A module system with applicative functors and recursive path references

Keiko Nakata

Abstract

When developing a large software program, it is useful to decompose the program into smaller parts and to reuse them in different contexts. Many modern programming languages provide some forms of module systems to facilitate such factoring of programs.

The ML module system is well-known for its flexibility in program structuring. A programmer can factor programs into hierarchy using nested *structures* and can define *functors*, which are functions over modules, to reuse program codes. Moreover, *signatures*, which represent types of modules, allow the programmer to control abstraction of modules. In spite of this flexibility, modules cannot be defined recursively in ML, since dependencies between modules must accord with the order of definitions. A complex program may be naturally decomposed into recursive modules. Yet, this constraint of ML will force the programmer to consolidate conceptually separate components into a single module, intruding on modular programming.

Introducing recursive modules is a natural way out of this predicament. Existing proposals, however, vary in expressiveness and verbosity. In this paper, we propose a type system for recursive modules, which can infer their signatures. Opaque signatures can also be given explicitly, to provide type abstraction either inside or outside the recursion. The type system is decidable, and is sound for a call-by-value semantics.

Acknowledgment

Jacques Garrigue strongly supported me throughout my work. We have had dense and fruitful discussions regularly and he gave me many many many useful suggestions and never discouraged my premature ideas. He always listened to me carefully and gave me appropriate advice and references. Our discussions were not limited to about recursive modules but about various topics on programming languages. All these discussions were exciting and useful to my work. I cannot thank him enough for his support.

Masahito Hasegawa supported me throughout my work. He supported me both technically and spiritually during my study at RIMS. Although our work is not closely related, he understood my work and gave me useful suggestions. In particular, his advice from a more technically fundamental point of view often made my thought clearer. I thank him very much.

Susumu Nishimura gave me useful suggestions on my work. He also carefully read my draft papers and gave me comments, which greatly helped me improve the drafts. I thank him very much.

Declaration

I declare that this work is entirely written by myself. The result presented in Part I is motivated by my previous work [48] and extends it from a technical point of view. The result of this whole thesis is condensed in [47], where proof and details are omitted.

Contents

List of Figures

Part I
Introduction

The ML module system

When developing a large software program, it is useful to decompose the program into smaller parts and to reuse them in different contexts. Module systems play an important role in facilitating such factoring of programs [29, 5]. Many modern programming languages provide some forms of module systems. Examples are class systems in object-oriented languages, the package mechanism in Java and the ML module system.

The family of ML programming languages, which includes Standard ML [46, 45] (hereafter, SML) and Objective Caml [42] (hereafter, O'Caml), provides a powerful mechanism for modular development of large programs, namely the ML module system [44, 40, 52]. Three important features of the module system are nested structures, functors and signature ascription. Here we introduce them by gradually extending a small example program in an attempt to build a versatile set module.

Nested structures Modules can be nested. That is, they can contain definitions of modules, in addition to definitions of types and core expressions. Hence they allow hierarchical decomposition of programs.

In Figure 1, we define a `Set` module representing sets of integers. We pack into the `Element` sub-module type and value components that are relevant to elements of those integer sets. Observe that module hierarchy also allows namespace management. The `Element` module contains a type component named `t`, which represents the type of elements held in sets; the `Set` module contains a type component of the same name, which represents the type of sets. A programmer can distinguish between these two components of the same name by referring to the former as `Set.Element.t` and the latter as `Set.t`. The ML scoping rule for backward references allows us to use `Element.t` to refer to `Set.Element.t` in the definition of `Set.t` [1]

[1] Precisely, we cannot use `Set.Element.t` in the definition of `Set.t` or `Set.sum`. This amounts to forward references, that the current ML module system does not allow.

```
module Set = struct
  module Element = struct
    type t = int
    val unit = 0
    val add x y = x + y
  end
  type t = Element.t list
  val empty = []
  val sum l = case l with
       [] ⇒ Element.unit
     | hd :: tl ⇒ Element.add hd (sum tl)
end
```

Figure 1: A `Set` module for integer sets

```
module FSet =
functor(X : sig type t val unit : t val add : t → t → t end) →
 struct
  module Element = X
  type t = Element.t list
  val empty = []
  val sum l = case l with
       [] ⇒ Element.unit
     | hd :: tl ⇒ Element.add hd (sum tl)
 end
```

Figure 2: A `FSet` functor for a parameterized set module

```
module AFSet =
functor(X : sig type t val unit : t val add : t → t → t end) →
 (struct
   module Element = X
   type t = Element.t list
   val empty = []
   val sum l = case l with
        [] ⇒ Element.unit
     | hd :: tl ⇒ Element.add hd (sum tl)
 end : sig type t val empty : t val sum : t → X.t)

module ASet = AFSet(Set.Element)
```

Figure 3: A `AFSet` functor, whose body is ascribed by a signature

Functors Functors are functions over modules, where their formal parameters are explicitly annotated with signatures. Signatures are types of modules. The body of a functor can refer to a component of the parameter as long as the parameter's signature says that it has this component. Functor application instantiates modules, where argument modules must implement all the components that the signature of the parameter requires and determine the behavior of the resulting modules. Functors are useful to ease code reuse.

In Figure 2, we define a functor `FSet`, a functorized version of the above `Set` module. When applied, `FSet` instantiates a module representing sets whose element type is determined by the argument module. Indeed, we can instantiate an equivalent of the above `Set` module by applying `FSet` to `Set.Element`, i.e., `FSet(Set.Element)`.

Signature ascription Modules can be ascribed by signatures. A signature does not have to mention all the components that the ascribed module contains but may only specify some of them translucently [27, 37, 43]. Thus a programmer can flexibly control accessibility of module components. Signatures improve modularity of programs.

In Figure 3, we define a functor `AFSet` by ascribing the above `FSet` functor with a signature. The signature abstracts away the underlying representation of sets and hides the sub-module `Element`. We can instantiate a module for integer sets by applying `AFSet` to `Set.Element`,

as we do for defining the module `ASet`. Due to the signature ascription, `ASet.empty` is the only value of type `ASet.t` that we can build.

In spite of this flexibility, the ML module system does not allow recursive modules. In ML, module dependencies must accord with the definition order. For instance, we cannot define the function `sum` before the `Element` submodule in Figure 1. Thus a programmer cannot define recursive functions or types across module boundaries. The absence of recursive modules is a major disadvantage of the ML module system, when compared to object-oriented languages, like Scala [1] and Java. These languages have supported recursive definitions across class boundaries from the beginning, and this feature is heavily used in practice.

The ML programming language enjoys strong type safety. Yet, due to the lack of recursive modules, a programmer may have to consolidate conceptually separate components into a single module, intruding on modular programming [56]. If ML had both recursive modules and this flexible module language, the programmer could enjoy a strongly type safe programming language with an equally strong expressive power.

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

Initialization: Suppose that a programmer can freely refer to value components of structures forward and backward. Then he might carelessly define value components cyclically like `val l = m val m = l`. Initialization of modules having such cyclic value definitions would either raise a runtime error or cause meaningless infinite computation. Boudol [6], Hirschowitz and Leroy [34, 33, 31, 32], and Dreyer [15] examined type systems which ensure safe initialization of recursive modules. Their type systems ensure that the initialization does not attempt to access undefined recursive variables. The above cyclic definitions will be rejected by their type systems because initialization of the value component `l` requires an access to itself. This path is not the main focus of this thesis.

Type checking: Designing a type system for recursive modules is another important and non-trivial issue; this is the main focus of this thesis. Suppose that a programmer can layout modules in any order regardless of their dependencies. Then, it might happen that a function returns a value whose type is not yet defined at the point where the function is defined. To type

check the function, a type system should somehow know about the type, which is going to be defined in the following part of the program.

Type checking recursive modules

To type check recursive modules, existing proposals [11, 56, 16, 41] rely on signature annotations from a programmer. The programmer has to assist the type checker by writing enough type information so that recursive modules do not burden the type checker with forward references.

The amount of required annotations varies in each proposal and depends on where to enforce type abstraction. In the context of recursive modules, a programmer can enforce type abstraction inside the recursion by giving signatures individually to modules, or outside the recursion by writing a single signature for the whole recursive modules. In all proposals, a programmer has to write two different signatures for the same module to enforce abstraction outside the recursion; one of the signatures is solely for assisting the type checker and does not affect the resulting signature of the module. Moreover, due to the annotation requirement a programmer cannot use type inference during development. This is unfortunate since a lot of useful inference algorithms have been and will be developed to support smooth development of programs.

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

Cyclic type specifications in signatures

To annotate recursive modules with signatures, existing type systems allow some forms of recursive references between signatures. To develop a practical algorithm for judging type equality, one may want to ensure that manifest type specifications in signatures do not declare cyclic types. For instance, one may want to forbid programmers from writing the following cyclic signature:

```
sig type t = s type s = t end
```

Detection of cyclic type specifications is not a trivial task when the module language supports both applicative functors [38, 18] and recursive signatures, as O'Caml does. Applicative functors give us more flexibility in expressing type sharing constraints between modules by allowing type paths to contain functor application. For instance, with functors being applicative

```
module EAFSet =
functor(X : sig type t val unit : t val add : t → t → t end) →
 (struct
   module Set = AFSet(X)
   include Set
   val total l = case l with
        [] ⇒ Element.unit
     | hd :: tl ⇒ Element.add (Set.sum hd) (total tl)
 end : sig
   type t = AFSet(X).t val empty : t val sum : t → X.t
   val total : t list → X.t
 end)
```

Figure 4: An extension of the `AFSet` functor

`AFSet(Set.Element).t` is a valid type in Figure 3. We can further extend the `AFSet` functor with a new function `total`, preserving type equality with `AFset` as shown in Figure 4. While applicative functors are useful, there is the potential that a programmer may carelessly write cyclic type specifications by combining applicative functors and recursive signatures, in such way that a naïve check cannot detect the cycle. Here is one pathological example.

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

Compare the above functor definition with the definition below.

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

On the one hand, a type system would easily detect the latter cycle, since the unrolling of the type `G(X).t` would be exactly `G(X).t`. On the other hand, the former cycle is more difficult to detect, since the unrolling of the type `F(F(X)).t` would yield the following infinite rewriting sequence.

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

Observe that this sequence is not merely cyclic, but produces types of arbitrary long length. In fact, O'Caml type checker diverges for the former functor definition of `F`, since it attempts to build this infinite sequence internally in an attempt to detect cycles.

The situation could become harder, if one wants to keep recursive definitions like:

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

Neither `H` nor `H'` contains cycles. Hence, from the programmer's perspective, there would be no reason to disallow them.

The three examples we have seen are simple. Hence one may easily distinguish between them, judging that only the last one is legal. When recursive modules define more complex types, however, this issue becomes harder to decide.

Potential existence of cyclic type definitions

Another subtle issue involved in recursive modules is how to account for the potential existence of cyclic type definitions in structures, when their implementations are hidden by signatures. For instance, should a type checker reject the program below?

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

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

Existing type systems take different stands on this issue.

In Russo's system [56], a programmer has to write forward declarations for

recursive modules, in which implementations of types other than datatypes cannot be hidden. Thus cyclic type definitions are never hidden by signatures. At the same time, a programmer cannot enforce type abstraction inside recursive modules.

Dreyer's work [16] focuses on type abstraction inside recursive modules. He requires the absence of cyclic type definitions whether or not they are hidden inside signatures. To ensure the absence of cycles without peeking inside signatures, he puts a restriction on types whose implementation can be hidden. As a consequence, the use of structural types is restricted. For instance, his type system would reject the following program, which uses a polymorphic variant type [24] and a list type to represent trees and forests, respectively. (Here we use a polymorphic variant type, which is supported only in O'Caml, since the core language we want to support is that of O'Caml. Yet, a similar restriction could arise in the context of SML, when one attempts to use a record type to represent trees.)

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

By replacing the polymorphic variant type with an usual datatype, one can make this program typable in Dreyer's system. Polymorphic variant types, however, have their own merits that datatypes do not have.

The path O'Caml chose is a more liberal one. It does not care about potential cycles in type definitions, as long as signatures do not specify cycles. The type checker will report an error when signatures contain cyclic type specifications. (The type checker can diverge since, as we mentioned above, recursive modules and applicative functors together make it difficult to detect cycles in a terminating way.) O'Caml has a very expressive core language, whose constructs include structural types such as object types [54] and polymorphic variant types. Moreover, the path it chose keeps flexibility in using these types and in abstracting them away by signatures.

Our proposal of a type system for recursive modules

The goal of our work is to make recursive modules an ordinary construct of the module language for ML programmers. We want to use them easily in

everyday programming, possibly combining with other constructs of the core and the module languages. With this goal in mind, we propose in this thesis a type system for recursive modules which overcomes as much of the difficulties discussed above as possible. Concretely, we follow the path O'Caml chose but extend it by 1) enabling type inference; 2) providing a terminating procedure to detect cyclic type specifications, in the presence of applicative functors; 3) formalizing the type system and proving its soundness, but allowing the potential of cyclic type definitions hidden inside signatures, thus leaving flexibility in using structural types. At the current stage, we confine ourselves to first-order functors. We defer it to future developments to accommodate higher-order functors.

The rest of this thesis is organized into two parts in the following way.

Part II We tackle the first two of the aforementioned difficulties in typing recursive modules, that is, type inference and detection of cycles in type specifications. For a formal study, we design a calculus, named *Marguerite*, which supports recursive modules and applicative functors but does not signature ascription. We develop "expansion algorithms" which can resolve recursive references between modules by tracing module and type abbreviations. These algorithms are terminating; they will either output the result of the expansion or raise an error when they cannot prove that input recursive modules do not contain cyclic or dangling type specifications. Using these algorithms, we design a type system for *Marguerite* and prove that the type system is decidable and sound for a call-by-value operational semantics.

Part III We extend *Marguerite* with signature ascription to make the module language full-fledged. The extended language is named *Traviata*. We reformulate the type system of *Marguerite* for *Traviata*. The resulting type system is two-phased, that is, it consists of a type reconstruction part and a type-correctness check part. The former part is an application of the result of Part I; the latter corresponds to a standard type checking of modules. We prove the type system is still decidable and sound.

Part II
Abbreviation expansion for recursive modules

In this part, we focus on developing "expansion algorithms" for resolving recursive references between modules. The motivation of the algorithms are to reduce types into canonical forms for judging type equality. One can think of canonical forms of types as abbreviation-free types that are obtained by replacing abbreviations with their definitions. To expand types, the algorithms trace abbreviations. Yet we have to be careful to keep them terminating, since the source program may contain dangling or cyclic abbreviations. The algorithms may raise an error when they cannot prove that both the input type and the source program do not contain dangling or cyclic references.

Every type has a unique canonical form, in which all references are resolved. The type system judges type equality by reducing types into canonical forms using the expansion algorithms and by comparing their syntactic equality. For decidability of type checking, termination of the algorithms is critical.

We design the expansion algorithms to be terminating independently of well-typedness of the source program. We cannot rely on well-typedness to keep the algorithms terminating, since we need a type equality judgment to type check the program and our type equality judgment requires types to be in canonical form. The algorithms are developed separately from the type system and proved to be terminating for any input.

This separation has the following two useful consequences.

1. Typing rules are kept straightforward. This is particularly useful to extend the type system later with more expressive language constructs. These new constructs may be accompanied by rather complex typing rules, so we would like to add them without polluting their typing rules with specifics to the expansion algorithms.

2. It is easy to accommodate a possible extension of the algorithms, that is, when we come up with cleverer expansion algorithms we can replace the current ones with the new ones without or with little change in typing rules.

For a formal study, we design in this part a calculus, named *Marguerite*, which supports nested recursive structures and applicative functors. *Marguerite* does not support signature ascription, on which we focus in the next part of this thesis. In this part, we explain the expansion algorithms in detail and prove their termination. We present a type system for *Marguerite*, where the expansion algorithms play an important role in judging type equality. Decidability of the type system is obtained as an immediate consequence of termination of the algorithms. We also prove a soundness result of the type system; the result includes that the expansion algorithms are consistent with the intuition for well-typed programs.

The rest of this part is organized as follows. In the next section, we overview the main features of *Marguerite* using an example. In Section 2, we give the syntax for *Marguerite*. In Section 3 and 4, we develop expansion algorithms for reducing module paths and types, respectively. In Section 5, we present the type system and in Section 6 we prove a soundness result. In Section 7, we discuss how to apply the expansion algorithms to define a core type inference algorithm.

1 Example

In this section, we present a scenario where recursive modules naturally arise and explain difficulties involved in type checking recursive modules, using an example given in Figure 5.

The top-level structure contains three sub-modules `S`, `Tree` and `Forest`, where `Tree` and `Forest` are defined in a mutually recursive way. The module `S` is an abbreviation for a module `IntSet`, which is we assume given in a library. The module `Tree` represents trees whose leaves and nodes are labeled with integers. The module `Forest` represents unordered sets of those integer trees.

To enable forward references between modules, we extend the top-level structure with an implicitly typed declaration of a *self variable*. Components of the top-level structure can refer to each other recursively using the self variable, regardless of definition ordering. For instance in the example, the top-level structure declares a self variable named `TF`, which is used inside `Tree` and `Forest` for recursive references to each other. We keep the usual ML scoping rules for implicit backward references. Thus the function `Tree.split` can refer to the `Leaf` and `Node` constructors without going through the self variable. It was possible to use `Tree`, instead of `TF.Tree`, inside `Forest`, since `Tree` is a backward reference for `Forest`. But the explicit notation seems clearer.

Let us explain the implementations of `Tree` and `Forest` in detail. Two types `Tree.t` and `Forest.t` refer to each other recursively. On the one hand, the datatype definition of `Tree.t` involves a type name `s`, which is an abbreviation for the type `TF.Forest.t`, a reference to the type `Forest.t`. On the other hand, the type `Forest.t` is a synonym for the type `T.t list`, where the type `T.t` is an abbreviation for `TF.Tree.t`, a reference to the type `Tree.t`. Two functions `Tree.labels` and `Forest.labels` call each other recursively. These functions calculate the sets of integers that a tree and a forest contain, respectively. Using these functions, we define the function `Forest.incr`, which augments a given forest only if a given tree contains original labels that are not contained in the forest.

The function `split` in `Tree` cuts off the root node of a given tree, then returns the resulting forest. The function `sweep` in `Forest` gathers leaves from a given forest. These two functions also make recursive references. The second case branch of `Tree.split` depends on the fact that a forest is a list of trees; `Forest.sweep` constructs and deconstructs trees through the

```
struct (TF)
  module S = IntSet
  module Tree = struct
   module F = TF.Forest
   type s = F.t
   datatype t = Leaf of int | Node of int * s
   val labels = λx.case x of Leaf i ⇒ TF.S.singleton i
     | Node (i, f) ⇒ TF.S.add i (F.labels f)
   val split = λx.case x of Leaf i ⇒ [Leaf i]
     | Node (i, f) ⇒ (Leaf i) :: f
  end
  module Forest = struct
   module T = TF.Tree
   type t = T.t list
   val labels = λx.case x of [] ⇒ TF.S.empty
     | hd :: tl ⇒ TF.S.union (T.labels hd) (labels tl)
   val incr = λf.λt.let l1 = labels f in
     let l2 = T.labels t in
     if TF.S.diff l2 l1 != TF.S.empty then (t :: f) else f
   val sweep = λx.case x of [] ⇒ []
     | (T.Leaf y) :: tl ⇒ (T.Leaf y) :: (sweep tl)
     | (T.Node y) :: tl ⇒ sweep tl
  end
 end
```

Figure 5: Tree and forest

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

Figure 6: A signature for `Tree` and `Forest`

constructors `Leaf` and `Node`, which are declared inside `Tree`.

Judging type equality The main difficulty in type checking this example is in judging type equality. For instance, let us consider type checking the second branch of the function `Tree.split`. For the list cons operation `(Leaf i) :: f` to be well-typed, `f` must be a list of trees. In the datatype definition of `Tree.t`, the constructor `Node` is described to contain an integer and a forest. By tracing underlined four abbreviations, a type system could expand the type `Tree.s` into `TF.Tree.t list`. Then it would conclude that the list cons operation is well-typed.

In this simple well-typed example, there is clearly no potential of divergence in tracing abbreviations. Having both recursive modules and applicative functors, however, a programmer might carelessly write pathologically cyclic abbreviations which are hard to detect. Then a naïve way of tracing abbreviations may diverge, causing non-terminating type checking. In Section 3 and 4, we examine such pathological examples and develop "expansion algorithms" which trace abbreviations in a terminating way for reducing types into abbreviation-free forms.

Type inference To type check the example, a type system also needs to support type inference. Suppose that we want to give a signature in Figure 6 to the example, where we extend usual ML signatures with implicitly typed declarations of self variables to allow recursive references inside signatures. The signature enforces type abstraction by hiding underlying implementations of the types `Tree.t` and `Forest.t`. Moreover it does not mention functions `Tree.labels` and `Forest.labels`. Since the implementation of the function `Forest.incr` relies on these two functions, a type system has to infer their types to type check `Forest.incr`.

Indeed, without type inference, a programmer has to write two different

signatures to enforce desired abstraction; one for the abstraction, which is given in Figure 6 and one for assisting the type checker, which we will examine below.

To avoid presenting too verbose signature annotations, we consider in the following examination the program in Figure 5 without the module abbreviation `module F = TF.Forest` inside `Tree`. We can dispense with abbreviations by replacing them with their definitions; yet abbreviations are useful in practical programs [57].

To type check the example in Dreyer's system [16] or O'Caml [42], a programmer has to write fully manifest signatures of `Tree` and `Forest`, that is, he has to present the type checker with the following signatures:

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

and

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

In Russo's system [56], the self variable `TF` of the top-level structure must be annotated with the recursive signature below. In his system, a recursive signature contains a typed declaration of a self variable to support forward references in the signature.

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

These additional signature annotations are indispensable in existing proposals and must be given beforehand. Then, the type checker first type checks the example assisted by these manifest signatures. Once this succeeds, type

abstraction can be enforced using the signature given in Figure 6.

Marguerite supports type inference unlike other proposals, hence it does not need the assistance of signature annotations. Indeed, it has an ability to reconstruct the fully manifest signatures of `Tree` and `Forest`, which the programmer has to write by himself in Dreyer's and O'Caml type systems. This implies that the signature in Figure 6 is sufficient for *Marguerite* to type check the example and to enforce type abstraction together. In Section 7, we explain how we define a type inference algorithm using our abbreviation expansion algorithms. In the next part of this thesis we examine and formalize how to type check the example when the signature in Figure 6 is given by the programmer.

2 Syntax

We give the syntax for the module language of *Marguerite* in Figure 7. It is based on Leroy's applicative functor calculus [38]. We use M as a metavariable for module names, X for names of module variables and Z for names of self variables. For simplicity, we distinguish them syntactically, however the context could tell them apart without this distinction. We also use t as a metavariable for type names, l for value names and c for constructor names.

Every module expression and signature is labeled with an integer. We use these integer labels to keep expansion algorithms terminating. For instance, a module expression E is a module expression body E_d labeled with an integer i. One can think of the integer label i of E_d^i as the location of E_d in the source program. For the interest of brevity, we may omit integer labels when they are not used. For the interest of clarity, we may write additional parentheses, for instance $(\texttt{functor}(\texttt{X} : \texttt{sig type t end}^2) \to \texttt{X}^3)^1$. We use metavariables i, j for integers.

A module expression body E_d is either a structure, a functor or a module path. A structure is a sequence of module, type and value definitions. A type definition may generate a new datatype or may be an alias for another type. In particular, that structures can contain sub-modules is an important feature of the ML module system. A functor is a function over modules. Signatures for functor arguments must be given explicitly. A functor can only be applied to a module which implements the specified signature of the argument. A signature is a sequence of specifications labeled with an integer. A type specification may expose the underlying implementation of the specified type (datatype and manifest type specifications) or may hide the implementation (abstract type specification). A value is specified with its type.

A module path is a reference to a module. The flexible referencing mechanism given by module paths is a key feature of *Marguerite*. A module path may refer to a module at any level of nesting within the recursive structure, regardless of component ordering. Moreover, module paths can contain simple cases of functor applications, where the functor and its arguments themselves are paths. Concretely, module paths are constructed from self variables, the dot notation [9] "$.M$", which represents access to the sub-module named M of a structure, and functor applications. The syntax of module paths in Figure 7 restricts module paths not to contain paths of the forms $X.M$ and $X(p)$. We explain this later.

Module expression

$$E \quad ::= \quad E_d^i$$

Module expression bodies

$$\begin{array}{lcll} E_d & ::= & \texttt{struct}\ D_1 \ldots D_n\ \texttt{end} & \textit{structure} \\ & | & \texttt{functor}\ (X : S) \to E & \textit{functor} \\ & | & p & \textit{module path} \end{array}$$

Definitions

$$\begin{array}{lcll} D & ::= & \texttt{module}\ M = E & \textit{module def.} \\ & | & \texttt{datatype}\ t = c\ \texttt{of}\ \tau & \textit{datatype def.} \\ & | & \texttt{type}\ t = \tau & \textit{type abbreviation} \\ & | & \texttt{val}\ l = e & \textit{value def.} \end{array}$$

Signature

$$S \quad ::= \quad S_d^i$$

Signature body

$$S_d \quad ::= \quad \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} \qquad \textit{structure type}$$

Specifications

$$\begin{array}{lcll} B & ::= & \texttt{datatype}\ t = c\ \texttt{of}\ \tau & \textit{datatype type spec.} \\ & | & \texttt{type}\ t = \tau & \textit{manifest type spec.} \\ & | & \texttt{type}\ t & \textit{abstract type spec.} \\ & | & \texttt{val}\ l : \tau & \textit{value spec.} \end{array}$$

Module identifiers

$$mid \quad ::= \quad Z \mid mid.M \mid mid(p)$$

Module paths

$$p, q, r \quad ::= \quad mid \mid X$$

Program

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

Figure 7: Syntax for the module language

$$
\begin{array}{rcl}
\textit{Core types} & & \\
\tau & ::= & \mathtt{1} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2 \mid p.t \\
\textit{Core expressions} & & \\
e & ::= & x \mid () \mid (e_1, e_2) \mid \pi_i(e) \mid (\lambda x.e : \tau) \mid e_1(e_2) \\
& \mid & p.c\ e \mid \mathtt{case}\ e\ \mathtt{of}\ p.c\ x \Rightarrow e \mid p.l
\end{array}
$$

Figure 8: Syntax for the core language

For the sake of simplicity, we assume that functor applications only contain module paths but not structures or functors. This does not reduce the expressive power of the language [39] and we believe that in several situations we can allow a larger class of functor applications, following Leroy's proposal [40].

A program is a top-level structure extended with an implicitly typed declaration of a self variable. A self variable is bound inside the top-level structure where the variable is declared. In this thesis, we only consider a bunch of recursive modules but not ordinary ones (i.e., non-recursive modules).

To develop a decidable type system, we impose a *first-order structure restriction* that requires functors 1) not to take functors as arguments and 2) not to access sub-modules of arguments. The first restriction means that our functors are not higher-order, while they can still return functors. The second restriction implies that a programmer has to pass sub-modules as independent parameters to a functor instead of passing a single module which contains all the sub-modules. The restriction on the syntax of module paths is consistent with this restriction.

In Figure 8, we give the syntax for the core language of *Marguerite*.

A core type is either a unit type $\mathtt{1}$, an arrow type $\tau_1 \to \tau_2$, a pair type $\tau_1 * \tau_2$ or a type path $p.t$, which refers to a type component named t in the structure that the module path p refers to. A core expression is either a core variable (variable, for short) x, a null $()$, a pair (e_1, e_2), a projection $\pi_i(e)$, an abstraction $(\lambda x.e : \tau)$, an application $e_1(e_2)$, a value construction $p.c\ e$ or deconstruction $\mathtt{case}\ e\ \mathtt{of}\ p.c\ x \Rightarrow e$, or a value path $p.l$, which refers to a value component named l in the structure that the module path p refers to.

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

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

`functor(X : sig type t val l : X.t end)` $\rightarrow$ `X`

is legal in *Marguerite*, which should be understood as

`functor(X : sig type t val l : t end)` $\rightarrow$ `X`

This convention is convenient when proving a soundness result, as the syntax of paths is kept uniform, that is, every path is prefixed by either a self variable or a module variable. In Section 13, we give an example where this convention is useful.

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

In the formalization, 1) function definitions are explicitly type annotated; 2) a path is always prefixed by either a self variable or a module variable. Our examples do not stick to these rules. Instead, we have assumed that there is an elaboration phase, prior to type checking, that adds type annotations for functions by running a type inference algorithm for the core language. The original program may still require some type annotations, to avoid running into the polymorphic recursion problem [30]. In Section 7, we discuss the details of this inference algorithm. The elaboration phase also infers omitted self variables, to complete implicit backward references.

We assume the following three conventions: 1) a program does not contain free module variables or free self variables; 2) all binding occurrences of module variables use distinct names; 3) any sequence of module definitions, datatype definitions, type abbreviations, value definitions, datatype specifications, manifest and abstract type specifications, and value specifications does not contain duplicate definitions or specifications for the same name.

3 Module path expansion

In this section, we develop a module path expansion algorithm for determining the module that a module path refers to. The type system uses the algorithm in the following three contexts.

1. To type check a functor application $p_1(p_2)$, the type system expands p_1 to make sure that p_1 indeed refers to a functor definition and to discover the argument signature of the functor which p_2 must implement (Section 5).

2. The type expansion algorithm is defined on top of the module path expansion algorithm (Section 4).

3. The type system decides an order for type inference using the module path expansion algorithm (Section 7).

The module path expansion algorithm reduces module paths into *located forms*. Intuitively, a module path p is in located form when, for all paths q contained in p, the reference of q is already resolved. To define formally, we introduce a *look-up judgment*.

Look-up judgment A *program environment* Δ is a mapping from a self variable to a top-level structure and from module variables to signatures. For a program P, the program environment of P, written Δ_P, is the program environment whose domain exactly contains the self variable declared in the top-level structure of P and all module variables appearing in P and which sends the self variable to P and module variables to their own signatures specified in P. We write $dom(\Delta)$ to denote the domain of Δ.

Given a module environment Δ, we define in Figure 10 a look-up judgment which determines the module that a given module path refers to with respect to Δ. We use θ as a metavariable for *module variable bindings*, which map module variables to module paths and write $dom(\theta)$ to denote the domain of θ. The judgment $\Delta \vdash p \mapsto (\theta, K)$ means that the module path p resolves to the module description K, where each module variable X is bound to $\theta(X)$ *w.r.t.*[2] Δ. We write ϵ to denote the empty module variable binding, that is, a module variable binding whose domain is empty. We use

[2]with respect to

the notation convention in Figure 9. In particular, we use K_d as a metavariable for *module description bodies*, which are either module expression bodies or signature bodies, and K for *module descriptions*, which are either module expressions or signatures. For a module variable binding θ, $\theta[X \mapsto p]$ denotes a module variable binding whose domain is $dom(\theta) \cup \{X\}$ and which maps X to p and coincides with θ on $dom(\theta) \backslash \{X\}$.

Let us examine each rule of the look-up. For self variables and module variables, the judgment consults the program environment Δ. A path $p.M$ resolves to the sub-module named M in the structure that p resolves to. Hence p_1 must resolve to a structure. A path $p_1(p_2)$ resolves to the body of the functor that p_1 resolves to, where the module variable binding is augmented with a new binding $[X \mapsto p_2]$.

The look-up judgment only holds for module paths whose references are already resolved. For instance, consider the program P_1 in Figure 11. Let Δ_{P_1} be the program environment of P_1, or $\Delta_{P_1} = [\mathtt{Z} \mapsto P_1, \mathtt{X} \mapsto \mathtt{sig\ type\ t\ end}^3]$. We have a derivation whose conclusion is:

$$\Delta_{P_1} \vdash \mathtt{Z.M}_1(\mathtt{Z.M}_2).\mathtt{M}_{11} \mapsto ([\mathtt{X} \mapsto \mathtt{Z.M}_2], \mathtt{struct\ end}^5)$$

but no derivation for the path $\mathtt{Z.M}_3.\mathtt{M}_{11}$.

Corresponding to the convention of the absence of free module variables in programs, we assume that any program variable environment we consider in this thesis does not contain free module variables. Precisely,

Definition 1 *A program variable environment Δ does not contain free module variables if, for any module path p other than module variables, when $\Delta \vdash p \mapsto (\theta, K)$ then the following two conditions hold.*

1. *$MVars(K) \subseteq dom(\theta)$*
2. *For all X in $dom(\theta)$, $MVars(\Delta(X)) \subseteq dom(\theta)$.*

For a module description K, $MVars(K)$ denotes the set of free module variables in K.

Located forms Now we define located forms, which are output of the module path expansion. A module path p is in located form if and only if p does not contain a module path which resolves to a module path according to the look-up judgment. Precisely,

Module description	K	$::=$	K_d^i
Module description bodies	K_d	$::=$	$E_d \mid S_d$
	$\mathtt{ss}$	$::=$	$\mathtt{struct} \mid \mathtt{sig}$

Figure 9: Notation convention

$$\textbf{[lk-self]}\ \frac{}{\Delta \vdash Z \mapsto (\epsilon, \Delta(Z))} \qquad \textbf{[lk-mvar]}\ \frac{}{\Delta \vdash X \mapsto (\epsilon, \Delta(X))}$$

$$\textbf{[lk-dot]}\ \frac{\Delta \vdash p \mapsto (\theta, \mathtt{struct} \ldots \mathtt{module}\ M = E \ldots \mathtt{end}^i)}{\Delta \vdash p.M \mapsto (\theta, E)}$$

$$\textbf{[lk-app]}\ \frac{\Delta \vdash p_1 \mapsto (\theta, (\mathtt{functor}(X : S) \to E)^i)}{\Delta \vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], E)}$$

Figure 10: Look-up

$$\begin{array}{l}
\mathtt{struct}\ (\mathtt{Z}) \\
\quad \mathtt{module}\ \mathtt{M}_1 = (\mathtt{functor}(\mathtt{X}\ :\ \mathtt{sig}\ \mathtt{type}\ \mathtt{t}\ \mathtt{end}^3) \to \\
\quad\quad \mathtt{struct} \\
\quad\quad\quad \mathtt{module}\ \mathtt{M}_{11} = \mathtt{struct}\ \mathtt{end}^5 \\
\quad\quad\quad \mathtt{module}\ \mathtt{M}_{12} = \mathtt{X}^6 \\
\quad\quad \mathtt{end}^4)^2 \\
\quad \mathtt{module}\ \mathtt{M}_2 = \mathtt{struct}\ \mathtt{type}\ \mathtt{t} = \mathtt{int}\ \mathtt{end}^7 \\
\quad \mathtt{module}\ \mathtt{M}_3 = \mathtt{Z}.\mathtt{M}_1(\mathtt{Z}.\mathtt{M}_2)^8 \\
\mathtt{end}^1
\end{array}$$

Figure 11: A program P_1

Definition 2 *A module path p is in located form w.r.t. a program environment Δ if the following two conditions hold:*

- $\Delta \vdash p \mapsto (\theta, K_d^i)$ *where K_d is not a module path.*
- *For all q in $args(p)$, q is in located form w.r.t. Δ.*

For a module path p, $args(p)$ denotes the set of module paths appearing inside p as functor arguments. Precisely:

$$args(Z) = \emptyset \qquad args(X) = \emptyset$$
$$args(p.M) = args(p) \quad args(p_1(p_2)) = args(p_1) \cup \{p_2\}$$

We say that a module variable binding θ is in located form *w.r.t.* Δ if and only if, for all X in $dom(\theta)$, $\theta(X)$ is in located form *w.r.t.* Δ.

