  let emit_exp env e i = match e with
    | Cadd (e1, e2) ->
        ... X.sel_add env arg1 arg2 i ...
    | Cif (econd, eif, eelse) ->
      let (cond, e1, e2) = X.sel_cond econd in
                    ⋮
  let sel_cond e = ...
  let sel_add env e1 e2 i =
      let (r1, i1) = emit_exp env e1 i in
      let (r2, i2) = emit_exp env e2 i1 in
        (X.Ladd, concat r1 r2, i2)
(* The entry point to the translation *)
  let emit e = emit_exp empty_env e (init ())
end
```

Figure 3. A machine independent translation

a list of translation functions tailored to the target machine. The functor `FSelectgen`[2] in Figure 3 is such a generic functor, which implements translation functions for the general machine. The target language `instr` contains, besides machine independent instructions, a special instruction `Lspec` to support machine specific instructions. While `FSelectgen` takes two functions `sel_cond` and `sel_add` as arguments, which may exploit the optimizing instructions for conditionals and arithmetic that the target machine provides, it also defines default implementations, which do not exploit such optimizing instructions, for both functions. Note that function `emit_exp`, which translates a source language ex-

[1] It is inspired by the OCaml implementation and was suggested by Xavier Leroy.

[2] The OCaml implementation uses its object system rather than recursive modules, and a virtual class `selector_generic` in the selectgen.ml file is a counterpart of the functor `FSelectgen`.

```
module rec ProcA : sig ... end = struct
  module Selectgen = FSelectgen(ProcA.Selection)
  module Selection = struct
    type spec_op = Laddi of int
    type instr = Ladd | ... | Lspec of spec_op
    and instr_s = instr list
        ⋮          (* a bunch of declarations *)
    let sel_add env e1 e2 i = match e1 with
      | Cint n ->
        let (r, i') = Selectgen.emit_exp env e2 i
          in (Lspec (Laddi n), r, i')
      | _ -> Selectgen.sel_add env e1 e2 i
    let sel_cond = Selectgen.sel_cond
    let emit = Selectgen.emit
  end
end
```

Figure 4. A translation tailored to a specific target machine with a special addition instruction

pression, uses machine dependent functions `X.sel_add` and `X.sel_cond` rather than their default implementations.

Figure 4 shows an example of instantiating the general functor `FSelectgen` to a specific machine. The recursive module `ProcA` is a machine dependent translation that is obtained by taking the fixpoint of `FSelectgen`. The target machine provides a special instruction `Laddi` for the addition with integer constants, so the module `Selection` defines a new implementation for the function `sel_add` which exploits the instruction `Laddi`. By tying the recursive knot between `FSelectgen` and `ProcA.Selection`, we obtain translation functions tailored to the target machine. The module `Selection` reuses the default implementation of `sel_cond` provided by `FSelectgen`. For different target machines with different optimizing instructions, we can similarly obtain machine dependent translations by taking the fixpoint of `FSelectgen`.

We believe that all these examples provide strong evidence that recursive modules are useful in practice and give us more expressive power to improve modularity.

3. Technical Challenges

This section presents technical challenges for typechecking recursive modules, namely the double vision problem [9] and cyclic type definitions. As suggested in the introduction, we address these issues separately, which allows us to clarify the problems involved and propose a simpler solution. In the following, we first look at the double vision problem and then cyclic type definitions, examining previous approaches and explaining our approach along the way.

3.1 The Double Vision Problem

The double vision problem refers to the inability to identify the external name of an abstract type, which is cyclically imported, with its internal name inside the recursive module where it is defined. For example, consider Figure 5, a variant of Figure 1. Here, we slightly modify the definitions of types `Tree.t` and `Forest.f` so that we can use Figure 5 as our running example, expressible in our language: `Tree.t` is defined as a pair of an integer and a forest, where the forest represents the branches of a tree, while `Forest.f` as a datatype representing a list of trees. Hence, we need not extend the entire system with parameterized type definitions such as `'a list`, which is feasible but beyond the scope of this paper. Both modules `Tree` and `Forest` now contain a new function called `plant` but with a different type, which inserts a tree into a forest.

```
module rec Forest : sig
  type f
  val empty : unit -> f
  val plant : Tree.t -> f -> f
end = struct
  type f = Nil | Cons of Tree.t * f
  let empty () = Nil
  let plant x y = Cons (x, y)
end
and Tree : sig
  type t
  val empty : unit -> t
  val plant : int -> Forest.f -> Forest.f
           -> Forest.f
end = struct
  type t = int * Forest.f
  let empty () = 0, Forest.empty ()
  let plant i x y = Forest.plant (i, x) y
end
```

Figure 5. The double vision problem

The example does not typecheck in OCaml 3.12.0 because of the function `plant` in `Tree`. Specifically, the expression `(i, x)` is of type `int * Forest.f` while the function `Forest.plant` expects a value of type `Tree.t`, which is an abstract type and thus cannot be identified with `int * Forest.f`. Inside `Tree`, however, `Tree.t` is merely a recursive reference to, or equivalently an external name of, `t` which equals `int * Forest.f`. Hence, inside `Tree`, the type system should also identify `Tree.t` with `int * Forest.f`. The inability to do so is called the double vision problem. OCaml partially solves this problem by automatically adding a type equation pointing from the internal definition to the external definition, when typechecking the body of a recursive module. However, in OCaml, a type definition has at most one type equation and therefore this approach is only applicable to datatype definitions, but not to type abbreviations or datatype definitions with equations.

Previous approaches

To our knowledge, only RMC by Dreyer [8] and $F^{\curlyvee}$ by Montagu and Rémy [26] solve the double vision problem in a type-theoretic way. The key idea of RMC is to separate the creation of the name for an abstract type from the definition of the type. Specifically, by elaboration similar to the Definition of Standard ML [24], the RMC type system first creates a semantic type variable and assigns it to both external and internal names of an abstract type. The semantic type variable is then defined only once and the RMC type system ensures its definition to be visible only inside the module where the type variable is defined. For example, the RMC type system first assigns a new type variable α to both `Tree.t` and `t` inside `Tree`. Then, α is defined as `int * Forest.f` inside `Tree`. As `Forest.plant` expects a value of type `Tree.t`, or equivalently α, which is known to equal `int * Forest.f` inside `Tree`, the function `plant` in `Tree` typechecks in RMC. We remark that the type variable α is abstract outside `Tree`. The typability in RMC, however, depends on the order of definitions (see Section 3.2 in this paper for details). This is why we should define `Forest` before `Tree`. Indeed, if the order is reversed, the resultant program does not typecheck in RMC.

$\mathsf{F}^{\curlyvee}$ by Montagu and Rémy is a variant of explicitly typed System F extended with recursive types and values, intended as a core language for modules. $\mathsf{F}^{\curlyvee}$ decomposes the introduction and elimination of existential types into more atomic constructs. In $\mathsf{F}^{\curlyvee}$, one may refer to a single type component via using either its external name or internal name, and thus the double vision problem arises. $\mathsf{F}^{\curlyvee}$ addresses the problem simply by locally enriching the typing context with equations between external and internal names. For example, when typechecking the module `Tree`, $\mathsf{F}^{\curlyvee}$ locally extends the typing context with an equation $\forall(\mathtt{t} \lhd \mathtt{Tree.t} = \mathtt{int * Forest.f})$. The equation means that the external name `Tree.t` can be viewed internally as `t` which is defined as `int * Forest.f`. Hence, Figure 5 also typechecks in $\mathsf{F}^{\curlyvee}$ with the help of proper encodings. ($\mathsf{F}^{\curlyvee}$ does not support paths, so `t` and `Tree.t` should be represented as two distinct type variables.)

Our approach: Path substitutions

We address the double vision problem in a similar way to $\mathsf{F}^{\curlyvee}$: our type system also extends the typing context locally with path substitutions, which are mappings from external names to internal names of abstract types. This is in contrast to RMC, which modifies the typing context globally. Specifically, as in most previous work, we employ recursive structures with explicitly typed recursion variables. We, however, do not distinguish between forward and backward references, and allow components of structures to refer to each other (recursively) only via recursion variables. This makes module component access uniform and simplifies the use of path substitutions. To solve the double vision problem, when typechecking a recursive module, we add into the typing context a path substitution of its internal recursion variable for its external path. Then, when checking equivalence on abstract types, we convert their external names to internal names (whose definitions are available to the type system) by applying path substitutions, and thus identify their two different names.

```
module type ST(type u) = rec(Y)sig
  type t
  val plant : int -> u -> u -> u
end
module type SF(type u) = rec(Z)sig
  type f
  val plant : u -> Z.f -> Z.f
end
module type S = rec(X)sig
  module Tree : ST(X.Forest.f)
  module Forest  : SF(X.Tree.t)
end

rec(X : S)struct
  module Tree : ST(X.Forest.f) =
  rec(Y : ST(X.Forest.f) with
    type t = int * X.Forest.f)
  struct
    type t = int * X.Forest.f
    let plant (i : int) (x : X.Forest.f)
      (y : X.Forest.f) = X.Forest.plant (i, x) y
  end
  module Forest : SF(X.Tree.t) =
  rec(Z : SF(X.Tree.t) with
    type f = Nil | Cons of X.Tree.t * Z.f)
  struct
    type f = Nil | Cons of X.Tree.t * Z.f
    let plant (x : X.Tree.t) (y : Z.f)
      = Z.Cons (x, y)
  end
end
```

Figure 6. Encodings of `Tree` and `Forest` modules in our language

To illustrate how we exploit path substitutions in solving the double vision problem, consider Figure 6, an encoding of Figure 5 in the abstract syntax of our language (extended with top-level parameterized signatures [17] for concision). To simplify the code, we omit the `empty` function in both `Tree` and `Forest` modules. The code may look a bit verbose because it is written in the abstract syntax, but in practice, most of the type annotations can be automatically inferred, as discussed in Section 8.2. We wrap `Tree` and `Forest` modules in a top-level recursive module `X` and assign them recursion variables `Y` and `Z`, respectively. Both `Y` and `Z` are annotated with their principal signatures. To typecheck `plant` in `Tree`, from which the double vision problem arises, the type system should identify an abstract type `X.Tree.t`, which is the type of the first argument of `X.Forest.plant`, with `int * X.Forest.f`, which is the type of `(i, x)`. To do that, when typechecking `Tree`, the type system locally enriches the typing context with a path substitution $\mathtt{X.Tree} \mapsto \mathtt{Y}$. Then, by rewriting `X.Tree.t` to `Y.t`, which equals `int *`

X.Forest.f, the type system typechecks plant. In this way, we solve the double vision problem.

Our approach to the double vision problem differs from those of RMC and $\mathsf{F}^{\curlyvee}$ in the following two points. First, we address the problem in a purely syntactic way, as opposed to RMC adopting SML-style elaboration into semantic objects. Second, we address the problem in the context of path-based recursive modules, whereas $\mathsf{F}^{\curlyvee}$ considers a low-level core language in the style of System F, and has to be extended to encode full fledged ML modules supporting recursion. Our approach is close to the OCaml implementation of recursive modules, but our approach to handling path substitutions is more general and allows us to typecheck more programs.

3.2 Cyclic Type Definitions

Besides the double vision problem, another subtle issue in typechecking recursive modules is to determine what kinds of cyclic type definitions to support. Independently of recursive type definitions, recursive modules allow us to indirectly introduce recursion in types and also to hide it through opaque sealings. Hence, we need to distinguish different kinds of recursion in types. We say that a type cycle is *guarded* if it goes through, or is intercepted by, a datatype. Otherwise, it is *unguarded*. We say that an unguarded type cycle is *opaque* if it is hidden through opaque sealings. Otherwise, it is *transparent*.[3]

Examples of type cycles

The example below, written in OCaml syntax, illustrates guarded and transparent type cycles in an observable signature.