The module path expansion algorithm reduces a module path into a located form or raises an error when it cannot prove that the input path does not contain dangling or cyclic references. The basic idea of the algorithm is straightforward. It traces module abbreviations until either the input module path resolves to a structure or a functor or it is reduced to a module variable. To keep the algorithm terminating, we have to be careful about the potential existence of cyclic module abbreviations. Below we give two pathological examples which contain cycles.

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

The first example is:

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

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

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

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

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

$$\frac{\Delta, \emptyset \vdash p \rightsquigarrow_g q \quad \Delta \vdash \mathit{varnlz}(q) = r}{\Delta \vdash p \rightsquigarrow r}$$

Figure 12: Module path expansion

3.1 Module path expansion algorithm

We define the module path expansion in Figure 12. It is a composite of *ground expansion* and *variable normalization*. The inference rule in Figure 12 means that the expansion reduces a given module path p into q *w.r.t.* a given program environment Δ, if the ground expansion reduces p into q, written $\Delta, \emptyset \vdash p \rightsquigarrow_g q$ and the variable normalization reduces q into r, written $\Delta \vdash \mathit{varnlz}(q) = r$. We may say that q is the located form of p when $\Delta \vdash p \rightsquigarrow q$.

The ground expansion and the variable normalization are defined below. Both are terminating (Proposition 2 and 3). As a result, the module expansion is also terminating (Proposition 4).

3.1.1 Ground expansion

The ground expansion is ground in the sense that it does not rely on functor arguments. It either reduces a module path into a *pre-located form* or raises an error when 1) it cannot prove that the input module path does not contain cyclic or dangling references, or 2) the input module path does not obey the first-order structure restriction.

We first introduce pre-located forms, the central idea for defining the ground expansion. A module path p is in pre-located form if and only if p does not contain a module path which resolves to a module identifier according to the look-up judgment. Precisely,

Definition 3 *A module path p is in pre-located form w.r.t. a program environment Δ if the following two conditions hold:*

- $\Delta \vdash p \mapsto (\theta, K_d^i)$ *and K_d is not a module identifier. (Hence K_d can be a module variable.)*
- *For all q in $\mathit{args}(p)$, q is in pre-located form w.r.t. Δ.*

The locution "pre-located form" indicates that we can turn a pre-located form into a located form by substituting functor arguments, as we show in Proposition 3.

We say that a module variable binding θ is in pre-located form *w.r.t.* Δ if and only if, for all X in $dom(\theta)$, $\theta(X)$ is in pre-located form *w.r.t.* Δ.

The important feature of pre-located forms is that they satisfy a substitution property, as stated in Lemma 1 below. We first define length of module paths, which we use to prove the lemma.

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

For a module path p and a module variable binding θ, we write $\theta(p)$ to denote the module path obtained by applying the substitution θ to p. Precisely,

$$\theta(Z) = Z \qquad \theta(X) = \begin{cases} X & \text{when } X \notin dom(\theta) \\ p & \text{when } X \in dom(\theta) \text{ and } \theta(X) = p \end{cases}$$
$$\theta(p.M) = \theta(p).M \quad \theta(p_1(p_2)) = \theta(p_1)(\theta(p_2))$$

Lemma 1 (Substitution property) *Let p and θ be in pre-located form w.r.t. Δ. Then $\theta(p)$ is in pre-located form w.r.t. Δ.*

Proof. By induction on the length of p. □

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

Lemma 2 *Let p be in pre-located form w.r.t. Δ. If $\Delta \vdash p \mapsto (\theta, K)$, then θ is in pre-located form w.r.t. Δ.*

Proof. By induction on the derivation of $\Delta \vdash p \mapsto (\theta, K)$. □

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

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

[gnlz-mvar]

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

[gnlz-self]

$$\frac{}{\Delta, \Sigma \vdash Z \rightsquigarrow_g Z}$$

[gnlz-def1]

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

[gnlz-pth1]

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

[gnlz-def2]

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

[gnlz-pth2]

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

Figure 13: Ground expansion

The module variable binding $[\mathtt{X} \mapsto \mathtt{L}]$ is in pre-located form, but the module path `L.M` is not (because `L.M` resolves to a module identifier).

We define the ground expansion in Figure 13. The judgment $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ means that the ground expansion reduces the module path p into the module path q with Σ locked *w.r.t.* the program environment Δ. We use Σ as a metavariable for sets of integers. The notation $\Sigma \uplus i$ means $\Sigma \cup \{i\}$ whenever $i \notin \Sigma$. We may say that q is the pre-located form of p *w.r.t.* Δ when $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ holds for some Σ.

Observe that for any program environment Δ, module path p and lock Σ, proof search for $\Delta, \Sigma \vdash p \rightsquigarrow_g \_$ is deterministic, where "_" is a place-holder. In other words, for any Δ, p and Σ we can search a derivation tree whose conclusion is $\Delta, \Sigma \vdash p \rightsquigarrow_g \_$ in a deterministic way. The search may fail with no applicable rules. When it is successful, we find a module path q such that $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ holds. In this way, we regard the inference rules of the ground expansion as defining an algorithm which takes Δ, p and Σ as input then either returns q when the search succeeds in building a derivation tree of $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ or raises an error when the search fails. We prove termination of the proof search later in Proposition 2.

Let us examine each rule in Figure 13. The first two rules **[gnlz-mvar]**

and [**gnlz-self**] are straightforward. For a path of the form $p.M$, the ground expansion first reduces the prefix p ([**gnlz-def1**][**gnlz-pth1**]). Suppose that p' is the pre-located form of p. Then there are two cases depending on whether $p'.M$ resolves to a module identifier or not. When $p'.M$ resolves to a module description other than a module identifier ([**gnlz-def1**]), then $p'.M$ is in pre-located form and the ground expansion terminates. When $p'.M$ resolves to a module identifier q ([**gnlz-pth1**]), then the ground expansion traces the abbreviation q. This is the key rule, hence we explain it in detail.

As a simple case, suppose that q is in pre-located form *w.r.t.* Δ. Then $\Delta, \Sigma \uplus i \vdash q \rightsquigarrow_g q$ holds immediately whenever i is not in Σ (see Lemma 6) and the ground expansion returns $\theta(q)$. By Lemma 1 and 2, we know that $\theta(q)$ is in pre-located form. In general, q is not necessarily in pre-located form. Hence, the ground expansion reduces q first to obtain its pre-located form in the premise $\Delta, \Sigma \uplus i \vdash q \rightsquigarrow_g r$, then applies the substitution θ to r.

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

The rules [**gnlz-def2**] and [**gnlz-pth2**] for paths of the form $p_1(p_2)$ are similar to [**gnlz-exp1**] and [**gnlz-pth1**], respectively.

3.1.2 Well-definedness and termination

Here we prove that the ground expansion does reduce module paths into pre-located forms unless it raises an error and that it is terminating.

Proposition 1 (Well-definedness of the ground expansion) *For any program environment Δ, lock Σ and module paths p, q, if $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ then q is in pre-located form w.r.t. Δ.*

Proof. By induction on the derivation of $\Delta, \Sigma \vdash p \rightsquigarrow_g q$ and by case on the last rule used. Use Lemma 1 and 2 for the rules [**gnlz-pth1**] and [**gnlz-pth2**]. □

We prove termination by defining well-founded relations.

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

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

Proof. Below, we define a well-founded relation $>_{g_\Delta}$ on pairs (p, Σ) of a module path p and a lock Σ *w.r.t.* Δ. It is easy to check that if $\Delta, \Sigma_2 \vdash p_2 \rightsquigarrow_g \_$ is a premise of $\Delta, \Sigma_1 \vdash p_1 \rightsquigarrow_g \_$, then $(p_1, \Sigma_1) >_{g_\Delta} (p_2, \Sigma_2)$. Thus, if there is an infinitely deep derivation tree of the ground expansion, then there is an infinitely descending sequence in $>_{g_\Delta}$. This contradicts well-foundedness of $>_{g_\Delta}$. By Köning's lemma on finitely branching trees, we obtain the proposition.

$(p_1, \Sigma_1) >_{g_\Delta} (p_2, \Sigma_2)$ holds if and only if either of the following three conditions holds. We write $\mathit{IntLabs}_\Delta$ to denote the set of integer labels appearing in Δ.

1. $p_1 = p_1'.M$ and $p_2 = p_1'$ and $\Sigma_1 = \Sigma_2$.
2. $p_1 = p_{11}(p_{12})$ and $p_2 = p_{1i}$ with i being either 1 or 2, and $\Sigma_1 = \Sigma_2$.
3. i is not in Σ_1 and $\Sigma_2 = \Sigma_1 \cup \{i\} \subseteq \mathit{IntLabs}_\Delta$

Well-foundedness of $>_{g_\Delta}$ follows from the finiteness of $\mathit{IntLabs}_\Delta$. □

3.2 Variable normalization

The variable normalization turns pre-located forms into located forms. In Figure 14 we define the variable normalization using two functions *varnlz* and *varsubst*. When the input module path resolves to a module variable, these functions recursively substitute for the variable the module path which is bound to the variable, according to the look-up judgment.

The proposition 3 below indicates that by combining the ground expansion and the variable normalization, we can calculate located forms.

Lemma 3 *Let p be in located form w.r.t. Δ. If $\Delta \vdash p \mapsto (\theta, K)$, then θ is in located form w.r.t. Δ.*

$$\frac{}{\Delta \vdash \mathit{varnlz}(X) = X} \quad \frac{}{\Delta \vdash \mathit{varnlz}(Z) = Z}$$

$$\frac{\Delta \vdash \mathit{varnlz}(p) = p' \quad \Delta \vdash \mathit{varsubst}(p'.M) = q}{\Delta \vdash \mathit{varnlz}(p.M) = q}$$

$$\frac{\Delta \vdash \mathit{varnlz}(p_1) = p_1' \quad \Delta \vdash \mathit{varnlz}(p_2) = p_2' \quad \Delta \vdash \mathit{varsubst}(p_1'(p_2')) = q}{\Delta \vdash \mathit{varnlz}(p_1(p_2)) = q}$$

$$\frac{\Delta \vdash p \mapsto (\theta, X^i)}{\Delta \vdash \mathit{varsubst}(p) = \theta(X)} \qquad \frac{\Delta \vdash p \mapsto (\theta, K_d^i) \quad K_d \neq X}{\Delta \vdash \mathit{varsubst}(p) = p}$$

Figure 14: Variable normalization

Proof. By induction on the derivation of $\Delta \vdash p \mapsto (\theta, K)$. □

Lemma 4 *Let p be in pre-located form w.r.t. Δ. If $\Delta \vdash p \mapsto (\theta, K_d^i)$ with $K_d \neq X$ and $\Delta \vdash \mathit{varnlz}(p) = q$, then $\Delta \vdash q \mapsto (\theta_1, K_d^i)$ where, for all X in $\mathit{dom}(\theta)$, $\Delta \vdash \mathit{varnlz}(\theta(X)) = \theta_1(X)$.*

Proof. By induction on the length of p. Observe that by definition on the look-up, $\mathit{dom}(\theta) = \mathit{dom}(\theta_1)$. □

Proposition 3 *When p is in pre-located form w.r.t. Δ, then there is a unique q in located form w.r.t. Δsuch that $\Delta \vdash \mathit{varnlz}(p) = q$.*

Proof. By induction on the length of p. We show the main case.
When $p = p_1.M$. By induction hypothesis, there is a unique p_2 in located form such that $\Delta \vdash \mathit{varnlz}(p_1) = p_2$. Since $p_1.M$ is in pre-located form, $\Delta \vdash p_1 \mapsto (\theta, \texttt{struct} \ldots \texttt{end}^i)$. By Lemma 4, we obtain the lemma. □

3.3 Termination and well-definedness of the module path expansion

Finally we prove that the module path expansion is terminating and that it does reduces module paths into located forms unless the ground expansion raises an error. We also present some lemmas that are used later in this thesis.

Proposition 4 (Termination of the module path expansion) *For any program environment Δ and module path p, proof search for $\Delta \vdash p \rightsquigarrow \_$ will terminate.*

Proof. The proposition is an immediate consequence of Proposition 2 and Proposition 3. □

Proposition 5 (Well-definedness of the module path expansion) *For any program environment Δ and module paths p, q, if $\Delta \vdash p \rightsquigarrow q$, then q is in located form w.r.t. Δ.*

Proof. By hypothesis, we have $\Delta, \emptyset \vdash p \rightsquigarrow_g p'$ and $\Delta \vdash \mathit{varnlz}(p') = q$. By Proposition 1, p' is in pre-located form *w.r.t.* Δ. By Proposition 3, q is in located form *w.r.t.* Δ. □

The following lemmas are proven by easy induction.

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

Lemma 6 *Let p be in pre-located form w.r.t. Δ. Then $\Delta, \Sigma \vdash p \rightsquigarrow_g p$ for any Σ.*

Lemma 7 *Let p be in located form w.r.t. Δ. Then $\Delta \vdash \mathit{varnlz}(p) = p$.*

Lemma 8 *Let p be in located form w.r.t. Δ. Then $\Delta \vdash p \rightsquigarrow p$.*

Proof. By Lemma 6 and 7. Recall that pre-located forms include located forms. □

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

4 Type expansion

In this section, we develop a type expansion algorithm, which reduces types into canonical forms by unrolling type abbreviations. The purpose of the type expansion is to define type equality. Each type has a unique canonical form unless it does not contain dangling or cyclic references. Hence, once types are reduced into canonical forms we can judge their equality in a syntactic way.

Located types We first introduce canonical forms of types, named *located types*, which are output from the type expansion. A located type consists of *simple located types* and unit types. A simple located type is an abstract type, i.e.

Definition 5 *A simple located type w.r.t. a program environment Δ is a type path $p.t$ where p is in located form w.r.t. Δ and either $\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end}^i)$ or $\Delta \vdash p \mapsto (\theta, \mathtt{sig} \ldots \mathtt{type}\ t \ldots \mathtt{end}^i)$ holds.*

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

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

Then we define located types as follows.

Definition 6 *A located type w.r.t. a program environment Δ is a type τ where each type τ' in $\mathit{typaths}(\tau)$ is a simple located type w.r.t. Δ.*

4.1 Type expansion algorithm

We define the type expansion in Figure 15. The judgment $\Delta; \Omega \vdash \tau \downarrow \tau'$ means that the expansion reduces the type τ into the type τ' where Ω is locked *w.r.t.* the program environment Δ. We use Ω as a metavariable for sets of pairs (i, t) of an integer i and a type name t.

Observe that for any program environment Δ, lock Ω and type τ, proof search for $\Delta; \Omega \vdash \tau \downarrow \_$ is deterministic. We regard inference rules of the type

[tnlz-uni]

$$\frac{}{\Delta;\Omega \vdash \mathtt{1} \downarrow \mathtt{1}}$$

[tnlz-arr]

$$\frac{\Delta;\Omega \vdash \tau_1 \downarrow \tau_1' \quad \Delta;\Omega \vdash \tau_2 \downarrow \tau_2'}{\Delta;\Omega \vdash \tau_1 \to \tau_2 \downarrow \tau_1' \to \tau_2'}$$

[tnlz-pair]

$$\frac{\Delta;\Omega \vdash \tau_1 \downarrow \tau_1' \quad \Delta;\Omega \vdash \tau_2 \downarrow \tau_2'}{\Delta;\Omega \vdash \tau_1 * \tau_2 \downarrow \tau_1' * \tau_2'}$$

[tnlz-dtyp]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end}^i)}{\Delta;\Omega \vdash p.t \downarrow p'.t}$$

[tnlz-atyp]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t \ldots \mathtt{end}^i)}{\Delta;\Omega \vdash p.t \downarrow p'.t}$$

[tnlz-abb]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p' \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t = \tau_1 \ldots \mathtt{end}^i) \quad \Delta;\Omega \uplus (i,t) \vdash \tau_1 \downarrow \tau_2 \quad \Delta;\Omega \vdash \theta(\tau_2) \downarrow \tau}{\Delta;\Omega \vdash p.t \downarrow \tau}$$

Figure 15: Type expansion

expansion as defining an algorithm which takes Δ, Ω and τ as input then either returns τ' as output when the search succeeds in building a derivation tree for $\Delta;\Omega \vdash \tau \downarrow \tau'$ or raises an error when the search fails. We prove termination of the proof search later in Proposition 7.

Let us examine each rule of the type expansion. The first three rules **[tnlz-uni]**, **[tnlz-arr]** and **[tnlz-pair]** are straightforward.

For a type type $p.t$, the expansion first reduces its prefix p into a located form p' to determine the module that p refers to (**[tnlz-dtyp]**[**tnlz-atyp**][**tnlz-abb**]). When the module path expansion fails, then the type expansion fails too. Even though the module path expansion succeeds, the type expansion may fail, if p' resolves to a functor; in that case the type path $p'.t$ is dangling, hence so is $p.t$. When the module path expansion succeeds in reducing p into p' and when p' resolves to a structure or structure type, the type expansion continues. There are four possible cases:

1) The structure (type) does not contain a type component named t. In this case $p.t$ is dangling.

2) It contains a datatype definition or specification named t ([**tnlz-dtyp**]).

3) It contains an abstract type specification named t ([**tnlz-atyp**]).

4) It contains a type abbreviation or manifest type specification named t ([**tnlz-abb**]).

For the cases 2) and 3), the expansion terminates immediately returning the type $p'.t$, which is already a located type. The last case 4) is very important and we will explain in detail.

When t is an alias for another type, then the expansion should trace the aliased type while avoiding divergence possibly caused by cyclic abbreviations. The rule [**tnlz-abb**] says that to reduce $\theta(\tau_1)$, for which the type $p'.t$ is alias, the expansion 1) first reduces τ_1 into a located type τ_2 without applying the module variable binding θ to τ_1, 2) then reduces the type $\theta(\tau_2)$ by applying θ to the newly obtained type τ_2. When reducing τ_1, the expansion augments the lock Ω with a new entry (i, t), which is released when reducing $\theta(\tau_2)$.

Compare the rule [**tnlz-abb**] to the rule [**gnlz-pth1**] of the ground expansion. Both handle abbreviations and have similar premises except that the type expansion continues after applying the module variable binding θ to the newly obtained type τ_2, while the ground expansion terminates immediately after applying θ to the newly obtained path r. Since located types do not satisfy a substitution property like module paths in located form do, it does not necessarily hold that applying a module variable binding in located form to a located type produces a located type. Due to this difference, the type expansion appears to be more involved than the ground expansion. We first study a simple case in detail below, to give the intuition of the type expansion. Then we examine key cases by giving concrete examples in Example 1 and 2.

First, we prove two useful lemmas about the type expansion. Lemma 9 presents a weak substitution property that simple located types satisfy. Lemma 10 states that located types are invariant of the type expansion.

Lemma 9 (Weak substitution property) *Let a type path $p.t$ be a simple located type w.r.t. a program environment Δ, and θ be in located form w.r.t. Δ, and $MVars(p) \subseteq dom(\theta)$. Then either of the following two conditions holds.*

1. *$\theta(p.t)$ is a simple located type.*

2. *p is a module variable.*

Proof. By definition of simple located types. Use Lemma 5 to prove that $\theta(p)$ is in located form *w.r.t.* Δ. □

Lemma 10 *Let τ be a located type w.r.t. a program environment Δ, then $\Delta; \Omega \vdash \tau \downarrow \tau$ for any Ω.*

Proof. By induction on the structure of τ. We show the main case where $\tau = p.t$. By definition of simple located types, p is in located form *w.r.t.* Δ. By Lemma 8, $\Delta \vdash p \rightsquigarrow p$. The only applicable rule is either **[tnlz-dtyp]** or **[tnlz-atyp]**, hence we have the claim. □

Now let us study a simple case. Suppose that every type abbreviation and manifest type specification appearing in a program environment Δ abbreviates a simple located type. That is, suppose that, for all $\texttt{type}\ t = \tau$ appearing in Δ, τ is a simple located type *w.r.t.* Δ. To reduce a type path $p.t$, the expansion first reduces p. Let us assume that the module path expansion successfully reduces p into p' where p' is not a module variable and that $\Delta \vdash p' \mapsto (\theta, \texttt{ss}\ \ldots \texttt{type}\ t = \tau \ldots\ \texttt{end}^i)$ holds. Since τ is a simple located type, $\Delta; \Omega \vdash \tau \downarrow \tau$ holds immediately (Lemma 10). Hence, by Lemma 9, $\theta(\tau)$ is either a simple located type or $\tau = X.t_1$ for some module variable X and a type name t_1. When $\theta(\tau)$ is a simple located type, the expansion terminates successfully returning $\theta_1(\tau)$ as output. Otherwise, the expansion continues reducing $\theta(X).t_1$. Since $\theta(X)$ is in located form (Lemma 3) and located forms are invariant of the module path expansion (Lemma 8), we have $\Delta \vdash \theta(X) \rightsquigarrow \theta(X)$. Thus the only possible case where the expansion further continues is where $\Delta \vdash \theta(X) \mapsto (\theta_2, \texttt{ss}\ \ldots\ \texttt{type}\ t_1 = \tau_1 \ldots\ \texttt{end}^j)$ holds. Again, by Lemma 9, $\theta_2(\tau_1)$ is either a simple located type or else $\tau_1 = X_2.t_2$ for some X_2 and t_2. Here one should notice that $\theta_2(X_2)$ is structurally smaller than $\theta(X)$, since $\theta_2(X_2)$ literally appears inside $\theta(X)$. Since $\theta(X)$ is structurally finite, the expansion eventually terminates.

In general, type abbreviations may contain more complex types than simple located types and so may manifest type specifications. Yet, if the expansion knows all the type abbreviations and manifest type specifications that are looked up during the expansion of a type τ and if it has expanded these types in advance, it can reduce τ in a similar way to the above simple

case we examined. In other words, the expansion reduces types in an appropriate order so that a type τ is expanded only after all those types that are looked up during the expansion of τ have been expanded. The expansion simultaneously searches such an order and reduces types along the order. It uses locks Ω to ensure that the order does not contain cycles.

The following two examples are good exercises to understand how the type expansion works in more complex cases.

Example 1 *Consider a functor definition:*

```
module F =
 (functor(X : sig type t end[2]) → struct type t = F(F(X)).t end[3])[1]
```

The type `t` *in the body of the functor* `F` *defines a cyclic abbreviation. The type expansion raises an error for input* `F(F(X)).t`*, when attempting to lock* `(3,t)` *under the lock* {`(3,t)`} *during the reduction. If the expansion traced the abbreviation in the intuitive way, it would yield the following infinite sequence:*

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

Observe that this sequence is not merely cyclic, but produces types of arbitrary long length.

Example 2 *Consider the following program:*

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

The type `N.t` *has a valid reference, and the type expansion successfully reduces the type* `F(F(M)).t` *into* `int * int * int`*.*

Here are two important observations on this example.

1. *The expansion reduces* `L.t * int` *into* `X.t * int` *before reducing* `F(F(M)).t`*, since the expansion of* `F(F(M)).t` *looks up the type* `t` *defined in* `F`*'s body.*

2. *If we restricted the expansion from tracing the same abbreviation twice during the reduction instead of having the rule* [**tnlz-abb**]*, then the expansion could not reduce* `F(F(M)).t`*, since the abbreviation* `type t = L.t * int` *in* `F`*'s body is looked up twice.*

4.2 Well-definedness and termination

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

Proposition 6 (Well-definedness of the type expansion) *For any program environment Δ, lock Ω and types τ, τ', if $\Delta; \Omega \vdash \tau \downarrow \tau'$, then τ' is a located type w.r.t. Δ.*

Proof. By induction on the derivation of $\Delta; \Omega \vdash \tau \downarrow \tau'$ and by case on the last rule used. □

Proposition 7 (Termination of the type expansion) *For any program environment Δ, lock Ω and type τ, proof search for $\Delta; \Omega \vdash \tau \downarrow \_$ will terminate.*

Proof. Below, we define a well-founded relation $>_{t_\Delta}$ on pairs (τ, Ω) of a type τ and a lock Ω *w.r.t.* Δ. Using Lemma 9 and Proposition 6, it can be easily checked that if there is an infinitely deep derivation tree of the type expansion, then one can construct an infinitely descending sequence in $>_{t_\Delta}$ from the tree. This contradicts well-foundedness of $>_{t_\Delta}$. By Köning's lemma on finitely branching trees, we obtain the proposition.

$(\tau_1, \Omega_1) >_{t_\Delta} (\tau_2, \Omega_2)$ holds if and only if either of the following four conditions holds. We write $\mathit{IntLabs}_\Delta$ and Tnames_Δ to denote the set of integer labels and type names appearing in Δ, respectively.

1. $\Omega_1 = \Omega_2$ and $\tau_1 = \tau_{11} * \tau_{12}$ and either $\tau_2 = \tau_{11}$ or $\tau_2 = \tau_{12}$.
2. $\Omega_1 = \Omega_2$ and $\tau_1 = \tau_{11} \to \tau_{12}$ and either $\tau_2 = \tau_{11}$ or $\tau_2 = \tau_{12}$.
3. All the following three conditions hold.
   - $\Omega_1 = \Omega_2$.
   - $\tau_1 = p.t$ and $\Delta \vdash p \rightsquigarrow p_1$ and $\Delta \vdash p_1 \mapsto (\theta, \texttt{ss} \ \ldots \texttt{type}\ t = \tau' \ldots\ \texttt{end}^i)$.
   - For all τ in $\mathit{typaths}(\tau_2)$, either τ is a simple located type *w.r.t.* Δ or else $\tau = \theta(X).t_1$ for some module variable X in $\mathit{dom}(\theta)$ and some type name t_1.

4. (i,t) is not in Ω_1 and $\Omega_2 = \Omega_1 \cup \{(i,t)\} \subseteq \{(i,t) \mid i \in \mathit{IntLabs}_\Delta,\ t \in \mathit{Tnames}_\Delta\}$.

To prove well-foundedness of $>_{t_\Delta}$, we define a well-founded relation $>_{\tau_\Delta}$ on types *w.r.t.* Δ. Then we show that well-foundedness of $>_{\tau_\Delta}$ implies that of $>_{t_\Delta}$.

$\tau_1 >_{\tau_\Delta} \tau_2$ holds if and only if either of the following three conditions holds.

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

Note the slight but crucial difference between the second condition of the rule 3. of $>_{t_\Delta}$ and the first condition of the rule 3. of $>_{\tau_\Delta}$. In the latter, we do not expand p.

First we show well-foundedness of $>_{\tau_\Delta}$. Suppose that there is an infinitely descending sequence $\{\tau_i\}_{i=1}^\infty$ in $>_{\tau_\Delta}$. Such sequence can only be constructed using the rule 3. of $>_{\tau_\Delta}$ infinitely often. Hence there is an infinite sequence $\{p_i.t_i\}_{i=1}^\infty$ such that, for all i in $1, 2, \ldots$, p_{i+1} is in $\mathit{args}(p_i)$. Since the length of p_1 is finite, this is a contradiction. (Note that if a type path $p.t$ is a simple located type, then $\Delta \vdash p \mapsto (\theta, \mathtt{ss}\ \ \ldots \mathtt{type}\ t = \tau' \ldots\ \mathtt{end}^i)$ cannot hold.)

Now we show well-foundedness of $>_{t_\Delta}$. Suppose that there is an infinitely descending sequence in $>_{t_\Delta}$. Since $\{(i,t) \mid i \in \mathit{IntLabs}_\Delta,\ t \in \mathit{Tnames}_\Delta\}$ is finite, there is a lock Ω_0 such that there is an infinitely descending sequence $\{(\tau_i, \Omega_0)\}_{i=1}^\infty$ in $>_{t_\Delta}$. Let j be an integer such that $(\tau_j, \Omega_0) >_{t_\Delta} (\tau_{j+1}, \Omega_0)$ holds due to the rule 3. of $>_{t_\Delta}$. (It is easy to check that such j exists.) Let $\tau_j = p.t$. We have $\Delta \vdash p \rightsquigarrow p_1$ and $\Delta \vdash p_1 \mapsto (\theta, \mathtt{ss}\ \ldots \mathtt{type}\ t = \tau' \ldots\ \mathtt{end}^{i_1})$. By Proposition 5, p_1 is in located form *w.r.t.* Δ. By Lemma 3, for all X in $\mathit{dom}(\theta)$, $\theta(X)$ is also in located form *w.r.t.* Δ. Since located forms are invariant of the module path expansion (Lemma 8), it holds that, for all $k > j$, if $(\tau_k, \Omega_0) >_{t_\Delta} (\tau_{k+1}, \Omega_0)$ holds due to the rule 3. of $>_{t_\Delta}$ and $\tau_k = p'.t'$ for some p' and t', then $\Delta \vdash p' \rightsquigarrow p'$. Thus, $\{\tau_i\}_{i=j+1}^\infty$ is a descending sequence in $>_{\tau_\Delta}$. This contradicts well-foundedness of $>_{\tau_\Delta}$. □

5 Typing

In this section, we define a type system for *Marguerite*. Having defined expansion algorithms, the remaining part of the type system is straightforward.

5.1 Type equality

We define a type equivalence judgment in Figure 16, with an auxiliary judgment in Figure 17. The judgment $\Delta \vdash \tau_1 \equiv \tau_2$ states that two the types τ_1 and τ_2 are equivalent *w.r.t.* the program environment Δ. The type system checks equivalence between two arbitrary types by reducing them into located ones. Figure 17 defines a type equivalence judgment on located types. All rules are syntax directed and straightforward.

It would be easy to observe that the type equivalence judgment defines an equivalence relation. Recall that the type expansion is deterministic, that is, if $\Delta; \Omega \vdash \tau \downarrow \tau'$ and $\Delta; \Omega \vdash \tau \downarrow \tau''$ then $\tau' = \tau''$.

Decidability of the type equivalence judgment follows from termination of the type expansion.

Lemma 11 *For any program environment $\Delta$and types τ, τ', it is decidable whether $\Delta \vdash \tau \equiv \tau'$ holds or not.*

5.2 Core type reconstruction

The core type reconstruction algorithm infers types of expressions, but does not assure that the inferred types are correct. For instance, to reconstruct a type of an application $e_1(e_2)$ ([**rcnstr-app**] in Figure 18), it only reconstructs a type of e_1, which must be an arrow type $\tau' \to \tau$, then returns the result type τ. We defer ensuring that e_2 does have a type equivalent to τ' to a *well-typedness judgment* of the form $\Delta; \Gamma \vdash e : \tau$, which is defined later in Figure 20.

We define the core type reconstruction in Figure 18, with an auxiliary judgment in Figure 19. The judgment $\Delta; \Gamma; \Psi \vdash e :: \tau$ means that the reconstruction infers the type τ for the expression e under the type environment Γ with Ψ locked *w.r.t.* the program environment Δ. We use Ψ as a metavariable for pairs (i, l) of an integer i and a value name l and Γ for type environments, which assign located types to variables. For a type environment Γ, $dom(\Gamma)$ denotes the domain of Γ.

$$\frac{\Delta;\emptyset \vdash \tau_1 \downarrow \tau_1' \quad \Delta;\emptyset \vdash \tau_2 \downarrow \tau_2' \quad \vdash \tau_1' \equiv_\tau \tau_2'}{\Delta \vdash \tau_1 \equiv \tau_2}$$

Figure 16: Type equivalence

$$\frac{}{\vdash \mathtt{1} \equiv_\tau \mathtt{1}} \qquad \frac{\vdash \tau_1 \equiv_\tau \tau_1' \quad \vdash \tau_2 \equiv_\tau \tau_2'}{\vdash \tau_1 \to \tau_2 \equiv_\tau \tau_1' \to \tau_2'}$$

$$\frac{\vdash \tau_1 \equiv_\tau \tau_1' \quad \vdash \tau_2 \equiv_\tau \tau_2'}{\vdash \tau_1 * \tau_2 \equiv_\tau \tau_1' * \tau_2'} \qquad \frac{}{\vdash p.t \equiv_\tau p.t}$$

Figure 17: Type equivalence on located types

Observe that for any program environment Δ, type environment Γ, lock Ψ and expression e, proof search for $\Delta;\Gamma;\Psi \vdash e :: \_$ is deterministic. We regard inference rules of the reconstruction as defining an algorithm which takes Δ, Ψ, Γ and e as input then either returns τ as output when the search succeeds in building a derivation tree for $\Delta;\Gamma;\Psi \vdash e :: \tau$ or raises an error when the search fails. We prove termination of the proof search later in Proposition 8.

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

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

The rules in Figure 18 are mostly straightforward. Here, we focus on the rules [**rcnstr-vpth1**] and [**rcnstr-vpth2**] for reconstructing a type of a value path $p.l$. Firstly, the reconstruction determines the module that p refers to by expanding p into a located form. When either the module path expansion fails or the located form p' of p does not resolve to a structure (type) containing a value component named l, the reconstruction fails. Otherwise there are two possibilities: 1) When p' resolves to a structure type which contains a value specification `val` $l : \tau'$ with θ being the module variable binding ([**rcnstr-vpth1**]), then the reconstruction returns the located type of $\theta(\tau')$. 2) When p' resolves to a structure which contains a value definition `val` $l = e$ with θ being the module variable binding ([**rcnstr-vpth2**]), then the reconstruction returns the located type of $\theta(\tau')$, where τ' is the inferred type of e. This rule corresponds to the rule [**tnlz-abb**] of the type expansion.

[rcnstr-var]

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

[rcnstr-uni]

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

[rcnstr-prd]

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

[rcnstr-prj]

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

[rcnstr-fun]

$$\frac{\Delta;\emptyset \vdash \tau' \downarrow \tau}{\Delta;\Gamma;\Psi \vdash (\lambda x.e : \tau') :: \tau}$$

[rcnstr-app]

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

[rcnstr-cnstr]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash \mathit{cnstrlkup}(p', c) = (t, \tau)}{\Delta;\Gamma;\Psi \vdash p.c\ e :: p'.t}$$

[rcnstr-case]

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

[rcnstr-vpth1]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p' \mapsto (\theta, \mathtt{sig}\ \ldots\ \mathtt{val}\ l : \tau' \ldots \mathtt{end}^i) \quad \Delta;\emptyset \vdash \theta(\tau') \downarrow \tau}{\Delta;\Gamma;\Psi \vdash p.l :: \tau}$$

[rcnstr-vpth2]

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

Figure 18: Type reconstruction

$\Delta \vdash \mathit{cnstrlkup}(p, c) = (t, \tau)$ when
$\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau'\ \ldots \mathtt{end}^i)$ and $\Delta;\emptyset \vdash \theta(\tau') \downarrow \tau$

Figure 19: Datatype look-up

When inferring a type of e, the reconstruction augments the lock Ψ with an new entry $\{(i, l)\}$ to avoid divergence.

Observe that the third premise of the rule **[rcnstr-vpth2]** has an empty type environment. Hence the reconstruction always infers the same type for the same value path under whatever type environment, unless it raises an error.

Proposition 8 (Termination of the core type reconstruction) *For any program environment Δ, type environment Γ, lock Ψ and expression e, proof search for $\Delta; \Gamma; \Psi \vdash e :: \_$ will terminate.*

Proof. Below we define a well-founded relation $>_{v_\Delta}$ on pairs (e, Ψ) of an expression e and a lock Ψ *w.r.t.* Δ. It can be easily checked that if there is an infinitely deep derivation tree of the core type reconstruction, then one can construct an infinitely descending sequence in $>_{v_\Delta}$ from that tree. This contradicts well-foundedness of $>_{v_\Delta}$. By Köning's lemma on finitely branching trees, we obtain the claim.

We write $\mathit{IntLabs}_\Delta$ and Vnames_Δ to denote the set of integer labels and value names appearing in Δ, respectively.

$(e_1, \Psi_1) >_{v_\Delta} (e_2, \Psi_2)$ holds if and only if either of the following two conditions holds.

1. e_2 is structurally smaller than e_1 and $\Psi_1 = \Psi_2$.
2. $(i, l) \notin \Psi_1$ and $\Psi_2 = \Psi_1 \cup \{(i, l)\} \subseteq \{(i, l) \mid i \in \mathit{IntLabs}_\Delta,\ l \in \mathit{Vnames}_\Delta\}$.

The well-foundedness of $>_{v_\Delta}$ follows from the finiteness of $\{(i, l) \mid i \in \mathit{IntLabs}_\Delta,\ l \in \mathit{Vnames}_\Delta\}$. □

5.3 Typing rules

Finally we present well-typedness judgments for the module language and for the core language in Figure 20 and 21, respectively. Auxiliary judgments are found in Figure 22 and 23.

The judgments $\Delta \vdash E \diamond$ and $\Delta \vdash S \diamond$ mean that the module expression E and the signature S are well-typed *w.r.t.* the program environment Δ, respectively. The judgment $\Delta; \Gamma \vdash e : \tau$ means that the core expression e has the type τ under the type environment Γ *w.r.t.* Δ. The other judgments are read similarly.

Module expression & Signature

$$\frac{\Delta \vdash E_d \diamond}{\Delta \vdash E_d^i \diamond} \qquad \frac{\Delta \vdash S_d \diamond}{\Delta \vdash S_d^i \diamond}$$

Module expression bodies

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

Signature body

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

Definitions & Specifications

$$\frac{\Delta \vdash E \diamond}{\Delta \vdash \texttt{module}\ M = E \diamond} \qquad \frac{\Delta \vdash \tau \diamond}{\Delta \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau \diamond}$$

$$\frac{\Delta \vdash \tau \diamond}{\Delta \vdash \texttt{type}\ t = \tau \diamond} \qquad \frac{}{\Delta \vdash \texttt{type}\ t \diamond} \qquad \frac{\Delta; \emptyset \vdash e : \tau}{\Delta \vdash \texttt{val}\ l = e \diamond} \qquad \frac{\Delta \vdash \tau \diamond}{\Delta \vdash \texttt{val}\ l : \tau \diamond}$$

Figure 20: Typing rules

The purpose of well-typedness judgments is to ensure well-formedness of module paths (explained later) and correctness of the core type reconstruction. As we explained earlier, we do not require the reconstruction to be correct. Instead, the type system checks its correctness here.

All typing rules in Figure 20 and for core types in Figure 21 are straightforward. They traverse the constituents of given module expressions, signatures and others. When typing a functor, we do not extend the program environment Δ with a new binding $[X \mapsto S]$, assuming that Δ already contains that binding. Typing rules for expressions are analogous to those found in [51], except for the last rule. To check well-typedness of a value path, the type system consults the core type reconstruction, which is responsible for resolving $p.l$'s reference and inferring its type.

In Figure 22, we define a *well-formedness judgment* of module paths. The judgment $\Delta \vdash p$ wf means that the module path p is well-formed *w.r.t.* the program environment Δ. It ensures 1) that p does not contain dangling or cyclic references by checking expandability of p and 2) that functor applications contained in p are type-correct in the sense that a functor argument implements the signature of the functor's formal parameter.

Core types

$$\frac{}{\Delta \vdash \mathtt{1} \diamond} \qquad \frac{\Delta \vdash \tau_1 \diamond \quad \Delta \vdash \tau_2 \diamond}{\Delta \vdash \tau_1 \to \tau_2 \diamond} \qquad \frac{\Delta \vdash \tau_1 \diamond \quad \Delta \vdash \tau_2 \diamond}{\Delta \vdash \tau_1 * \tau_2 \diamond}$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta; \emptyset \vdash p.t \downarrow \tau}{\Delta \vdash p.t \diamond}$$

Core expressions

$$\frac{}{\Delta; \Gamma \vdash () : \mathtt{1}} \qquad \frac{x \in dom(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)}$$

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

$$\frac{\Delta \vdash \tau \diamond \quad \Delta; \emptyset \vdash \tau \downarrow \tau_1 \to \tau_2 \quad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_3 \quad \Delta \vdash \tau_2 \equiv \tau_3}{\Delta; \Gamma \vdash (\lambda x.e : \tau) : \tau_1 \to \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau \quad \Delta; \Gamma \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2 \equiv \tau_1}{\Delta; \Gamma \vdash e_1\ (e_2) : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash cnstrlkup(p', c) = (t, \tau_1) \\ \Delta; \Gamma \vdash e : \tau_2 \quad \Delta \vdash \tau_1 \equiv \tau_2\end{array}}{\Delta; \Gamma \vdash p.c\ e : p'.t}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow p' \\ \Delta \vdash cnstrlkup(p', c) = (t, \tau_2) \quad \Delta \vdash \tau_1 \equiv p'.t \quad \Delta; \Gamma, x : \tau_2 \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ p.c\ x \Rightarrow e_2 : \tau}$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta; \emptyset; \emptyset \vdash p.l :: \tau}{\Delta; \Gamma \vdash p.l : \tau}$$

Figure 21: Typing for the core language

$$\frac{X \in dom(\Delta)}{\Delta \vdash X \text{ wf}} \qquad \frac{Z \in dom(\Delta)}{\Delta \vdash Z \text{ wf}} \qquad \frac{\Delta \vdash p \text{ wf} \quad \Delta \vdash p.M \rightsquigarrow q}{\Delta \vdash p.M \text{ wf}}$$

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

Figure 22: Well-formed module paths

$$\frac{\Delta;\emptyset \vdash p.t \downarrow \tau}{\Delta \vdash p \triangleright \texttt{type}\ t} \qquad \frac{\Delta \vdash p.t \equiv \tau}{\Delta \vdash p \triangleright \texttt{type}\ t = \tau} \qquad \frac{\Delta;\emptyset;\emptyset \vdash p.l :: \tau' \quad \Delta \vdash \tau \equiv \tau'}{\Delta \vdash p \triangleright \texttt{val}\ l : \tau}$$

$$\frac{\Delta \vdash \mathit{cnstrlkup}(p, c) = (t, \tau') \quad \Delta \vdash \tau \equiv \tau'}{\Delta \vdash p \triangleright \texttt{datatype}\ t = c\ \texttt{of}\ \tau}$$

Figure 23: Realization

The type system checks type-correctness of functor applications by means of the *realization judgment* defined in Figure 23. The judgment $\Delta \vdash p \triangleright B$ means that the module path p resolves to a module which contains a component satisfying the specification B.

Let us examine each rule. For a module path p to satisfy an abstract type specification $\texttt{type}\ t$, p must resolve to a structure (type) which contains a type component named t. This is ensured by checking expandability of the type $p.t$. For p to satisfy a manifest type specification $\texttt{type}\ t = \tau$, p must resolve to a structure (type) whose type component t is equivalent to τ. This means that two types $p.t$ and τ are equivalent. For p to satisfy a value specification $\texttt{val}\ l : \tau$, p must resolve to either a structure containing a value component named l of type τ' or a structure type containing a value specification for l with type τ', where τ' is equivalent to τ. Observe that the rule consults core type reconstruction, instead of core typing (i.e., the first premise is $\Delta;\emptyset;\emptyset \vdash p.l :: \tau'$, not $\Delta;\emptyset \vdash p.l : \tau'$.). We do not require $p.l$ to be well-typed at this stage, avoiding a circular typing strategy. For p to satisfy a datatype specification $\texttt{datatype}\ t = c\ \texttt{of}\ \tau$, p must resolve to a structure (type) containing an equivalent datatype definition or specification, which has the same named constructor c whose argument type is equivalent to τ.

Definition 7 *A program P is well-typed if $\Delta_P \vdash P \diamond$ holds.*

Decidability of the type system is an immediate consequence of termination of the module path expansion, the type expansion and the core type reconstruction.

Proposition 9 (Decidability of the type system) *For any program P, it is decidable whether P is well-typed or not.*

Proof. Decidability of the realization judgment follows from termination of the type expansion (Proposition 7) and of the core type reconstruction

(Proposition 8) and decidability of the type equivalence judgment (Lemma 11). This and termination of the module path expansion (Proposition 4) result in decidability of the well-formedness judgment of module paths. Then the claim can be proven by induction on the structure of P, again using the same lemma and propositions. □

6 Soundness

In this section, we define a call-by-value operational semantics as small step reductions of core expressions and prove a soundness result with respect to the reductions.

We first define the intuitive expansion of module paths, named *normalization*, in Figure 24. We use normalization to resolve path references in the reductions. The judgment $\Delta \vdash p \rightsquigarrow_n q$ means that the normalization reduces the module path p into the module path q *w.r.t.* the program environment Δ. Normalization expands module paths by tracing module abbreviations in the intuitive way. Hence it may not be terminating. We prove in Proposition 11 that the module path expansion and the normalization coincide for well-typed programs. The proposition implies that normalization terminates for well-typed programs.

Values v and evaluation contexts L are:

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

where p does not contain module variables.

A small step reduction is defined with respect to a program environment Δ, which is either:

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

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

$$\Delta \vdash p.l \overset{\texttt{vpth}}{\to} \theta(e) \quad \begin{array}{l} \text{when } \Delta \vdash p \rightsquigarrow_n q \\ \text{and } \Delta \vdash q \mapsto (\theta, \texttt{struct} \ldots\ \texttt{val}\ l = e \ldots\ \texttt{end}^i) \end{array}$$

or an inner reduction obtained by induction:

$$\frac{\Delta \vdash e_1 \to e_2 \quad L \neq \{\}}{\Delta \vdash L\{e_1\} \to L\{e_2\}}$$

where write $\Delta \vdash e \to e'$ when e reduces into e' with one of the above three reductions.

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

When deconstructing a value through the case expression $\texttt{case}\ p.c\ v\ \texttt{of}$

$$\frac{}{\Delta \vdash X \rightsquigarrow_n X} \qquad \frac{}{\Delta \vdash Z \rightsquigarrow_n Z}$$

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

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

$$\frac{\Delta \vdash p_1 \rightsquigarrow_n p_1' \quad \Delta \vdash p_2 \rightsquigarrow_n p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, K_d^i) \quad K_d \neq q}{\Delta \vdash p_1(p_2) \rightsquigarrow_n p_1'(p_2')}$$

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

Figure 24: Normalization of module paths

$q.c\ x \Rightarrow e$, we do not explicitly check that p and q resolve to the same module. The type system already ensures that they expand into the same module path.

Proposition 10 (Soundness) *Let a program P be well-typed, and an expression e contain no module variables. When $\Delta_P; \emptyset \vdash e : \tau$, we have the following two results.*

1. *If $\Delta_P \vdash e \to e'$, then $\Delta_P; \emptyset \vdash e' : \tau'$ with $\Delta_P \vdash \tau \equiv \tau'$.*
2. *Either e is a value or else there is some e' with $\Delta_P \vdash e \to e'$.*

6.1 Proof of the soundness

The soundness result can be proven in a standard way for the most part. The only difficulty in the proof is about the reduction rule $\overset{\texttt{vpth}}{\to}$. Below we prove progress and subject reduction properties for this rule in Proposition 12 and 14, respectively.

We have already shown decidability of the type system in Proposition 9. Locks Σ, Ω and Ψ are useful only for the decidability result. For soundness, we are interested in derivation trees which prove well-typedness of programs, but not in how we can construct the trees. Hence, in the proof below, we use judgments of the ground expansion, the type expansion and the core type reconstruction that do not hold locks. For instance, we may say that

[ugnlz-mv]

$$\frac{}{\Delta \vdash X \rightsquigarrow_{ug} X}$$

[ugnlz-sf]

$$\frac{}{\Delta \vdash Z \rightsquigarrow_{ug} Z}$$

[ugnlz-def1]

$$\frac{\Delta \vdash p \rightsquigarrow_{ug} p' \quad \Delta \vdash p'.M \mapsto (\theta, K_d^i) \quad K_d \notin mid}{\Delta \vdash p.M \rightsquigarrow_{ug} p'.M}$$

[ugnlz-pth1]

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

[ugnlz-def2]

$$\frac{\Delta \vdash p_1 \rightsquigarrow_{ug} p_1' \quad \Delta \vdash p_2 \rightsquigarrow_{ug} p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, K_d^i) \quad K_d \notin mid}{\Delta \vdash p_1(p_2) \rightsquigarrow_{ug} p_1'(p_2')}$$

[ugnlz-pth2]

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

Figure 25: Unsafe ground-normalization

$\Delta \vdash p \rightsquigarrow_g q$ holds, when $\Delta, \emptyset \vdash p \rightsquigarrow_g q$ can be proven by the inference rules that are same as the rules for the ground expansion (Figure 13) but that do not use locks. (It is clear that whether or not the inference rules use locks does not affect output of the ground expansion. The ground expansion without locks may diverge and the ground expansion with locks may raise more errors than without.)

We first define a sanity condition on program variable environments.

Definition 8 *A program environment Δ is well-formed if both the following conditions hold.*

1. *for all X in $dom(\Delta)$, $\Delta \vdash \Delta(X) \diamond$*
2. *for all Z in $dom(\Delta)$, $\Delta \vdash \Delta(Z) \diamond$*

Note that if a program P is well-typed then so is the program environment of P.

We first show in Proposition 11 that the module path expansion coincides with the normalization for well-typed module paths. The proof proceeds in two steps: 1) we prove in Lemma 19 that the ground expansion coincides

with the unsafe ground expansion defined in Figure 25; then 2) we prove in Lemma 24 that the composition of the unsafe one and the variable normalization coincides with the normalization. For the unsafe ground expansion, we use judgments of the form $\Delta \vdash p \rightsquigarrow_{ug} q$. In rules [**ugnlz-pth1**] and [**ugnlz-pth2**], the unsafe one applies θ to q before expanding q, whereas the original one applies θ to the result of expansion of q in rules [**gnlz-pth1**] and [**gnlz-pth2**].

For a module variable binding θ, we write $MVars(\theta)$ to denote the set of module variables contained in the range of θ, or $MVars(\theta) = \bigcup_{X \in dom(\theta)} MVars(\theta(X))$. For module variable environments θ_1 and θ_2, their composition $\theta_1 \circ \theta_2$ denotes a module variable environment θ_3 such that $dom(\theta_3) = dom(\theta_2)$ and, for all X in $dom(\theta_3)$, $\theta_3(X) = \theta_1(\theta_2(X))$. Then the following three lemmas can be proven by easy induction.

Lemma 12 *Let p be not a module variable and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p \mapsto (\theta_1, K)$, then $\Delta \vdash \theta(p) \mapsto (\theta \circ \theta_1, K)$ and $MVars(\theta_1) \subseteq dom(\theta)$.*

Lemma 13 *If $\Delta \vdash p \rightsquigarrow_{ug} q$ then q is in pre-located form w.r.t. Δ.*

Lemma 14 *Let p be in pre-located form w.r.t. Δ. Then $\Delta \vdash p \rightsquigarrow_{ug} p$.*

Lemma 15 *Let θ be in pre-located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p \rightsquigarrow_{ug} q$, then $\Delta \vdash \theta(p) \rightsquigarrow_{ug} \theta(q)$ and $MVars(q) \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta \vdash p \rightsquigarrow_{ug} q$ and by case on the last rule used. Use above three lemmas. □

Lemma 16 *Let θ be in pre-located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p \rightsquigarrow_{g} q$, then $\Delta \vdash \theta(p) \rightsquigarrow_{g} \theta(q)$ and $MVars(q) \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta \vdash p \rightsquigarrow_{g} q$ and by case on the last rule used. Use Lemma 1 and 6. □

Corollary 1 *Let θ be in located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p \rightsquigarrow_{g} q$, then $\Delta \vdash \theta(p) \rightsquigarrow_{g} \theta(q)$ and $MVars(q) \subseteq dom(\theta)$.*

Lemma 17 *Let θ and p be in pre-located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$, and θ' be such that $dom(\theta) = dom(\theta')$ and, for all X in $dom(\theta')$, $\Delta \vdash varnlz(\theta(X)) = \theta'(X)$. If $\Delta \vdash varnlz(p) = q$, then $\Delta \vdash varnlz(\theta(p)) = \theta'(q)$ and $MVars(q) \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta \vdash varnlz(p) = q$ and by case on the last rule used. □

Lemma 18 *Let θ be in pre-located form w.r.t. Δ, and θ' be such that $dom(\theta) = dom(\theta')$ and, for all X in $dom(\theta')$, $\Delta \vdash varnlz(\theta(X)) = \theta'(X)$. If $\Delta \vdash p \rightsquigarrow q$ and $MVars(p) \subseteq dom(\theta)$, then $\Delta \vdash \theta(p) \rightsquigarrow \theta'(q)$ and $MVars(q) \subseteq dom(\theta)$.*

Proof. By Lemma 16 and 17. □

Lemma 19 *If $\Delta \vdash p \rightsquigarrow_g q$, then $\Delta \vdash p \rightsquigarrow_{ug} q$.*

Proof. By induction on the derivation of $\Delta \vdash p \rightsquigarrow_g q$ and by case on the last rule used. We show the main case.
[**gnlz-pth1**] Suppose $p = p_1.M$ and $\Delta \vdash p_1 \rightsquigarrow_g p_1'$ and $\Delta \vdash p_1'.M \mapsto (\theta, r^i)$ and $r \neq X$ and $\Delta \vdash r \rightsquigarrow_g q_1$ and $q = \theta(q_1)$. By induction hypothesis, $\Delta \vdash p_1 \rightsquigarrow_{ug} p_1'$ and $\Delta \vdash r \rightsquigarrow_{ug} q_1$. By Proposition 1 and Lemma 2, θ is in pre-located form *w.r.t.* Δ. Since Δ does not contain free module variables, $MVars(r) \subseteq dom(\theta)$. By Lemma 15, $\Delta \vdash \theta(r) \rightsquigarrow_{ug} \theta(q_1)$. □

The two lemmas below are proven by easy induction.

Lemma 20 *If $\Delta \vdash p \rightsquigarrow_n q$ then q is in located form w.r.t. Δ.*

Lemma 21 *Let p be in located form w.r.t. Δ. Then $\Delta \vdash p \rightsquigarrow_n p$.*

Lemma 22 *Let p be in pre-located form w.r.t. Δ. If $\Delta \vdash varnlz(p) = q$, then $\Delta \vdash p \rightsquigarrow_n q$.*

Proof. By induction on the structure of p. Use Lemma 20 and 21. □

Lemma 23 *Let θ be in pre-located form w.r.t. Δ and θ' be such that $dom(\theta) = dom(\theta')$ and, for all X in $dom(\theta)$, $\Delta \vdash varnlz(\theta(X)) = \theta'(X)$. If $\Delta \vdash \theta(p) \rightsquigarrow_n q$ and $MVars(p) \subseteq dom(\theta)$, then $\Delta \vdash \theta'(p) \rightsquigarrow_n q$.*

Proof. By induction on the structure of p. For the case where p is a module variable, use Proposition 3, and Lemma 21 and 22. □

Lemma 24 *If $\Delta \vdash p \rightsquigarrow_{ug} q$ and $\Delta \vdash varnlz(q) = r$, then $\Delta \vdash p \rightsquigarrow_n r$.*

Proof. By induction on the derivation of $\Delta \vdash p \rightsquigarrow_{ug} q$ and by case on the last rule used. We show the main case.
[**ugnlz-pth1**] Suppose $p = p_1.M$ and $\Delta \vdash p_1 \rightsquigarrow_{ug} p_1'$ and $\Delta \vdash p_1'.M \mapsto (\theta, r^i)$ and $r \neq X$ and $\Delta \vdash \theta(r) \rightsquigarrow_{ug} q$. By Proposition 1 and 3, $\Delta \vdash \mathit{varnlz}(p_1') = p_1''$ and $\Delta \vdash \mathit{varnlz}(q) = q'$ for some p_1'' and q'. By induction hypothesis, $\Delta \vdash p \rightsquigarrow_n p_1''$ and $\Delta \vdash \theta(r) \rightsquigarrow_n q'$. We have $\Delta \vdash p_1''.M \mapsto (\theta', r^i)$ where θ' is such that, for all X in $\mathit{dom}(\theta')$, $\Delta \vdash \mathit{varnlz}(\theta(X)) = \theta'(X)$. Since Δ does not contain free module variables, $\mathit{MVars}(r) \subseteq \mathit{dom}(\theta')$. By Lemma 23, $\Delta \vdash \theta'(r) \rightsquigarrow_n q'$. □

Lemma 25 *Let θ be in located form w.r.t. Δ. If $\Delta \vdash \mathit{varnlz}(p) = q$ and $\mathit{MVars}(p) \subseteq \mathit{dom}(\theta)$, then $\Delta \vdash \mathit{varnlz}(\theta(p)) = \theta(q)$ and $\mathit{MVars}(q) \subseteq \mathit{dom}(\theta)$.*

Proof. By induction on the derivation of $\Delta \vdash \mathit{varnlz}(p) = q$ and by case on the last rule used. For the case where p is a module variable X in $\mathit{dom}(\theta)$, use Lemma 7. □

Lemma 26 *Let θ be in located form w.r.t. Δ. If $\Delta \vdash p \rightsquigarrow q$ and $\mathit{MVars}(p) \subseteq \mathit{dom}(\theta)$, then $\Delta \vdash \theta(p) \rightsquigarrow \theta(q)$ and $\mathit{MVars}(q) \subseteq \mathit{dom}(\theta)$.*

Proof. By hypothesis, $\Delta \vdash p \rightsquigarrow_g p'$ and $\Delta \vdash \mathit{varnlz}(p') = q$. By Corollary 1, $\Delta \vdash \theta(p) \rightsquigarrow_g \theta(p')$. By Lemma 25, $\Delta \vdash \mathit{varnlz}(\theta(p')) = \theta(q)$. Thus we deduce $\Delta \vdash \theta(p) \rightsquigarrow \theta(q)$. □

Lemma 27 *If $\Delta \vdash p \diamond$, then $\Delta \vdash p \rightsquigarrow q$ for some q.*

Proof. By case on the structure of p. □

Proposition 11 *Suppose $\Delta \vdash p \diamond$, then $\Delta \vdash p \rightsquigarrow q$ if and only if $\Delta \vdash p \rightsquigarrow_n q$.*

Proof. By $\Delta \vdash p \diamond$ in the hypothesis and Lemma 27, $\Delta \vdash p \rightsquigarrow q'$ for some q'. Since derivations of the module path expansion are deterministic, $q = q'$. By definition of the module path expansion $\Delta \vdash p \rightsquigarrow_g p_1$ and $\Delta \vdash \mathit{varnlz}(p_1) = q$ for some p_1. By Lemma 19, $\Delta \vdash p \rightsquigarrow_{ug} p_1$. By Lemma 24, $\Delta \vdash p \rightsquigarrow_n q$. Since derivations of the normalization are deterministic, if $\Delta \vdash p \rightsquigarrow_n q_1$ and $\Delta \vdash p \rightsquigarrow_n q_2$ then q_1 and q_2 are identical. Thus we have the claim. □

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

Proposition 12 (Progress for the reduction $\overset{\texttt{vpth}}{\to}$) *Let a program P be well-typed. If $\Delta_P; \emptyset \vdash p.l : \tau$, then $\Delta_P \vdash p \rightsquigarrow_n q$ and $\Delta_P \vdash q \mapsto (\theta, \texttt{struct} \ \ldots \ \texttt{val} \ l = e \ldots \texttt{end}^i)$*

Proof. By $\Delta_P; \emptyset \vdash p.l : \tau$ in the hypothesis, $\Delta_P \vdash p \diamond$ and $\Delta_P \vdash p \rightsquigarrow p_1$ and $\Delta_P \vdash p_1 \mapsto (\theta', \texttt{struct} \ \ldots \ \texttt{val} \ l = e' \ \ldots \ \texttt{end}^j)$. By Proposition 11, $\Delta_P \vdash p \rightsquigarrow_n p_1$. □

Before proving a subject reduction property for the reduction $\overset{\texttt{vpth}}{\to}$, we prove in Proposition 13 that well-formedness of module paths is invariant of the module path expansion.

For module variable bindings, we define their well-formedness as follows.

Definition 9 *A module variable binding θ is well-formed w.r.t. a program environment Δ, written $\Delta \vdash \theta$ wf, if, for all X in $dom(\theta)$, the following two conditions hold.*

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

Lemma 28 *Let θ be in located form w.r.t. Δ and $MVars(\tau) \subseteq dom(\theta)$. If $\Delta \vdash \tau \downarrow \tau'$ and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(\tau) \equiv \theta(\tau')$ with $MVars(\tau') \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta \vdash \tau \downarrow \tau'$ and by case on the last rule used. We show the main case.
[tnlz-abb] Suppose $\tau = p.t$ and $\Delta \vdash p \rightsquigarrow p'$ and $\Delta \vdash p' \mapsto (\theta_1, \texttt{ss} \ \ldots \ \texttt{type} \, t = \tau_1 \ldots \texttt{end}^i)$ and $\Delta \vdash \tau_1 \downarrow \tau_1'$ and $\Delta \vdash \theta_1(\tau_1') \downarrow \tau'$. By Lemma 26, we have $\Delta \vdash \theta(p) \rightsquigarrow \theta(p')$. Now we have two cases.

- When p' is not a module variable, then $\Delta \vdash \theta(p') \mapsto (\theta \circ \theta_1, \texttt{ss} \ \ldots \ \texttt{type} \, t = \tau_1 \ldots \texttt{end}^i)$ by Lemma 12. By induction hypothesis, we have the claim.
- When $p' = X$ for some module variable X in $dom(\theta)$. Then, since θ_1 is an identity substitution, we have $\tau_1' = \tau'$ by Proposition 6 and Lemma 10. By well-formedness of θ, $\Delta \vdash \theta(X).t \equiv \theta(\tau_1)$. By induction hypothesis, we have the claim.

□

Corollary 2 *Let θ be in located form w.r.t. Δ and $MVars(\tau_1) \subseteq dom(\theta)$. If $\Delta \vdash \tau_1 \downarrow \tau_2$ and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(\tau_1) \downarrow \tau_3$ for some τ_3.*

Corollary 3 *Let θ be in located form w.r.t. Δ and $MVars(\tau) \cup MVars(\tau') \subseteq dom(\theta)$. If $\Delta \vdash \tau \equiv \tau'$ and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(\tau) \equiv \theta(\tau')$.*

We say that a type environment Γ is in located form *w.r.t.* a program environment Δ if and only if, for all x in $dom(\Gamma)$, $\Gamma(x)$ is a located type *w.r.t.* Δ.

Lemma 29 *Let Γ, Γ_1 and θ be in located form w.r.t. Δ and and $MVars(\Gamma) \cup MVars(e) \subseteq dom(\theta)$. Suppose that Γ_1 satisfies the two conditions: 1) $dom(\Gamma) = dom(\Gamma_1)$ and 2) for all x in $dom(\Gamma)$, $\Delta \vdash \theta(\Gamma(x)) \equiv \Gamma_1(x)$. If $\Delta; \Gamma \vdash e :: \tau$ and $\Delta \vdash \theta$ wf, then $\Delta; \Gamma_1 \vdash \theta(e) :: \tau_1$ with $\Delta \vdash \theta(\tau) \equiv \tau_1$ and $MVars(\tau) \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e :: \tau$ and by case on the last rule used. We show the main case.
[v-vpth1] Suppose $e = p.l$ and $\Delta \vdash p \rightsquigarrow p_1$ and
$\Delta \vdash p_1 \mapsto (\theta_1, \texttt{struct} \ \ldots \ \texttt{val} \ l = e_1 \ldots \texttt{end}^i)$ and $\Delta; \emptyset \vdash e_1 :: \tau_2$ and $\Delta \vdash \theta_1(\tau_2) \downarrow \tau$. By Lemma 26, $\Delta \vdash \theta(p) \rightsquigarrow \theta(p_1)$. By Lemma 12, $\Delta \vdash \theta(p_1) \mapsto (\theta \circ \theta_1, \texttt{struct} \ \ldots \ \texttt{val} \ l = e_1 \ldots \texttt{end}^i)$. By Lemma 28, we have $\Delta \vdash \theta \circ \theta_1(\tau_2) \equiv \theta(\tau)$, which also implies $\Delta \vdash \theta \circ \theta_1(\tau_2) \downarrow \tau_3$ for some τ_3. Thus we deduce $\Delta; \Gamma \vdash \theta(p).l :: \tau_3$. □

Lemma 30 *Let θ be in located form w.r.t. Δ and $MVars(p) \cup MVars(B) \subseteq dom(\theta)$. If $\Delta \vdash p \triangleright B$ and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(p) \triangleright \theta(B)$.*

Proof. We show the main case. Suppose $B = \texttt{val} \ l : \tau$. We have $\Delta; \emptyset \vdash p.l :: \tau_1$ and $\Delta \vdash \tau \equiv \tau_1$. By Lemma 29, $\Delta; \emptyset \vdash \theta(p).l :: \tau_2$ with $\Delta \vdash \theta(\tau_1) \equiv \tau_2$. By Lemma 3, $\Delta \vdash \theta(\tau) \equiv \theta(\tau_1)$. Since the type equivalence relation is transitive, $\Delta \vdash \tau_2 \equiv \theta(\tau)$. □

Lemma 31 *Let θ be in located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p$ wf and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(p)$ wf.*

Proof. By induction on the derivation of $\Delta \vdash p$ wf and by case on the last rule used. We show the main case.
Suppose $p = p_1(p_2)$. We have $\Delta \vdash p_1$ wf, $\Delta \vdash p_2$ wf, $\Delta \vdash p_1 \rightsquigarrow p_1'$, $\Delta \vdash p_2 \rightsquigarrow$

p'_2, $\Delta \vdash p_1(p_2) \rightsquigarrow q$, $\Delta \vdash p'_1 \mapsto (\theta_1, (\texttt{functor}(X \ : \ \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^j) \rightarrow E)^i)$ and, for all i in $1 \ldots n$, $\Delta \vdash p'_2 \triangleright \theta_1[X \mapsto p'_2](B_i)$. By induction hypothesis, $\Delta \vdash \theta(p_1)$ wf and $\Delta \vdash \theta(p_2)$ wf. By Lemma 26, $\Delta \vdash \theta(p_1) \rightsquigarrow \theta(p'_1)$, $\Delta \vdash \theta(p_2) \rightsquigarrow \theta(p'_2)$ and $\Delta \vdash \theta(p_1(p_2)) \rightsquigarrow \theta(q)$. By definition of the look-up, $\Delta \vdash \theta(p'_1) \mapsto (\theta \circ \theta_1, (\texttt{functor}(X \ : \ \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}^j) \rightarrow \ E)^i)$. By Lemma 30, for all i in $1 \ldots n$, $\Delta \vdash \theta(p'_2) \triangleright \theta \circ \theta_1[X \mapsto \theta(p'_2)](B_i)$. □

Lemma 32 *Let p be in pre-located form w.r.t. Δ. If $\Delta \vdash p$ wf and $\Delta \vdash varnlz(p) = q$ then $\Delta \vdash q$ wf.*

Proof. By induction on the derivation of $\Delta \vdash varnlz(p) = q$. □

Lemma 33 *Let θ be in pre-located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. If $\Delta \vdash p$ wf and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(p)$ wf.*

Proof. By induction on the derivation of $\Delta \vdash p$ wf and by case on the last rule used. Use Lemma 18 and 32. □

Lemma 34 *Let Δ be well-formed. If $\Delta \vdash p$ wf and $\Delta \vdash p \rightsquigarrow_{ug} q$, then $\Delta \vdash q$ wf.*

Proof. By induction on the derivation of $\Delta \vdash p \rightsquigarrow_{ug} q$ and by case on the last rule used. Use Lemma 33. □

Proposition 13 *Let Δ be well-formed. If $\Delta \vdash p$ wf and $\Delta \vdash p \rightsquigarrow q$, then $\Delta \vdash q$ wf.*

Proof. By hypothesis, we have $\Delta \vdash p \rightsquigarrow_g r$ and $\Delta \vdash varnlz(r) = q$. By Lemma 19, $\Delta \vdash p \rightsquigarrow_{ug} r$. By Lemma 34, 13 and 32, $\Delta \vdash q$ wf. □

Finally, we show a subject reduction property for the reduction $\overset{\texttt{vpth}}{\longrightarrow}$ in Proposition 14.