```
sig
  module rec X : sig
    type s = Int of X.s        (* guarded *)
    type t = X.t               (* transparent *)
    type u = int * X.u         (* transparent *)
  end
end
```

In the next example, the cycle may appear to be hidden through an opaque sealing, but actually it is still observable (hence transparent). The reason is that our type system identifies types t and L.t using path substitutions introduced to solve the double vision problem. In OCaml, where the double vision problem remains, the cycle would be left opaque.

```
module rec L : sig type t end =
 struct type t = int * L.t end (* transparent *)
```

In our system, opaque type cycles have to be built in a more subtle way. Here, we use mutual recursion between two modules to create an opaque type cycle. To make it transparent, both type definitions are needed simultaneously, but only one of them is available in each of M and N.

[3] Dreyer [8] uses the term "transparent" for "unguarded". We think our naming better matches the intuition that a transparent type cycle should be observable without peeking inside opaque sealings.

```
module rec M : sig type t end =
  struct type t = int * N.t end  (* opaque *)
and N : sig type t end =
  struct type t = int * M.t end  (* opaque *)
```

The distinction between the two last examples is semantical. In our system, type abbreviations, even cyclic ones, are handled in a structural way. In particular, inside L, L.t would be equal to any other type defined as an infinite sequence of int's such as type s = int * s. On the other hand, while M.t and N.t are related to each other inside M and N, they have no such infinite expansion.

Problems with unguarded type cycles

Transparent type cycles are problematic for keeping typechecking practical. For example, Crary et al. [6] support transparent type cycles such as the above X.u. For this, they extend their type theory with equi-recursive type constructors of higher kind. There is, however, no known practical algorithm for checking their equivalence.

While opaque type cycles are harmless with respect to typechecking, they may still introduce difficulties when proving type soundness. Indeed, proving a type preservation lemma requires proving typability in the presence of reduction, which may require removing opaque sealings. After removing sealings, all opaque type cycles become transparent. Hence, if we do not reject opaque type cycles altogether in the surface language, we eventually need to handle transparent ones when proving type soundness.

To reject opaque type cycles, however, may reduce the flexibility and expressive power of recursive modules, because it is generally impossible to detect such cycles without breaking abstraction. This is explained in the example below.

```
module type S = sig type t end
module F (X : S) : S = struct type t = X.t end
module G (X : S) : S = struct type t = int end
module rec Ffix : S = F(Ffix)
module rec Gfix : S = G(Gfix)
```

Ffix creates an opaque type cycle, whereas Gfix does not. Yet, they are both defined as fixpoints of functors with the same signature functor (X : S) -> S. Therefore, without peeking at the actual definitions of F and G, we cannot tell Ffix from Gfix and cannot reject Ffix while accepting Gfix. We can reject opaque type cycles conservatively only by rejecting some modules that do not contain such cycles at all.

Dreyer's approach

The RMC type system of Dreyer [8] rejects unguarded type cycles altogether conservatively, whether they are transparent or opaque. The reason is that RMC removes opaque sealings in its operational semantics (thus making opaque type cycles transparent) and uses an equational theory in which transparent type cycles are problematic [7]. Hence,

```
module Set (X : sig type t end)
  : sig type t = X.t type f end =
struct
  type t = X.t
  type f = Nil | Cons of t * f
end

module rec Tree : sig type t end
  = struct type t = int * Forest.f end
and Forest : sig type t = Tree.t type f end
  = Set(Tree)
```

Figure 7. Cyclic type definitions

the potential presence of unguarded type cycles has to be prevented. Specifically, in RMC, the internal definitions of abstract types in sealed modules and the type components of the argument modules in functor applications may not depend on any type variables marked as undefined in the typing context. A type variable is said to be undefined if the typechecker has not accepted its definition yet. In this way, RMC guarantees that even when opaque sealings are stripped away, there are still no transparent type cycles. As solving the double vision problem corresponds to locally stripping away opaque sealings in some sense, this approach is also compatible with Dreyer's approach to the double vision problem.

Dreyer's approach of rejecting unguarded type cycles, however, imposes restrictions on uses of functor fixpoints. Note that functor fixpoints are crucial for encoding objects and classes as shown in Figure 2 and useful for flexible code reuse as shown in Figure 4. In fact, Figure 4 also hides the double vision problem since types such as `spec_op` and `instr` have external names (through `ProcA`) as well as internal names (insides the module `Selection`). To illustrate, consider Figure 7, which is a variant of `Tree` and `Forest` modules defined using functor fixpoints. For clarity, we include only type definitions. In Figure 7, `Tree.t` and `Forest.f` refer to each other in a mutually recursive way: `Tree.t` is defined as `int * Forest.f`, and `Forest.f` as a datatype using `Tree.t` through the functor application `Set(Tree)`. The cycle between `Tree.t` and `Forest.f` is guarded because it is intercepted by the datatype `Forest.f`. The RMC type system, however, rejects Figure 7. The reason is that the internal definition of the abstract type `Tree.t` depends on the type variable for `Forest.f` which is marked as undefined in the typing context. To reverse the order of definitions does not help here because then the type variable for `Tree.t` is marked as undefined at the point of the application `Set(Tree)`.

To exploit functor fixpoints in RMC, one should write them in accordance with the following conditions. First, given a functor application `F(M)`, the module `M` must be defined before the application. For example, the first example below is allowed, while the second and third are not:

```
module rec M : S = ...  and N : T = F(M)
module rec N : T = F(M) and M : S = ...
module rec M : S = F(M)
```

Here, we assume that either the signature `S` declares an abstract type, or a type component of `M` refers to an abstract type of `N`. Second, if a type component `t` of the (argument) module `M` depends on an abstract type of `N` (whose definition appears after the definition of `M`), `t` must be defined as a datatype. The second condition similarly applies to general mutually recursive modules. For example, in Figure 5, if we change the order of definitions for modules `Tree` and `Forest`, they do not typecheck in RMC. This explains why we had to first define `Forest` before `Tree`, not vice versa, in Figure 5.

Our approach: Weak bisimilarity

In this paper, we accept all opaque type cycles. As a result, we need to deal with transparent type cycles in the soundness proof. We therefore first work with a type system permitting transparent type cycles and prove its type soundness. Then, we identify a practically implementable fragment, producing an algorithmic type system, by rejecting transparent type cycles. As transparent type cycles are observable, we can accurately detect them without peeking inside opaque sealings.

To account for transparent type cycles, we define a type equivalence relation using weak bisimilarity [23]. Using weak bisimilarity, we generalize extensional equality, or bisimilarity, on arbitrary (non-regular) infinite trees to account for type abbreviations. Roughly speaking, two types are equivalent if they have the same tree-like expansions by unfolding type abbreviations. For instance, types `t` and `s` are equivalent with respect to type abbreviations **type** `t = int * int`, **type** `s = u * u`, and **type** `u = int`. Weak bisimilarity also naturally handles the possibility that unfolding type abbreviations goes on forever without producing constructors, such as for **type** `t = t`. This allows us to step aside the problem with non-contractive types [26]. We still need to avoid equating all such vacuous cycles by carefully crafting their semantics.

We prove type soundness by a standard proof technique for weak bisimilarity, i.e. finding a weak bisimulation. The algorithmic type system is then obtained by additionally rejecting transparent type cycles, which renders type equivalence based on weak bisimilarity easily decidable.

Comparison with OCaml and $\mathsf{F}^{\curlyvee}$

Our approach to type cycles was inspired by OCaml, which also rejects only transparent type cycles. Due to our solving the double vision problem, however, we detect more transparent type cycles, which are opaque in OCaml, and therefore end up rejecting more of them. The soundness of OCaml's recursive modules is not yet proved, but here again weak bisimilarity may be used.

The handling of type cycles in $\mathsf{F}^{\curlyvee}$ [26] is harder to compare because $\mathsf{F}^{\curlyvee}$ itself is only a core language. Moreover,

Module paths			
p, q	$::=$	X, Y, Z	recursion variable
	$\mid$	$p.M$	
Module expressions			
m	$::=$	$\mathsf{rec}(X : S)s$	recursive structure
	$\mid$	$\mathsf{functor}\ (X : S) \to m$	functor
	$\mid$	$p_1(p_2)$	functor application
	$\mid$	$(m : S)$	opaque sealing
	$\mid$	p	module path
Structure bodies			
s	$::=$	$\mathsf{struct}\ d_1\ \ldots\ d_n\ \mathsf{end}$	structure definition
Definitions			
d	$::=$	$\mathsf{module}\ M = m$	module definition
	$\mid$	$\mathsf{datatype}\ t\ [= \tau'] = c\ \mathsf{of}\ \tau$	datatype definition
	$\mid$	$\mathsf{type}\ t = \tau$	type abbreviation
	$\mid$	$\mathsf{let}\ l = e$	value definition
Module types			
S	$::=$	$\mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}$	signature
	$\mid$	$\mathsf{functor}\ (X : S_1) \to S_2$	functor signature
Specifications			
D	$::=$	$\mathsf{module}\ M : S$	module spec.
	$\mid$	$\mathsf{datatype}\ t\ [= \tau'] = c\ \mathsf{of}\ \tau$	datatype spec.
	$\mid$	$\mathsf{type}\ t = \tau$	manifest type spec.
	$\mid$	$\mathsf{type}\ t$	abstract type spec.
	$\mid$	$\mathsf{val}\ l : \tau$	value spec.
Programs			
P	$::=$	$(\mathsf{rec}(X : S)s, e)$	

Figure 8. The abstract syntax of the module language

Core types	τ	$::=$	$1 \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid p.t$
Core expr.	e	$::=$	$x \mid () \mid \lambda x : \tau.e \mid e_1\ e_2$
		$\mid$	$(e_1, e_2) \mid \pi_i(e) \mid p.c\ e$
		$\mid$	$\mathsf{case}\ e\ \mathsf{of}\ p.c\ x \Rightarrow e' \mid p.l$

Figure 9. The abstract syntax of the core language

while it accommodates recursion between type definitions, the proposed translation from ML modules [25] does not include recursive modules. Technically, $\mathsf{F}^{\curlyvee}$ ensures syntactically that recursive types are always contractive, which allows $\mathsf{F}^{\curlyvee}$ to use an equational theory for recursive types. While this may make the syntactic theory simpler, this is already a restriction. As a result, if we freely extend the translation to recursive modules, it seems that functors suffer from the same problem as in RMC: an abstract type generated by a functor cannot be directly used in its input, making the construction of generative functor fixpoints impossible. This confirms our intuition that, at least for the semantics, it is better not to impose any restriction in terms of type cycles.

4. Syntax

Figure 8 gives the syntax of our module language, which is based on that of OCaml. We use X, Y, Z as metavariables for recursion variables and module variables, M, N for module names, t for type names, and l for value names.

As explained in Section 3.1, we extend structures with explicitly typed declarations of recursion variables. Every structure is a recursive structure $\mathsf{rec}(X : S)s$; every signature is a recursively dependent signature $\mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}$. Components of structures and signatures may refer to each other (recursively) only via recursion variables. A module path p, by which types, values, and submodules may be projected, is a recursion variable X followed by a series of module names separated by dots (.). Unlike OCaml, we do not support applicative functors [19], so module paths do not include functor application paths $p_1(p_2)$; we support only generative functors as in SML.

A module expression m is a recursive structure, a functor, a functor application, a sealed module, or a path. The forward declaration signature S in a recursive structure $\mathsf{rec}(X : S)s$ should contain all the necessary information to typecheck forward references in the body s. Functors are higher-order, meaning that they may take functors as arguments. Moreover, they behave generatively in the sense that each application of the same functor generates fresh abstract types from types that are declared abstractly in its signature. We ensure this generativity simply by requiring every functor application to be bound to a distinct name before being used and by allowing projections from modules only via module paths [20]. For simplicity, we permit only module paths in functor applications. Signature sealing is opaque as in OCaml and we do not support transparent signature sealing *à la* SML.

A module type S is either a recursively dependent signature or a functor signature.

In a structure $\mathsf{rec}(X : S)\mathsf{struct}\ d_1\ \ldots\ d_n\ \mathsf{end}$ and a signature $\mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}$, the variable X is bound in $d_1\ \ldots\ d_n$ and $D_1\ \ldots\ D_n$, respectively. Free and bound variables are defined in the usual way. We identify module expressions and module types modulo renaming bound variables.

A definition d is a binding of a module, a datatype, a type, or a value; a specification D declares a module, a datatype, a manifest type, an abstract type, or a value. For datatypes, we use an optional equality denoted by $[= \tau']$. $\mathsf{datatype}\ t = \tau' = c\ \mathsf{of}\ \tau$ means that the datatype t is a replication of another datatype τ' whose constructor c expects an argument of type τ.

A program P is a pair of a top-level recursive structure and a core expression to be evaluated. We assume that a program does not contain free variables.

Figure 9 gives the syntax of the core language, which is a simple functional language extended with type and value paths. A type path $p.t$ and a value path $p.l$ refer to the type component t and the value component l in the structure that p refers to, respectively. We use x as a metavariable for term variables and c for datatype constructors.

We introduce the core language mainly for an illustrative purpose such as for proving type soundness and its definition is largely orthogonal to the formulation of our type system.

In the rest of paper, we assume the following two useful conventions: 1) all binding occurrences of bound variables use distinct names; 2) no sequence of definitions or specifications includes duplicate definitions or specifications for the same name.

5. A Type System for Recursive Modules

Our type system solves the double vision problem by introducing path substitutions. It permits transparent as well as opaque type cycles by considering weakly bisimilar types as equivalent. A fragment allowing practical typechecking is presented in Section 7, by additionally rejecting transparent type cycles.

5.1 Typing Rules

For typing rules, we use module contexts Γ, path substitutions Δ, and core contexts Σ. A module context Γ is a finite mapping from recursion variables to their signatures; a set Δ of path substitutions is a mapping from external module paths to internal recursion variables; and a core context Σ is a finite mapping from core variables to core types:

Module contexts	Γ	$::=$	$\cdot \mid \Gamma, X : S$
Path substitutions	Δ	$::=$	$\cdot \mid \Delta, X.M \mapsto Y$
Core contexts	Σ	$::=$	$\cdot \mid \Sigma, x : \tau$