Lemma 35 *Let θ be in located form w.r.t. Δ and $MVars(\tau) \subseteq dom(\theta)$. If $\Delta \vdash \tau \diamond$ and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(\tau) \diamond$.*

Proof. By induction on the derivation of $\Delta \vdash \tau \diamond$ and by case on the last rule used. We show the main case.
Suppose $\tau = p.t$. Then we have $\Delta \vdash p$ wf and $\Delta \vdash p.t \downarrow \tau_1$. By Lemma 31, we have $\Delta \vdash \theta(p)$ wf. By Corollary 2, $\Delta \vdash \theta(p.t) \downarrow \tau_2$ for some τ_2. □

Lemma 36 *Let Δ be well-formed. If $\Delta \vdash \tau \diamond$ and $\Delta \vdash \tau \downarrow \tau'$, then $\Delta \vdash \tau' \diamond$.*

Proof. By induction on the derivation of $\Delta \vdash \tau \downarrow \tau'$ and by case on the last rule used. We show the main case.
[**tnlz-abb**] Suppose $\tau = p.t$ and $\Delta \vdash p \rightsquigarrow p'$ and $\Delta \vdash p' \mapsto (\theta, \texttt{ss} \ldots \texttt{type}\ t = \tau_1 \ldots \texttt{end}^i)$ and $\Delta \vdash \tau_1 \downarrow \tau_2$ and $\Delta \vdash \theta(\tau_2) \downarrow \tau'$. By Proposition 13, $\Delta \vdash p'$ wf, hence $\Delta \vdash \theta$ wf. By well-formedness ofΔ in the hypothesis, $\Delta \vdash \tau_1 \diamond$. By induction hypothesis, $\Delta \vdash \tau_2 \diamond$. By Lemma 35, $\Delta \vdash \theta(\tau_2) \diamond$, By induction hypothesis, $\Delta \vdash \tau' \diamond$. □

We say that a type environment Γ is well-formed, written $\Delta \vdash \Gamma$ wf, if and only if Γ is in located form *w.r.t.* Δ, and for all x in $dom(\Gamma)$, $\Delta \vdash \Gamma(x) \diamond$.

Lemma 37 *Let Δ and Γ be well-formed and θ and Γ_1 be in located form w.r.t. Δ and $MVars(\Gamma) \cup MVars(e) \subseteq dom(\theta)$. Suppose that Γ_1 satisfies the two conditions: 1) $dom(\Gamma) = dom(\Gamma_1)$ and 2) for all x in $dom(\Gamma)$, $\Delta \vdash \theta(\Gamma(x)) \equiv \Gamma_1(x)$. If $\Delta \vdash \theta$ wf and $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma_1 \vdash \theta(e) : \tau'$ for some τ' with $\Delta \vdash \tau' \equiv \theta(\tau)$ and $MVars(\tau) \subseteq dom(\theta)$.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash e : \tau$ and by case on the last rule used. We show the main cases.
Suppose $e = (\lambda x.e_1 : \tau_1)$ and $\Delta \vdash \tau_1 \diamond$ and $\Delta \vdash \tau_1 \downarrow \tau_2 \to \tau_3$ and $\Delta; \Gamma, x : \tau_2 \vdash e_1 : \tau_4$ and $\Delta \vdash \tau_4 \equiv \tau_3$. By Lemma 35 $\Delta \vdash \theta(\tau_1) \diamond$. By Lemma 28, $\Delta \vdash \theta(\tau_1) \downarrow \tau_5 \to \tau_6$ with $MVars(\tau_2) \cup MVars(\tau_3) \subseteq dom(\theta)$ and $\Delta \vdash \tau_5 \equiv \theta(\tau_2)$ and $\Delta \vdash \tau_6 \equiv \theta(\tau_3)$ By Lemma 36, $\Delta \vdash \tau_2 \diamond$. By induction hypothesis, $\Delta; \Gamma_1, x : \tau_5 \vdash \theta(e_1) : \tau_7$ with $\Delta \vdash \tau_7 \equiv \theta(\tau_4)$ and $MVars(\tau_4) \subseteq dom(\theta)$. By Corollary 3, $\Delta \vdash \theta(\tau_4) \equiv \theta(\tau_3)$, hence $\Delta \vdash \tau_7 \equiv \tau_6$. As a whole we have, $\Delta; \Gamma_1 \vdash \theta(\lambda x.e_1 : \tau_1) : \tau_5 \to \tau_6$ with $\Delta \vdash \theta(\tau_2 \to \tau_3) \equiv \tau_5 \to \tau_6$ and $MVars(\tau_2 \to \tau_3) \subseteq dom(\theta)$.
Suppose $e = \texttt{case}\ e_1\ \texttt{of}\ p.c\ x \Rightarrow e_2$ and $\Delta; \Gamma \vdash e_1 : \tau_1$ and $\Delta \vdash p$ wf and $\Delta \vdash p \rightsquigarrow p'$ and $\Delta \vdash cnstrlkup(p', c) = (t, \tau_2)$ and $\Delta \vdash \tau_1 \equiv p'.t$ and $\Delta; \Gamma, x : \tau_2 \vdash e_2 : \tau$. By induction hypothesis, $\Delta; \Gamma_1 \vdash \theta_1(e_1) : \tau_3$ with $\Delta \vdash \tau_3 \equiv \theta(\tau_1)$ and $MVars(\tau_1) \subseteq dom(\theta)$. By Lemma 31, $\Delta \vdash \theta(p)$ wf. By Lemma 26, $\Delta \vdash \theta(p) \rightsquigarrow \theta(p')$ with $MVars(p') \subseteq dom(\theta)$. By Lemma 13, $\Delta \vdash p'$ wf. By well-formedness of Δ and Lemma 35 and 36, $\Delta \vdash \tau_2 \diamond$. By hypothesis on θ, we have $\Delta \vdash cnstrlkup(\theta(p'), c) = (t, \tau_4)$ with $\Delta \vdash \tau_4 \equiv \theta(\tau_2)$ with $MVars(\tau_2) \subseteq dom(\theta)$. By Corollary 3 and transitivity of the type equivalence relation, $\Delta \vdash \tau_3 \equiv \theta(p').t$. By induction hypothesis, $\Delta; \Gamma_1, x : \tau_4 \vdash \theta(e_2) : \tau'$ with $\Delta \vdash \tau' \equiv \theta(\tau)$ and $MVars(\tau) \subseteq dom(\theta)$. □

Proposition 14 (Subject reduction for the reduction $\overset{\texttt{vpth}}{\rightarrow}$) *Suppose a program P is well-typed. If $\Delta_P; \emptyset \vdash p.l : \tau$ and $\Delta_P \vdash p \rightsquigarrow_n p'$ and $\Delta_P \vdash p' \mapsto (\theta, \texttt{struct } \ldots \texttt{val } l = e \ldots \texttt{end}^i)$ then $\Delta_P; \emptyset \vdash \theta(e) : \tau'$ with $\Delta_P \vdash \tau \equiv \tau'$.*

Proof. By Proposition 11, $\Delta_P \vdash p \rightsquigarrow p'$. By Proposition 13, $\Delta_P \vdash p'$ wf. By $\Delta_P; \emptyset \vdash p.l : \tau$ in the hypothesis, $\Delta_P; \emptyset \vdash p.l :: \tau$. Hence we have $\Delta_P; \emptyset \vdash e :: \tau_1$ and $\Delta_P \vdash \theta(\tau_1) \downarrow \tau$. By Lemma 37, $\Delta_P; \emptyset \vdash \theta(e) : \tau_2$ with $\Delta_P \vdash \theta(\tau_1) \equiv \tau_2$, hence $\Delta_P \vdash \tau \equiv \tau_2$. □

7 Type inference for the core language

A type inference algorithm for the core language can be defined by 1) determining an inference order using the module path expansion algorithm, then 2) running a standard core type inference algorithm, for instance one found in [36], along this order. Concretely, using the module path expansion, we build a call graph of functions (represented by a directed graph), which expresses how components in recursive modules depend on each other: the strongly connected components of the graph indicate sets of value components whose types should be inferred simultaneously, referring to each other monomorphically; by topologically sorting the connected components, we generalize types in a connected component before moving on to typing the next one. For instance in Figure 5, we build an inference order:

$$\{\texttt{Tree.labels, Forest.labels}\} \rightarrow \texttt{Tree.split} \rightarrow \texttt{Forest.incr} \rightarrow \{\texttt{Forest.sweep}\}$$

where braces specify strongly connected components. That is, `Tree.labels` and `Forest.labels` are mutually recursive, and `Forest.sweep` is a recursive function.

We must also check for well-formedness of types, as module variables should not escape their scope during unification. This can be checked after the inference in a straightforward way.

Explicit type annotations can be used to break dependencies in the call graph and to allow polymorphic recursion. Currently, we do not attempt to infer polymorphic recursion, whose complete type inference is known to be undecidable [30]. To define those functions, type annotations are required. Otherwise the inference will fail.

Part III
Recursive modules for programming

The ability to control abstraction of modules with explicit signatures is an important feature of the ML module system. A programmer can make a value component defined in a structure inaccessible to the outside by explicitly giving the structure a signature that does not mention the component. By specifying a type component of the structure as an abstract type in the signature, one can hide the underlying implementation of the type, thus can protect its invariants.

Supporting type abstraction between recursive modules gives rise to a subtle design issue. How to treat cyclic type definitions, when the cycles are hidden inside signatures? For instance, should a type system reject the program below?

```
module M1 = (struct type t = N1.t end : sig type t end)
and N1 = (struct type t = M1.t end : sig type t end)
```

If it should, then how can it detect the cycle? The type system is supposed to obey type abstraction, that is, it must not peek inside signatures so as to know underlying implementations of abstract types. Then it would be impossible to reject exactly cycles but allow all other valid cases. For instance, the type system should allow the program below, which does not contain cycles.

```
module M2 = (struct type t = N2.t end : sig type t end)
and N2 = (struct type t = int end : sig type t end)
```

Existing proposals take different stands on this issue. Russo's [56] and Dreyer's [17] type systems disallow cyclic type definitions whether or not cycles are hidden inside signatures. To prevent a programmer from defining cycles, they put restrictions on types which can be abstracted in signatures. As a result in Russo's system, a programmer cannot enforce type abstraction between recursive modules. This is not a desirable restriction. Dreyer's system is more lenient. Only types that depend on *non-stable* types cannot be abstracted. For instance in the above two programs, the types `N1.t` and `N2.t` are not stable inside `M1` and `M2`, respectively. Since the types

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

Figure 26: Tree and Forest with structural recursive types

`M1.t` and `M2.t` depend on these non-stable types, they cannot be abstracted in signatures. This means that Dreyer's system prohibits a programmer from writing neither of the above two programs, although the latter does not contain cycles. [3] This aside, Dreyer's restriction may be acceptable in practice for SML. Yet, for O'Caml, which supports structural recursive types such as polymorphic variant types and object types, his restriction seems still severe. Indeed, Dreyer's system would reject the program in Figure 26, which uses a polymorphic variant type and a list type to represent trees and forests, respectively. The type `Tree.t` depends on the type `Forest.t`, which is not stable inside `Tree`. Hence his system does not allow the type `Tree.t` to be abstracted in the signature.

O'Caml type checks all the three programs we have seen. It does not care whether or not cyclic type definitions are hidden inside signatures, as long as signatures themselves do not specify cycles. For instance, while O'Caml rejects:

```
module M3 = (struct type t = N3.t end : sig type t = N3.t end)
and N3 = (struct type t = M3.t end : sig type t = M3.t end)
```

it accepts:

```
module M4 = (struct type t = N4.t end : sig type t end)
and N4 = (struct type t = M4.t end : sig type t end)
```

In the former program, cycles in type definitions are visible since signatures specifies the cycles; in the latter, they are invisible.

[3] To be precise, it is possible to make the latter program typed in Dreyer's system by permuting the definition order of the modules `M2` and `N2`, that is, by defining `N2` first. Yet permutation does not always work. For instance, there is no way to make the following program typed in his system.

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

```
module F = functor(X : sig type t val eval : t -> int end) ->
 struct
 type t = Int of int | Pair of X.t * X.t
 val eval = λx.case x with Int y ⇒ y
   | Pair(y1, y2) ⇒ (X.eval y1) + (X.eval y2)
 end
module Eval = (F(Eval) : sig type t val eval : t -> int end)
```

Figure 27: Taking the fix-point of a functor

Now we face a design choice between

1. To disallow cyclic type definitions whether or not they are hidden inside signatures. This choice entails restrictions on non-cyclic type definitions as we have discussed above.

2. To disallow only cycles which are visible in signatures, but allow them when they are hidden inside signatures. A downside of this approach may be that a well-typed program may not type check anymore once signatures are erased. Besides, except for the experimental implementation inside O'Caml type checker, there is no formal account of this approach.

For our language, we prefer to the latter choice since we believe it is worth keeping liberal uses of polymorphic variant types and object types together with recursive modules. Our experience in programming with recursive modules in O'Caml is that recursive modules are even more useful when combined with other language constructs. Hence we do not want to restrict such possible combinations by following the former choice.

Moreover our design choice enables a new style of programming; a programmer can take the fix-point of a functor. For instance, we type check the program in Figure 27: the functor `F` defines an open recursion, where the formal argument `X` contains both type-level and value-level forwardings; then the module `Eval` closes the both level recursion simultaneously, by taking the fix-point of `F`. Except for O'Caml, no previous work by others on recursive modules have not explored this new style of programming. In Section 13, we give another example of this programming style by solving the notorious expression problem [60] in a type-safe and modular manner, in support of our design choices.

For a formal study, we design a language, named *Traviata*, in this part (Section 8 and 9). *Traviata* is an extension of *Marguerite* with signature ascription. To accommodate the extension, we divide the type system of *Traviata* into two part, namely, a reconstruction part and a type-correctness check part. In the reconstruction part, the type system infers fully manifest signatures of recursive modules (Section 10). We design an inference engine by using the expansion algorithms developed in Part I with little change. In particular, termination of the inference follows from that of the expansion algorithms. In the type-correctness check part, the type system type checks programs using the result of the reconstruction as type environment (Section 11). We prove that the type system is sound for a call-by-value operational semantics (Section 12).

8 Example

Traviata is an extension with signature ascription of *Marguerite*. We introduce this new feature using an example in Figure 28.

The toplevel structure contains two sub-modules `Tree` and `Forest`. The module `Tree` represents trees whose leaves and nodes are labeled with integers. The module `Forest` represents unordered sets of those integer trees.

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

Unlike the example of Figure 5 in Part I, we enforce type abstraction between `Tree` and `Forest` here, by sealing them with signatures individually. Each signature specifies the type component `t` as an abstract type, hence its underlying implementation is hidden to each other, that is, the type `Forest.t` is not equivalent to the type `Tree.t list` inside `Tree` and the constructors `Leaf` and `Node` are invisible inside `Forest`. One of reasons that signature ascription is useful is that a programmer can make it explicit that the outside of a signature does not depend on the inside of the signature. For instance, the function `Tree.max` does not depend on the underlying implementation of the type `Forest.t`, but only requires the module `Forest` to provide a function `max` of type `Forest.t` $\rightarrow$ `int`. Hence, it does not affect `Tree.max`'s behavior to modify the implementation of `Forest.t` to `(T.t * T.t) list`, as long as implementation of `Forest.max` is modified properly.

As seen in this example, we extend every structure and signature with an implicitly typed declaration of a self variable in *Traviata*, whereas we did only the toplevel structure in *Marguerite*. When a module is sealed with a signature, it is important that a programmer can declare a self variable inside the sealed module to refer to components which are only visible inside the module, but not outside. To enforce type abstraction properly, we require module paths to only contain bound self variables. For instance in Figure 28, the self variable `T` is bound inside `Tree` but unbound inside `Forest`. In this way, constructors `Leaf` and `Node` are only accessible inside `Tree` but not inside `Forest`, enforcing type abstraction of `Tree` towards `Forest`. As in *Marguerite*, we can keep the ML scoping rule for backward references in a practical system by providing an elaboration phase. Yet in the example we use complete paths by letting both forward and backward references go

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

Figure 28: Modules for trees and forests

through self variables for clarity.

As we mentioned in the beginning of this part, the ability to take fixpoints of functors is a useful feature of *Traviata*. This ability was not available in *Marguerite*, but is in *Traviata* thanks to signature ascription. Indeed, the module `Eval` may not be defined as:

```
module Eval = F(Eval)
```

since the module path expansion cannot safely reduce the path `F(Eval)` with this definition. By writing `Eval`'s signature explicitly as in Figure 27, a programmer can break possible cycles in type definitions that might arise from connecting the result of the instantiation of `F` to the argument.

9 Syntax

Figure 29 gives the syntax for the module language of *Traviata*. We use M as a metavariable for module names, X for names of module variables, Z for names of self variables, t for type names, l for (core) value names and c for constructor names.

For *Traviata*, we extend module expressions of *Marguerite* with a sealing construct of the form $(E : S)$, which seals the module expression E with the signature S. To seal functors and nested structures with signatures, we extend signatures with functor types and specifications with module specifications, respectively. Note that, compared to *Marguerite*, neither module expressions nor signatures are labeled with integers, which we explain later in Section 10.3.

As mentioned in the previous section, every structure and signature contains an implicitly typed declaration of a self variable. In the construct `struct` (Z) $D_1 \ldots D_n$ `end`, the self variable Z is bound in $D_1 \ldots D_n$. Similarly, in the construct `sig` (Z) $B_1 \ldots B_n$ `end`, the self variable Z is bound in $B_1 \ldots B_n$.

Figure 30 gives the syntax for module paths, which is same as *Marguerite*. In *Traviata*, a program may declare several self variables. We require module paths only to contain bound self variables. Otherwise type abstraction can be broken. Through the self variable declared in a structure, one can refer to any module named in that structure except for those hidden within sealed sub-structures.

The type system of *Traviata* uses expansion algorithms in a similar way that *Marguerite* does. For termination of the algorithms, we again put the first-order structure restriction on *Traviata* that requires functors not to take functors as arguments or to access sub-modules of arguments.

The core language of *Traviata* is same as *Marguerite*, which is repeated in Figure 31.

We assume the following five conventions: 1) a program does not contain free module variables or free self variables; 2) all binding occurrences of module or self variables use distinct names; 3) any sequence of module definitions, type abbreviations, datatype definitions, value definitions, module specifications, manifest and abstract type specifications, datatype specifications and value specifications does not contain duplicate definitions or specifications for the same name; 4) signatures for module variables are structure types that

Module expression

E	$::=$	$\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}$	*structure*
	$\mid$	$\texttt{functor}\ (X : S) \to E$	*functor*
	$\mid$	$(E : S)$	*sealing*
	$\mid$	p	*module path*

Definitions

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

Signature

S	$::=$	$\texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}$	*structure type*
	$\mid$	$\texttt{functor}(X : S_1) \to S_2$	*functor type*

Specifications

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

Program

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

Figure 29: The module language of *Traviata*

Module identifiers

mid	$::=$	$Z \mid mid.M \mid mid(p)$

Module paths

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

Figure 30: Syntax for module paths

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

Figure 31: The core language of *Traviata*

do not contain module specifications; 5) for any sealing construct $(E : S)$, neither E nor S is a functor (type). The fourth convention is consistent with the first-order structure restriction. The fifth convention does not diminish the expressive power of the language. Since functors cannot take functors as argument, direct sealing of functors has no use. Note that we still need functor types to seal structures which contain functors as sub-modules.

9.1 Elaboration

Prior to type checking, we elaborate the syntax for modules (Figure 29) and module paths (Figure 30) to Figure 32 and 33 respectively, in order to make it easier for the type system to manipulate module paths during type checking. The elaboration operation is summarized as follows:

1. To erase declarations of self variables for which self variables declared in outer structures or structure types can be substituted.

2. Responsively, to replace each self variable whose declaration is erased with a module path which refers to the structure or structure type that the self variable is declared.

3. To annotate each non-erased self variable with an identity module variable binding[4]. The domain of the binding exactly contains the module variables that are bound in the structure or structure type that the self variable is declared.

A module variable binding is a mapping from module variables to module paths. Unlike in *Marguerite*, we regard domains of module variable bindings as sequences of module variables. For a module variable binding $\theta = [X_1 \mapsto p_1, \ldots, X_n \mapsto p_n]$, $dom(\theta)$ denotes $\{\{X_1, \ldots, X_n\}\}$, where we use "{{" and "}}" to denote sequences. Application of a module variable binding θ to a module path p is defined inductively as follows:

$$\theta(Z^{\theta_1}) = Z^{\theta\circ\theta_1} \qquad \theta(X) = \begin{cases} X & \text{when } X \notin dom(\theta) \\ p & \text{when } X \in dom(\theta) \text{ and } \theta(X) = p \end{cases}$$

$$\theta(p.M) = \theta(p).M \quad \theta(p_1(p_2)) = \theta(p_1)(\theta(p_2))$$

For module variable bindings θ_1 and θ_2, their composition $\theta_1 \circ \theta_2$ denotes a

[4]Note that we will distinguish between identity module variable bindings and the empty module variable binding. The domain of an identity module variable binding is not empty.

Toplevel module expressions

$$
\begin{array}{rcl}
TE & ::= & \texttt{struct}\ (Z^\theta)\ D_1 \ldots D_n\ \texttt{end} \\
& | & (TE : TS) \\
& | & p
\end{array}
$$

Non-toplevel module expressions

$$
\begin{array}{rcl}
NE & ::= & \texttt{struct}\ D_1 \ldots D_n\ \texttt{end} \\
& | & \texttt{functor}\ (X : NS) \rightarrow NE \\
& | & (TE : TS) \\
& | & p
\end{array}
$$

Definitions

$$
\begin{array}{rcl}
D & ::= & \texttt{module}\ M = NE \\
& | & \texttt{datatype}\ t = c\ \texttt{of}\ \tau \mid \texttt{type}\ t = \tau \mid \texttt{val}\ l = e
\end{array}
$$

Toplevel signature

$$
TS \quad ::= \quad \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end}
$$

Non-toplevel signatures

$$
NS \quad ::= \quad \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} \mid \texttt{functor}(X : NS_1) \rightarrow NS_2
$$

Specifications

$$
\begin{array}{rcl}
B & ::= & \texttt{module}\ M : NS \\
& | & \texttt{datatype}\ t = c\ \texttt{of}\ \tau \mid \texttt{type}\ t = \tau \mid \texttt{type}\ t \mid \texttt{val}\ l : \tau
\end{array}
$$

Program

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

Module expressions

$$
E \quad ::= \quad TE \mid NE
$$

Signatures

$$
S \quad ::= \quad TS \mid NS
$$

Figure 32: The module language after elaboration

Module identifiers

$$
mid \quad ::= \quad Z^\theta \mid mid.M \mid mid(p)
$$

Module paths

$$
p, q, r \quad ::= \quad mid \mid X
$$

Figure 33: Module paths after elaboration

module variable environment such that $dom(\theta_1 \circ \theta_2) = dom(\theta_2)$ and, for all X in $dom(\theta_1 \circ \theta_2)$, $\theta_1 \circ \theta_2(X) = \theta_1(\theta_2(X))$. For a set of module variables $\mathcal{X}$ and a sequence of module variables Λ, we write $\mathcal{X} \subseteq \Lambda$ when all elements in $\mathcal{X}$ is also in Λ.

We examine two examples to deliver the intuition of the elaboration, then review the syntax after the elaboration and define a function for the elaboration operation.

The first example is:

```
struct (Z1)
  module M = struct (Z2) type s = int type t = Z2.s end
end
```

The declaration of $\mathtt{Z}_2$ is superfluous and all uses of $\mathtt{Z}_2$ can be substituted by $\mathtt{Z}_1$.M. Indeed, $\mathtt{Z}_1$.M refers to the structure that $\mathtt{Z}_2$ is declared. Hence, the above program is elaborated into:

```
struct (Z1^ε)
  module M = struct type s = int type t = Z1^ε.M.s end
end
```

The elaboration also annotated $\mathtt{Z}_1$ with the empty module variable binding ϵ, which is the module variable binding whose domain is empty. There are no module variables bound in the structure where $\mathtt{Z}_1$ is declared.

The elaboration cannot erase declarations of self variables in the outermost structures and structure types inside sealing. For instance in Figure 34, we keep the declaration of $\mathtt{Z}_4$, but erase that of $\mathtt{Z}_5$. The use of $\mathtt{Z}_5$ can be substituted by $\mathtt{Z}_4$.N. We do not expand types during elaboration. The type definition of u in the module N is elaborated into `type u = Z4^ε.s * Z4^ε.N.t`, not into `type u = int * Z4^ε.N.t`. Hence, elaboration does not diverge. We keep declarations of self variables in the outermost sealing signatures. The type system uses these self variables when type checking a sealing construct. Hence, the declaration of $\mathtt{Z}_2$ is kept, but not that of $\mathtt{Z}_3$. As a whole, Figure 34 is elaborated into Figure 35.

Now let us review the syntax of modules after elaboration (Figure 32). Module expressions and signatures are divided into toplevels and non-toplevels, where toplevels declare self variables but non-toplevels do not. We nominate module expressions and signatures as toplevels when they are immediate sub-constructs of sealing. Due to the conventions described earlier, neither toplevel module expression nor signature cannot be a functor (type). Pro-

```
struct (Z1)
 module M = (struct (Z4)
  type s = int
  module N = struct (Z5)
   datatype t = A type u = Z4.s * Z5.t end
 end : sig (Z2)
  type s
  module N : sig (Z3) type t type u = Z2.s * Z3.t end
 end)
end
```

Figure 34: Example of elaboration

```
struct (Z1ϵ)
 module M = (struct (Z4ϵ)
  type s = int
  module N = struct datatype t = A type u = Z4ϵ.s * Z4ϵ.N.t end
 end : sig (Z2ϵ)
  type s
  module N : sig type t type u = Z2ϵ.s * Z2ϵN.t end
 end)
end
```

Figure 35: Result of elaboration

grams are toplevel structures.

We often say module expressions to denote both toplevel and non-toplevel module expressions together and use E as a metavariable for them. Similarly, we say signatures to denote toplevel and non-toplevel signatures and use S as a metavariable for them.

Figure 33 gives the syntax for module paths after elaboration. Self variables are annotated with module variable bindings. Otherwise module paths have the same syntax as before.

In Figure 36, we define a function *elb* for the elaboration operation. The notation $[Z \mapsto p]D$ denotes substitution of p for Z in D. The notation $[Z \mapsto p]B$ is read similarly. The behavior of *elb* is already summarized in the beginning of this subsection. We use three helper functions. The function *elb_nt* traverses non-top levels, hence it erases declarations of self variables (in $(\star)$-labeled rules). The function *elb_t* does top levels, hence it annotates self variables with module variable bindings (in $(\star\star)$-labeled rules). The function *elb_mv* operates on signatures of functor arguments. It substitutes a functor's formal parameter for the self variable declared in the parameter's signature. Recall our convention that a module variable is bound inside its own signature. Hence *elb_mv* does not introduce unbound module variables.

In the rest of the thesis, we only consider programs after elaboration.

$elb(\texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}) = \texttt{struct}\ (Z^\epsilon)\ D'_1 \ldots D'_n\ \texttt{end}$
where $D'_i = elb\_nt(\epsilon, Z^\epsilon, [Z \mapsto Z^\epsilon]D_i)$

$(\star)$ $elb\_nt(\theta, p, \texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end})$
$= \texttt{struct}\ elb\_nt(\theta, p, [Z \mapsto p]D_1) \ldots elb\_nt(\theta, p, [Z \mapsto p]D_n)\ \texttt{end}$

$elb\_nt(\theta, p, \texttt{functor}(X : S) \rightarrow\ E)$
$= \texttt{functor}(X : elb\_mv(X, S)) \rightarrow elb\_nt(\theta[X \mapsto X], p(X), E)$

$elb\_nt(\theta, p, (E : S)) = (elb\_t(\theta, E) :\ elb\_t(\theta, S))$

$elb\_nt(\theta, p, q) = q$

$(\star)$ $elb\_nt(\theta, p, \texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end})$
$= \texttt{sig}\ elb\_nt(\theta, p, [Z \mapsto p]B_1) \ldots elb\_nt(\theta, p, [Z \mapsto p]B_n)\ \texttt{end}$

$elb\_nt(\theta, p, \texttt{functor}(X : S_1) \rightarrow\ S_2)$
$= \texttt{functor}(X : elb\_mv(X, S_1)) \rightarrow elb\_nt(\theta[X \mapsto X], p(X), S_2)$

$elb\_nt(\theta, p, \texttt{module}\ M = E) = \texttt{module}\ M = elb\_nt(\theta, p.M, E)$

$elb\_nt(\theta, p, D) = D$ when D is not a module definition

$elb\_nt(\theta, p, \texttt{module}\ M : S) = \texttt{module}\ M : elb\_nt(\theta, p.M, S)$

$elb\_nt(\theta, p, S) = S$ when S is not a module specification

$(\star\star)$ $elb\_t(\theta, \texttt{struct}\ (Z)\ D_1 \ldots D_n\ \texttt{end}) = \texttt{struct}\ (Z^\theta)\ D'_1 \ldots D'_n\ \texttt{end}$
where $D'_i = elb\_nt(\theta, Z^\theta, [Z \mapsto Z^\theta]D_i)$

$elb\_t(\theta, (E : S)) = (elb\_t(\theta, E) : elb\_t(\theta, S))$

$elb\_t(\theta, p) = p$

$(\star\star)$ $elb\_t(\theta, \texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}) = \texttt{sig}\ (Z^\theta)\ B'_1 \ldots B'_n\ \texttt{end}$
where $B'_i = elb\_nt(\theta, Z^\theta, [Z \mapsto Z^\theta]B_i)$

$elb\_mv(X, \texttt{sig}\ (Z)\ B_1 \ldots B_n\ \texttt{end}) = [Z \mapsto X](\texttt{sig}\ B_1 \ldots B_n\ \texttt{end})$

Figure 36: Elaboration operation

10 Reconstruction

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

In this section we explain the reconstruction part; in the next section we do the type-correctness check part.

The rest of this section is organized as follows. We first introduce lazy program types, which are output from the reconstruction (Section 10.1). Then we define a look-up judgment for using programs and lazy program types as lookup tables (Section 10.2). We define expansion algorithms for *Traviata*, by adapting those for *Marguerite* (Section 10.3). Finally, we present an algorithm for reconstructing lazy program types from programs (Section 10.4).

10.1 Lazy module types

In Figure 37, we give the syntax for lazy module types, which we use as types of modules during type checking. The syntax mimics that for module expressions (Figure 32). We have toplevel and non-toplevel lazy signatures, where only toplevels declare self variables. Both toplevel and non-toplevel lazy signatures may be lazy sealing types $(TT : TS)$ or lazy paths types p. We use lazy sealing types to check type-correctness of a sealing construct $(TE : TS)$ of module expressions (in rule (33) in Figure 55). We use lazy path types to instantiate signatures lazily (in rule (59) in Figure 57). In the construct `sig` (Z^θ) $C_1 \ldots C_n$ `end`, the name Z is bound in $C_1 \ldots C_n$. A lazy program type is a toplevel lazy structure type. We may say lazy signatures to denote toplevel and non-toplevel lazy signatures together and use T as a metavariable for them. Note that lazy signatures include signatures.

Lazy path types are important for keeping a flexible module abbreviation mechanism. For instance, for the implementation of the `Tree` module in Figure 28, the type system reconstructs a lazy signature:

Toplevel lazy signatures

TT	::=	$\texttt{sig}\ (Z^\theta)\ C_1 \ldots C_n\ \texttt{end}$	*lazy structure type*
	|	$(TT : TS)$	*lazy sealing type*
	|	p	*lazy path type*

Non-toplevel lazy signatures

NT	::=	$\texttt{sig}\ C_1 \ldots C_n\ \texttt{end}$	
	|	$\texttt{functor}(X : NS) \rightarrow NT$	*lazy functor type*
	|	$(TT : TS)$	
	|	p	

Lazy specifications

C	::=	$\texttt{module}\ M : NT$	*lazy module spec.*
	|	$\texttt{datatype}\ t = c\ \texttt{of}\ \tau$	
	|	$\texttt{type}\ t = \tau$	
	|	$\texttt{type}\ t$	
	|	$\texttt{val}\ l : \tau$	

Lazy program type

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

Lazy signatures

T	::=	$TT \mid NT$

Figure 37: Lazy module types

Toplevels	O	::=	$\texttt{struct}\ (Z^\theta)\ D_1 \ldots D_n\ \texttt{end}$
		|	$\texttt{sig}\ (Z^\theta)\ C_1 \ldots C_n\ \texttt{end}$
		|	$\texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end}$
Toplevel module descriptions	TK	::=	$TE \mid TS \mid TT$
Non-toplevel module descriptions	NK	::=	$NE \mid NS \mid NT$
Module descriptions	K	::=	$TK \mid NK$
Module components	J	::=	$D \mid C \mid B$
	$:=$	::=	$= \mid :$
	$\texttt{ss}$	::=	$\texttt{struct} \mid \texttt{sig}$

Figure 38: Notation convention

```
sig (T^ϵ)
 module F = TF^ϵ.Forest
 datatype t = Leaf of int | Node of int * TF^ϵ.Forest.t
 val max : T^ϵ.t → int
end
```

The module abbreviation `module F = TF`$^\epsilon$`.Forest` is kept using a lazy path type. We cannot expand it out to a structure type, which would require infinitely nesting structure types. In addition, lazy path types make it possible for *Traviata* to support fully applicative functors. We examine it in detail in Section 14.

We use the notation convention in Figure 38. In particular, we use O as a metavariable for toplevels, which are either toplevel structures, toplevel structure types or toplevel lazy structure types, and K for *module descriptions*, which are either module expressions, signatures or lazy signatures, and J for *module components*, which are either definitions, specifications or lazy specifications.

10.2 Look-up

We introduce *self variable environments*, *module variable environments* and *variable environments* as the corresponding notions in *Traviata* to program environments in *Marguerite*.

A self variable environment is a mapping from names of self variables to pairs of a module description and a sequence of module variables. We use self variable environments as look-up tables when resolving module path ref-

$$\frac{\Delta = (\mu, \nu)}{\Delta \vdash X \mapsto (\epsilon, \nu(X))} \ (1)$$

$$\frac{\Delta = (\mu, \nu) \quad \mu(Z) = (K, \Lambda) \quad dom(\theta) = \Lambda}{\Delta \vdash Z^{\theta} \mapsto (\theta, K)} \ (2)$$

$$\frac{\Delta \vdash p \mapsto (\theta, \texttt{ss} \ \ldots \texttt{module} \ M := K \ldots \texttt{end}) \quad K \neq (K_1 : K_2)}{\Delta \vdash p.M \mapsto (\theta, K)} \ (3)$$

$$\frac{\Delta \vdash p \mapsto (\theta, \texttt{ss} \ \ldots \texttt{module} \ M := K \ldots \texttt{end}) \quad K = (K_1 : K_2)}{\Delta \vdash p.M \mapsto (\theta, K_2)} \ (4)$$

$$\frac{\Delta \vdash p_1 \mapsto (\theta, \texttt{functor}(X : NS) \to K) \quad K \neq (K_1 : K_2)}{\Delta \vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], K)} \ (5)$$

$$\frac{\Delta \vdash p_1 \mapsto (\theta, \texttt{functor}(X : NS) \to K) \quad K = (K_1 : K_2)}{\Delta \vdash p_1(p_2) \mapsto (\theta[X \mapsto p_2], K_2)} \ (6)$$

Figure 39: Look-up

$$\begin{aligned}
&\mathit{mkselfenv}(\texttt{ss} \ (Z^{\theta}) \ J_1 \ldots J_n \ \texttt{end}) \\
&\quad = (Z, (\texttt{ss} \ (Z^{\theta}) \ J_1 \ldots J_n \ \texttt{end}, dom(\theta))) \cup \textstyle\bigcup_i \mathit{mkselfenv}(J_i) \\
&\mathit{mkselfenv}(\texttt{ss} \ J_1 \ldots J_n \ \texttt{end}) = \textstyle\bigcup_i \mathit{mkselfenv}(J_i) \\
&\mathit{mkselfenv}(\texttt{functor}(X : S) \to \ K) = \mathit{mkselfenv}(K) \\
&\mathit{mkselfenv}(\texttt{module} \ M := K) = \mathit{mkselfenv}(K) \\
&\mathit{mkselfenv}(J) = \emptyset \ \text{when } J \text{ is not a module definition} \\
&\qquad\qquad\qquad\qquad \text{or (lazy) module specification.}
\end{aligned}$$

Figure 40: Self variable environments of module descriptions

erences with the look-up judgment defined later. For a module description K, the self variable environment of K, written μ_K, is the self variable environment whose domain exactly contains all names of self variables declared in K and which sends a name Z of a self variable to the pair (O, Λ), where Z is the self variable of the toplevel O and Λ sets out all module variables bound in O in the binding order. Precisely, the self variable environment of K is computed by the function *mkselfenv* defined in Figure 40. Then μ_K is defined by:

$$\mu_K(Z) \quad = \quad \begin{cases} (K', \Lambda) & \text{when } (Z, (K', \Lambda)) \in \mathit{mkselfenv}(K) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We use μ as a metavariable for self variable environments and Λ for sequences of module variables. We write $\mathit{dom}(\mu)$ and μ_ϵ to denote the domain of μ and a self variable environment of the empty domain, respectively. For self variable environments μ_1 and μ_2 we write $\mu_1\mu_2$ to denote a self variable environment such that $\mathit{dom}(\mu_1\mu_2) = \mathit{dom}(\mu_1) \cup \mathit{dom}(\mu_2)$ and for any Z in $\mathit{dom}(\mu_1\mu_2)$,

$$\mu_1\mu_2(Z) = \begin{cases} \mu_2(Z) & \text{when } Z \text{ is in } \mathit{dom}(\mu_2) \\ \mu_1(Z) & \text{otherwise} \end{cases}$$

A module variable environment is a mapping from module variables to signatures. For a module description K, the module variable environment of K, written ν_K, is the module variable environment whose domain exactly contains all the module variables appearing in K and which sends a module variable to its own signature specified in K.

We use ν as a metavariable for module variable environments. We write $\mathit{dom}(\nu)$, ν_ϵ and $\nu_1\nu_2$ with similar meanings to those for self variable environments. That is, $\mathit{dom}(\nu)$ denotes the domain of ν, and ν_ϵ denotes a module variable environment whose domain is empty. For module variable environments ν_1 and ν_2, $\nu_1\nu_2$ denotes a module variable environment such that $\mathit{dom}(\nu_1\nu_2) = \mathit{dom}(\nu_1) \cup \mathit{dom}(\nu_2)$ and for any X in $\mathit{dom}(\nu_1\nu_2)$,

$$\nu_1\nu_2(X) = \begin{cases} \nu_2(X) & \text{when } X \text{ is in } \mathit{dom}(\nu_2) \\ \nu_1(X) & \text{otherwise} \end{cases}$$

A variable environment is a pair of a self variable environment and a module variable environment. For a module description K, the variable environment of K, written Δ_K, is (μ_K, ν_K). For variable environments $\Delta_1 = (\mu_1, \nu_1)$ and $\Delta_2 = (\mu_2, \nu_2)$, we write $\Delta_1\Delta_2$ to denote $(\mu_1\mu_2, \nu_1\nu_2)$.

```
struct (Z_1^ε)
 module M = (struct (Z_2^ε)
  module N = struct type t = int end
 end : sig (Z_3^ε) module N = sig type t end end)
end
```

Figure 41: A program P_1

In Figure 39, we define a look-up judgment for *Traviata* to use variable environments as look-up tables. The judgment $\Delta \vdash p \mapsto (\theta, K)$ means that the module path p resolves to the module description K *w.r.t.* the variable environment Δ, where each module variable X is bound to $\theta(X)$.

Now let us examine each rule of the look-up. For a module variable, the judgment consults the variable environment Δ, where the signature of X should be found. For a self variable Z^θ, the judgment again consults Δ, where the toplevel that Z is declared should be found. The side condition $dom(\theta) = \Lambda$ ensures coherence of the annotation θ, that is, that all free module variables in K must be bound by θ. Next two rules (3) and (4) handle module paths of the form $p.M$. A module path $p.M$ resolves to the sub-module named M in the module that p resolves to. Hence p must resolve to either a structure of a (lazy) structure type. The two rules distinguish whether M is bound to a sealing construct $(K_1 : K_2)$ or not; when it is, then $p.M$ resolves to the sealing part K_2. Thus, the judgment prevents peeking inside sealed modules from outside them. The last two rules (5) and (6) handle module paths of the form $p_1(p_2)$. When p_1 resolves to either a functor or a (lazy) functor type, then $p_1(p_2)$ does to the body of the functor, where the module variable environment is augmented with a new binding $[X \mapsto p_2]$. Again the two rules distinguish whether the body is a sealing construct or not.

For instance in Figure 41, the module paths $\mathtt{Z}_1^\epsilon.\mathtt{M.N}$ and $\mathtt{Z}_2^\epsilon.\mathtt{N}$ resolve to `sig type t end` and `struct type t = int end`, respectively.

For brevity, we extend the look-up judgment to handle type and value paths in Figure 42. All rules are as expected.

Corresponding to the convention of the absence of free module variables in programs, we assume that any variable environment we consider in this thesis does not contain free module variables. Precisely,

$$\frac{\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t \ldots \mathtt{end})}{\Delta \vdash p.t \mapsto (\theta, \mathtt{type}\ t)} \quad \frac{\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{type}\ t = \tau \ldots \mathtt{end})}{\Delta \vdash p.t \mapsto (\theta, \mathtt{type}\ t = \tau)}$$

$$\frac{\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \ldots \mathtt{end})}{\Delta \vdash p.l \mapsto (\theta, \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau)}$$

$$\frac{\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{val}\ l : \tau \ldots \mathtt{end})}{\Delta \vdash p.l \mapsto (\theta, \mathtt{val}\ l : \tau)} \quad \frac{\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{val}\ l = e \ldots \mathtt{end})}{\Delta \vdash p.l \mapsto (\theta, \mathtt{val}\ l = e)}$$

Figure 42: Look-up for type and value paths

Definition 10 *A variable environment $\Delta = (\mu, \nu)$ does not contain free module variables if, for any module path p other than a module variable, when $\Delta \vdash p \mapsto (\theta, K)$ then the following two conditions hold.*

1. *$MVars(K) \subseteq dom(\theta)$*
2. *For all X in $dom(\theta)$, $MVars(\nu(X)) \subseteq dom(\theta)$.*

10.3 Expansion algorithms

From a technical point of view, expansion algorithms we use for *Traviata* are mostly same as those we used for *Marguerite*. In particular, their termination and well-definedness (i.e., that the module path expansion reduces module paths into located forms and that the type expansion does types into located types) are proven in a similar way. We adapt them for *Traviata* in the following two ways.

1. Expansions in *Traviata* use module paths instead of integers as locks.
2. The ground expansion performs path compression so that module paths after expansion contain the innermost self variables. For instance in Figure 41, the module path $\mathtt{Z}_1^\epsilon\mathtt{.M.N}$ expands into $\mathtt{Z}_3^\epsilon\mathtt{.N}$. This is useful for defining type equality.

Location equivalence When checking whether or not a module path is already held in a lock, the expansions use a *location equivalence judgment*, defined in Figure 43. The judgment $\vdash p_1 \doteq p_2$ means that the module paths p_1 and p_2 are location equivalent. Two module paths are location equivalent if

$$\frac{}{\vdash X \doteq X} \qquad \frac{}{\vdash Z^{\theta_1} \doteq Z^{\theta_2}} \qquad \frac{\vdash p_1 \doteq p_2}{\vdash p_1.M \doteq p_2.M} \qquad \frac{\vdash p_1 \doteq q_1}{\vdash p_1(p_2) \doteq q_1(q_2)}$$

Figure 43: Location equivalence

and only if they have the syntactically same structure in disregard of functor arguments. It is easy to observe that when two module paths are location equivalent then they resolve to the same module description at the same location, according to the look-up judgment.

Module path expansion The module path expansion algorithm reduces module paths into located forms. For *Traviata* we adapt located forms from *Marguerite* so that they contain the innermost self variables.

We first define two auxiliary functions *subpaths* and *head* on module paths. For a given module path p, $\mathit{subpaths}(p)$ returns the set of sub-paths contained in the trunk of p, and $\mathit{head}(p)$ returns the self variable or the module variable at the head of p. Precisely,

$$\mathit{subpaths}(X) = \{X\} \qquad \mathit{subpaths}(Z^\theta) = \{Z^\theta\}$$
$$\mathit{subpaths}(p.M) = \{p.M\} \cup \mathit{subpaths}(p)$$
$$\mathit{subpaths}(p_1(p_2)) = \{p_1(p_2)\} \cup \mathit{subpaths}(p_1)$$

and

$$\mathit{head}(X) = X \qquad \mathit{head}(Z^\theta) = Z^\theta$$
$$\mathit{head}(p.M) = \mathit{head}(p) \qquad \mathit{head}(p_1(p_2)) = \mathit{head}(p_1)$$

Then, located forms are defined as follows.

Definition 11 *A module path p is in located form w.r.t. a variable environment Δ if the following three conditions hold.*

- $\Delta \vdash p \mapsto (\theta, K)$ *where K is not a module path.*
- *For all q in $\mathit{subpaths}(p)$ other than $\mathit{head}(p)$, if $\Delta \vdash q \mapsto (\theta', K')$ then K' is not a toplevel.*
- *For all q in $\mathit{args}(p)$, q is in located form w.r.t. Δ.*

For a module path p, $\mathit{args}(p)$ denotes the set of module paths appearing inside p as functor arguments, or:

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

We say that a module variable binding θ is in located form *w.r.t.* Δ if and only if, for all X in $dom(\theta)$, $\theta(X)$ is in located form *w.r.t.* Δ.

In Figure 44, we define the ground expansion for *Traviata*. We use Π as a metavariable for sets of module paths. The notation $\Pi \uplus_p q$ means $\Pi \cup \{q\}$ whenever Π does not contain a module path r such that $\vdash q \doteq r$. Compared to the ground expansion in *Marguerite*, we introduced two new rules [**gnlz-comps1**] [**gnlz-comps2**] to perform path compression. When a module path resolves to a toplevel, then the ground expansion substitutes the self variable declared in the toplevel for the module path. The other rules are same as those in *Marguerite*.

Variable normalization and module path expansion for *Traviata* are defined by the same inference rules as those in *Marguerite*. We repeat their definitions in Figure 45 and 46, respectively.

Termination and well-definedness of the module path expansion are proven in a similar way to in *Marguerite*. We only need a sanity condition on annotations of self variables, which we assume to hold for any input module path to the module path expansion.

Definition 12 *A module path p has located variables w.r.t. a variable environment Δ if all the self variables contained in p are in located form w.r.t. Δ.*

Definition 13 *A variable environment Δ has located variables if all the module paths appearing in Δ have located variables w.r.t. Δ.*

After the elaboration described in Section 9.1, all self variables in a program P are annotated with identity module variable bindings which have appropriate domains. Hence all module paths in P have located variables *w.r.t.* Δ_P.

The lemma below ensures that the ground expansion cannot augment a lock infinitely often.

Lemma 38 *For any variable environment Δ, let $\mathcal{P}$ be the set of module paths in located form w.r.t. Δ. The quotient set of $\mathcal{P}$ by the location equivalence relation is finite.*

[gnlz-mvar]

$$\frac{}{\Delta, \Pi \vdash X \rightsquigarrow_g X}$$

[gnlz-self]

$$\frac{}{\Delta, \Pi \vdash Z^{\theta} \rightsquigarrow_g Z^{\theta}}$$

[gnlz-comps1]

$$\frac{\Delta, \Pi \vdash p \rightsquigarrow_g p' \quad \Delta \vdash p'.M \mapsto (\theta, \texttt{ss}\ (Z^{\theta'}) \ldots \texttt{end})}{\Delta, \Pi \vdash p.M \rightsquigarrow_g \theta(Z^{\theta'})}$$

[gnlz-def1]

$$\frac{\Delta, \Pi \vdash p \rightsquigarrow_g p' \quad \Delta \vdash p'.M \mapsto (\theta, K) \quad K \notin mid \quad K \neq O}{\Delta, \Pi \vdash p.M \rightsquigarrow_g p'.M}$$

[gnlz-pth1]

$$\frac{\Delta, \Pi \vdash p \rightsquigarrow_g p' \quad \Delta \vdash p'.M \mapsto (\theta, q) \quad q \neq X \quad \Delta, \Pi \uplus_p p'.M \vdash q \rightsquigarrow_g r}{\Delta, \Pi \vdash p.M \rightsquigarrow_g \theta(r)}$$

[gnlz-comps2]

$$\frac{\Delta, \Pi \vdash p_1 \rightsquigarrow_g p_1' \quad \Delta, \Pi \vdash p_2 \rightsquigarrow_g p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, \texttt{ss}\ (Z^{\theta'}) \ldots \texttt{end})}{\Delta, \Pi \vdash p_1(p_2) \rightsquigarrow_g \theta(Z^{\theta'})}$$

[gnlz-def2]

$$\frac{\Delta, \Pi \vdash p_1 \rightsquigarrow_g p_1' \quad \Delta, \Pi \vdash p_2 \rightsquigarrow_g p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, K) \quad K \notin mid \quad K \neq O}{\Delta, \Pi \vdash p_1(p_2) \rightsquigarrow_g p_1'(p_2')}$$

[gnlz-pth2]

$$\frac{\Delta, \Pi \vdash p_1 \rightsquigarrow_g p_1' \quad \Delta, \Pi \vdash p_2 \rightsquigarrow_g p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, q) \quad \Delta, \Pi \uplus_p p_1'(p_2') \vdash q \rightsquigarrow_g r}{\Delta, \Pi \vdash p_1(p_2) \rightsquigarrow_g \theta(r)}$$

Figure 44: Ground expansion

$$\frac{}{\Delta \vdash \mathit{varnlz}(X) = X} \qquad \frac{}{\Delta \vdash \mathit{varnlz}(Z^\theta) = Z^\theta}$$

$$\frac{\Delta \vdash \mathit{varnlz}(p) = p' \quad \Delta \vdash \mathit{varsubst}(p'.M) = q}{\Delta \vdash \mathit{varnlz}(p.M) = q}$$

$$\frac{\Delta \vdash \mathit{varnlz}(p_1) = p_1' \quad \Delta \vdash \mathit{varnlz}(p_2) = p_2' \quad \Delta \vdash \mathit{varsubst}(p_1'(p_2')) = q}{\Delta \vdash \mathit{varnlz}(p_1(p_2)) = q}$$

$$\frac{\Delta \vdash p \mapsto (\theta, X)}{\Delta \vdash \mathit{varsubst}(p) = \theta(X)} \qquad \frac{\Delta \vdash p \mapsto (\theta, K) \quad K \neq X}{\Delta \vdash \mathit{varsubst}(p) = p}$$

Figure 45: Variable normalization

$$\frac{\Delta, \emptyset \vdash p \rightsquigarrow_g q \quad \Delta \vdash \mathit{varnlz}(q) = r}{\Delta \vdash p \rightsquigarrow r}$$

Figure 46: Module path expansion

Proof. Suppose that all module descriptions appearing in Δ are labeled with distinct natural numbers. Let n be the greatest number among these labels. Let p_1 and p_2 be in located form *w.r.t.* Δ and $\vdash p_1 \doteq p_2$. By definition, we have $\Delta \vdash p_1 \mapsto (\theta_1, K_1^{i_1})$ and $\Delta \vdash p_2 \mapsto (\theta_2, K_2^{i_2})$, where i_1 and i_2 are labels. By induction on the derivation of $\vdash p_1 \doteq p_2$, we prove $i_1 = i_2$. Hence we conclude that the number of the elements of the quotient set of $\mathcal{P}$ by the location equivalence relation is less than $n + 1$. □

Proposition 15 *For any variable environment Δ having located variables and module path p having located variables w.r.t. Δ, proof search for $\Delta \vdash p \rightsquigarrow \_$ will terminate.*

Proposition 16 *For any variable environment Δ having located variables and module path p having located variables w.r.t. Δ., if $\Delta \vdash p \rightsquigarrow q$, then q is in located form w.r.t. Δ.*

Type expansion We define the type expansion for *Traviata* in Figure 47. We use Π_τ as a metavariable for sets of type paths. The notation $\Pi_\tau \uplus_\tau p.t$ means $\Pi_\tau \cup \{p.t\}$ whenever Π_τ does not contain a type path $q.t$ such that $\vdash p \doteq q$. Except that we use type paths as locks, inference rules in Figure 47 are same as those for the type expansion of *Marguerite*.

$$\textbf{[tnlz-uni]}\quad \frac{}{\Delta; \Pi_\tau \vdash \mathtt{1} \downarrow \mathtt{1}}$$

$$\textbf{[tnlz-arr]}\quad \frac{\Delta; \Pi_\tau \vdash \tau_1 \downarrow \tau_1' \quad \Delta; \Pi_\tau \vdash \tau_2 \downarrow \tau_2'}{\Delta; \Pi_\tau \vdash \tau_1 \to \tau_2 \downarrow \tau_1' \to \tau_2'} \qquad \textbf{[tnlz-pair]}\quad \frac{\Delta; \Pi_\tau \vdash \tau_1 \downarrow \tau_1' \quad \Delta; \Pi_\tau \vdash \tau_2 \downarrow \tau_2'}{\Delta; \Pi_\tau \vdash \tau_1 * \tau_2 \downarrow \tau_1' * \tau_2'}$$

$$\textbf{[tnlz-atyp]}\quad \frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.t \mapsto (\theta, \mathtt{type}\ t)}{\Delta; \Pi_\tau \vdash p.t \downarrow p'.t}$$

$$\textbf{[tnlz-dtyp]}\quad \frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.t \mapsto (\theta, \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau)}{\Delta; \Pi_\tau \vdash p.t \downarrow p'.t}$$

$$\textbf{[tnlz-abb]}\quad \frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.t \mapsto (\theta, \mathtt{type}\ t = \tau_1) \quad \Delta; \Pi_\tau \uplus_\tau p'.t \vdash \tau_1 \downarrow \tau_2 \quad \Delta; \Pi_\tau \vdash \theta(\tau_2) \downarrow \tau_3}{\Delta; \Pi_\tau \vdash p.t \downarrow \tau_3}$$

Figure 47: Type expansion

Located types in *Traviata* are defined in the exactly same way as in *Marguerite*.

Definition 14 *A simple located type w.r.t. a variable environment Δ is a type path $p.t$ where p is in located form w.r.t. Δ and either $\Delta \vdash p.t \mapsto (\theta, \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau)$ or $\Delta \vdash p.t \mapsto (\theta, \mathtt{type}\ t)$ holds.*

Definition 15 *A located type w.r.t. a variable environment Δ is a type τ where every type path $p.t$ in $typaths(\tau)$ is a simple located type w.r.t. Δ.*

The function *typaths* was defined in Section 4.

Termination and well-definedness of the type expansion are proven in a similar way as in *Marguerite*. Again we need a sanity condition which we assume to hold for any input type to the type expansion.

Definition 16 *A type τ has located variables w.r.t. a variable environment Δ if if all the module paths contained in τ have located variables w.r.t. Δ.*

Proposition 17 *For any variable environment* Δ *having located variables, lock* Π_τ*, and type* τ_1 *having located variables w.r.t.* Δ*, proof search for* $\Delta; \Pi_\tau \vdash \tau_1 \downarrow \_$ *will terminate.*

Proposition 18 *For any variable environment* Δ *having located variables, lock* Π_τ*, and type* τ_1 *having located variables w.r.t.* Δ *and type* τ_2*, if* $\Delta; \Pi_\tau \vdash \tau_1 \downarrow \tau_2$*, then* τ_2 *is a located type w.r.t.* Δ*.*

Core type reconstruction We define a core type reconstruction algorithm for *Traviata* in Figure 48 with an auxiliary judgment in Figure 49. We use Π_e as a metavariable for sets of value paths. The notation $\Pi_e \uplus_e p.l$ means $\Pi_e \cup \{p.l\}$ whenever Π_e does not contain a value path $q.l$ such that $\vdash p \doteq q$. Except that we use value paths as locks, inference rules in Figure 48 are same as those for the core type reconstruction in *Marguerite*.

Termination of the core type reconstruction can be proven in a similar way to in *Marguerite*. We need a sanity condition which we assume to hold for any input expression to the reconstruction.

Definition 17 *A core expression e has a located variables w.r.t. a variable environment* Δ *if all the module paths contained in e have located variables w.r.t.* Δ*.*

Definition 18 *A type environment* Γ *is in located form w.r.t. a variable environment* Δ *if, for all x in* $dom(\Gamma)$*,* $\Gamma(x)$ *is a located type w.r.t.* Δ*.*

Proposition 19 *For any variable environment* Δ *having located variables, type environment* Γ *in located form w.r.t.* Δ*, lock* Π_e*, and expression e having located variables w.r.t.* Δ*, proof search for* $\Delta; \Gamma; \Pi_e \vdash e :: \_$ *will terminate.*

10.4 Lazy program type reconstruction

In Figure 50, we define an algorithm which reconstructs a lazy program type from a given program, with functions found in Figure 51. The judgments $\Delta \vdash E \triangleright T$ means that the reconstruction infers the lazy signature T for the module expression E *w.r.t.* the variable environment Δ. The other judgments are read similarly.

Observe that for any variable environment Δ and a program P, proof search for $\Delta \vdash P \triangleright \_$ is deterministic. We regard inference rules of the

[rcnstr-var]

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

[rcnstr-uni]

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

[rcnstr-prd]

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

[rcnstr-prj]

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

[rcnstr-fun]

$$\frac{\Delta;\emptyset \vdash \tau' \downarrow \tau}{\Delta;\Gamma;\Pi_e \vdash (\lambda x.e_1 : \tau') :: \tau}$$

[rcnstr-app]

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

[rcnstr-cnstr]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash \mathit{cnstrlkup}(p', c) = (t, \tau)}{\Delta;\Gamma;\Pi_e \vdash p.c\ e_1 :: p'.t}$$

[rcnstr-case]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash \mathit{cnstrlkup}(p', c) = (t, \tau_1) \quad \Delta;\Gamma, x : \tau_1;\Pi_e \vdash e_2 :: \tau}{\Delta;\Gamma;\Pi_e \vdash \mathtt{case}\ e_1\ \mathtt{of}\ p.c\ x \Rightarrow\ e_2 :: \tau}$$

[rcnstr-vpth1]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.l \mapsto (\theta, \mathtt{val}\ l : \tau') \quad \Delta;\emptyset \vdash \theta(\tau') \downarrow \tau}{\Delta;\Gamma;\Pi_e \vdash p.l :: \tau}$$

[rcnstr-vpth2]

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.l \mapsto (\theta, \mathtt{val}\ l = e_1) \quad \Delta;\emptyset;\Pi_e \uplus_e p'.l \vdash e_1 :: \tau' \quad \Delta;\emptyset \vdash \theta(\tau') \downarrow \tau}{\Delta;\Gamma;\Pi_e \vdash p.l :: \tau}$$

Figure 48: Core type reconstruction

$\Delta \vdash \mathit{cnstrlkup}(p, c) = (t, \tau)$ when
$\Delta \vdash p \mapsto (\theta, \mathtt{ss} \ldots \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau' \ \ldots \mathtt{end})$ and $\Delta;\emptyset \vdash \theta(\tau') \downarrow \tau$

Figure 49: Datatype look-up

Module expressions

$$\frac{\Delta \vdash D_1 \ \triangleright C_1 \ \ldots \ \Delta \vdash D_n \ \triangleright C_n}{\Delta \vdash \texttt{struct}\ (Z^\theta)\ D_1 \ \ldots \ D_n\ \texttt{end} \ \triangleright \texttt{sig}\ (Z^\theta)\ C_1 \ \ldots \ C_n\ \texttt{end}} \ (7)$$

$$\frac{\Delta \vdash D_1 \ \triangleright C_1 \ \ldots \ \Delta \vdash D_n \ \triangleright C_n}{\Delta \vdash \texttt{struct}\ D_1 \ \ldots \ D_n\ \texttt{end} \ \triangleright \texttt{sig}\ C_1 \ \ldots \ C_n\ \texttt{end}} \ (8)$$

$$\frac{\Delta \vdash NS \ \triangleright NS' \quad \Delta \vdash E \ \triangleright T}{\Delta \vdash \texttt{functor}(X : NS) \to E \ \triangleright \texttt{functor}(X : NS') \to T} \ (9)$$

$$\frac{\Delta \vdash TS \ \triangleright TS' \quad \mathit{manif}(TE, TS) = TS_2 \quad \Delta(\mu_{TS_2}, \nu_\epsilon) \vdash TE \ \triangleright TT}{\Delta \vdash (TE : TS) \ \triangleright (TT : TS')} \ (10) \qquad \frac{\Delta \vdash p \rightsquigarrow q}{\Delta \vdash p \ \triangleright q} \ (11)$$

Signatures

$$\frac{\Delta \vdash B_1 \ \triangleright B'_1 \ldots \Delta \vdash B_n \ \triangleright B'_n}{\Delta \vdash \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end} \ \triangleright \texttt{sig}\ (Z^\theta)\ B'_1 \ldots B'_n\ \texttt{end}} \ (12)$$

$$\frac{\Delta \vdash B_1 \ \triangleright B'_1 \ldots \Delta \vdash B_n \ \triangleright B'_n}{\Delta \vdash \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} \ \triangleright \texttt{sig}\ B'_1 \ldots B'_n\ \texttt{end}} \ (13)$$

$$\frac{\Delta \vdash NS \ \triangleright NS' \quad \Delta \vdash S \ \triangleright S'}{\Delta \vdash \texttt{functor}(X : NS) \to S \ \triangleright \texttt{functor}(X : NS') \to S'} \ (14)$$

Definitions and Specifications

$$\frac{\Delta \vdash NE \ \triangleright NT}{\Delta \vdash \texttt{module}\ M = NE \ \triangleright \texttt{module}\ M : NT} \ (15)$$

$$\frac{\Delta \vdash NS \ \triangleright NS'}{\Delta \vdash \texttt{module}\ M : NS \ \triangleright \texttt{module}\ M : NS'} \ (16)$$

$$\frac{\Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau \ \triangleright \texttt{datatype}\ t = c\ \texttt{of}\ \tau'} \ (17)$$

$$\frac{\Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{type}\ t = \tau \ \triangleright \texttt{type}\ t = \tau'} \ (18)$$

$$\frac{\Delta; \emptyset; \emptyset \vdash e :: \tau}{\Delta \vdash \texttt{val}\ l = e \ \triangleright \texttt{val}\ l : \tau} \ (19) \qquad \frac{\Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{val}\ l : \tau \ \triangleright \texttt{val}\ l : \tau'} \ (20)$$

Figure 50: Lazy program type reconstruction

$$manif(\texttt{ss}\ (Z^\theta)\ \ldots \texttt{end}, TS) = update(Z^\theta, TS)$$

$$manif((TK : \texttt{sig}\ (Z^\theta) \ldots \texttt{end}), TS) = update(Z^\theta, TS)$$

$$manif(p, TS) = update(p, TS)$$

$$update(p, \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end})$$
$$= \texttt{sig}\ (Z^\theta)\ update(p, B_1) \ldots update(p, B_n)\ \texttt{end}$$

$$update(p, \texttt{sig}\ B_1 \ldots B_n\ \texttt{end})$$
$$= \texttt{sig}\ update(p, B_1) \ldots update(p, B_n)\ \texttt{end}$$

$$update(p, \texttt{functor}(X : NS) \rightarrow\ NS') = update(p(X), NS')$$

$$update(p, \texttt{type}\ t) = \texttt{type}\ t = p.t$$

$$update(p, \texttt{datatype}\ t = c\ \texttt{of}\ \tau) = \texttt{type}\ t = p.t$$

$$update(p, \texttt{module}\ M : NS) = update(p.M, NS)$$

$$update(p, \texttt{type}\ t = \tau) = \texttt{type}\ t = \tau$$

$$update(p, \texttt{val}\ l : \tau) = \texttt{val}\ l : \tau$$

Figure 51: Manifestation of type specifications

reconstruction as defining an algorithm which takes Δ and P as input then either returns U when the search succeeds in building a derivation tree for $\Delta \vdash P \triangleright U$ or else raises an error when the search fails. We prove termination of the proof search in Proposition 20 below.

The reconstruction is mostly a straightforward composite of the module path and the type expansions and the core type reconstruction except for the rule (10), which reconstructs a lazy signature for a sealing construct. This rule aside, the task of the reconstruction is summarized as follows.

- When a module expression is a module path, then the reconstruction expands the path (in rule (11)).
- For a type definition or type specification, it expands all types contained in the definition or specification (in rules (17), (18)).
- For a value specification, it expands the specified type (in rule (20)).
- For a value definition, it consults the core type reconstruction (in rule (19)).

Hence, the reconstruction fails when either the module path expansion, the type expansion or the core type reconstruction fails.

Now we examine the rule (10) for a sealing construct. In the first premise $\Delta \vdash TS : TS'$, the reconstruction infers the lazy signature TS' for the sealing signature TS by expanding each type in TS into a located type. Before inferring a lazy signature for the sealed module expression TE, the reconstruction enriches the variable environment so as to recover type equality between the sealing signature and the sealed module expression, by adding type equality constraint to abstract type and datatype specifications in the sealing signature.

The function *manif* in Figure 51 formalizes this manifestation operation. It takes two arguments, a toplevel module description TK and a toplevel signature TS, then returns a toplevel signature which is built from TS by connecting every abstract type and datatype specification in TS to its correspondence in TK. The functionality of *manif* is to find the module path which resolves to the toplevel inside a sealing construct. Then a helper function *update* adds type equality constraint, traversing the constituents of the sealing signature.

Definition 19 *A program has located variables w.r.t. a variable environment Δ if all the module paths appearing in P have located variables w.r.t. Δ.*

Proposition 20 *If a program P has located variables w.r.t. Δ_P, proof search for $\Delta_P \vdash P \ \triangleright \_$ will terminate.*

Proof. By induction on the structure of P. Termination of the function *update* is easily proved by structural induction on the input. Then the claim is an immediate consequence of termination of the module path expansion (Proposition 15), the type expansion (Proposition 17) and the core type reconstruction (Proposition 19). □

In the rest of the thesis, we only consider module variable environments having located variables and module paths having located variables *w.r.t.* specified variable environments. Thanks to Proposition 16, there is no possibility of breaking this assumption through expansions.

11 Type-correctness check

In this section, we present the latter part of the type system, namely the type-correctness check part.

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

To circumvent difficulties arising from forward references, other existing type systems rely on signature annotations from a programmer. As we examined in Section 1, this requirement can compel the programmer to write two different signatures for the same module. Moreover, the programmer cannot rely on type inference during development due to the requirement. This is unfortunate since a lot of useful inference algorithms have been and will be developed to support smooth development of programs.

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

There are three tasks to be completed in this type-correctness check part.