Note that unlike usual substitutions which substitute paths for variables, Δ substitutes variables for paths. Moreover, Δ maintains only the shortest external path for each internal recursion variable. For example, when $X.M.N$ and $Y.N$ are external paths for Z, Δ contains only $Y.N \mapsto Z$ and not $X.M.N \mapsto Z$. In this case, by construction, Δ must contain $X.M \mapsto Y$ and thus we can deduce $X.M.N \mapsto Z$.

Our typing rules use the following judgments:

$\Gamma; \Delta; p \vdash m : S$	Well-typed modules
$\Gamma; \Delta; X \vdash d : D$	Well-typed definitions
$\Gamma; \Delta; p \vdash S_1 \leq S_2$	S_1 is a subtype of S_2
$\Gamma; \Delta; X \vdash D_1 \leq D_2$	D_1 is a sub-specification of D_2
$\Gamma \vdash S$ wf	Well-formed signatures
$\Gamma \vdash D$ wf	Well-formed specifications
$\vdash P : (S, \tau)$	Well-typed programs
$\Gamma; \Delta; \Sigma \vdash e : \tau$	Well-typed expressions
$\Gamma \vdash \tau$ wf	Well-formed types
$\Gamma \vdash p \ni D$	The signature of p contains D
$\Gamma; \Delta \vdash \tau_1 \approx \tau_2$	τ_1 is weakly bisimilar to τ_2

The judgment $\Gamma; \Delta; p \vdash m : S$ includes *context path* p, which is the external path for module m. The judgment $\Gamma; \Delta; X \vdash d : D$ includes as a context path the recursion variable X of the module to which definition d belongs. By exploiting context paths, our type system generates path substitutions to solve the double vision problem when typechecking recursive structures. The subtyping judgments for module types and specifications also include a context path so that our type system solves the double vision problem arising when checking subtyping relations. Subtyping rules are explained in Section 5.4. We use a dummy path ϕ as a placeholder for path contexts. In typing rules, ill-formed paths such as $\phi.M$ never appear. The membership judgment $\Gamma \vdash p \ni D$ means that under context Γ, the signature of module p contains a specification D (see Section 5.2 for details). We use weak bisimilarity $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$ for type equivalence, which exploits path substitutions Δ to identify external names with internal names of abstract types (see Section 5.5).

Figure 10 shows typing rules for the module language. Most of the rules are standard and similar to those in the literature [18, 19, 28, 37]; we highlight only distinguishing features.

To solve the double vision problem, the rule typ-str adds a new path substitution $p \mapsto X$ to the typing context. When typechecking the structure body, the type system exploits the fact that the context path p is an external reference of the recursion variable X. For example, when typechecking the module `Tree` in Figure 6, the rule typ-str adds a substitution `X.Tree` $\mapsto$ `Y` into the context. Our type system generates path substitutions for the shortest external paths only: the rule typ-mdef sets the context path to the shortest external path $X.M$ of module m; if m is a recursive structure whose recursion variable is Y, the rule typ-str then adds a substitution $X.M \mapsto Y$.

The rule typ-str requires that the forward declaration signature S be equivalent to the resultant signature. Here, by equivalence $\Gamma; \Delta; p \vdash S_1 \equiv S_2$, we mean that both $\Gamma; \Delta; p \vdash S_1 \leq S_2$ and $\Gamma; \Delta; p \vdash S_2 \leq S_1$ hold. The reason why we require the equivalence is simply that components that are not specified in S cannot be reached; their definitions are dead code. Recall that we allow components of structures to refer to each other only via recursion variables.

The rule typ-path assigns module type S to module path $q.M$ if the signature of module q contains a module specification module $M : S$. The rule typ-self assigns module type S/q to module path q if q is of type S. Roughly speaking, strengthening S/q [15, 18] makes each abstract type t in S manifestly equal to itself by changing specification type t in S to type $t = q.t$. For further discussion on strengthening, we refer the reader to Section 5.3.

The subsumption rule typ-seal accounts for signature sealing and coerces the type S' of module m into S. Here, we use $[p]$ to mean p if m is a recursive structure in which case the double vision problem may arise; otherwise, it means a dummy path ϕ.

Module expressions $\boxed{\Gamma; \Delta; p \vdash m : S}$

$$\frac{X \in \mathsf{dom}(\Gamma)}{\Gamma; \Delta; p \vdash X : \Gamma(X)}\ \textsf{typ-var} \qquad \frac{\Gamma \vdash q \ni \mathsf{module}\ M : S}{\Gamma; \Delta; p \vdash q.M : S}\ \textsf{typ-path} \qquad \frac{\Gamma; \Delta; p \vdash q : S}{\Gamma; \Delta; p \vdash q : S/q}\ \textsf{typ-self}$$

$$\frac{\Gamma \vdash S \text{ wf} \quad \Gamma, X : S; \Delta, p \mapsto X; X \vdash d_i : D_i \quad (1 \leq i \leq n) \quad \Gamma; \Delta; p \vdash \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end} \equiv S}{\Gamma; \Delta; p \vdash \mathsf{rec}(X : S)\mathsf{struct}\ d_1\ \ldots\ d_n\ \mathsf{end} : \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}}\ \textsf{typ-str}$$

$$\frac{\Gamma \vdash S \text{ wf} \quad \Gamma, X : S; \Delta; \phi \vdash m : S'}{\Gamma; \Delta; p \vdash \mathsf{functor}\ (X : S) \to m : \mathsf{functor}\ (X : S) \to S'}\ \textsf{typ-fun} \qquad \frac{\Gamma \vdash S \text{ wf} \quad \Gamma; \Delta; p \vdash m : S' \quad \Gamma; \Delta; [p] \vdash S' \leq S}{\Gamma; \Delta; p \vdash (m : S) : S}\ \textsf{typ-seal}$$

$$\frac{\Gamma; \Delta; p \vdash q_1 : \mathsf{functor}\ (X : S_1) \to S_2 \quad \Gamma; \Delta; p \vdash q_2 : S_1' \quad \Gamma; \Delta; \phi \vdash S_1' \leq S_1}{\Gamma; \Delta; p \vdash q_1(q_2) : [X \mapsto q_2]S_2}\ \textsf{typ-app}$$

Definitions $\boxed{\Gamma; \Delta; X \vdash d : D}$

$$\frac{\Gamma; \Delta; X.M \vdash m : S}{\Gamma; \Delta; X \vdash \mathsf{module}\ M = m : \mathsf{module}\ M : S}\ \textsf{typ-mdef} \qquad \frac{\Gamma \vdash \tau \text{ wf}}{\Gamma; \Delta; X \vdash \mathsf{type}\ t = \tau : \mathsf{type}\ t = \tau}\ \textsf{typ-type}$$

$$\frac{\Gamma \vdash \tau \text{ wf} \quad [\Gamma \vdash q \ni \mathsf{datatype}\ t = c\ \mathsf{of}\ \tau]}{\Gamma; \Delta; X \vdash \mathsf{datatype}\ t\ [= q.t] = c\ \mathsf{of}\ \tau : \mathsf{datatype}\ t\ [= q.t] = c\ \mathsf{of}\ \tau}\ \textsf{typ-data} \qquad \frac{\Gamma; \Delta; \cdot \vdash e : \tau}{\Gamma; \Delta; X \vdash \mathsf{let}\ l = e : \mathsf{val}\ l : \tau}\ \textsf{typ-val}$$

Module types $\boxed{\Gamma \vdash S \text{ wf}}$

$$\frac{\Gamma, X : \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end} \vdash D_i \text{ wf} \quad (1 \leq i \leq n)}{\Gamma \vdash \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end} \text{ wf}}\ \textsf{wf-sig} \qquad \frac{\Gamma \vdash S_1 \text{ wf} \quad \Gamma, X : S_1 \vdash S_2 \text{ wf}}{\Gamma \vdash \mathsf{functor}\ (X : S_1) \to S_2 \text{ wf}}\ \textsf{wf-fsig}$$

Specifications $\boxed{\Gamma \vdash D \text{ wf}}$

$$\frac{\Gamma \vdash S \text{ wf}}{\Gamma \vdash \mathsf{module}\ M : S \text{ wf}}\ \textsf{wf-mspec} \qquad \frac{\Gamma \vdash \tau \text{ wf} \quad [\Gamma \vdash q \ni \mathsf{datatype}\ t = c\ \mathsf{of}\ \tau]}{\Gamma \vdash \mathsf{datatype}\ t\ [= q.t] = c\ \mathsf{of}\ \tau \text{ wf}}\ \textsf{wf-data}$$

$$\frac{\Gamma \vdash \tau \text{ wf}}{\Gamma \vdash \mathsf{type}\ t = \tau \text{ wf}}\ \textsf{wf-type} \qquad \frac{}{\Gamma \vdash \mathsf{type}\ t \text{ wf}}\ \textsf{wf-abs} \qquad \frac{\Gamma \vdash \tau \text{ wf}}{\Gamma \vdash \mathsf{val}\ l : \tau \text{ wf}}\ \textsf{wf-vspec}$$

Programs $\boxed{\vdash P : (S, \tau)}$

$$\frac{\cdot; \cdot; \phi \vdash \mathsf{rec}(X : S)s : S' \quad X : S; \cdot; \cdot \vdash e : \tau}{\vdash (\mathsf{rec}(X : S)s, e) : (S', \tau)}\ \textsf{typ-prog}$$

Figure 10. Typing rules for the module language

Since we allow cyclic type definitions, the rules typ-type and wf-type do not check whether there is a type cycle involving t. They only check that the type τ is well-formed, i.e. whether every type path used in τ is declared in the typing context.

The rule typ-prog sets the context path to a dummy path ϕ and inspects the top-level module $\mathsf{rec}(X : S)s$. As the recursion variable X is bound in the top-level expression e (to be evaluated), it inspects e under the context $(X : S, \cdot, \cdot)$.

Figure 11 shows typing rules for the core language. As most of the rules are standard, we highlight only distinguishing features; see Appendix A for the omitted rules. The rule c-typ-path assigns type τ to value path $p.l$ if the signature of module p contains a value specification $\mathsf{val}\ l : \tau$. A type path $p.t$ is well-formed if the signature of module p contains a specification for the type t (the rules c-wf-data, c-wf-type, and c-wf-abs).

5.2 Membership

In Figures 10 and 11, we use the membership judgment $\Gamma \vdash p \ni D$ which is inspired by νObj of Odersky et al. [30]. The judgment means that under context Γ, the signature of module p contains a specification D. The rule mem-spec below precisely describes this idea:

$$\frac{\Gamma; \Delta; q \vdash p : \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}}{\Gamma \vdash p \ni [X \mapsto p]D_i}\ \textsf{mem-spec}$$

Since the recursion variable X is local, the rule mem-spec replaces it with the corresponding external module path p. In the premise, any Δ and q may be used.

Core expressions $\boxed{\Gamma; \Delta; \Sigma \vdash e : \tau}$

$$\frac{\Gamma \vdash p \ni \mathsf{val}\ l : \tau}{\Gamma; \Delta; \Sigma \vdash p.l : \tau}\ \textsf{c-typ-path}$$

Core types $\boxed{\Gamma \vdash \tau\ \mathrm{wf}}$

$$\frac{\Gamma \vdash p \ni \mathsf{datatype}\ t\ [= \tau'] = c\ \mathsf{of}\ \tau}{\Gamma \vdash p.t\ \mathrm{wf}}\ \textsf{c-wf-data}$$

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t = \tau}{\Gamma \vdash p.t\ \mathrm{wf}}\ \textsf{c-wf-type} \qquad \frac{\Gamma \vdash p \ni \mathsf{type}\ t}{\Gamma \vdash p.t\ \mathrm{wf}}\ \textsf{c-wf-abs}$$

Figure 11. Typing rules for the core language (excerpt)

5.3 Type Strengthening

To propagate type equalities across module boundaries, we introduce a type strengthening operation [15, 18]. To illustrate, consider the following code (written in OCaml syntax):