1. To check type-correctness of core expressions. (Recall that the core type reconstruction does not ensure type-correctness of expressions that it reconstructs types for.)

2. To check well-formedness of module paths, that is, to check that functor applications contained in the paths are type-correct and that the paths do not contain cyclic or dangling references.

3. To check that, for every sealing construct $(TE : TS)$, the module expression TE inhabits the signature TS.

11.1 Type equality

We define a type equivalence judgment in Figure 52, with auxiliary judgments in Figure 53 and 54. The judgment $\Delta \vdash \tau_1 \equiv \tau_2$ means that the

$$\frac{\Delta; \emptyset \vdash \tau_1 \downarrow \tau_1' \quad \Delta; \emptyset \vdash \tau_2 \downarrow \tau_2' \quad \vdash \tau_1' \equiv_\tau \tau_2'}{\Delta \vdash \tau_1 \equiv \tau_2} \ (21)$$

Figure 52: Type equivalence

$$\frac{}{\vdash \mathtt{1} \equiv_\tau \mathtt{1}} \ (22) \qquad \frac{\vdash \tau_1 \equiv_\tau \tau_1' \quad \vdash \tau_2 \equiv_\tau \tau_2'}{\vdash \tau_1 \to \tau_2 \equiv_\tau \tau_1' \to \tau_2'} \ (23)$$

$$\frac{\vdash \tau_1 \equiv_\tau \tau_1' \quad \vdash \tau_2 \equiv_\tau \tau_2'}{\vdash \tau_1 * \tau_2 \equiv_\tau \tau_1' * \tau_2'} \ (24) \qquad \frac{\vdash p_1 \equiv_p p_2}{\vdash p_1.t \equiv_\tau p_2.t} \ (25)$$

Figure 53: Equivalence on located types

two types τ_1 and τ_2 are equivalent *w.r.t.* the variable environment Δ. As in *Marguerite*, the type system judges type equivalence by reducing types into located ones. Figure 53 defines an equivalence judgment on located types and Figure 54 does on module paths in located forms. Both judgments are syntactic and straightforward. The rule (29) on self variables may appear unfamiliar, however. Two self variables are equivalent if and only if 1) they have the same name and 2) they are annotated with module variable bindings which have the same domain and map each module variable in the domain to equivalent module paths.

11.2 Typing rules

We present typing rules for type-correctness check of the module language and of the core language in Figure 55 and 56, respectively. Auxiliary judgments and functions are found in Figure 51 to 59.

The judgment $\Delta \vdash E : T$ means that the module expression E of the lazy signature T is type-correct *w.r.t.* the variable environment Δ. The judgment $\Delta; \Gamma \vdash e : \tau$ means that the core expression e of the type τ is type-correct

$$\frac{}{\vdash X \equiv_p X} \ (26) \quad \frac{\vdash p_1 \equiv_p p_2}{\vdash p_1.M \equiv_p p_2.M} \ (27) \quad \frac{\vdash p_1 \equiv_p q_1 \quad \vdash p_2 \equiv_p q_2}{\vdash p_1(p_2) \equiv_p q_1(q_2)} \ (28)$$

$$\frac{dom(\theta_1) = dom(\theta_2) \quad \forall X \in dom(\theta_1), \ \vdash \theta_1(X) \equiv_p \theta_2(X)}{\vdash Z^{\theta_1} \equiv_p Z^{\theta_2}} \ (29)$$

Figure 54: Equivalence on module paths in located forms

Module expressions

$$\frac{\Delta \vdash D_1 : C_1 \ \ldots \ \Delta \vdash D_n : C_n}{\Delta \vdash \texttt{struct}\ (Z^\theta)\ D_1\ \ldots\ D_n\ \texttt{end} : \texttt{sig}\ (Z^\theta)\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (30)$$

$$\frac{\Delta \vdash D_1 : C_1 \ \ldots \ \Delta \vdash D_n : C_n}{\Delta \vdash \texttt{struct}\ D_1\ \ldots\ D_n\ \texttt{end} : \texttt{sig}\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (31)$$

$$\frac{\Delta \vdash NS : NS' \quad \Delta \vdash NE : NT}{\Delta \vdash \texttt{functor}(X : NS) \to NE : \texttt{functor}(X : NS') \to NT} \quad (32)$$

$$\frac{\begin{array}{c}\Delta \vdash TS : TS' \quad \mathit{manif}(TT, TS) = TS_2 \\ \Delta(\mu_{TS_2}, \nu_\epsilon) \vdash TE : TT \quad \Delta(\mu_{TS_2}, \nu_\epsilon) \vdash TT < TS'\end{array}}{\Delta \vdash (TE : TS) : (TT : TS')} \quad (33)$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow q}{\Delta \vdash p : q} \quad (34)$$

Signatures

$$\frac{\Delta \vdash B_1 : B_1' \ldots \Delta \vdash B_n : B_n'}{\Delta \vdash \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ (Z^\theta)\ B_1' \ldots B_n'\ \texttt{end}} \quad (35)$$

$$\frac{\Delta \vdash B_1 : B_1' \ldots \Delta \vdash B_n : B_n'}{\Delta \vdash \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ B_1' \ldots B_n'\ \texttt{end}} \quad (36)$$

$$\frac{\Delta \vdash NS_1 : NS_1' \quad \Delta \vdash NS_2 : NS_2'}{\Delta \vdash \texttt{functor}(X : NS_1) \to NS_2 : \texttt{functor}(X : NS_1') \to NS_2'} \quad (37)$$

Definitions and Specifications

$$\frac{\Delta \vdash NE : NT}{\Delta \vdash \texttt{module}\ M = NE : \texttt{module}\ M : NT} \quad (38)$$

$$\frac{\Delta; \emptyset \vdash e : \tau}{\Delta \vdash \texttt{val}\ l = e : \texttt{val}\ l : \tau} \quad (39)$$

$$\frac{\Delta \vdash \tau \diamond \quad \Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau : \texttt{datatype}\ t = c\ \texttt{of}\ \tau'} \quad (40)$$

$$\frac{\Delta \vdash \tau \diamond \quad \Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{type}\ t = \tau : \texttt{type}\ t = \tau'} \quad (41)$$

$$\frac{\Delta \vdash NS : NS'}{\Delta \vdash \texttt{module}\ M : NS : \texttt{module}\ M : NS'} \quad (42)$$

$$\frac{}{\Delta \vdash \texttt{type}\ t : \texttt{type}\ t} \quad (43) \qquad \frac{\Delta \vdash \tau \diamond \quad \Delta; \emptyset \vdash \tau \downarrow \tau'}{\Delta \vdash \texttt{val}\ l : \tau : \texttt{val}\ l : \tau'} \quad (44)$$

Figure 55: Typing rules for the module language

Core types

$$\frac{}{\Delta \vdash \mathtt{1} \diamond} \ (45) \qquad \frac{\Delta \vdash \tau_1 \diamond \quad \Delta \vdash \tau_2 \diamond}{\Delta \vdash \tau_1 \to \tau_2 \diamond} \ (46) \qquad \frac{\Delta \vdash \tau_1 \diamond \quad \Delta \vdash \tau_2 \diamond}{\Delta \vdash \tau_1 * \tau_2 \diamond} \ (47)$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta; \emptyset \vdash p.t \downarrow \tau}{\Delta \vdash p.t \diamond} \ (48)$$

Core expressions

$$\frac{}{\Delta; \Gamma \vdash () : \mathtt{1}} \ (49) \qquad \frac{x \in dom(\Gamma)}{\Delta; \Gamma \vdash x : \Gamma(x)} \ (50)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2} \ (51) \qquad \frac{\Delta; \Gamma \vdash e : \tau_1 * \tau_2}{\Delta; \Gamma \vdash \pi_i(e) : \tau_1} \ (52)$$

$$\frac{\Delta \vdash \tau \diamond \quad \Delta; \emptyset \vdash \tau \downarrow \tau_1 \to \ \tau_2 \quad \Delta; \Gamma, x : \tau_1 \vdash e : \tau_3 \quad \Delta \vdash \tau_2 \equiv \tau_3}{\Delta; \Gamma \vdash (\lambda x.e : \tau) : \tau_1 \to \tau_2} \ (53)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash e_2 : \tau_3 \quad \Delta \vdash \tau_1 \equiv \tau_3}{\Delta; \Gamma \vdash e_1 \ (e_2) : \tau_2} \ (54)$$

$$\frac{\begin{array}{c}\Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash \mathit{cnstrlkup}(p', c) = (t, \tau_1) \\ \Delta; \Gamma \vdash e : \tau_2 \quad \Delta \vdash \tau_1 \equiv \tau_2\end{array}}{\Delta; \Gamma \vdash p.c \ e : p'.t} \ (55)$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \tau_1 \quad \Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow p' \\ \Delta \vdash \mathit{cnstrlkup}(p', c) = (t, \tau_2) \quad \Delta \vdash \tau_1 \equiv p'.t \quad \Delta; \Gamma, x : \tau_2 \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ p.c\ x \Rightarrow e_2 : \tau} \ (56)$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p'.l \mapsto (\theta, \mathtt{val}\ l : \tau') \quad \Delta; \emptyset \vdash \theta(\tau') \downarrow \tau}{\Delta; \Gamma \vdash p.l : \tau} \ (57)$$

Figure 56: Typing rules for the core language

$$\frac{\Delta \vdash TS < S}{\Delta \vdash (TT : TS) < S} \ (58)$$

$$\frac{\Delta \vdash p \rightsquigarrow p' \quad \Delta \vdash p' \mapsto (\theta, T) \quad \Delta \vdash \theta(T) < S}{\Delta \vdash p < S} \ (59)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\}, \ \Delta \vdash C_{\sigma(i)} < B_i}{\Delta \vdash \texttt{sig}\ [(Z_1^{\theta_1})]\ C_1 \ldots C_n\ \texttt{end} < \texttt{sig}\ (Z_2^{\theta_2})\ B_1 \ldots B_m\ \texttt{end}} \ (60)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\}, \ \Delta \vdash C_{\sigma(i)} < B_i}{\Delta \vdash \texttt{sig}\ [(Z_1^{\theta_1})]\ C_1 \ldots C_n\ \texttt{end} < \texttt{sig}\ \ B_1 \ldots B_m\ \texttt{end}} \ (61)$$

$$\frac{\Delta \vdash NS_2 < [X_1 \mapsto X_2]NS_1 \quad \Delta \vdash [X_1 \mapsto X_2]T < S}{\Delta \vdash \texttt{functor}(X_1 : NS_1) \to T < \texttt{functor}(X_2 : NS_2) \to S} \ (62)$$

$$\frac{}{\Delta \vdash \texttt{type}\ t < \texttt{type}\ t} \ (63) \qquad \frac{}{\Delta \vdash \texttt{type}\ t = \tau < \texttt{type}\ t} \ (64)$$

$$\frac{}{\Delta \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau < \texttt{type}\ t} \ (65)$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2}{\Delta \vdash \texttt{type}\ t = \tau_1 < \texttt{type}\ t = \tau_2} \ (66)$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2}{\Delta \vdash \texttt{val}\ l : \tau_1 < \texttt{val}\ l : \tau_2} \ (67)$$

$$\frac{\Delta \vdash \tau_1 \equiv \tau_2}{\Delta \vdash \texttt{datatype}\ t = c\ \texttt{of}\ \tau_1 < \texttt{datatype}\ t = c\ \texttt{of}\ \tau_2} \ (68)$$

$$\frac{\Delta \vdash T < S}{\Delta \vdash \texttt{module}\ M : T < \texttt{module}\ M : S} \ (69)$$

Figure 57: Subtyping

$$\frac{\Delta = (\mu, \nu) \quad X \in dom(\nu)}{\Delta \vdash X \text{ wf}} \ (70) \qquad \frac{\Delta = (\mu, \nu) \quad Z \in dom(\mu)}{\Delta \vdash Z^{\theta} \text{ wf}} \ (71)$$

$$\frac{\Delta \vdash p \text{ wf} \quad \Delta \vdash p.M \rightsquigarrow q}{\Delta \vdash p.M \text{ wf}} \ (72)$$

$$\frac{\begin{array}{c}\Delta \vdash p_1 \text{ wf} \quad \Delta \vdash p_2 \text{ wf} \\ \Delta \vdash p_1 \rightsquigarrow p_1' \quad \Delta \vdash p_2 \rightsquigarrow p_2' \quad \Delta \vdash p_1(p_2) \rightsquigarrow q \\ \Delta \vdash p_1' \mapsto (\theta, \texttt{functor}\ (X : NS) \to T) \quad \Delta \vdash p_2' \triangleright \theta[X \mapsto p_2'](NS)\end{array}}{\Delta \vdash p_1(p_2) \text{ wf}} \ (73)$$

Figure 58: Well-formed module paths

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

$$\frac{\Delta \vdash \mathit{cnstrlkup}(p, c) = (t, \tau') \quad \Delta \vdash \tau \equiv \tau'}{\Delta \vdash p \triangleright \texttt{datatype}\ t = c\ \texttt{of}\ \tau}$$

Figure 59: Realization

under the type environment Γ *w.r.t.* Δ. Other judgments are read similarly.

Typing rules in Figure 55 and 56 are mostly straightforward and similar to the typing rules in *Marguerite*, except that here they associate lazy module types to typed objects. Observe that the type system enriches the variable environment when checking type-correctness of a sealing construct (in the rule (33)), in the same way as the reconstruction did.

In Figure 57, we define a subtyping judgment between lazy signatures and signatures ((58) to (62)) and between lazy specifications and specifications ((63) to (69)). We write $\texttt{sig}\ [(Z^\theta)]\ C_1 \ldots C_n\ \texttt{end}$ to denote $\texttt{sig}\ (Z^\theta)\ C_1 \ldots C_n\ \texttt{end}$ and $\texttt{sig}\ C_1 \ldots C_n\ \texttt{end}$ together. The only interesting rule is (59), which checks subtyping between a lazy path type and a signature. The rule instantiates the lazy signature of the module that p refers to, by determining the referred module with the module path expansion.

We define a well-formedness judgment for module paths in Figure 58 and a realization judgment in Figure 59. Both judgments are same as those in *Marguerite*. Note that the type system applies the rule (71) only to self variables which carry identity module variable bindings.

Definition 20 *A program P is well-typed if and only if $\Delta_P \vdash P \ \triangleright U$ and $\Delta_U \vdash P : U$ hold.*

Finally, we prove in Proposition 21 that the type system is decidable.

Lemma 39 *For any variable environment Δ, we have the following two results.*

1. *For any lazy signature T and signature S, it is decidable whether $\Delta \vdash T < S$ holds or not.*

2. *For any lazy specification C and specification B, it is decidable whether $\Delta \vdash C < B$ holds or not.*

Proof. By simultaneous induction on the structures of S and B. Most cases are straightforward. Yet, induction hypothesis does not immediately apply to rules (59) and (58).
For the rule (58), TS is a structure type by syntactic convention. Hence only applicable rules to the premise $\Delta \vdash TS < S$ is either (60) or (61). Both rules deconstruct S.
For the rule (59), since p' is in located form *w.r.t.* Δ, $\theta(T)$ is not a module path. Hence applicable rules to the premise $\Delta \vdash \theta(T) < S$ is among (58), (60), (61) and (62). The last three rules deconstruct S. The first rule eventually deconstructs S as examined above. □

Proposition 21 (Decidability of the type system) *For any program P, it is decidable whether P is well-typed or not.*

Proof. In Proposition 20, we have already proved termination of the proof search for $\Delta_P \vdash P \ \triangleright \_$. Hence it is sufficient to prove decidability of the judgment $\Delta_U \vdash P : U$. Decidability of the realization judgment and the well-formedness judgment follows from termination of the module path expansion (Proposition 15), the type expansion (Proposition 17) and the core type reconstruction (Proposition 19). Then the claim is proven by induction on the structure of P using lemma 39. □

12 Soundness

In this section, we define a call-by-value operational semantics as small step reductions of module paths and core expressions, then prove a soundness result with respect to the reductions.

We define an *erasure look-up judgment* in Figure 60 with an auxiliary function in Figure 61. During reductions we use this judgment in order to look up concrete module expressions instead of sealing signatures. The judgment $\Delta \vdash p \mapsto_{er} (\theta, K)$ means that the module path p resolves to the module description K when all sealings are erased *w.r.t.* the variable environment Δ, where each module variable X is bound to $\theta(X)$. The judgment is supposed to be used for module paths containing no module variables.

Correspondingly to the erasure look-up judgment, we introduce *erasure environments* for mapping self variables to module expressions inside sealing. The erasure environment of a module description K is a self variable environment whose domain exactly contains all self variables declared in K and which sends a name Z of a self variable to a pair (K', Λ) satisfying the following three conditions:

1. When $\mu_K(Z) = (K'', \Lambda')$, then $\Lambda = \Lambda'$.
2. When Z is declared in a toplevel structure in K, then K' is the toplevel structure.
3. When Z is declared in a toplevel sealing signature in K, then K' is the innermost module expression in the sealing construct. (Hence K' is not a sealing construct.)

In Figure 62, we define a small step normalization of module paths. The judgment $\Delta \vdash p \to q$ means that the module path p reduces into the module path q in one step *w.r.t.* the variable environment Δ. The normalization traces module abbreviations in the intuitive way and expands module paths in a lazy strategy in the sence that functor arguments are not reduced. Note that in the context of *Traviata* well-typed programs may still contain cyclic module abbreviations since signatures can hide these cycles, whereas in the context of *Marguerite* they may not.

Values v and evaluation contexts L are:

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

$$\frac{\Delta = (\mu, \nu) \quad \mu(Z) = (K, \Lambda) \quad dom(\theta) = \Lambda}{\Delta \vdash Z^{\theta} \mapsto_{er} (\theta, K)}$$

$$\frac{\Delta \vdash p \mapsto_{er} (\theta, \texttt{ss}\ \ldots \texttt{module}\ M = K \ldots \texttt{end})}{\Delta \vdash p.M \mapsto_{er} (\theta, erase(K))}$$

$$\frac{\Delta \vdash p_1 \mapsto_{er} (\theta, \texttt{functor}(X : NS) \to K)}{\Delta \vdash p_1(p_2) \mapsto_{er} (\theta[X \mapsto p_2], erase(K))}$$

Figure 60: Erasure look-up

$$erase((K : S)) = erase(K)$$
$$erase(K) = K \quad \text{when } K \neq (K' : S)$$

Figure 61: Sealing erasure

where p does not contain module variables.

A small step reduction of core expressions is defined with respect to a variable environment Δ, which is either:

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

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

$$\Delta \vdash p.l \overset{\texttt{mp}}{\to} p'.l \quad \text{when } \Delta \vdash p \to p'$$

$$\Delta \vdash p.l \overset{\texttt{vpth}}{\to} \theta(e) \quad \text{when } \Delta \vdash p \mapsto_{er} (\theta, \texttt{struct} \ldots\ \texttt{val}\ l = e \ldots\ \texttt{end})$$

or an inner reduction obtained by induction:

$$\frac{\Delta \vdash e_1 \to e_2 \quad L \neq \{\}}{\Delta \vdash L\{e_1\} \to L\{e_2\}}$$

where write $\Delta \vdash e \to e'$ when e reduces into e' with one of the above four reductions.

We assume that the outermost toplevel structure of a program P declares a self variable named $\texttt{Z}_0$ and that it contains a value component named `main`. Evaluation of P begins by reducing the defining expression of `main`.

Proposition 22 (Soundness) *Let a program P be well-typed and μ be the erasure environment of P. Then the reduction of $\texttt{Z}_0\texttt{.main}$ w.r.t. (μ, ν_ϵ) either returns a value or else gives rise to an infinite reduction sequence.*

We cannot state a subject reduction property in the context of *Traviata*.

$$\frac{\Delta \vdash p \to p'}{\Delta \vdash p.M \to p'.M} \quad \frac{\Delta \vdash p \to p'}{\Delta \vdash p(q) \to p'(q)} \quad \frac{\Delta \vdash p \mapsto_{er} (\theta, q)}{\Delta \vdash p \to \theta(q)}$$

Figure 62: Small step normalization of module paths

For the decidability result, the type system of *Traviata* rejects cyclic type abbreviations. for proving subject reduction, we want to establish a type equivalence relation which can account for these cycles. In proof, we define another type system, called *TraviataX*, which may not be decidable but can handle cycles. We prove that *TraviataX* is sound for the operational semantics of this section, by proving subject reduction and progress properties. Then, we prove that if a program P is well-typed in *Traviata*, then so is in *TraviataX*.

12.1 Proof of the soundness

As we did when proving the soundness result for *Marguerite*, below we use judgments of the ground expansion, the type expansion and the core type reconstruction that do not hold locks.

12.1.1 Results from *Marguerite*

Most lemmas we proved for the soundness result of *Marguerite* also hold in the context of *Traviata*. In particular, lemmas listed below can be proven in a similar way to in *Marguerite*.

Lemma 40 *Let p be not a module variable. If $\Delta \vdash p \mapsto (\theta_1, K)$ and $MVars(p) \subseteq dom(\theta)$, then $\Delta \vdash \theta(p) \mapsto (\theta \circ \theta_1, K)$ with $MVars(\theta_1) \subseteq dom(\theta)$.*

Lemma 41 *Let p be in located form w.r.t. Δ. Then $\Delta \vdash p \rightsquigarrow p$.*

Lemma 42 *Let τ be a located type w.r.t. Δ, then $\Delta; \Pi_\tau \vdash \tau \downarrow \tau$ for any Π_τ.*

Definition 21 *A variable environment $\Delta = (\mu, \nu)$ is well-formed, written $\Delta \vdash \Delta$ wf, if both the following conditions hold.*

1. *for all X in $dom(\nu)$, $\Delta \vdash \nu(X) : \nu(X)$*
2. *for all Z in $dom(\mu)$, when $\mu(Z) = (T, \Lambda)$ then $\Delta \vdash T : T$*

Lemma 43 *Let p and θ be in located form w.r.t. Δ and $MVars(p) \subseteq dom(\theta)$. Then $\theta(p)$ is in located form w.r.t. Δ.*

Lemma 44 *Let p be in located form w.r.t. Δ. If $\Delta \vdash p \mapsto (\theta, K)$, then θ is in located form w.r.t. Δ.*

Definition 22 *A module variable binding θ is well-formed w.r.t. a variable environment Δ, written $\Delta \vdash \theta$ wf, if, for all X in $dom(\theta)$, the following three conditions hold.*

- *$\theta(X)$ is in located form w.r.t. Δ.*
- *$\Delta \vdash \theta(X)$ wf*
- *When $\Delta \vdash X \mapsto (\theta', \texttt{sig}\ B_1 \ldots B_n\ \texttt{end})$ then $\forall i \in \{1, \ldots, n\}$, $MVars(B_i) \subseteq dom(\theta)$ and $\Delta \vdash \theta(X) \triangleright \theta(B_i)$. (Note that by definition of the look-up, $\theta' = \epsilon$.)*

Lemma 45 *If $MVars(p) \subseteq dom(\theta)$ and $\Delta \vdash p$ wf and $\Delta \vdash \theta$ wf, then $\Delta \vdash \theta(p)$ wf.*

Lemma 46 *If $\Delta \vdash \Delta$ wf and $\Delta \vdash p$ wf and $\Delta \vdash p \rightsquigarrow q$, then $\Delta \vdash q$ wf.*

Lemma 47 *If $MVars(\tau) \subseteq dom(\theta)$ and $\Delta \vdash \theta$ wf and $\Delta \vdash \tau\ \diamond$, then $\Delta \vdash \theta(\tau)\ \diamond$.*

Lemma 48 *If $\Delta \vdash \Delta$ wf and $\Delta \vdash \tau\ \diamond$ and $\Delta \vdash \tau \downarrow \tau'$, then $\Delta \vdash \tau'\ \diamond$.*

We say that a type environment Γ is in located form *w.r.t.* a variable environment Δ if and only if, for all x in $dom(\Gamma)$, $\Gamma(x)$ is a located type *w.r.t.* Δ. We also say that a type environment Γ is well-formed *w.r.t.* Δ, written $\Delta \vdash \Gamma$ wf, if and only if Γ is in located form *w.r.t.* Δ, and, for all x in $dom(\Gamma)$, $\Delta \vdash \Gamma(x)\ \diamond$.

Lemma 49 *Suppose $\Delta \vdash \Delta$ wf and $\Delta \vdash \theta$ wf and $\Delta \vdash \Gamma$ wf and $MVars(\Gamma) \cup MVars(e) \subseteq dom(\theta)$. Suppose also that Γ_1 is in located form w.r.t. Δ and satisfies two conditions: 1) $dom(\Gamma) = dom(\Gamma_1)$ and 2) for all x in $dom(\Gamma)$, $\Delta \vdash \theta(\Gamma(x)) \equiv \Gamma_1(x)$. If $\Delta; \Gamma \vdash e : \tau$, then $\Delta; \Gamma_1 \vdash \theta(e) : \tau'$ for some τ' with $\Delta \vdash \tau' \equiv \theta(\tau)$ and $MVars(\tau) \subseteq dom(\theta)$.*

$$\frac{}{\Delta \vdash X \rightsquigarrow_n X} \qquad \frac{}{\Delta \vdash Z^{\theta} \rightsquigarrow_n Z^{\theta}}$$

$$\frac{\Delta \vdash p \rightsquigarrow_n p' \quad \Delta \vdash p'.M \mapsto (\theta, \texttt{ss}\ (Z^{\theta'}) \ldots \texttt{end})}{\Delta \vdash p.M \rightsquigarrow_n \theta(Z^{\theta'})}$$

$$\frac{\Delta \vdash p \rightsquigarrow_n p' \quad \Delta \vdash p'.M \mapsto (\theta, K) \quad K \neq q \quad K \neq O}{\Delta \vdash p.M \rightsquigarrow_n p'.M}$$

$$\frac{\Delta \vdash p \rightsquigarrow_n p' \quad \Delta \vdash p'.M \mapsto (\theta, q) \quad \Delta \vdash \theta(q) \rightsquigarrow_n r}{\Delta \vdash p.M \rightsquigarrow_n r}$$

$$\frac{\Delta \vdash p_1 \rightsquigarrow_n p_1' \quad \Delta \vdash p_2 \rightsquigarrow_n p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, \texttt{ss}\ (Z^{\theta'}) \ldots \texttt{end})}{\Delta \vdash p_1(p_2) \rightsquigarrow_n \theta(Z^{\theta'})}$$

$$\frac{\Delta \vdash p_1 \rightsquigarrow_n p_1' \quad \Delta \vdash p_2 \rightsquigarrow_n p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, K) \quad K \neq q \quad K \neq O}{\Delta \vdash p_1(p_2) \rightsquigarrow_n p_1'(p_2')}$$

$$\frac{\Delta \vdash p_1 \rightsquigarrow_n p_1' \quad \Delta \vdash p_2 \rightsquigarrow_n p_2' \quad \Delta \vdash p_1'(p_2') \mapsto (\theta, q) \quad \Delta \vdash \theta(q) \rightsquigarrow_n r}{\Delta \vdash p_1(p_2) \rightsquigarrow_n r}$$

Figure 63: Normalization of module paths in *Traviata*

To prove that the module path expansion coincides with the intuitive normalization for well-formed module paths, we need adapt the normalization of *Marguerite* to perform path compression. In Figure 63, we define normalization of module paths for *Traviata*.

Lemma 50 *Suppose* Δ *has located variables and so does* p *w.r.t.* Δ*. When* $\Delta \vdash p$ wf*, then* $\Delta \vdash p \rightsquigarrow q$ *if and only if* $\Delta \vdash p \rightsquigarrow_n q$*.*

12.1.2 Type system *TraviataX*

TraviataX only provides a type-correctness check part, but not a reconstruction part. Given a program P, its expected lazy program type U and a variable environment Δ, it checks that U is a correct type of P *w.r.t.* Δ.

Here are two notable differences between *Traviata* and *TraviataX*.

1. The type equivalence relation in *TraviataX* is defined by confluence of types with respect to a rewriting relation on types. In this way, *TraviataX* handles cyclic type abbreviations.

$$\frac{\Delta \vdash \tau_1 \to \tau_1'}{\Delta \vdash \tau_1 \to \tau_2 \to \tau_1' \to \tau_2} \quad \frac{\Delta \vdash \tau_2 \to \tau_2'}{\Delta \vdash \tau_1 \to \tau_2 \to \tau_1 \to \tau_2'}$$

$$\frac{\Delta \vdash \tau_1 \to \tau_1'}{\Delta \vdash \tau_1 * \tau_2 \to \tau_1' * \tau_2} \quad \frac{\Delta \vdash \tau_2 \to \tau_2'}{\Delta \vdash \tau_1 * \tau_2 \to \tau_1 * \tau_2'}$$

$$\frac{\Delta \vdash p.t \mapsto (\theta, \texttt{type}\ t = \tau)}{\Delta \vdash p.t \to \theta(\tau)}$$

$$\frac{\Delta \vdash p.t \mapsto (\theta, \texttt{datatype}\ t = p_1.t_1 = c\ \texttt{of}\ \tau)}{\Delta \vdash p.t \to \theta(p_1.t_1)}$$

Figure 64: A small step reduction of types

2. *TraviataX* does not enrich the variable environment when type checking a sealing construct like *Traviata* does. Instead we provide *TraviataX* a variable environment which already contains type equality constraint between sealing signatures and corresponding sealed module expressions, by performing the manifestation operation that the function *manif* (Figure 51) does beforehand. This is further explained in Section 12.1.3.

Variable environments that *TraviataX* uses may contain *manifest datatype specifications* of the form "$\texttt{datatype}\ t = p_1.t_1 = c\ \texttt{of}\ \tau$". We extend the range of the metavariable C to contain manifest datatype specifications and write "$\texttt{datatype}\ t\ [= p_1.t_1] = c\ \texttt{of}\ \tau$" to denote $\texttt{datatype}\ t = c\ \texttt{of}\ \tau$ and $\texttt{datatype}\ t\ = p_1.t_1 = c\ \texttt{of}\ \tau$ together. We assume that manifest datatype specifications do not appear in signatures of module variables, precisely, for any variable environment Δ, when $\Delta \vdash X \mapsto (\theta, \texttt{sig}\ B_1 \ldots B_n\ \texttt{end})$ or $\Delta \vdash p \mapsto (\theta, \texttt{functor}(X\ :\ \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}) \to\ K)$, then any B_i is not a manifest datatype specification.

Type equality We define a small step reduction relation on types in Figure 64. The judgment $\Delta \vdash \tau \to \tau'$ states that the type τ reduces into the type τ' in one step *w.r.t.* the variable environment Δ. The notation $\Delta \vdash \tau \Rightarrow \tau'$ means that there is a sequence of zero or more reductions from τ to τ'. Formally, $\Delta \vdash \tau \Rightarrow \tau'$ holds if and only if either $\tau = \tau'$ or else there are types $\tau_0 = \tau, \tau_1, \ldots, \tau_n = \tau'$ with $n \geq 1$ such that $\Delta \vdash \tau_0 \to \tau_1, \Delta \vdash \tau_1 \to \tau_2, \ldots, \Delta \vdash \tau_{n-1} \to \tau_n$. We also call n the length of the reductions

A type equivalence relation in *TraviataX* is defined as confluence with respect to this reduction relation.

Definition 23 *Two types τ_1 and τ_2 are equivalent w.r.t. a variable environment Δ, written $\Delta \vdash_X \tau_1 \equiv \tau_2$, if there is τ_3 such that $\Delta \vdash \tau_1 \Rightarrow \tau_3$ and $\Delta \vdash \tau_2 \Rightarrow \tau_3$.*

Corollary 4 below states that the type equivalence relation is transitive.

Lemma 51 *If $\Delta \vdash \tau_1 \Rightarrow \tau_2$ and $\Delta \vdash \tau_1 \Rightarrow \tau_3$, then $\Delta \vdash \tau_2 \Rightarrow \tau_4$ and $\Delta \vdash \tau_3 \Rightarrow \tau_4$ for some τ_4.*

Proof. It is easy to observe that if $\Delta \vdash \tau_1 \rightarrow \tau_2$ and $\Delta \vdash \tau_1 \rightarrow \tau_3$, then either $\tau_2 = \tau_3$ or there is τ_4 such that $\Delta \vdash \tau_2 \rightarrow \tau_4$ and $\Delta \vdash \tau_3 \rightarrow \tau_4$. Hence the reflexive closure of this reduction relation on types satisfies the diamond property, from which the claim follows. □

Corollary 4 *If $\Delta \vdash_X \tau_1 \equiv \tau_2$ and $\Delta \vdash_X \tau_2 \equiv \tau_3$, then $\Delta \vdash_X \tau_2 \equiv \tau_3$.*

Typing rules We present *TraviataX*'s typing rules for type-correctness check of the module and of the core languages in Figure 65 and 66, respectively. Auxiliary judgments are found in Figure 68 to 71, where we define $\Delta \vdash_X \theta$ wf below. The subscript X is used to distinguish judgments in *TraviataX* from those in *Traviata*.

Definition 24 *A module variable binding θ is well-formed in TraviataX w.r.t. a variable environment Δ, written $\Delta \vdash_X \theta$* wf*, if, for all X in $dom(\theta)$, the following two conditions hold.*

- $\Delta \vdash_X \theta(X)$ wf
- *When $\Delta \vdash X \mapsto (\theta', \texttt{sig}\ B_1 \ldots B_n\ \texttt{end})$, then, for all i in $1, \ldots, n$, $MVars(B_i) \subseteq dom(\theta)$ and $\Delta \vdash_X \theta(X) \triangleright \theta(B_i)$.*

Most rules in figures are similar to those in *Traviata*. We use the rule (82) for type checking manifest datatype specifications to state well-formedness of variable environments (Definition 28). As we indicated above, *TraviataX* does not enrich the variable environment when type checking a sealing construct (rule (77)).