```
module type S = sig type t end
module M : S = struct type t = int end
module N = M
module F (X : S) = X
module L = F(M)
```

In the above code, the module `M` declares an abstract type `t` via a sealing. If the type system assigned exactly the same signature `S` to both modules `M` and `N`, the abstract types `M.t` and `N.t` would be considered different. Here, we assume that two abstract types are equal only if their paths are the same. However, since `N` is merely another name for `M`, the type system should identify the two types `M.t` and `N.t`. Strengthening extends the signature of `M` so that the abstract type `t` in the signature becomes manifestly equal to itself (using its external path `M.t`). By assigning the strengthened signature **sig type** `t = M.t` **end** to `N`, the type system can now identify `M.t` with `N.t`. The same technique applies to the module `L`, which is another alias for `M` obtained by applying the identity functor `F`. In order to identify `M.t` and `L.t`, `F` should have type **functor** `(X : S) -> S` **with type** `t = X.t` rather than just **functor** `(X : S) -> S`.

The strengthening operation S/q extends the signature S of module q and makes each abstract type t in S manifestly equal to itself using its external path. It is defined as follows:

$$\begin{aligned}
(\mathsf{rec}(X)\mathsf{sig}\ \overline{D}\ \mathsf{end})/q &\triangleq \mathsf{rec}(X)\mathsf{sig}\ \overline{D}/q\ \mathsf{end} \\
(D_1\ \ldots\ D_n)/q &\triangleq D_1/q\ \ldots\ D_n/q \\
(\mathsf{functor}\ (X : S_1) \to S_2)/q &\triangleq \mathsf{functor}\ (X : S_1) \to S_2 \\
(\mathsf{module}\ M : S)/q &\triangleq \mathsf{module}\ M : S/q.M \\
(\mathsf{datatype}\ t = c\ \mathsf{of}\ \tau)/q &\triangleq \mathsf{datatype}\ t = q.t = c\ \mathsf{of}\ \tau \\
(\mathsf{datatype}\ t = p.t = c\ \mathsf{of}\ \tau)/q &\triangleq \mathsf{datatype}\ t = p.t = c\ \mathsf{of}\ \tau \\
(\mathsf{type}\ t = \tau)/q &\triangleq \mathsf{type}\ t = \tau \\
(\mathsf{type}\ t)/q &\triangleq \mathsf{type}\ t = q.t \\
(\mathsf{val}\ l : \tau)/q &\triangleq \mathsf{val}\ l : \tau
\end{aligned}$$

For notational convenience, we write $\overline{D}$ for a sequence of specifications $D_1\ \ldots\ D_n$. Datatype and manifest type specifications may have only one type equality as in OCaml. Hence, if a datatype is already a replication of another one, we do not strengthen it again. We also do not strengthen manifest type specifications.

The rule typ-self in Figure 10 uses the strengthening operation: if module path q is of module type S, it is also of module type S/q. Using the rule typ-self, our type system correctly propagates type equalities across module boundaries including first-order functor boundaries. To fully propagate type equalities across higher-order functor boundaries, however, we need to extend our system with applicative functors [19] or singleton kinds [39], which is beyond the scope of this paper.

5.4 Subtyping Rules

Figure 12 shows subtyping rules. As the double vision problem may arise in checking subtyping relations, the rules sub-sig and sub-mod update the context path appropriately as in the rules typ-str and typ-mdef in Figure 10. The rules sub-sig and sub-fsig implicitly assume alpha-conversion of bound variables, as indicated by the use of the same recursion variable X. The rule sub-sig for subtyping on signatures permits depth, width, and permutation subtyping as in record subtyping. The rule sub-fsig for subtyping on functor signatures is, as usual, contravariant in argument signatures and covariant in result signatures. The rules for subtyping on specifications are standard except that we use weak bisimilarity $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$ for type equivalence.

5.5 Type Equivalence by Weak Bisimilarity

This section defines our type equivalence relation as the largest weak bisimulation [23] of a labeled transition system on types [28]. The labeled transition system is parameterized by the typing context (Γ, Δ). Alternatively, we could have defined type equivalence more declaratively using mixed induction-coinduction in the style of [29]. Indeed, our account of cyclic types and type abbreviations relies on the ability to mix induction and coinduction, which weak bisimilarity naturally supports by definition.

Figure 13 shows the labeled transition system on types, which consists of two sets of rules: one for labeled transitions and the other for silent transitions. Informally, two types are weakly bisimilar, or equivalent, if their tree-like expansions share the same pattern of labeled transitions. Moreover, given a type τ, all the types reachable from τ by silent transitions are equivalent to τ. Technically, labeled transitions analyze the structure of a given type, while silent transitions allow us to analyze trivially equivalent types such as type abbreviations and datatype replications.

For labeled transitions, we use 0 to denote a stuck state to which no transition can be applied. The rules l-tran-unit, l-tran-abs, and l-tran-data transform the unit type 1, abstract types $p.t$, and datatypes $q.t$ to 0 with the labels 1, $p.t$, and

$$\frac{\Gamma, X : \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end}; \Delta, p \mapsto X; X \vdash D_{\sigma(i)} \leq D'_i \quad (1 \leq i \leq m) \quad \sigma : \{1, \cdots, m\} \mapsto \{1, \cdots, n\}}{\Gamma; \Delta; p \vdash \mathsf{rec}(X)\mathsf{sig}\ D_1\ \ldots\ D_n\ \mathsf{end} \leq \mathsf{rec}(X)\mathsf{sig}\ D'_1\ \ldots\ D'_m\ \mathsf{end}}\ \text{sub-sig}$$

$$\frac{\Gamma; \Delta; \phi \vdash S'_1 \leq S_1 \quad \Gamma, X : S'_1; \Delta; \phi \vdash S_2 \leq S'_2}{\Gamma; \Delta; p \vdash \mathsf{functor}\ (X : S_1) \to S_2 \leq \mathsf{functor}\ (X : S'_1) \to S'_2}\ \text{sub-fsig} \qquad \frac{\Gamma; \Delta; X.M \vdash S \leq S'}{\Gamma; \Delta; X \vdash \mathsf{module}\ M : S \leq \mathsf{module}\ M : S'}\ \text{sub-mod}$$

$$\frac{\Gamma; \Delta \vdash \tau_1 \approx \tau_2 \quad \Gamma; \Delta \vdash \tau'_1 \approx \tau'_2}{\Gamma; \Delta; X \vdash \mathsf{datatype}\ t = \tau'_1 = c\ \mathsf{of}\ \tau_1 \leq \mathsf{datatype}\ t = \tau'_2 = c\ \mathsf{of}\ \tau_2}\ \text{sub-data}$$

$$\frac{\Gamma; \Delta \vdash \tau_1 \approx \tau_2}{\Gamma; \Delta; X \vdash \mathsf{datatype}\ t\ [= \tau'_1] = c\ \mathsf{of}\ \tau_1 \leq \mathsf{datatype}\ t = c\ \mathsf{of}\ \tau_2}\ \text{sub-data}_2 \qquad \frac{\Gamma; \Delta \vdash \tau_1 \approx \tau_2}{\Gamma; \Delta; X \vdash \mathsf{type}\ t = \tau_1 \leq \mathsf{type}\ t = \tau_2}\ \text{sub-type}$$

$$\frac{}{\Gamma; \Delta; X \vdash \mathsf{type}\ t \leq \mathsf{type}\ t}\ \text{sub-abs-abs} \qquad \frac{}{\Gamma; \Delta; X \vdash \mathsf{type}\ t = \tau \leq \mathsf{type}\ t}\ \text{sub-type-abs}$$

$$\frac{}{\Gamma; \Delta; X \vdash \mathsf{datatype}\ t\ [= \tau'] = c\ \mathsf{of}\ \tau \leq \mathsf{type}\ t}\ \text{sub-data-abs} \qquad \frac{\Gamma; \Delta \vdash \tau_1 \approx \tau_2}{\Gamma; \Delta; X \vdash \mathsf{val}\ l : \tau_1 \leq \mathsf{val}\ l : \tau_2}\ \text{sub-val}$$

Figure 12. Subtyping rules

Labeled transition rules

$$\Gamma; \Delta \vdash 1 \overset{1}{\rightharpoonup} 0 \qquad \text{(l-tran-unit)}$$
$$\Gamma; \Delta \vdash \tau_1 \to \tau_2 \overset{\mathrm{ar}_i}{\rightharpoonup} \tau_i \qquad \text{(l-tran-fun)}$$
$$\Gamma; \Delta \vdash \tau_1 * \tau_2 \overset{\mathrm{prd}_i}{\rightharpoonup} \tau_i \qquad \text{(l-tran-prod)}$$

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t \quad p \mapsto q \notin \Delta}{\Gamma; \Delta \vdash p.t \overset{p.t}{\rightharpoonup} 0}\ \text{l-tran-abs}$$

$$\frac{\Gamma \vdash p \ni \mathsf{datatype}\ t = c\ \mathsf{of}\ \tau \quad p \mapsto q \notin \Delta}{\Gamma; \Delta \vdash p.t \overset{p.t}{\rightharpoonup} 0}\ \text{l-tran-data}$$

$$\frac{\Gamma; \Delta \vdash p_1.t_1 \rightharpoonup \cdots \rightharpoonup p_n.t_n \rightharpoonup p_1.t_1}{\Gamma; \Delta \vdash p_1.t_1 \overset{p_i.t_i}{\rightharpoonup} 0}\ \text{l-tran-cycle}$$

Silent transition rules

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t = \tau}{\Gamma; \Delta \vdash p.t \rightharpoonup \tau}\ \text{s-tran-type}$$

$$\frac{\Gamma \vdash p \ni \mathsf{type}\ t \quad p \mapsto q \in \Delta}{\Gamma; \Delta \vdash p.t \rightharpoonup q.t}\ \text{s-tran-abs}$$

$$\frac{\Gamma \vdash p \ni \mathsf{datatype}\ t = c\ \mathsf{of}\ \tau \quad p \mapsto q \in \Delta}{\Gamma; \Delta \vdash p.t \rightharpoonup q.t}\ \text{s-tran-data}$$

$$\frac{\Gamma \vdash p \ni \mathsf{datatype}\ t = q.t = c\ \mathsf{of}\ \tau}{\Gamma; \Delta \vdash p.t \rightharpoonup q.t}\ \text{s-tran-alias}$$

Figure 13. A labeled transition system

$q.t$, respectively. As their labels are distinct from each other, the unit type, abstract types, and datatypes are equivalent to only themselves (and their aliases). The rule l-tran-cycle allows us to transform a path $p_1.t_1$ pertaining to a silent cycle (which otherwise would go on forever without emitting any label) into 0, with the label $p_i.t_i$ of any member of this cycle; as a result, all members of this cycle are seen as equivalent, but they are still distinct from members of other silent cycles. The rule l-tran-fun transforms a function type $\tau_1 \to \tau_2$ to either τ_1 with the label ar_1 or τ_2 with the label ar_2. Similarly, the rule l-tran-prod transforms a product type $\tau_1 * \tau_2$ to either τ_1 with the label prd_1 or τ_2 with the label prd_2.

Silent transition rules cover type abbreviations, path substitutions, and datatype replications. First, the rule s-tran-type unfolds a type abbreviation. Second, the rules s-tran-abs and s-tran-data inspect path substitutions Δ so as to transform an external type path $p.t$ to its internal type path $q.t$. Finally, the rule s-tran-alias transforms a replicated datatype $p.t$ to its original one $q.t$.

For the silent transition rules s-tran-type, s-tran-abs, and s-tran-data, we use a new judgment $p \mapsto q \in \Delta$ which is defined as follows:

- $p \mapsto X \in \Delta, p \mapsto X, \Delta'$
- if $p \mapsto q \in \Delta$, then $p.M \mapsto q.M \in \Delta$ for any M.

As we now have transition rules, we are ready to define weak bisimilarity. Let us start with defining several new notations. We write $\Gamma; \Delta \vdash \tau_1 \rightharpoonup^* \tau_2$ for the transitive reflexive closure of the single-step silent transition. We write $\Gamma; \Delta \vdash \tau_1 \overset{l}{\rightharpoonup}{}^* \tau_2$ to mean $\Gamma; \Delta \vdash \tau_1 \rightharpoonup^* \tau_3, \Gamma; \Delta \vdash \tau_3 \overset{l}{\rightharpoonup} \tau_4$, and $\Gamma; \Delta \vdash \tau_4 \rightharpoonup^* \tau_2$ for some τ_3 and τ_4. For a binary relation $\mathcal{S}$, we write $\tau_1 \mathcal{S} \tau_2$ to mean $(\tau_1, \tau_2) \in \mathcal{S}$ and define its inverse as $\{(\tau_1, \tau_2) \mid (\tau_2, \tau_1) \in \mathcal{S}\}$.

Definition 5.1 (Weak simulation). *A binary relation $\mathcal{S}$ over types is a weak simulation under context (Γ, Δ) if and only if, whenever $\Gamma; \Delta \vdash \tau_1 \mathcal{S} \tau_2$,*

1. *if $\Gamma; \Delta \vdash \tau_1 \rightharpoonup \tau'_1$, then there exists some τ'_2 such that $\Gamma; \Delta \vdash \tau_2 \rightharpoonup^* \tau'_2$ and $\Gamma; \Delta \vdash \tau'_1 \mathcal{S} \tau'_2$;*
2. *if $\Gamma; \Delta \vdash \tau_1 \overset{l}{\rightharpoonup} \tau'_1$, then there exists some τ'_2 such that $\Gamma; \Delta \vdash \tau_2 \overset{l}{\rightharpoonup}{}^* \tau'_2$ and $\Gamma; \Delta \vdash \tau'_1 \mathcal{S} \tau'_2$.*

Definition 5.2 (Weak bisimulation and weak bisimilarity). *A binary relation $\mathcal{S}$ is said to be a weak bisimulation if both $\mathcal{S}$ and its inverse are weak simulations. We say that τ_1 and τ_2 are weakly bisimilar under (Γ, Δ), written $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$, if there exists a weak bisimulation $\mathcal{S}$ such that $\Gamma; \Delta \vdash \tau_1 \mathcal{S} \tau_2$.*

To illustrate how weak bisimulations work, we show how to typecheck the function `plant` in `Tree` in Figure 6. Specifically, we show a weak bisimulation $\mathcal{S}$ such that `X.Tree.t` and `int * X.Forest.f` are weakly bisimilar. When typechecking `plant`, the typing context is as follows:

```
Γ = X : S
    Y : rec(Y)sig
          type t = int * X.Forest.f
          val plant : int -> X.Forest.f
                -> X.Forest.f -> X.Forest.f
        end
Δ =  X.Tree ↦ Y
```

- We let $\mathcal{S}_0$ be $\{$(`X.Tree.t`, `int * X.Forest.f`)$\}$.
- Since $\Gamma; \Delta \vdash$ `X.Tree.t` $\rightharpoonup$ `Y.t` by the rule s-tran-abs, we let $\mathcal{S}_1$ be $\{$(`Y.t`, `int * X.Forest.f`)$\}$.
- Since $\Gamma; \Delta \vdash$ `Y.t` $\rightharpoonup$ `int * X.Forest.f` by the rule s-tran-type, we let $\mathcal{S}_2$ be $\{$(`int * X.Forest.f`, `int * X.Forest.f`)$\}$.

Then, $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_{Id}$ is a desired weak bisimulation where $\mathcal{S}_{Id} = \{(\tau, \tau) \mid \tau \text{ is a type}\}$. Hence, the function `plant` in `Tree` typechecks.

Conversely, `X.t` and `X.u` below are not weakly bisimilar.

```
rec(X)sig type t = X.t type u = X.u end
```

They both can go on forever without generating any label, but they can also respectively emit the labels `X.t` and `X.u` which are distinct.

6. Operational Semantics

This section presents a light-weight small-step operational semantics similar to that of *Traviata* [28]. Specifically, we give the operational semantics for a program (m, e) where the evaluation begins by reducing the given expression e under the top-level recursive structure m.

For core expressions, we use a call-by-value reduction strategy using evaluation contexts. For module expressions, especially for functor applications, we use a call-by-name reduction strategy using module environments, which are mappings from recursion variables to their definitions. We do not statically reject ill-founded recursions and let them diverge. We choose this style of operational semantics in order to be independent of the evaluation strategy of recursive modules and to be able to successfully evaluate our motivating examples, which OCaml can handle as well. Additionally, it makes it easy to establish a type soundness result at the surface language level. However, this semantics is not required by the type system, and we may eventually choose a more usual backpatching operational semantics as in RTG [7] or OCaml.

$$
\begin{array}{llll}
\textit{Values} & v & ::= & () \mid \lambda x:\tau.e \mid (v_1, v_2) \mid p.c\ v \\
\textit{Contexts} & \kappa & ::= & \{\} \mid \kappa\ e \mid v\ \kappa \mid (\kappa, e) \mid (v, \kappa) \\
 & & \mid & \pi_i(\kappa) \mid p.c\ \kappa \mid \textsf{case}\ \kappa\ \textsf{of}\ p.c\ x \Rightarrow e \\
\textit{Environments} & \mathcal{E} & ::= & \cdot \mid \mathcal{E}, X \mapsto m
\end{array}
$$

$$
\begin{array}{rcl}
(\lambda x.\tau : e)\ v & \to_\beta & [x \mapsto v]e \\
\pi_i(v_1, v_2) & \to_\beta & v_i \\
\textsf{case}\ p.c\ v\ \textsf{of}\ q.c\ x \Rightarrow e & \to_\beta & [x \mapsto v]e
\end{array}
$$

$$
\frac{\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'}{\mathcal{E} \vdash r.l \to r'.l \mid \mathcal{E}'}\ \textsf{red-path}
\qquad
\frac{\mathcal{E} \vdash p \ni \textsf{let}\ l = e}{\mathcal{E} \vdash p.l \to e \mid \mathcal{E}}\ \textsf{red-val}
$$

$$
\frac{e \to_\beta e'}{\mathcal{E} \vdash e \to e' \mid \mathcal{E}}\ \textsf{red-beta}
\qquad
\frac{\mathcal{E} \vdash e \to e' \mid \mathcal{E}' \quad \kappa \neq \{\}}{\mathcal{E} \vdash \kappa\{e_1\} \to \kappa\{e'\} \mid \mathcal{E}'}\ \textsf{red-ctx}
$$

Figure 14. Small-step call-by-value operational semantics using evaluation contexts for the core language

Since the evaluation of a functor application may diverge without leading to a structure, we introduce extended module paths which contain functor applications as components.

$$
\begin{array}{llll}
\textit{Module paths} & p & ::= & X \mid p.M \\
\textit{Extended module paths} & r & ::= & r(p) \mid r.M \mid p \\
\textit{Core expressions} & e & ::= & \cdots \mid r.l
\end{array}
$$

The extended module paths only appear during evaluation; they may not be used in a source program. Note that in a functor application path $r(p)$, the argument p is a normal module path. This is because we adopt the call-by-name reduction strategy for module expressions. Core expressions e now contain extended value paths $r.l$.

Figure 14 shows the definition of values v, evaluation contexts κ, module environments $\mathcal{E}$, and reduction rules for the core language. We use the judgment $\mathcal{E} \vdash e \to e' \mid \mathcal{E}'$ which means that under environment $\mathcal{E}$, expression e reduces to e' with extended environment $\mathcal{E}'$. If the reduction of e does not involve the reduction of a functor application, $\mathcal{E}$ and $\mathcal{E}'$ are the same. The rule red-path expands a value path $r.l$ into another one $r'.l$ by a one-step expansion of its module path r. The rule red-val unfolds the definition of a value path $p.l$ if p is a structure containing a value definition let $l = e$. The rule red-beta covers the usual β-reductions, denoted by $e \to_\beta e'$. The rule red-ctx is an inner reduction rule following the call-by-value reduction order.

Figure 15 shows call-by-name reduction rules for the module language. The path normalization judgment $\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'$ means a one-step expansion of module abbreviation r.

The module reduction judgment $\mathcal{E} \vdash r \to m \mid \mathcal{E}'$ means that under environment $\mathcal{E}$, extended module path r reduces

Path normalization $\boxed{\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'}$

$$\frac{\mathcal{E} \vdash r \rightarrow r' \mid \mathcal{E}'}{\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'}\ \textsf{pn-red}$$

$$\frac{\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'}{\mathcal{E} \vdash r.M \rightsquigarrow_n r'.M \mid \mathcal{E}'}\ \textsf{pn-path}$$

$$\frac{\mathcal{E} \vdash r \rightsquigarrow_n r' \mid \mathcal{E}'}{\mathcal{E} \vdash r(p) \rightsquigarrow_n r'(p) \mid \mathcal{E}'}\ \textsf{pn-app}$$

Module reduction $\boxed{\mathcal{E} \vdash r \rightarrow m \mid \mathcal{E}'}$

$$\frac{\begin{array}{c}\mathcal{E} \vdash r \rightarrow (\textsf{functor}\ (X : S) \rightarrow m) \mid \mathcal{E} \quad m \hookrightarrow m' \\ m' = \textsf{rec}(Y : S')s \quad Y \notin \textsf{dom}(\mathcal{E})\end{array}}{\mathcal{E} \vdash r(p) \rightarrow Y \mid \mathcal{E}, Y \mapsto [X \mapsto p]m'}\ \textsf{red-app}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash r \rightarrow (\textsf{functor}\ (X : S) \rightarrow m) \mid \mathcal{E} \\ m \hookrightarrow m' \quad m' \text{ is not a structure}\end{array}}{\mathcal{E} \vdash r(p) \rightarrow [X \mapsto p]m' \mid \mathcal{E}}\ \textsf{red-app}_2$$

$$\frac{\mathcal{E} \vdash p \ni \textsf{module}\ M = m \quad m \hookrightarrow m'}{\mathcal{E} \vdash p.M \rightarrow m' \mid \mathcal{E}}\ \textsf{red-mdef}$$

$$\frac{X \in \textsf{dom}(\mathcal{E})}{\mathcal{E} \vdash X \rightarrow \mathcal{E}(X) \mid \mathcal{E}}\ \textsf{red-var}$$

Membership $\boxed{\mathcal{E} \vdash p \ni d}$

$$\frac{\mathcal{E} \vdash p \rightarrow \textsf{rec}(X : S)\textsf{struct}\ d_1\ \ldots\ d_n\ \textsf{end} \mid \mathcal{E}}{\mathcal{E} \vdash p \ni [X \mapsto p]d_i}\ \textsf{mem-def}$$

Unsealing $\boxed{m \hookrightarrow m'}$

$$\frac{m \text{ is not a signature sealing}}{m \hookrightarrow m}\ \textsf{unseal} \qquad \frac{m \hookrightarrow m'}{(m : S) \hookrightarrow m'}\ \textsf{unseal}_2$$

Figure 15. Call-by-name reduction rules for the module language

to its definition m with extended environment $\mathcal{E}'$. The rule red-app chooses an alpha-equivalent structure of the functor body m' without sealings, if its recursion variable Y is already used in the domain of the environment $\mathcal{E}$. Hence, the functor application path $r(p)$ reduces to a new recursion variable Y while the environment is extended with a mapping from Y to its definition. The rule red-mdef is defined in terms of module paths instead of extended module paths because it uses as its premise a membership judgment, which is also defined in terms of module paths.

The membership judgment $\mathcal{E} \vdash p \ni d$ means that under environment $\mathcal{E}$, module p contains a definition d. It is defined in terms of module paths because functor application paths are always first replaced with recursion variables by the rule red-app before being applied to the membership judgment.

$$\frac{}{\Gamma; \Delta; \Omega \vdash 1 \searrow 1}\ \textsf{n-unit} \qquad \frac{\Gamma \vdash p \ni \textsf{type}\ t \quad p \mapsto q \notin \Delta}{\Gamma; \Delta; \Omega \vdash p.t \searrow p.t}\ \textsf{n-abs}$$

$$\frac{\Gamma; \Delta; \Omega \vdash \tau_1 \searrow \tau_1' \quad \Gamma; \Delta; \Omega \vdash \tau_2 \searrow \tau_2'}{\Gamma; \Delta; \Omega \vdash \tau_1 \rightarrow \tau_2 \searrow \tau_1' \rightarrow \tau_2'}\ \textsf{n-fun}$$

$$\frac{\Gamma; \Delta; \Omega \vdash \tau_1 \searrow \tau_1' \quad \Gamma; \Delta; \Omega \vdash \tau_2 \searrow \tau_2'}{\Gamma; \Delta; \Omega \vdash \tau_1 * \tau_2 \searrow \tau_1' * \tau_2'}\ \textsf{n-prod}$$