Module expressions

$$\frac{\Delta \vdash_X D_1 : C_1 \ \ldots \ \Delta \vdash_X D_n : C_n}{\Delta \vdash_X \texttt{struct}\ (Z^\theta)\ D_1\ \ldots\ D_n\ \texttt{end} : \texttt{sig}\ (Z^\theta)\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (74)$$

$$\frac{\Delta \vdash_X D_1 : C_1 \ \ldots \ \Delta \vdash_X D_n : C_n}{\Delta \vdash_X \texttt{struct}\ D_1\ \ldots\ D_n\ \texttt{end} : \texttt{sig}\ C_1\ \ldots\ C_n\ \texttt{end}} \quad (75)$$

$$\frac{\Delta \vdash_X NS : NS' \quad \Delta \vdash_X E : T}{\Delta \vdash_X \texttt{functor}(X : NS) \to E : \texttt{functor}(X : NS') \to T} \quad (76)$$

$$\frac{\Delta \vdash_X E : T \quad \Delta \vdash_X S : S' \quad \Delta \vdash_X T < S'}{\Delta \vdash_X (E : S) : (T : S')} \quad (77) \qquad \frac{\Delta \vdash_X p \text{ wf}}{\Delta \vdash_X p : p} \quad (78)$$

Definitions and Specifications

$$\frac{\Delta \vdash_X E : T}{\Delta \vdash_X \texttt{module}\ M = E : \texttt{module}\ M : T} \quad (79)$$

$$\frac{\Delta \vdash_X S : S'}{\Delta \vdash_X \texttt{module}\ M : S : \texttt{module}\ M : S'} \quad (80)$$

$$\frac{\Delta \vdash_X \tau \diamond}{\Delta \vdash_X \texttt{datatype}\ t = c\ \texttt{of}\ \tau : \texttt{datatype}\ t = c\ \texttt{of}\ \tau} \quad (81)$$

$$\frac{\Delta \vdash_X p_1 \text{ wf} \quad \Delta \vdash_X \tau \diamond \quad \Delta \vdash_X \mathit{cnstrlkup}(p_1, c) = (t_1, \tau') \quad \Delta \vdash_X \tau \equiv \tau'}{\Delta \vdash_X \texttt{datatype}\ t = p_1.t_1 = c\ \texttt{of}\ \tau : \texttt{datatype}\ t = p_1.t_1 = c\ \texttt{of}\ \tau} \quad (82)$$

$$\frac{\Delta \vdash_X \tau \diamond}{\Delta \vdash_X \texttt{type}\ t = \tau : \texttt{type}\ t = \tau} \quad (83) \qquad \frac{}{\Delta \vdash_X \texttt{type}\ t : \texttt{type}\ t} \quad (84)$$

$$\frac{\Delta; \emptyset \vdash_X e : \tau}{\Delta \vdash_X \texttt{val}\ l = e : \texttt{val}\ l : \tau} \quad (85) \qquad \frac{\Delta \vdash_X \tau \diamond}{\Delta \vdash_X \texttt{val}\ l : \tau : \texttt{val}\ l : \tau} \quad (86)$$

Signatures

$$\frac{\Delta \vdash_X B_1 : B_1' \ldots \Delta \vdash_X B_n : B_n'}{\Delta \vdash_X \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ (Z^\theta)\ B_1' \ldots B_n'\ \texttt{end}} \quad (87)$$

$$\frac{\Delta \vdash_X B_1 : B_1' \ldots \Delta \vdash_X B_n : B_n'}{\Delta \vdash_X \texttt{sig}\ B_1 \ldots B_n\ \texttt{end} : \texttt{sig}\ B_1' \ldots B_n'\ \texttt{end}} \quad (88)$$

$$\frac{\Delta \vdash_X NS : NS' \quad \Delta \vdash_X S : S'}{\Delta \vdash_X \texttt{functor}(NX : S) \to S : \texttt{functor}(X : NS') \to S'} \quad (89)$$

Figure 65: Typing rules for the module language in *TraviataX*

Core types

$$\frac{}{\Delta \vdash_X \mathtt{1} \diamond} \ (90)$$

$$\frac{\Delta \vdash_X \tau_1 \diamond \quad \Delta \vdash_X \tau_2 \diamond}{\Delta \vdash_X \tau_1 \to \tau_2 \diamond} \ (91) \qquad \frac{\Delta \vdash_X \tau_1 \diamond \quad \Delta \vdash_X \tau_2 \diamond}{\Delta \vdash_X \tau_1 * \tau_2 \diamond} \ (92)$$

$$\frac{\Delta \vdash_X p \text{ wf} \quad \Delta \vdash p.t \mapsto (\theta, C)}{\Delta \vdash_X p.t \diamond} \ (93)$$

Core expressions

$$\frac{\Delta; \Gamma \vdash_X e : \tau' \quad \Delta \vdash_X \tau \equiv \tau' \quad \Delta \vdash_X \tau \diamond}{\Delta; \Gamma \vdash_X e : \tau} \ (94)$$

$$\frac{}{\Delta; \Gamma \vdash_X () : \mathtt{1}} \ (95) \qquad \frac{x \in dom(\Gamma)}{\Delta; \Gamma \vdash_X x : \Gamma(x)} \ (96)$$

$$\frac{\Delta; \Gamma \vdash_X e_1 : \tau_1 \quad \Delta; \Gamma \vdash_X e_2 : \tau_2}{\Delta; \Gamma \vdash_X (e_1, e_2) : \tau_1 * \tau_2} \ (97) \qquad \frac{\Delta; \Gamma \vdash_X e : \tau_1 * \tau_2}{\Delta; \Gamma \vdash_X \pi_i(e) : \tau_1} \ (98)$$

$$\frac{\begin{array}{c}\Delta \vdash_X \tau \diamond \quad \Delta \vdash_X \tau \equiv \tau_1 \to \tau_2 \\ \Delta \vdash_X \tau_1 \to \tau_2 \diamond \quad \Delta; \Gamma, x : \tau_1 \vdash_X e : \tau_2\end{array}}{\Delta; \Gamma \vdash (\lambda x.e : \tau) : \tau} \ (99)$$

$$\frac{\Delta; \Gamma \vdash_X e_1 : \tau' \to \tau \quad \Delta; \Gamma \vdash_X e_2 : \tau'}{\Delta; \Gamma \vdash_X e_1 \ (e_2) : \tau} \ (100)$$

$$\frac{\Delta \vdash_X p \text{ wf} \quad \Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau_1) \quad \Delta; \Gamma \vdash_X e : \tau_1}{\Delta; \Gamma \vdash_X p.c \ e : p.t} \ (101)$$

$$\frac{\begin{array}{c}\Delta \vdash_X p \text{ wf} \quad \Delta; \Gamma \vdash_X e_1 : p.t \\ \Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau_2) \quad \Delta; \Gamma, x : \tau_2 \vdash_X e_2 : \tau\end{array}}{\Delta; \Gamma \vdash_X \mathtt{case}\ e_1\ \mathtt{of}\ p.c\ x \Rightarrow e_2 : \tau} \ (102)$$

$$\frac{\Delta \vdash_X p \text{ wf} \quad \Delta \vdash p.l \mapsto (\theta, \mathtt{val}\ l : \tau)}{\Delta; \Gamma \vdash_X p.l : \theta(\tau)} \ (103)$$

Figure 66: Typing rules for the core language in *TraviataX*

$$\frac{p \text{ is in located form } w.r.t.\ \Delta}{\Delta \vdash p \text{ lctd}}$$

Figure 67: Located form judgment

$$\Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \theta(\tau)) \quad \text{when}$$
$$\Delta \vdash p \mapsto (\theta, \texttt{ss} \ldots \texttt{datatype}\ t\ [= p_1.t_1] = c\ \texttt{of}\ \tau\ \ldots \texttt{end})$$

Figure 68: Datatype look-up in *TraviataX*

$$\frac{\Delta \vdash_X S < S'}{\Delta \vdash_X (T : S) < S'} \ (104)$$

$$\frac{\Delta \vdash_X NS' < [X \mapsto X']NS \quad \Delta \vdash_X [X \mapsto X']T < S}{\Delta \vdash_X \texttt{functor}(X : NS) \to T < \texttt{functor}(X' : NS') \to S} \ (105)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\},\ \Delta \vdash_X C_{\sigma(i)} < B_i}{\Delta \vdash_X \texttt{sig}\ [(Z_1^{\theta_1})]\ C_1 \ldots C_n\ \texttt{end} < \texttt{sig}\ (Z_2^{\theta_2})\ B_1 \ldots B_m\ \texttt{end}} \ (106)$$

$$\frac{\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\} \ \ \forall i \in \{1, \ldots, m\},\ \Delta \vdash_X C_{\sigma(i)} < B_i}{\Delta \vdash_X \texttt{sig}\ [(Z_1^{\theta_1})]\ C_1 \ldots C_n\ \texttt{end} < \texttt{sig}\ \ B_1 \ldots B_m\ \texttt{end}} \ (107)$$

$$\frac{}{\Delta \vdash_X \texttt{type}\ t < \texttt{type}\ t} \ (108) \qquad \frac{}{\Delta \vdash_X \texttt{type}\ t = \tau < \texttt{type}\ t} \ (109)$$

$$\frac{}{\Delta \vdash_X \texttt{type}\ t = c\ \texttt{of}\ \tau < \texttt{type}\ t} \ (110)$$

$$\frac{\Delta \vdash_X \tau_1 \equiv \tau_2}{\Delta \vdash_X \texttt{type}\ t = \tau_1 < \texttt{type}\ t = \tau_2} \ (111)$$

$$\frac{\Delta \vdash_X \tau_1 \equiv \tau_2}{\Delta \vdash_X \texttt{datatype}\ t = c\ \texttt{of}\ \tau_1 < \texttt{datatype}\ t = c\ \texttt{of}\ \tau_2} \ (112)$$

$$\frac{\Delta \vdash_X \tau_1 \equiv \tau_2}{\Delta \vdash_X \texttt{val}\ l : \tau_1 < \texttt{val}\ l : \tau_2} \ (113)$$

$$\frac{\Delta \vdash_X T < S}{\Delta \vdash_X \texttt{module}\ M : T < \texttt{module}\ M : S} \ (114)$$

Figure 69: Subtyping in *TraviataX*

$$\frac{\Delta = (\mu, \nu) \quad X \in dom(\nu)}{\Delta \vdash_X X \text{ wf}} \ (115) \qquad \frac{\Delta = (\mu, \nu) \quad Z \in dom(\mu) \quad \Delta \vdash_X \theta \text{ wf}}{\Delta \vdash_X Z^\theta \text{ wf}} \ (116)$$

$$\frac{\Delta \vdash_X p \text{ wf} \quad \Delta \vdash p.M \text{ lctd}}{\Delta \vdash_X p.M \text{ wf}} \ (117)$$

$$\frac{\begin{array}{c}\Delta \vdash p_1(p_2) \text{ lctd} \quad \Delta \vdash_X p_1 \text{ wf} \quad \Delta \vdash_X p_2 \text{ wf} \\ \Delta \vdash p_1 \mapsto (\theta, \texttt{functor}\ (X\ :\ \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}) \to T) \\ \forall i \in \{1, \ldots, n\},\ \Delta \vdash_X p_2 \triangleright \theta[X \mapsto p_2](B_i)\end{array}}{\Delta \vdash_X p_1(p_2) \text{ wf}} \ (118)$$

Figure 70: Well-formed module paths in *TraviataX*

$$\frac{\Delta \vdash p.t \mapsto (\theta, C)}{\Delta \vdash_X p \triangleright \texttt{type}\ t} \qquad \frac{\Delta \vdash_X p.t \equiv \tau}{\Delta \vdash_X p \triangleright \texttt{type}\ t = \tau} \qquad \frac{\Delta; \emptyset \vdash_X p.l : \tau}{\Delta \vdash_X p \triangleright \texttt{val}\ l : \tau}$$

$$\frac{\Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau') \quad \Delta \vdash_X \tau \equiv \tau'}{\Delta \vdash_X p \triangleright \texttt{datatype}\ t = c\ \texttt{of}\ \tau}$$

Figure 71: Realization in *TraviataX*

Figure 67 defines a *located form judgment*, which *Traviata* does not use. *TraviataX* requires well-typed programs only contain module paths in located form. Note also that it does not instantiate lazy path types during subtyping checking (Figure 69). This implies that module abbreviations are appropriately inline expanded according to sealing signatures in well-typed programs. (We explain this further in Section 12.1.3.) These two requirements make soundness proof simpler, since evaluation of well-typed programs does not trace module abbreviations, hence we need not consider the reduction $\xrightarrow{\texttt{mp}}$.

We introduce a sanity condition which ensures consistency between lazy program types and variable environments.

Definition 25 *A lazy program type U conforms with a variable environment Δ if, for any value path $p.l$, if $\Delta \vdash p.l \mapsto (\theta, C)$ then $\Delta_U \vdash p.l \mapsto (\theta, C)$.*

Definition 26 *Let a lazy program type U conform with a variable environment Δ. Then U is a correct-type of a program P w.r.t. Δ in TraviataX if $\Delta \vdash_X P : U$ holds.*

Soundness We establish a soundness result for *TraviataX* by proving a subject reduction property (in Proposition 24) and a progress property (in

Proposition 23).

Firstly, we prove in Lemma 54 that well-formed module variable bindings preserve type equality.

Lemma 52 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(\tau_1) \subseteq dom(\theta)$. *If* $\Delta \vdash \tau_1 \to \tau_2$ *then then* $MVars(\tau_2) \subseteq dom(\theta)$ *and* $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_2)$.

Proof. By induction on the derivation of $\Delta \vdash \tau_1 \to \tau_2$ and by case on the last rule used. We show the main case where $\tau_1 = p.t$ and $\Delta \vdash p.t \mapsto (\theta_1, \texttt{type}\ t = \tau_3)$.

1. When p is not a module variable. Then by Lemma 40, $\Delta \vdash \theta(p).t \mapsto (\theta \circ \theta_1, \texttt{type}\ t = \tau_3)$ and $MVars(\theta_1) \subseteq dom(\theta)$. We deduce $\Delta \vdash \theta(p).t \to \theta \circ \theta_1(\tau_3)$. Hence $\Delta \vdash_X \theta(p).t \equiv \theta \circ \theta_1(\tau_3)$. Since Δ does not contain free module variables, $MVars(\tau_3) \subseteq dom(\theta_1)$. Thus we have $MVars(\theta_1(\tau_3)) \subseteq dom(\theta)$.

2. When $p = X$. Then $\theta_1 = \epsilon$ and $\tau_2 = \tau_3$. The claim follows from well-formedness of θ.

□

Lemma 53 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(\tau_1) \subseteq dom(\theta)$. *If* $\Delta \vdash \tau_1 \Rightarrow \tau_2$ *then* $MVars(\tau_2) \subseteq dom(\theta)$ *and* $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_2)$.

Proof. By induction on the length of $\Delta \vdash \tau_1 \Rightarrow \tau_2$.

1. When $\tau_1 = \tau_2$. We immediately have the claim.

2. When $\Delta \vdash \tau_1 \Rightarrow \tau_3$ and $\Delta \vdash \tau_3 \to \tau_2$. By induction hypothesis, $MVars(\tau_3) \subseteq dom(\theta)$ and $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_3)$. By Lemma 52, $MVars(\tau_2) \subseteq dom(\theta)$ and $\Delta \vdash_X \theta(\tau_3) \equiv \theta(\tau_2)$. By Corollary 4, $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_2)$.

□

Lemma 54 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(\tau_1) \cup MVars(\tau_2) \subseteq dom(\theta)$. *If* $\Delta \vdash_X \tau_1 \equiv \tau_2$, *then* $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_2)$.

Proof. By definition, there is τ_3 such that $\Delta \vdash \tau_1 \Rightarrow \tau_3$ and $\Delta \vdash \tau_2 \Rightarrow \tau_3$. By Lemma 53, we have $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_3)$ and $\Delta \vdash_X \theta(\tau_2) \equiv \theta(\tau_3)$. By Corollary 4, $\Delta \vdash_X \theta(\tau_1) \equiv \theta(\tau_2)$. □

Using Lemma 54, we prove that well-formed module variable bindings preserve well-typedness of module paths (in Lemma 60), types (in Lemma 61) and core expressions (in Lemma 69). This is a similar path we followed when proving the subject reduction property in *Marguerite*.

Lemma 55 *If* $\Delta; \Gamma \vdash_X e : \tau_1$ *and* $\Delta; \Gamma \vdash_X e : \tau_2$*, then* $\Delta \vdash_X \tau_1 \equiv \tau_2$

Proof. By induction on the derivations of $\Delta; \Gamma \vdash_X e : \tau_1$ and $\Delta; \Gamma \vdash_X e : \tau_2$. □

Lemma 56 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(p) \subseteq dom(\theta)$*. If* $\Delta \vdash p.t \mapsto (\theta_1, C_1)$ *then* $\Delta \vdash \theta(p).t \mapsto (\theta_2, C_2)$*.*

Proof. When p is not a module variable, then we have $\Delta \vdash \theta(p).t \mapsto (\theta \circ \theta_1, C_1)$ by Lemma 40. When $p = X$, then we have the claim by well-formedness of θ. □

Lemma 57 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(p) \subseteq dom(\theta)$*.*
If $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau)$*, then* $\Delta \vdash_X cnstrlkup(\theta(p), c) = (t, \tau_1)$ *with* $MVars(\tau) \subseteq dom(\theta)$ *and* $\Delta \vdash_X \theta(\tau) \equiv \tau_1$*.*

Proof. Analogous to Lemma 56. □

Lemma 58 *Suppose* $\Delta \vdash p$ lctd *and* $\Delta \vdash_X \theta$ wf *and* $MVars(p) \subseteq dom(\theta)$*. If* $\Delta; \emptyset \vdash_X p.l : \tau$*, then there is* τ' *such that* $\Delta \vdash_X \tau \equiv \tau'$ *and* $MVars(\tau') \subseteq dom(\theta)$ *and* $\Delta; \emptyset \vdash_X \theta(p).l : \theta(\tau')$*.*

Proof. By $\Delta; \emptyset \vdash_X p.l : \tau$ in the hypothesis and Lemma 55, we have $\Delta \vdash p.l \mapsto (\theta_1, \texttt{val}\ l : \tau_1)$ and $\Delta \vdash_X \theta_1(\tau_1) \equiv \tau$. We have two cases to consider.

1. When p is not a module variable. By Lemma 40, $\Delta \vdash \theta(p).l \mapsto (\theta \circ \theta_1, \texttt{val}\ l : \tau_1)$ with $MVars(\theta_1) \subseteq dom(\theta)$, hence $\Delta; \emptyset \vdash_X \theta(p).l : \theta \circ \theta_1(\tau_1)$. By the absence of free module variables in Δ and $MVars(\theta_1) \subseteq dom(\theta)$, $MVars(\theta_1(\tau_1)) \subseteq dom(\theta)$.

2. When $p = X$. Then $\theta_1 = \epsilon$ and we have the claim by the well-formedness of θ.

□

Lemma 59 *Suppose* $\Delta \vdash p$ lctd *and* $\Delta \vdash_X \theta$ wf *and* $MVars(p) \cup MVars(B) \subseteq dom(\theta)$. *If* $\Delta \vdash_X p \triangleright B$, *then* $\Delta \vdash_X \theta(p) \triangleright \theta(B)$.

Proof. By case on B.

1. Suppose $B = \texttt{val}\ l : \tau$. By hypothesis, $\Delta; \Gamma \vdash_X p.l : \tau$. By Lemma 58, there is τ' such that $\Delta \vdash_X \tau \equiv \tau'$ and $MVars(\tau') \subseteq dom(\theta)$ and $\Delta; \Gamma \vdash_X \theta(p).l : \theta(\tau')$. By Lemma 54, $\Delta \vdash_X \theta(\tau) \equiv \theta(\tau')$, which concludes $\Delta \vdash_X \theta(p) \triangleright \texttt{val}\ l : \theta(\tau)$.

2. Suppose $B = \texttt{type}\ t$. By hypothesis, $\Delta \vdash p.t \mapsto (\theta_1, C)$. By Lemma 56, $\Delta \vdash \theta(p).t \mapsto (\theta_2, C_2)$.

3. When $B = \texttt{type}\ t = \tau$. By hypothesis, $\Delta \vdash_X p.t \equiv \tau$. By Lemma 54, $\Delta \vdash_X \theta(p.t) \equiv \theta(\tau)$, which concludes $\Delta \vdash_X \theta(p) \triangleright \texttt{type}\ t = \theta(\tau)$.

4. When $B = \texttt{datatype}\ t = c\ \texttt{of}\ \tau$. By hypothesis, $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau')$ with $\Delta \vdash_X \tau \equiv \tau'$. By Lemma 57, $\Delta \vdash_X cnstrlkup(\theta(p), c) = (t, \tau_2)$ with $MVars(\tau') \subseteq dom(\theta)$ and $\Delta \vdash_X \theta(\tau') \equiv \tau_2$. By Lemma 54 $\Delta \vdash_X \theta(\tau) \equiv \tau'$. By Corollary 4, $\Delta \vdash_X \theta(\tau) \equiv \tau_2$, from which the claim follows.

□

Lemma 60 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(p) \subseteq dom(\theta)$. *If* $\Delta \vdash_X p$ wf, *then* $\Delta \vdash_X \theta(p)$ wf.

Proof. By induction on the derivation of $\Delta \vdash_X p$ wf and by case on the last rule used. We show the main case where $p = p_1(p_2)$.
By $\Delta \vdash_X p_1(p_2)$ wf in the hypothesis, we have $\Delta \vdash_X p_1$ wf and $\Delta \vdash_X p_2$ wf and $\Delta \vdash p_1 \mapsto (\theta_1, \texttt{functor}(X : \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}) \to K)$ and, for all i in $1, \ldots, n$, $\Delta \vdash_X p_2 \triangleright \theta_1[X \mapsto p_2](B_i)$. Since Δ does not contain free module variables, $MVars(B_i) \subseteq dom(\theta_1) \cup \{X\}$. By induction hypothesis, $\Delta \vdash_X \theta(p_1)$ wf and $\Delta \vdash_X \theta(p_2)$ wf. Due to the first-order structure restriction, p_1 cannot be a module variable. By Lemma 40, $\Delta \vdash \theta(p_1) \mapsto (\theta \circ \theta_1, \texttt{functor}(X : \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}) \to K)$ and $MVars(\theta_1) \subseteq dom(\theta)$. For any i in $1, \ldots, n$, $MVars(\theta_1[X \mapsto p_2](B_i)) \subseteq dom(\theta)$, by $MVars(\theta_1) \subseteq dom(\theta)$, $MVars(p_2) \subseteq dom(\theta)$, and $MVars(B_i) \subseteq dom(\theta_1) \cup \{X\}$. By Lemma 59, we conclude $\Delta \vdash_X \theta(p_2) \triangleright (\theta \circ \theta_1)[X \mapsto \theta(p_2)](B_i)$ for all i in $1, \ldots, n$. □

Lemma 61 *Suppose* $\Delta \vdash_X \theta$ wf *and* $MVars(\tau) \subseteq dom(\theta)$. *If* $\Delta \vdash_X \tau \diamond$, *then* $\Delta \vdash_X \theta(\tau) \diamond$.

Proof. By induction on the derivation of $\Delta \vdash_X \tau \diamond$ and by case on the last rule used. We show the main case where $\tau = p.t$. By hypothesis, we have $\Delta \vdash_X p$ wf and $\Delta \vdash p.t \mapsto (\theta', C)$. We have $\Delta \vdash_X \theta(p)$ wf by Lemma 60, and $\Delta \vdash \theta(p).t \mapsto (\theta'', C')$ by Lemma 56. □

Definition 27 *A type environment* Γ *is well-formed w.r.t. a variable environment* Δ, *written* $\Delta \vdash_X \Gamma$ wf, *if, for all* x *in* $dom(\Gamma)$, $\Delta \vdash_X \Gamma(x) \diamond$.

Lemma 62 *Suppose* $\Delta \vdash \Gamma_1$ wf *and* $\Delta \vdash \Gamma_2$ wf *and* $dom(\Gamma_1) = dom(\Gamma_2)$ *and, for all* x *in* $dom(\Gamma_1)$, $\Delta \vdash_X \Gamma_1(x) \equiv \Gamma_2(x)$. *If* $\Delta; \Gamma_1 \vdash_X e : \tau$, *then* $\Delta; \Gamma_2 \vdash_X e : \tau$.

Proof. By induction on the derivation of $\Delta; \Gamma_1 \vdash_X e : \tau$ and by case on the last rule used. The main case is where $e = x$ and $\Delta; \Gamma_1 \vdash_X e : \tau$ is deduced from $\Gamma_1(x) = \tau$ (rule (96)). By $dom(\Gamma_1) = dom(\Gamma_2)$ in the hypothesis, $\Delta; \Gamma_2 \vdash_X e : \Gamma_2(x)$. By $\Delta \vdash_X \Gamma_1(x) \equiv \Gamma_2(x)$ and $\Delta \vdash \Gamma_1$ wf in the hypothesis, we deduce $\Delta; \Gamma_2 \vdash_X e : \tau$. □

Definition 28 *A variable environment* $\Delta = (\nu, \mu)$ *is well-formed, written* $\Delta \vdash \Delta$ wf, *if the following two conditions hold.*

1. *For all* X *in* $dom(\nu)$, $\Delta \vdash_X \nu(X) : \nu(X)$.
2. *For all* Z *in* $dom(\mu)$, *let* $\mu(Z) = (T, \Lambda)$. *There is a derivation for* $\Delta \vdash_X T : T$, *where we replace the typing rule (77) in Figure 65 with the following rule by removing subtyping checking.*

$$\frac{\Delta \vdash_X K : T \quad \Delta \vdash_X S : S'}{\Delta \vdash_X (K : S) : (T : S')}$$

Lemma 63 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X p_1$ wf *and* $\Delta \vdash p_1.t \to \tau$, *then* $\Delta \vdash_X \tau \diamond$.

Proof. By $\Delta \vdash p_1.t \to \tau$, we have $\tau = \theta(\tau')$ with either $\Delta \vdash p_1.t \mapsto (\theta, \texttt{type}\ t = \tau')$ or $\Delta \vdash p_1.t \mapsto (\theta, \texttt{datatype}\ t = \tau' = c\ \texttt{of}\ \tau'')$. By $\Delta \vdash_X \Delta$ wf, $\Delta \vdash_X \tau' \diamond$. Since $\Delta \vdash_X p_1$ wf, we have $\Delta \vdash_X \theta$ wf. When p_1 is a module variable then $\theta = \epsilon$ and $\tau = \tau'$. Suppose p_1 is not a module variable. Since Δ does not contain free module variables, $MVars(\tau') \subseteq dom(\theta)$. By Lemma 61, $\Delta \vdash_X \theta(\tau') \diamond$. □

Lemma 64 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X \tau \diamond$ *and* $\Delta \vdash \tau \rightarrow \tau'$*, then* $\Delta \vdash_X \tau' \diamond$*.*

Proof. By induction on the derivation of $\Delta \vdash \tau \rightarrow \tau'$. Use Lemma 63 for the case where τ is a type path. □

Corollary 5 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X \tau \diamond$ *and* $\Delta \vdash \tau \Rightarrow \tau'$*, then* $\Delta \vdash_X \tau' \diamond$*.*

Lemma 65 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X p_1$ wf *and* $\Delta \vdash p_1.t_1 \rightarrow \tau$ *and* $\Delta \vdash_X cnstrlkup(p_1, c) = (t_1, \tau_1)$*, then* $\tau = p_2.t_2$ *and* $\Delta \vdash_X p_2.t_2 \diamond$ *and* $\Delta \vdash_X cnstrlkup(p_2, c) = (t_2, \tau_2)$ *with* $\Delta \vdash_X \tau_1 \equiv \tau_2$*.*

Proof. By Corollary 5, $\Delta \vdash_X \tau \diamond$. By $\Delta \vdash_X \Delta$ wf and $\Delta \vdash_X cnstrlkup(p_1, c) = (t, \tau_1)$ and $\Delta \vdash p_1.t \rightarrow \tau$, we have $\Delta \vdash p_1.t \mapsto (\theta_1, \texttt{datatype}\ t = p_3.t_3 = c\ \texttt{of}\ \tau_3)$ and $\tau = \theta(p_3.t_3)$ and $\tau_1 = \theta_1(\tau_3)$. By $\Delta \vdash_X \Delta$ wf, $\Delta \vdash p_3.t_3 \mapsto (\theta_2, \texttt{datatype}\ t_3 = p_4.t_4 = c\ \texttt{of}\ \tau_4)$ with $\Delta \vdash_X \tau_3 \equiv \theta_2(\tau_4)$. When p_1 is a module variable, then we immediately have the claim. Suppose p_1 is not a module variable. Since Δ does not contain free module variables, $MVars(p_3) \cup MVars(\tau_3) \subseteq dom(\theta_1)$. By $\Delta \vdash_X p_1$ wf, $\Delta \vdash_X \theta_1$ wf. By Lemma 57, $\Delta \vdash_X cnstrlkup(\theta_1(p_3), c) = (t_3, \tau_5)$ with $MVars(\theta_2(\tau_4)) \subseteq dom(\theta_1)$ and $\Delta \vdash_X \tau_5 \equiv \theta_1(\theta_2(\tau_4))$. By Corollary 4 and Lemma 54, $\Delta \vdash_X \tau_5 \equiv \tau_1$. □

Corollary 6 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X p_1$ wf *and* $\Delta \vdash_X cnstrlkup(p_1, c) = (t_1, \tau_1)$ *and* $\Delta \vdash p_1.t_1 \Rightarrow \tau$*, then* $\tau = p_2.t_2$ *and* $\Delta \vdash_X p_2.t_2 \diamond$ *and* $\Delta \vdash_X cnstrlkup(p_2, c) = (t_2, \tau_2)$ *with* $\Delta \vdash_X \tau_1 \equiv \tau_2$*.*

Lemma 66 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X p$ wf *and* $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau)$*, then* $\Delta \vdash_X \tau \diamond$*.*

Proof. $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau)$ implies $\Delta \vdash p.t \mapsto (\theta, \texttt{datatype}\ t\ [= p_1.t_1] = \tau')$ and $\tau = \theta(\tau')$. When p is a module variable, then the claim follows immediately from $\Delta \vdash_X \Delta$ wf. When p not a module variable, then, since $\Delta \vdash_X \theta$ wf, the claim follows from $\Delta \vdash_X \Delta$ wf and Lemma 61. □

Lemma 67 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X \Gamma$ wf *and* $\Delta; \Gamma \vdash_X e : \tau$*, then* $\Delta \vdash_X \tau \diamond$*.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash_X e : \tau$ and by case on the last rule used. □

Lemma 68 *If* $\Delta; \Gamma \vdash_X e : \tau$ *and* x *is not in* $dom(\Gamma)$*, then* $\Delta; \Gamma, x : \tau' \vdash_X e : \tau$*.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash_X e : \tau$ and by case on the last rule used. □

Lemma 69 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X \Gamma$ wf *and* $\Delta \vdash_X \Gamma_1$ wf *and* $\Delta \vdash_X \theta$ wf *and* $MVars(e) \subseteq dom(\theta)$*. Suppose also that* $dom(\Gamma) = dom(\Gamma_1)$ *and, for all* x *in* $dom(\Gamma)$*, there is* τ *such that* $\Delta \vdash_X \Gamma(x) \equiv \tau$ *and* $MVars(\tau) \subseteq dom(\theta)$ *and* $\Delta \vdash_X \theta(\tau) \equiv \Gamma_1(x)$*. If* $\Delta; \Gamma \vdash_X e : \tau$*, then there is* τ' *such that* $\Delta \vdash_X \tau \equiv \tau'$ *and* $MVars(\tau') \subseteq dom(\theta)$ *and* $\Delta; \Gamma_1 \vdash_X \theta(e) : \theta(\tau')$*.*

Proof. By induction on the derivation of $\Delta; \Gamma \vdash_X e : \tau$ and by case on the last rule used. We show the main cases.
rule (94) Suppose $\Delta; \Gamma \vdash_X e : \tau_1$ and $\Delta \vdash_X \tau \equiv \tau_1$ and $\Delta \vdash_X \tau \diamond$. By induction hypothesis, there is τ_2 such that $\Delta \vdash_X \tau_1 \equiv \tau_2$ and $MVars(\tau_2) \subseteq dom(\theta)$ and $\Delta; \Gamma_1 \vdash_X \theta(e) : \theta(\tau_2)$. By transitivity of the type equivalence (Corollary 4), $\Delta \vdash_X \tau \equiv \tau_2$.
rule (99) Suppose $e = (\lambda x.e_1 : \tau)$ and $\Delta \vdash_X \tau \diamond$ and $\Delta \vdash_X \tau \equiv \tau_1 \to \tau_2$ and $\Delta \vdash_X \tau_1 \to \tau_2 \diamond$ and $\Delta; \Gamma, x : \tau_1 \vdash_X e_1 : \tau_2$. By Lemma 61, $\Delta \vdash_X \theta(\tau) \diamond$. By $\Delta \vdash_X \tau \equiv \tau_1 \to \tau_2$, there is τ_3 and τ_4 such that $\Delta \vdash \tau \Rightarrow \tau_3 \to \tau_4$ and $\Delta \vdash \tau_1 \Rightarrow \tau_3$ and $\Delta \vdash \tau_2 \Rightarrow \tau_4$. By Lemma 53, $\Delta \vdash_X \theta(\tau) \equiv \theta(\tau_3 \to \tau_4)$ with $MVars(\tau_3) \cup MVars(\tau_4) \subseteq dom(\theta)$. By Corollary 5 and Lemma 61, $\Delta \vdash_X \theta(\tau_3) \diamond$. Since $\Delta \vdash_X \tau_1 \equiv \tau_3$, by induction hypothesis, there is τ_5 such that $\Delta \vdash_X \tau_2 \equiv \tau_5$ and $MVars(\tau_5) \subseteq dom(\theta)$ and $\Delta; \Gamma_1, x : \theta(\tau_3) \vdash_X \theta(e_1) : \theta(\tau_5)$. By Corollary 4 and Lemma 54, $\Delta \vdash_X \theta(\tau_5) \equiv \theta(\tau_4)$. By Corollary 5 and Lemma 61, $\Delta \vdash_X \theta(\tau_4) \diamond$, by which and the rule (94) we deduce $\Delta; \Gamma_1 \vdash_X \theta(\lambda x.e_1 : \tau) : \theta(\tau)$.
rule (101) Suppose $e = p.c\ e_1$ and $\Delta \vdash_X p$ wf and $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau_1)$ and $\Delta; \Gamma \vdash_X e_1 : \tau_1$ and $\tau = p.t$. We have $\Delta \vdash_X \theta(p)$ wf by Lemma 60, and $\Delta \vdash_X cnstrlkup(\theta(p), c) = (t, \tau_3)$ with $MVars(\tau_1) \subseteq dom(\theta)$ and $\Delta \vdash_X \theta(\tau_1) \equiv \tau_3$ by Lemma 57. By induction hypothesis, there is τ_4 such that $\Delta \vdash_X \tau_2 \equiv \tau_4$ and $MVars(\tau_4) \subseteq dom(\theta)$ and $\Delta; \Gamma_1 \vdash_X \theta(e_1) : \theta(\tau_4)$. By Corollary 4 and Lemma 54 and 66 and the rule (94), $\Delta; \Gamma_1 \vdash_X \theta(e_1) : \tau_3$, with which we deduce $\Delta; \Gamma_1 \vdash_X \theta(p.c\ e_1) : \theta(p).t$

$$\frac{\Delta \vdash p \mapsto_{er} (\theta, \texttt{ss} \ldots \texttt{val}\ l : \tau \ldots \texttt{end})}{\Delta \vdash p.l \mapsto_{er} (\theta, \texttt{val}\ l : \tau)} \quad \frac{\Delta \vdash p \mapsto_{er} (\theta, \texttt{ss} \ldots \texttt{val}\ l = e \ldots \texttt{end})}{\Delta \vdash p.l \mapsto_{er} (\theta, \texttt{val}\ l = e)}$$

Figure 72: Erasure look-up for value paths

rule (102) Suppose $e = \texttt{case}\ e_1\ \texttt{of}\ p.c\ x \Rightarrow\ e_2$ and $\Delta \vdash_X p$ wf and $\Delta; \Gamma \vdash_X e_1 : p.t$ and $\Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau_2)$ and $\Delta; \Gamma, x : \tau_2 \vdash_X e_2 : \tau$. We have $\Delta \vdash_X \theta(p)$ wf by Lemma 60. By induction hypothesis, there is τ_3 such that $\Delta \vdash_X p.t \equiv \tau_3$ and $\mathit{MVars}(\tau_3) \subseteq \mathit{dom}(\theta)$ and $\Delta; \Gamma_1 \vdash_X \theta(e_1) : \theta(\tau_3)$. By Lemma 61 and 54, $\Delta; \Gamma_1 \vdash_X \theta(e_1) : \theta(p).t$. By Lemma 57, $\Delta \vdash_X \mathit{cnstrlkup}(\theta(p), c) = (t, \tau_3)$ with $\mathit{MVars}(\tau_2) \subseteq \mathit{dom}(\theta)$ and $\Delta \vdash_X \tau_3 \equiv \theta(\tau_2)$. We have the claim by induction hypothesis.
rule (103) By Lemma 58 and 68. □

As we we did for the look-up judgment, we extend the erasure look-up judgment for value paths in Figure 72.