$$\frac{\Gamma \vdash p \ni \textsf{datatype}\ t = c\ \textsf{of}\ \tau \quad p \mapsto q \notin \Delta}{\Gamma; \Delta; \Omega \vdash p.t \searrow p.t}\ \textsf{n-data}$$

$$\frac{\Gamma \vdash p \ni \textsf{type}\ t = \tau \quad \Gamma; \Delta; \Omega \uplus p.t \vdash \tau \searrow \tau'}{\Gamma; \Delta; \Omega \vdash p.t \searrow \tau'}\ \textsf{n-type}$$

$$\frac{\Gamma \vdash p \ni \textsf{type}\ t \quad p \mapsto q \in \Delta \quad \Gamma; \Delta; \Omega \uplus p.t \vdash q.t \searrow \tau}{\Gamma; \Delta; \Omega \vdash p.t \searrow \tau}\ \textsf{n-abs}_2$$

$$\frac{\begin{array}{c}\Gamma \vdash p \ni \textsf{datatype}\ t = c\ \textsf{of}\ \tau \\ p \mapsto q \in \Delta \quad \Gamma; \Delta; \Omega \uplus p.t \vdash q.t \searrow \tau'\end{array}}{\Gamma; \Delta; \Omega \vdash p.t \searrow \tau'}\ \textsf{n-data}_2$$

$$\frac{\begin{array}{c}\Gamma \vdash p \ni \textsf{datatype}\ t = q.t = c\ \textsf{of}\ \tau \\ \Gamma; \Delta; \Omega \uplus p.t \vdash q.t \searrow \tau'\end{array}}{\Gamma; \Delta; \Omega \vdash p.t \searrow \tau'}\ \textsf{n-self}$$

Figure 16. Type normalization

As in the rule mem-spec in Section 5.2, the rule mem-def replaces the local recursion variable X with its valid external path p.

The unsealing judgment $m \hookrightarrow m'$ strips away sealed signatures in the module m.

Theorem 6.1 (Soundness). *Let a program (m, e) be well-typed. Then, the evaluation of e either returns a value or else gives rise to an infinite reduction sequence.*

We prove type soundness by the usual progress and type preservation properties. Progress is easy, but type preservation is subtle: it does not hold when a value is passed across opaque sealings. We address this by making all path substitutions available globally; hence, we can always obtain the most precise type of a given value using path substitutions. This is indeed equivalent to removing opaque sealings, thus breaking type abstraction. The key lemma in the soundness proof states that the program remains well-typed after making all path substitutions available globally, or equivalently removing sealings. We prove the lemma by finding an appropriate weak bisimulation in the absence of sealings, which equates all the types that are weakly bisimilar in the presence of sealings.

7. An Algorithmic Type System

This section presents an algorithmic type system which is obtained by rejecting transparent type cycles. To reject such cycles, we first define type normalization which fails if and only if a type contains a dangling reference or a transparent type cycle. Then, we extend with type normalization the rules for checking the well-formedness of types in Figure 10: now a type is well-formed if and only if its normalization succeeds. Finally, instead of weak bisimilarity, we employ an algorithmic type equivalence relation based on type normalization, which makes our type system clearly decidable.

Figure 16 defines a type normalization algorithm. The judgment $\Gamma; \Delta; \Omega \vdash \tau_1 \searrow \tau_2$ means that the algorithm normalizes type τ_1 into type τ_2 with respect to Γ and Δ. We use a metavariable Ω for a set of type paths $p.t$. The notation $\Omega \uplus p.t$ means $\Omega \cup \{p.t\}$ whenever $p.t \notin \Omega$. Normalization fails either if it tries to unfold a type path $p.t$ which is already a member of Ω or if the membership judgment for $p.t$ fails. The former indicates that the definition of $p.t$ constitutes a transparent type cycle, whereas the latter happens when $p.t$ is a dangling reference.

To reject transparent type cycles and thereby the program containing such cycles altogether, we replace $\Gamma \vdash S$ wf and $\Gamma \vdash D$ wf with new well-formedness judgments $\Gamma; \Delta \vdash S$ wf and $\Gamma; \Delta \vdash D$ wf in all typing rules in Figure 10. Moreover, in the rules typ-type and wf-type, we check whether a given type τ is normalizable, using the judgment $\Gamma; \Delta; \emptyset \vdash \tau \searrow \tau'$:

$$\frac{\Gamma; \Delta; \emptyset \vdash \tau \searrow \tau'}{\Gamma; \Delta; p \vdash \mathsf{type}\ t = \tau : \mathsf{type}\ t = \tau}\ \textsf{typ-type}$$

$$\frac{\Gamma; \Delta; \emptyset \vdash \tau \searrow \tau'}{\Gamma \vdash \mathsf{type}\ t = \tau\ \mathrm{wf}}\ \textsf{wf-type}$$

As for type equivalence, given two arbitrary types, we first normalize them and then check equivalence on the normalized types by syntactic equality. We employ a new judgment $\Gamma; \Delta \vdash \tau_1 \equiv \tau_2$ for the algorithmic type equivalence relation, which means that two types τ_1 and τ_2 are equivalent under Γ and Δ, and is defined as follows:

$$\frac{\Gamma; \Delta; \emptyset \vdash \tau_1 \searrow \tau_1' \quad \Gamma; \Delta; \emptyset \vdash \tau_2 \searrow \tau_2' \quad \tau_1' = \tau_2'}{\Gamma; \Delta \vdash \tau_1 \equiv \tau_2}\ \textsf{eq-types}$$

The algorithmic type system uses type equivalence $\Gamma; \Delta \vdash \tau_1 \equiv \tau_2$ instead of weak bisimilarity $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$.

Thanks to the rules $\textsf{n-abs}_2$ and $\textsf{n-data}_2$ in Figure 16, which exploit path substitutions, we still solve the double vision problem while the type system remains decidable.

Lemmas 7.1 and 7.2 below imply that our algorithmic type equivalence is sound and complete with respect to weak bisimilarity if the program does not contain transparent type cycles.

Lemma 7.1. $\Gamma; \Delta \vdash \tau_1 \equiv \tau_2$ *if and only if* $\Gamma; \Delta \vdash \tau_1 \approx \tau_2$, $\exists \tau_1'$, $\Gamma; \Delta; \emptyset \vdash \tau_1 \searrow \tau_1'$ *and* $\exists \tau_2'$, $\Gamma; \Delta; \emptyset \vdash \tau_2 \searrow \tau_2'$.

Lemma 7.2. *For any* τ, Γ, Δ, *and* Ω, *it is decidable whether there exists a type* τ' *such that* $\Gamma; \Delta; \Omega \vdash \tau \searrow \tau'$.

As a corollary of Lemmas 7.1 and 7.2 and Theorem 6.1, the algorithmic type system is sound and decidable.

Theorem 7.3. *The algorithmic type system is sound and decidable.*

Without type parameters, our language of types can express only regular types. As practical algorithms are available both for equality and unification[4] on regular types, we could have built an algorithmic type system based on the full weak bisimilarity, without excluding transparent type cycles. However, extending our language with type parameters introduces non-regular equi-recursive types. Strictly speaking, equality of equi-recursive types with type parameters is still decidable [6]. Solomon [38] has shown it to be equivalent to the equivalence problem for deterministic pushdown automata (DPDA), which has been shown decidable by Sénizergues [36]. There is, unfortunately, no known practical algorithm for DPDA-equivalence, and it is not known whether there exists any algorithm for unification either. Therefore, in the case of extending our language with type parameters, the normalization based type system becomes crucial in providing concrete algorithms for equality, unification, and typechecking.

Technical considerations are not the only reason that one would want to reject transparent type cycles. If we were to allow them in the surface language, we would not only have to accept them in definitions, but also infer them in core language expressions, so as to keep the principality of type inference. Experience with early versions of OCaml, which put no restriction on regular types, has shown that they often lead the typechecker to accept wrong programs, whose types may be unsatisfiable. For instance, a slightly erroneous version of `List.append` would lead to inferring the type $\mu\alpha.\,\alpha$ `list`, which allows only the value *nil*. To avoid this problem, subsequent versions of OCaml restrict regular types to object and polymorphic variant types. Disallowing transparent type cycles during type inference is sufficient to avoid this issue.

8. Discussion

8.1 Separate Compilation

Our type system does not support separate compilation of mutually recursive modules in its current form. To illustrate, consider Figure 17. In order to compile the two recursive modules `Tree` and `Forest` separately, we turn each module into a functor that is parameterized over the recursive variable `X` and later tie the recursive knot. Unfortunately, the functor `TreeFn` suffers from the double vision problem. The reason is that there is no way in the body of `TreeFn` to connect `X.Tree.t` with `V.t`, which is the implementation of `X.Tree.t` that the body of `TreeFn` provides.

To solve the double vision problem while supporting separate compilation, we require the programmer to explicitly specify that the defining functor body be an implementation of (a submodule of) the functor parameter, for example, using the notation $\boxed{\textbf{where}\ \texttt{X.Tree} \mapsto \texttt{V}}$. Whenever it

[4] Unification is required for type inference in the core language.

```
module type ST(type u) = rec(Y)sig
  type t
  val plant : int -> u -> u -> u
end
module type SF(type u) = rec(Z)sig
  type f
  val plant : u -> Z.f -> Z.f
end
module type S = rec(X)sig
  module Tree : ST(X.Forest.f)
  module Forest  : SF(X.Tree.t)
end

module TreeFn (X : S) =
rec(V : ST(X.Forest.f) with
  type t = int * X.Forest.f) where X.Tree ↦ V
struct
  type t = int * X.Forest.f
  let plant (i : int) (x : X.Forest.f)
    (y : X.Forest.f) = X.Forest.plant (i, x) y
  (* error without the equation in the box *)
  (* as V.t = int * X.Forest.f ≠ X.Tree.t *)
end

module ForestFn (X : S) =
rec(W : SF(X.Tree.t) with
  type f = Nil | Cons of X.Tree.t * W.f)
struct
  type f = Nil | Cons of X.Tree.t * W.f
  let plant (x : X.Tree.t) (y : W.f)
    = W.Cons (x, y)
end

module TreeForest = rec(X : S)struct
  module Tree = TreeFn(X)
  module Forest = ForestFn(X)
end
```

Figure 17. Separate compilation

encounters such an equation, the type system simply adds a path substitution X.Tree ↦ V into the typing context. Then, we can conclude that X.Tree.t and V.t are weakly bisimilar (and also equivalent by the algorithmic type equivalence relation), and thus solve the double vision problem while supporting separate compilation. When declaring and applying the functor TreeFn, the type system additionally checks if the signature of the functor body V is a subtype of that of the argument X.Tree.

This mechanism can be further simplified if we consider a file based approach, similar to OCaml's "packing" mechanism [3]. The OCaml compiler provides two options: "-for-pack *Name*" tells it that a module should be compiled as though it were a substructure of a module *Name*; -pack then combines several separately compiled modules into a hierarchical structure, whose name must match the one declared by -for-pack. An explicit sealing signature can be provided for this structure. To extend this approach to recursive modules, one just has to provide in advance the signature for *Name*, allowing it to be used recursively in all modules compiled with "-for-pack *Name*". The already available sealing mechanism is sufficient to ensure the validity of the provided signature. Note that this is essentially the same functionality as in the functor based scheme; we are just generating automatically the **where** clauses according to the name of the compiled module and its -for-pack information.

To our knowledge, no previous work supports separate compilation of recursive modules using functors while solving the double vision problem. Whereas others rely on mixin-style recursive linking to support separate compilation [10, 12, 32], which deviates much from ML modules, we only slightly extend the notation of functors. We remark, however, that our approach may be considered a restricted form of mixin-style recursive linking.

8.2 Type Reconstruction

The type system requires that each recursion variable be annotated with the principal signature and the argument of each function with a core type. We can easily relax this restriction by requiring annotations only for components used in forward references as in [34]. Furthermore, there are several ways to reduce redundancy in annotations. For example, for nested recursive modules, one needs to specify forward declaration signatures only for leaf-level modules, which are structures containing no module definitions. Starting with those explicitly annotated signatures, a type reconstruction algorithm may build up signatures for enclosing modules in the bottom up fashion. For function definitions that do not use forward references, we may use the usual type inference algorithm to infer their types.

9. Related Work

9.1 Recursive Modules

Crary et al. [6] were the first to propose a type-theoretic foundation of recursive modules. They introduce recursively dependent signatures and recursive structures, and interpret them in the context of a phase distinction formalism [16]. To avoid the double vision problem, their calculus requires forward declaration signatures to be transparent. For example, this means in Figure 5 that the specifications of Tree.t and Forest.f should reveal their definitions, thus preventing data abstraction between Tree and Forest. To support cyclic type definitions, they use equi-recursive type constructors [4] of higher kind, for which there is no known practical algorithm for checking equivalence.

Russo [2, 34] proposes a recursive module extension to Standard ML [24]. In his system, any abstract type components must be implemented as being manifestly equal to themselves. For example, in Figure 5, one must write **type** t = Tree.t in Tree and **type** f = Forest.f in Forest.

Hence, forward declaration signatures may effectively include only manifest type specifications and datatype specifications as in [6].

Although OCaml [3, 21] provides a powerful and flexible support for recursive modules, it does not have a formal specification for typechecking recursive modules nor its soundness proof. Furthermore, it does not fully solve the double vision problem. To avoid this problem, OCaml requires that abstract type specifications in forward declaration signatures be defined internally by datatype definitions. Specifically, when typechecking the body of a recursive module, type abbreviations cannot be strengthened to be equal to their external paths, while datatype definitions can be. As for cyclic type definitions, our approach of rejecting only transparent type cycles is inspired by OCaml. We, however, end up observing more transparent type cycles due to our more robust approach to the double vision problem. Moreover, OCaml's support of applicative functors makes it more difficult to accurately detect transparent type cycles [27].

Nakata and Garrigue [28] propose a recursive module calculus called *Traviata*, which is based on OCaml. While the use of weak bisimilarity is inspired by *Traviata*, there are several major differences. First, *Traviata* supports only first-order applicative functors [19], while we support higher-order generative functors. This restriction to first-order functors is required to render type equivalence decidable in the presence of applicative functors. Indeed, path normalization is generally undecidable with applicative functors [14]. Second, in *Traviata*, the programmer has to manually specify type coercion from internal type paths to external ones to solve the double vision problem, while our type system solves this problem in a simple type theoretic way using path substitutions. Finally, in *Traviata*, the programmer need not specify forward declaration signatures; the type system can infer them. In contrast, we require the programmer to annotate such signatures as in [3, 6, 8, 34].

Dreyer [7] proposes RTG which gives a logical account of recursive type generativity. He later extends its ideas and techniques in the context of recursive modules and proposes RMC [8]. Recently, Montagu and Rémy [26] propose F^{Υ}, which is a variant of System F, by decomposing the introduction and elimination of existential types into more atomic constructs. Although F^{Υ} is a core calculus, it may be considered a simple logical foundation of recursive modules. In contrast to RTG and F^{Υ}, we do not investigate a logical interpretation of recursive modules, but focus on how to address their typing issues from a practical point of view. Our type system typechecks more recursive modules that are useful in practice. For detailed comparisons with RMC and F^{Υ}, we refer the reader to Section 3.

9.2 Units and Mixin Modules

Instead of extending the ML module system with recursive modules, several authors [10–12, 30, 32] have investigated mixin-style recursive composition [5] in conjunction with ML-style abstraction mechanisms and hierarchical composition. Compared with recursive module extensions to ML, the main benefit of using mixin-style recursive linking is that it is more suitable for separate compilation of mutually recursive definitions.

Flatt and Felleisen [12] propose units as a recursive linking mechanism for Scheme. Units may be considered a generalization of functors, where imports may refer to exports. Owens and Flatt [32] later extend units with ML-style nested modules to provide hierarchical composition. They also provide encodings of ML-style structures and first-order functors, but without module abbreviations, in their unit language. In their language, an abstract type cannot be referred to in two ways in a given scope, and thus only a restricted form of the double vision problem arises when linking units, which can be easily avoided. Their source language does not include recursive type or value definitions, but it is later extended with the rec construct for creating recursive definitions to define the operational semantics and prove type soundness. The rec construct may introduce transparent type cycles and they provide an axiomatic system for defining a type equivalence on these cyclic types. In contrast, we account for transparent type cycles and type abbreviations by means of weak bisimilarity, or mixed induction-coinduction, which has a semantic foundation.

Recently, Dreyer and Rossberg [10] propose MixML as a unifying framework for ML modules and mixin modules. MixML is able to express most interesting features of the ML module system including recursive modules via relatively simple encodings. Moreover, by mixin composition, it also supports separate compilation of mutually recursive definitions. Since their mixin composition requires bidirectional ML-style signature matching in the presence of data abstraction and recursion, they need to solve a bidirectional version of the double vision problem. For proving type soundness, they elaborate MixML to an internal language, showing that well-typed programs in MixML are translated into well-typed programs in the internal language and the internal language is type-sound. Again, they have to deal with transparent type cycles in their soundness proof. Since the complete proof, including the definition of type equivalence in the presence of type cycles, is not published yet at the moment of writing this paper, further comparison with their approach has to be deferred to future work.

While integrating ML modules with mixin modules is a promising direction for further investigation, it requires rather drastic changes to the current implementation of the ML module system such as the OCaml implementation. In contrast, our system may be regarded as a lightweight extension of the current implementation of recursive modules in OCaml. In particular, we generalize OCaml's approach to the double vision problem and cyclic type definitions, and also prove type soundness, which is one of our main contributions.

10. Conclusion

This paper proposes a syntactic type system for recursive modules which both solves the double vision problem and accounts for cyclic type definitions. To solve the double vision problem, we introduce path substitutions to locally maintain consistency between external and internal views of recursive modules. To account for cyclic type definitions, we define a type equivalence relation by weak bisimilarity. Our type system typechecks flexible uses of recursive modules such as functor fixpoints, whose uses are restricted in previous work. We show that the type system is sound with respect to a call-by-value operational semantics. As weak bisimulations are hard to handle in practice, we also propose an algorithmic type equivalence relation that is compatible with our solution to the double vision problem. The algorithmic type system still typechecks flexible uses of functor fixpoints. Future work will include extending our system with first-class modules as in [33] and applicative functors [19], both of which are already available in OCaml.

Acknowledgments

We thank Andreas Rossberg, Didier Rémy, and the anonymous reviewers for their helpful comments. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2011-0000977), Mid-career Researcher Program through NRF funded by the MEST (2010-0022061), and National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and Experts Education. Keiko Nakata was supported by the EU FP7 project 231620 Highly Adaptable and Trustworthy Software using Formal Models (HATS) and the European Regional Development Fund (ERDF) through the Estonian Centre of Excellence in Computer Science (EXCS).

References

[1] Amthing. `https://forge.ocamlcore.org/projects/amthing/`.

[2] Moscow ML. `http://www.itu.dk/~sestoft/mosml.html`.

[3] OCaml. `http://caml.inria.fr/ocaml/`.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[5] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *ICCL '92: Proceedings of the 1992 International Conference on Computer Languages*, pages 282–290. IEEE Computer Society, 1992.

[6] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[7] D. Dreyer. Recursive type generativity. *Journal of Functional Programming*, 17(4-5):433–471, 2007.

[8] D. Dreyer. A type system for recursive modules. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 289–302, 2007.

[9] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2005.

[10] D. Dreyer and A. Rossberg. Mixin' up the ML module system. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 307–320, 2008.

[11] D. Duggan and C. Sourelis. Mixin modules. In *ICFP '96: Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, 1996.

[12] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[13] J. Garrigue. Private rows: Abstracting the unnamed. In *APLAS '06: Proceedings of the 4th Asian Symposium on Programming Languages and Systems*, pages 44–60. Springer-Verlag, 2006.

[14] J. Garrigue and K. Nakata. Path resolution for recursive nested modules. Available at `http://www.math.nagoya-u.ac.jp/~garrigue/papers/`, 2010.

[15] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, 1994.

[16] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[17] R. Harper and B. C. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8. MIT press, 2005.

[18] X. Leroy. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[19] X. Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, 1995.

[20] X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(05):667–698, 1996.

[21] X. Leroy. A proposal for recursive modules in Objective Caml, 2003. Available at `http://caml.inria.fr/about/papers.en.html`.

[22] D. MacQueen. Modules for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 198–207, 1984.

[23] R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge University Press, 1999.

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

[25] B. Montagu. *Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types*. PhD thesis, École Polytechnique, Palaiseau, France, December 2010.

[26] B. Montagu and D. Rémy. Modeling abstract types in modules with open existential types. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 354–365, 2009.

[27] K. Nakata. Bug report, nr. 0003674. `http://caml.inria.fr/mantis/view.php?id=3674`, 2005.

[28] K. Nakata and J. Garrigue. Recursive modules for programming. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 74–86, 2006.

[29] K. Nakata and T. Uustalu. Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction. In *SOS '10: Seventh Workshop on Structural Operational Semantics*, pages 57–75, 2010.

[30] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 201–224. Springer-Verlag, 2003.

[31] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[32] S. Owens and M. Flatt. From structures and functors to modules and units. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 87–98, 2006.

[33] C. V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.

[34] C. V. Russo. Recursive structures for Standard ML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 2001.

[35] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 248–274. Springer-Verlag, 2003.

[36] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages, and Programming*, pages 671–681. Springer-Verlag, 1997.

[37] Z. Shao. Transparent modules with fully syntatic signatures. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming*, pages 220–232, 1999.

[38] M. Solomon. Type definitions with type parameters (extended abstract). In *POPL '78: Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 31–38, 1978. Full version available as technical report DAIMI PB-54.

[39] C. A. Stone and R. Harper. Deciding type equivalence in a language with singleton kinds. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–227, 2000.

A. Typing Rules for the Core Language

Core expressions $\boxed{\Gamma; \Delta; \Sigma \vdash e : \tau}$

$$\frac{x : \tau \in \Sigma}{\Gamma; \Delta; \Sigma \vdash x : \tau}\ \textsf{c-typ-var} \qquad \frac{}{\Gamma; \Delta; \Sigma \vdash () : 1}\ \textsf{c-typ-unit}$$

$$\frac{\Gamma \vdash \tau \text{ wf} \quad \Gamma; \Delta; \Sigma, x : \tau \vdash e : \tau'}{\Gamma; \Delta; \Sigma \vdash \lambda x : \tau.e : \tau \to \tau'}\ \textsf{c-typ-lam}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \to \tau \\ \Gamma; \Delta; \Sigma \vdash e_2 : \tau_2 \quad \Gamma; \Delta \vdash \tau_1 \approx \tau_2\end{array}}{\Gamma; \Delta; \Sigma \vdash e_1\ e_2 : \tau}\ \textsf{c-typ-app}$$

$$\frac{\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \quad \Gamma; \Delta; \Sigma \vdash e_2 : \tau_2}{\Gamma; \Delta; \Sigma \vdash (e_1, e_2) : \tau_1 * \tau_2}\ \textsf{c-typ-pair}$$

$$\frac{\Gamma; \Delta; \Sigma \vdash e : \tau_1 * \tau_2}{\Gamma; \Delta; \Sigma \vdash \pi_i(e) : \tau_i}\ \textsf{c-typ-proj}$$

$$\frac{\begin{array}{c}\Gamma \vdash q \ni \textsf{datatype}\ t\ [= \tau''] = c\ \textsf{of}\ \tau \\ \Gamma; \Delta; \Sigma \vdash e : \tau' \quad \Gamma; \Delta \vdash \tau \approx \tau'\end{array}}{\Gamma; \Delta; \Sigma \vdash q.c\ e : q.t}\ \textsf{c-typ-con}$$

$$\frac{\begin{array}{c}\Gamma; \Delta; \Sigma \vdash e_1 : \tau_1 \\ \Gamma \vdash q \ni \textsf{datatype}\ t\ [= \tau'] = c\ \textsf{of}\ \tau \\ \Gamma; \Delta \vdash \tau_1 \approx q.t \quad \Gamma; \Delta; \Sigma, x : \tau \vdash e_2 : \tau_2\end{array}}{\Gamma; \Delta; \Sigma \vdash \textsf{case}\ e_1\ \textsf{of}\ q.c\ x \Rightarrow e_2 : \tau_2}\ \textsf{c-typ-case}$$

$$\frac{\Gamma \vdash p \ni \textsf{val}\ l : \tau}{\Gamma; \Delta; \Sigma \vdash p.l : \tau}\ \textsf{c-typ-path}$$

Core types $\boxed{\Gamma \vdash \tau \text{ wf}}$

$$\frac{}{\Gamma \vdash 1 \text{ wf}}\ \textsf{c-wf-unit} \qquad \frac{\Gamma \vdash \tau_1 \text{ wf} \quad \Gamma \vdash \tau_2 \text{ wf}}{\Gamma \vdash \tau_1 \to \tau_2 \text{ wf}}\ \textsf{c-wf-fun}$$

$$\frac{\Gamma \vdash \tau_1 \text{ wf} \quad \Gamma \vdash \tau_2 \text{ wf}}{\Gamma \vdash \tau_1 * \tau_2 \text{ wf}}\ \textsf{c-wf-prod}$$

$$\frac{\Gamma \vdash p \ni \textsf{datatype}\ t\ [= \tau'] = c\ \textsf{of}\ \tau}{\Gamma \vdash p.t \text{ wf}}\ \textsf{c-wf-data}$$

$$\frac{\Gamma \vdash p \ni \textsf{type}\ t = \tau}{\Gamma \vdash p.t \text{ wf}}\ \textsf{c-wf-type} \qquad \frac{\Gamma \vdash p \ni \textsf{type}\ t}{\Gamma \vdash p.t \text{ wf}}\ \textsf{c-wf-abs}$$

Figure 18. Typing rules for the core language