Lemma 70 *Suppose* $\Delta \vdash_X \theta_1$ wf *and* $\Delta \vdash_X \theta_2$ wf *and* $\mathit{MVars}(\theta_2) \subseteq \mathit{dom}(\theta_1)$, *then* $\Delta \vdash_X \theta_1 \circ \theta_2$ wf.

Proof. By Lemma 60, for all X in $\mathit{dom}(\theta_2)$, $\Delta \vdash_X \theta_1 \circ \theta_2(X)$ wf. For any X in $\mathit{dom}(\theta_2)$, let $\Delta(X) = \texttt{sig}\ B_1 \ldots B_n\ \texttt{end}$. By $\Delta \vdash_X \theta_2$ wf, for all i in $1, \ldots, n$, $\Delta \vdash_X \theta_2(X) \triangleright \theta_2(B_i)$. By Lemma 59, $\Delta \vdash_X \theta_1 \circ \theta_2(X) \triangleright \theta_1 \circ \theta_2(B_i)$. □

Definition 29 *A type path p.t is stable w.r.t. a variable environment* Δ *if* $\Delta \vdash p.t \mapsto (\theta, \texttt{datatype}\ t\ [= p_1.t_1] = c\ \texttt{of}\ \tau)$ *holds.*

Lemma 71 *For any types* τ_1, τ_2 *and stable type p.t w.r.t.* Δ, *there are no derivations for*

1. $\Delta; \Gamma \vdash_X () : \tau_1 * \tau_2$
2. $\Delta; \Gamma \vdash_X () : \tau_1 \to \tau_2$
3. $\Delta; \Gamma \vdash_X () : p.t.$
4. $\Delta; \Gamma \vdash_X (v_1, v_2) : \tau_1 \to \tau_2.$
5. $\Delta; \Gamma \vdash_X (v_1, v_2) : \texttt{1}.$

6. $\Delta; \Gamma \vdash_X (v_1, v_2) : p.t.$

7. $\Delta; \Gamma \vdash_X (\lambda x.e : \tau) : \tau_1 * \tau_2.$

8. $\Delta; \Gamma \vdash_X (\lambda x.e : \tau) : \mathtt{1}.$

9. $\Delta; \Gamma \vdash_X (\lambda x.e : \tau) : p.t.$

10. $\Delta; \Gamma \vdash_X p.c\ v : \tau_1 \to \tau_2$

11. $\Delta; \Gamma \vdash_X p.c\ v : \tau_1 * \tau_2$

12. $\Delta; \Gamma \vdash_X p.c\ v : \mathtt{1}$

Proof. Observe that for any types τ_1, τ_2 and stable type $p.t$ *w.r.t.* Δ, none of $\Delta \vdash_X \tau_1 * \tau_2 \equiv \mathtt{1}$, $\Delta \vdash_X \tau_1 \to \tau_2 \equiv \mathtt{1}$, $\Delta \vdash_X \tau_1 \to \tau_2 \equiv p.t$, $\Delta \vdash_X \tau_1 \to \tau_2 \equiv \tau_1 * \tau_2$ $\Delta \vdash_X \tau_1 * \tau_2 \equiv p.t$ or $\Delta \vdash_X \tau_1 \to \tau_2 \equiv p.t$ holds. □

Lemma 72 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta; \emptyset \vdash_X v : p.t$ *and* $\Delta \vdash_X cnstrlkup(p, c) = (t, \tau)$*, then* $v = p_1.c\ v_1$ *with* $\Delta; \emptyset \vdash_X v_1 : \tau$ *for some* p_1 *and* v_1*.*

Proof. By Lemma 71, $v = p_1.c_1\ v_1$ for some p_1 and v_1. By $\Delta; \emptyset \vdash_X p_1.c_1\ v_1 : p.t$ and Lemma 55, $\Delta \vdash_X cnstrlkup(p_1, c_1) = (t_1, \tau_1)$ and $\Delta; \emptyset \vdash_X v_1 : \tau_1$ and $\Delta \vdash_X p_1.t_1 \equiv p.t$. Hence, there is τ_3 such that $\Delta \vdash p_1.t_1 \Rightarrow \tau_3$ and $\Delta \vdash p.t \Rightarrow \tau_3$. By Lemma 67, $\Delta \vdash_X p$ wf and $\Delta \vdash_X p_1$ wf. By Corollary 6, $c = c_1$ and $\Delta \vdash_X \tau_1 \equiv \tau$. We deduce $\Delta; \emptyset \vdash_X v_1 : \tau$ by Lemma 66 and the rule (94). □

Lemma 73 *Suppose* $\Delta \vdash_X \Delta$ wf *and* $\Delta \vdash_X \Gamma$ wf *and* $\Delta \vdash_X \tau' \diamond$ *and* $\Delta; \Gamma, x : \tau' \vdash_X e : \tau$ *and* $\Delta; \Gamma \vdash_X e' : \tau'$*, then* $\Delta; \Gamma \vdash_X [x \mapsto e']e : \tau$*.*

Proof. By induction on the derivation of $\Delta; \Gamma, x : \tau' \vdash_X e : \tau$ and by case on the last rule used. □

Proposition 23 (Progress in *TraviataX*) *Suppose that a lazy program type* U *conforms with a variable environment* Δ *and that* U *is a correct type of a program* P *w.r.t.* Δ *in TraviataX and* $\Delta \vdash_X \Delta$ wf*. Let* μ *be the erasure environment of* P*. If* $\Delta; \emptyset \vdash_X e : \tau$ *and* $MVars(e) = \emptyset$*, then either* e *is a value or else there is some* e' *with* $(\mu, \nu_\epsilon) \vdash e \to e'$*. Particularly, if* $\Delta; \emptyset \vdash_X p.l : \tau$ *and* $MVars(p) = \emptyset$*, then* $(\mu, \nu_\epsilon) \vdash p.l \overset{\mathtt{vpth}}{\to} e'$*.*

Proof. By induction on the derivation of $\Delta; \emptyset \vdash_X e : \tau$. We show the main cases.
Suppose $e = p.l$. By $\Delta; \emptyset \vdash_X p.l : \tau$ in the hypothesis, $\Delta \vdash p.l \mapsto (\theta, \texttt{val}\ l : \tau_1)$. Since U conforms with Δ, $\Delta_U \vdash p.l \mapsto (\theta, \texttt{val}\ l : \tau_1)$. By $\Delta \vdash_X P : U$, $(\mu_{,}\nu_\epsilon) \vdash p.l \mapsto_{er} (\theta_1, \texttt{val}\ l = e_1)$, from which $(\mu_{,}\nu_\epsilon) \vdash p.l \overset{\texttt{vpth}}{\rightarrow} \theta_1(e_1)$ follows.
Suppose $e = e_1(e_2)$. When either e_1 or e_2 is not a value, then by induction hypothesis, we have $(\mu_{,}\nu_\epsilon) \vdash e_1 \rightarrow e_1'$ or $(\mu_{,}\nu_\epsilon) \vdash e_2 \rightarrow e_2'$. Suppose e_1 and e_2 are values v_1 and v_2, respectively. By well-typedness of $v_1(v_2)$, $\Delta; \emptyset \vdash_X v_1 : \tau_1 \rightarrow \tau_2$. By Lemma 71, $v_1 = (\lambda x.e_3 : \tau_3)$, hence we have $(\mu_{,}\nu_\epsilon) \vdash (\lambda x.e_3 : \tau_3)v_2 \overset{\texttt{fun}}{\rightarrow} [x \mapsto v_2]e_3$.
Suppose $e = \texttt{case}\ e_1\ \texttt{of}\ p.c\ x \Rightarrow e_2$. When e_1 is not a value, then by induction hypothesis $(\mu_{,}\nu_\epsilon) \vdash e_1 \rightarrow e_1'$. Suppose e_1 is a value v_1. By well-typedness of e, $\Delta; \emptyset \vdash_X v_1 : p.t$ where $\Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau_1)$. By Lemma 72, $v_1 = p_1.c\ v_2$ for some p_1 and v_2. Now we have $(\mu_{,}\nu_\epsilon) \vdash \texttt{case}\ p_1.c\ v_2\ \texttt{of}\ p.c\ x \Rightarrow e_2 \rightarrow [x \mapsto v_2]e_2$ □

Proposition 24 (Subject reduction in *TraviataX*) *Suppose that a lazy program type U conforms with a variable environment Δ and that U is a correct type of a program P w.r.t. Δ in TraviataX and $\Delta \vdash_X \Delta$* wf. *Let μ be the erasure environment of P. If $\Delta; \emptyset \vdash_X e : \tau$ and $\mathit{MVars}(e) = \emptyset$ and $(\mu_{,}\nu_\epsilon) \vdash e \rightarrow e'$, then $\Delta; \emptyset \vdash_X e' : \tau$ with $\mathit{MVars}(e') = \emptyset$.*

Proof. By induction on the derivation of $(\mu_{,}\nu_\epsilon) \vdash e \rightarrow e'$ and by case on the last rule used. We show the main cases.
Suppose e is a value path $p.l$ and $(\mu_{,}\nu_\epsilon) \vdash p.l \mapsto_{er} (\theta, \texttt{val}\ l = e_1)$ and $e' = \theta(e_1)$. Since P does not contain free module variables, $\mathit{MVars}(e') = \emptyset$. Let μ' be the erasure environment of U and $(\mu', \nu_\epsilon) \vdash p.l \mapsto_{er} (\theta, \texttt{val}\ l : \tau_1)$ and $\Delta_U \vdash p.l \mapsto (\theta_1, \texttt{val}\ l : \tau_2)$. By Lemma 55, $\Delta \vdash_X \tau \equiv \theta_1(\tau_2)$. By $\Delta \vdash_X P : U$, $\Delta; \emptyset \vdash_X e_1 : \tau_1$. Let $\mathit{dom}(\theta) = \{\{X_1, \ldots, X_n\}\}$ and $\mathit{dom}(\theta_1) = \{\{X_1', \ldots, X_n'\}\}$ and $\theta_2 = [X_1 \mapsto X_1', \ldots, X_n \mapsto X_n']$. We have $\theta = \theta_1 \circ \theta_2$. By $\Delta \vdash_X P : U$ and Lemma 70, $\Delta \vdash_X \theta_1 \circ \theta_2$ wf. By Lemma 69, there is τ_3 such that $\Delta \vdash_X \tau_1 \equiv \tau_3$ and $\mathit{MVars}(\tau_3) \subseteq \mathit{dom}(\theta_1 \circ \theta_2)$ and $\Delta; \emptyset \vdash_X \theta_1 \circ \theta_2(e_1) : \theta_1 \circ \theta_2(\tau_3)$, which means $\Delta; \emptyset \vdash_X \theta(e_1) : \theta(\tau_3)$. By $\Delta \vdash_X P : U$, $\Delta \vdash_X \theta_2(\tau_1) \equiv \tau_2$. By Lemma 54, $\Delta \vdash_X \theta(\tau_3) \equiv \tau$. By Lemma 67, we deduce $\Delta; \emptyset \vdash_X \theta(e) : \tau$.
Suppose $e = \texttt{case}\ v\ \texttt{of}\ p.c\ x \Rightarrow e_2$. By $\Delta; \emptyset \vdash_X e : \tau$ and Lemma 55, $\Delta \vdash_X p$ wf and $\Delta; \emptyset \vdash_X v : p.t$ and $\Delta \vdash_X \mathit{cnstrlkup}(p, c) = (t, \tau_1)$ and $\Delta; x : \tau_1 \vdash_X e_2 : \tau'$ with $\Delta \vdash_X \tau' \equiv \tau$. By Lemma 72, $v_1 = p_1.c\ v_2$ for some

p_1 and v_2 with $\Delta; \emptyset \vdash_X v_2 : \tau_1$. By Lemma 73, $\Delta; \emptyset \vdash_X [x \mapsto v]e_2 : \tau'$, by which and Lemma 67 we conclude $\Delta; \emptyset \vdash_X [x \mapsto v]e_2 : \tau$. □

12.1.3 From *Traviata* to *TraviataX*

In this section, we convert well-typedness in *Traviata* to well-typedness in *TraviataX*. As we explained in the beginning of Section 12.1.2, type equivalence relations and ways of type checking a sealing construct are two notable differences between *Traviata* and *TraviataX*. We prove in Corollary 7 that the type equivalence relation in *Traviata* is included in that of *TraviataX*. To bridge the latter gap, we provide *TraviataX* the fully manifest variable environment that already contains all type equality constraint added by the manifestation operation during type-correctness check in *Traviata*. This variable environment is built in a straightforward way by, roughly, applying the function *manif* throughout the reconstructed lazy program type of a program.

TraviataX requires well-typed programs to only contain module paths in located form. Hence we need expand module paths beforehand. Moreover, it does not have the ability to instantiate lazy path types during subtyping checking. This requires us to inline expand module abbreviations into structures and functors that the abbreviating paths refer to.

Our operational semantics evaluates modules in a lazy way, that is, it only evaluates components of modules that are accessed and functor application does not trigger any reductions. Hence, neither expansion of module paths nor inline path expansion has any impact on the semantics of programs.

Full manifestation of type specifications When type checking a sealing construct $(E : S)$, *Traviata* enriched the variable environment to make manifest abstract type and datatype specifications in the sealing signature S. To build fully manifest variable environments, we apply a similar operation throughout reconstructed lazy program types using the function *fullmanif* defined in Figure 73.

We used *manif* to add type equality constraint locally to a single sealing construct, whereas we do *fullmanif* to add the constraint globally throughout a lazy program type. The function *fullmanif* traverses the constituents of a lazy program type. The behavior of *manifX* and *updateX* is identical to that of *manif* and *update* in Figure 51 respectively, except for the $(\star)$-labeled rule. To make a datatype specification manifest, *updateX* introduces a manifest

$$
\begin{array}{ll}
 & \mathit{fullmanif}(\texttt{sig}\ (Z^\theta)\ C_1 \ldots C_n\ \texttt{end}) \\
 & = \texttt{sig}\ (Z^\theta)\ \mathit{fullmanif}(C_1) \ldots \mathit{fullmanif}(C_n)\ \texttt{end} \\
 & \mathit{fullmanif}(\texttt{sig}\ C_1 \ldots C_n\ \texttt{end}) = \texttt{sig}\ \mathit{fullmanif}(C_1) \ldots \mathit{fullmanif}(C_n)\ \texttt{end} \\
 & \mathit{fullmanif}(\texttt{functor}(X : S) \rightarrow\ T) = \texttt{functor}(X : S) \rightarrow\ \mathit{fullmanif}(T) \\
 & \mathit{fullmanif}((T : S)) = (\mathit{fullmanif}(T){:}\mathit{manifX}(T, S)) \\
 & \mathit{fullmanif}(p) = p \\
 & \mathit{fullmanif}(\texttt{module}\ M : T) = \texttt{module}\ M : \mathit{fullmanif}(T) \\
 & \mathit{fullmanif}(C) = C \ \text{ when } C \text{ is not a lazy module specification} \\
 & \mathit{manifX}(\texttt{ss}\ (Z^\theta)\ \ldots \texttt{end}, TS) = \mathit{updataX}(Z^\theta, TS) \\
 & \mathit{manifX}((TT : \texttt{sig}\ (Z^\theta) \ldots \texttt{end}), TS) = \mathit{updataX}(Z^\theta, TS) \\
 & \mathit{manifX}(p, TS) = \mathit{updataX}(p, TS) \\
 & \mathit{updateX}(p, \texttt{sig}\ (Z^\theta)\ B_1 \ldots B_m\ \texttt{end}) \\
 & = \texttt{sig}\ (Z^\theta)\ \mathit{updataX}(p, B_1) \ldots \mathit{updataX}(p, B_m)\ \texttt{end} \\
 & \mathit{updateX}(p, \texttt{sig}\ B_1 \ldots B_m\ \texttt{end}) \\
 & = \texttt{sig}\ \mathit{updataX}(p, B_1) \ldots \mathit{updataX}(p, B_m)\ \texttt{end} \\
 & \mathit{updateX}(p, \texttt{functor}(X : S) \rightarrow\ S') \\
 & = \texttt{functor}(X : S) \rightarrow \mathit{updateX}(p(X), S') \\
 & \mathit{updataX}(p, \texttt{module}\ M : S) = \texttt{module}\ M : \mathit{updateX}(p.M, S) \\
 & \mathit{updataX}(p, \texttt{type}\ t) = \texttt{type}\ t = p.t \\
 & \mathit{updataX}(p, \texttt{type}\ t = \tau) = \texttt{type}\ t = \tau \\
(\star) & \mathit{updataX}(p, \texttt{datatype}\ t = c\ \texttt{of}\ \tau) = \texttt{datatype}\ t = p.t = c\ \texttt{of}\ \tau \\
 & \mathit{updataX}(p, \texttt{val}\ l : \tau) = \texttt{val}\ l : \tau
\end{array}
$$

Figure 73: Full manifestation of type specifications

datatype specification, instead of a manifest specification. This avoids erasing the necessary type information.

Inline path expansion For every module expression in a program which seals a module path p with a signature S, we inline expand p into the structure or functor that p refers to so that after the expansion the nesting level inside the sealing becomes same as that of the sealing signature S.

In Figure 74, we define the inline path expansion operation on programs. We write $\Delta \vdash e \prec e'$ to denote that e' is obtained from e by expanding all module paths contained in e into located forms *w.r.t.* Δ. The judgment $\Delta \vdash K \prec K'$ means that the module description K inline path expands into the module description K' *w.r.t.* the variable environment Δ and the judgment $\Delta \vdash K \prec_S K'$ means that K does into K' along the signature S *w.r.t.* Δ. The other judgments are read similarly.

The ($\star$)-labeled rule uses two helper functions and is the only important rule in Figure 74. The function *outer* replaces every occurrence of a sealing construct $(T : S)$ with S in the given lazy signature. The function *inst* is defined in Figure 75. It instantiates module expressions from lazy signatures, by adapting their syntax (e.g. to turn the keyword `sig` to `struct`) and by reifying value specifications into value definitions (in the ($\star$)-labeled rule). The reification uses *inst*'s first argument, which keeps track of the location, by means of module paths, of the value specification to be reified. Let us return to the ($\star$)-labeled rule in Figure 74. To inline expand a module path p along a signature S, we instantiate a module expression from the lazy signature that p refers to. Since q is in located form, T is not a module path.

The inline path expansion operation on lazy program types is defined by the same inference rules as those in Figure 74, except that we replace the ($\star$)-labeled rule with the rule:

$$\frac{\Delta \vdash p \rightsquigarrow q \quad \Delta \vdash q \mapsto (\theta, T) \quad \Delta \vdash \mathit{outer}(\theta(T)) \sqsubset_S K}{\Delta \vdash p \sqsubset_S K}$$

For inline path expansion on lazy program types, we use judgments of the forms $\Delta \vdash K \sqsubset K'$ and $\Delta \vdash K \sqsubset_S K'$, instead of $\Delta \vdash K \prec K'$ and $\Delta \vdash K \prec_S K'$ respectively.

Lemma 74 *Suppose* $\Delta \vdash K : T$*, then* $\Delta \vdash T \sqsubset T'$ *for some* T'*.*

Proof. We prove if $\Delta \vdash T < S$ then $\Delta \vdash T \sqsubset_S T'$ for some T', by induction

$$\frac{\Delta \vdash J_1 \prec J_1' \ \ldots \ \Delta \vdash J_n \prec J_n'}{\Delta \vdash \mathtt{ss}\ (Z^\theta)\ J_1 \ldots J_n\ \mathtt{end} \prec \mathtt{ss}\ (Z^\theta)\ J_1' \ldots J_n'\ \mathtt{end}}$$

$$\frac{\Delta \vdash J_1 \prec J_1' \ \ldots \ \Delta \vdash J_n \prec J_n'}{\Delta \vdash \mathtt{ss}\ J_1 \ldots J_n\ \mathtt{end} \prec \mathtt{ss}\ J_1' \ldots J_n'\ \mathtt{end}}$$

$$\frac{\Delta \vdash S \prec S' \quad \Delta \vdash K \prec K'}{\Delta \vdash \mathtt{functor}(X : S) \to\ K \prec \mathtt{functor}(X : S') \to\ K'}$$

$$\frac{\Delta \vdash TS \prec S \quad \mathit{manif}(TK, TS) = TS' \quad \Delta(\mu_{TS'}, \nu_\epsilon) \vdash TK \prec_{TS} K}{\Delta \vdash (TK : TS) \prec (K : S)}$$

$$\frac{\Delta \vdash p \rightsquigarrow q}{\Delta \vdash p \prec q}$$

$$\frac{\Delta \vdash K \prec K'}{\Delta \vdash \mathtt{module}\ M := K \prec \mathtt{module}\ M := K'}$$

$$\frac{}{\Delta \vdash \mathtt{type}\ t \prec \mathtt{type}\ t} \quad \frac{\Delta \vdash \tau \downarrow \tau'}{\Delta \vdash \mathtt{type}\ t = \tau \prec \mathtt{type}\ t = \tau'}$$

$$\frac{\Delta \vdash \tau \downarrow \tau'}{\Delta \vdash \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau \prec \mathtt{datatype}\ t = c\ \mathtt{of}\ \tau'}$$

$$\frac{\Delta \vdash \tau \downarrow \tau'}{\Delta \vdash \mathtt{val}\ l : \tau \prec \mathtt{val}\ l : \tau'} \quad \frac{\Delta \vdash e \prec e'}{\Delta \vdash \mathtt{val}\ l = e \prec \mathtt{val}\ l = e'}$$

$$\frac{\Delta \vdash TS \prec S \quad \Delta \vdash TK \prec K}{\Delta \vdash (TK : TS) \prec_{S_1} (K : S)}$$

$$\frac{\forall i \in \{1, \ldots, n\}, \Delta \vdash J_i \prec_{B_{\sigma(i)}} J_i' \text{ when } i \text{ exists, otherwise } \Delta \vdash J_i \prec J_i'}{\Delta \vdash \mathtt{ss}\ J_1 \ldots J_n\ \mathtt{end} \prec_{\mathtt{sig}\ [(Z_1^{\theta_1})]\ B_1 \ldots B_m\ \mathtt{end}} \mathtt{ss}\ J_1' \ldots J_n'\ \mathtt{end}}$$

$$\frac{\forall i \in \{1, \ldots, n\}, \Delta \vdash J_i \prec_{B_{\sigma(i)}} J_i' \text{ when } i \text{ exists, otherwise } \Delta \vdash J_i \prec J_i'}{\Delta \vdash \mathtt{ss}\ (Z^\theta)\ J_1 \ldots J_n\ \mathtt{end} \prec_{\mathtt{sig}\ [(Z_1^{\theta_1})]\ B_1 \ldots B_m\ \mathtt{end}} \mathtt{ss}\ (Z^\theta)\ J_1' \ldots J_n'\ \mathtt{end}}$$

$$\frac{\Delta \vdash S_1 \prec S_1' \quad \Delta \vdash K \prec_S K'}{\Delta \vdash \mathtt{functor}(X_1 : S_1) \to K \prec_{\mathtt{functor}(X_2 : S_2) \to S} \mathtt{functor}(X_1 : S_1') \to K'}$$

$$\frac{\Delta \vdash p \rightsquigarrow q \quad \Delta \vdash q \mapsto (\theta, T) \quad \Delta \vdash \mathit{inst}(q, \mathit{outer}(\theta(T))) \prec_S K}{\Delta \vdash p \prec_S K}\ (\star)$$

$$\frac{\Delta \vdash K \prec_S K'}{\Delta \vdash \mathtt{module}\ M := K \prec_{\mathtt{module}\ M : S} \mathtt{module}\ M := K'}$$

$$\frac{J \neq \mathtt{module}\ M := K \quad \Delta \vdash J \prec J'}{\Delta \vdash J \prec_B J'}$$

Figure 74: Inline path expansion

$$
\begin{array}{ll}
 & inst(p, \texttt{sig}\ (Z^\theta)\ C_1 \ldots C_n\ \texttt{end}) \\
 & = \texttt{struct}\ (Z^\theta)\ inst(p, C_1) \ldots inst(p, C_n)\ \texttt{end} \\
 & inst(p, \texttt{sig}\ C_1 \ldots C_n\ \texttt{end}) \\
 & = \texttt{struct}\ inst(p, C_1) \ldots inst(p, C_n)\ \texttt{end} \\
 & inst(p, \texttt{functor}(X : S) \rightarrow\ T) = \\
 & \texttt{functor}(X : S) \rightarrow\ inst(p(X), T) \\
 & inst(p, q) = q \\
 & inst(p, \texttt{module}\ M : T) = \texttt{module}\ M = inst(p.M, T) \\
 & inst(p, \texttt{type}\ t) = \texttt{type}\ t \\
 & inst(p, \texttt{type}\ t = \tau) = \texttt{type}\ t = \tau \\
 & inst(p, \texttt{datatype}\ t = c\ \texttt{of}\ \tau) = \texttt{datatype}\ t = c\ \texttt{of}\ \tau \\
(\star) & inst(p, \texttt{val}\ l : \tau) = \texttt{val}\ l = p.l
\end{array}
$$

Figure 75: Instantiation of module expressions

on the derivation of $\Delta \vdash T < S$. Then the lemma is proven by induction on the derivation of $\Delta \vdash K : T$. □

Lemma 75 *Suppose* $\Delta \vdash K : T$ *and* $\Delta \vdash T \sqsubset T'$*, then* $\Delta \vdash K \prec K'$ *for some* K'*.*

Proof. By induction on the structure of K. □

Correctness We prove in Corollary 7 that the type equivalence relation in *Traviata* is included in that of *TraviataX*. Then Proposition 25 states the main claim of this subsection.

Lemma 76 *Suppose that all module paths contained in* Δ *and* τ, τ' *are in located form w.r.t.* Δ *and that all types contained in* Δ *are located types w.r.t.* Δ*. If* $\Delta \vdash \tau \downarrow \tau'$*, then* $\Delta \vdash \tau \Rightarrow \tau'$*.*

Proof. By induction on the derivation of $\Delta \vdash \tau \downarrow \tau'$ and by case on the last rule used. We show the main case.
[**tnlz-abb**] Suppose $\tau = p.t$ and $\Delta \vdash p \rightsquigarrow p'$ and $\Delta \vdash p'.t \mapsto (\theta, \texttt{type}\ t = \tau_1)$ and $\Delta \vdash \tau_1 \downarrow \tau_2$ and $\Delta \vdash \theta(\tau_2) \downarrow \tau'$. Since located forms are invariant of the module path expansion (Lemma 41) , $p = p'$. Similarly, since located

types are invariant of the type expansion (Lemma 42), $\tau_1 = \tau_2$. By induction hypothesis, $\Delta \vdash \theta(\tau_2) \Rightarrow \tau'$. □

Corollary 7 *Suppose that all module paths contained in Δ and τ, τ' are in located form w.r.t. Δ and that all types contained in Δ are located types w.r.t. Δ. If $\Delta \vdash \tau \equiv \tau'$, then $\Delta \vdash_X \tau \equiv \tau'$.*

Lemma 77 *Let $\Delta_1 = (\mu, \nu_1)$ and $\Delta_2 = (\mu, \nu_2)$ be such that X is not in $dom(\nu_1)$ and $dom(\nu_2) = dom(\nu_1) \cup \{X\}$.*

1. *If $\Delta_1 \vdash_X K : T$, then If $\Delta_2 \vdash_X K : T$.*
2. *If $\Delta_1; \Gamma \vdash_X e : \tau$, then $\Delta_2; \Gamma \vdash_X e : \tau$.*
3. *If $\Delta_1 \vdash_X \tau \diamond$ then $\Delta_2 \vdash_X \tau \diamond$.*
4. *If $\Delta_1 \vdash_X p$ wf then $\Delta_2 \vdash_X p$ wf.*
5. *If $\Delta_1 \vdash_X p \triangleright B$ then $\Delta_2 \vdash_X p \triangleright B$.*
6. *If $\Delta_1 \vdash_X T < S$ then $\Delta_2 \vdash_X T < S$.*

Proof. By easy induction. □

Lemma 78 *Suppose T does not contain a sealing construct and $\Delta \vdash T : T$. For any θ in located form w.r.t. (μ, ν) such that $(\mu, \nu) \vdash \theta$ wf and $MVars(T) \subseteq dom(\theta)$, let T' be the lazy signature obtained from $\theta(T)$ by expanding all module paths and types into located forms and located types w.r.t. (μ, ν) and by renaming bound module variables so that all binding occurrences of module variables use distinct names and that $dom(\nu)$ and $dom(\nu_{T'})$ are disjoint, then $(\mu, \nu\nu_{T'}) \vdash T' : T'$.*

Proof. By induction on the structure of T. Use lemmas in Section 12.1.1. □

Proposition 25 *Suppose $\Delta_P \vdash P \triangleright U$ and $\Delta_U \vdash P : U$ and $\Delta_U \vdash P \prec P'$ and $\Delta_U \vdash U \prec U'$. Suppose that we have renamed bound module variables in U' so that all binding occurrences of module variables use distinct names. Correspondingly, suppose that we have renamed bound module variables in P' so that $\nu_{P'}$ and $\nu_{U'}$ coincide. Then we have $\Delta_{fullmanif(U)}(\mu_\epsilon, \nu_{U'}) \vdash_X P' : U'$.*

Proof. We say that a variable environment Δ is available in $\Delta_U \vdash P : U$, if the derivation of $\Delta_U \vdash P : U$ contains a judgment whose variable environment is Δ. Observe that $\Delta_{\mathit{fullmanif}(U)}$ contains all type equality constraints that any Δ available in $\Delta_U \vdash P : U$ contains. Hence Corollary 7 and Lemma 46 and 78 together prove by induction on the derivation of $\Delta_U \vdash P : U$ that, if we replace the rule (77) in Figure 65 with the rule:

$$\frac{\Delta \vdash_X E : T \quad \Delta \vdash_X S : S'}{\Delta \vdash_X (E : S) : (T : S')}$$

then there is a derivation for $\Delta_{\mathit{fullmanif}(U)}(\mu_\epsilon, \nu_{U'}) \vdash_X P' : U'$. Again by induction on the derivation of $\Delta_U \vdash P : U$, it is proven that if the derivation of $\Delta_U \vdash U \prec U'$ contains $\Delta \vdash T \prec_S T'$ then $\Delta \vdash T < S$ and $\Delta \vdash T' < S$. This means that for any sealing construct $(T : S)$ in U', there is Δ available in $\Delta_U \vdash P : U$ such that $\Delta \vdash T < S$. Hence, by Corollary 7, for any sealing construct $(T : S)$ in U', $\Delta_{\mathit{fullmanif}(U)} \vdash_X T < S$. By Lemma 77, we conclude $\Delta_{\mathit{fullmanif}(U)}(\mu_\epsilon, \nu_{U'}) \vdash_X P' : U'$. □

13 The expression problem

In this section, we present an advanced example of recursive modules, by giving a solution to the expression problem [22, 60].

The expression problem, originally named by Phil Wadler, dates back to Cook, who first discussed this problem [10]. It is one of the most fundamental problems a programmer faces during the development of extensible software. Here, we paraphrase a typical example of this problem in the following way: suppose that we have a small expression language, composed of a recursively defined datatype and operations on this datatype; then we want to extend the expression language in two dimensions, that is, to extend the datatype with new constructors and to add new operations that can handle both existing and new constructors. That a programming language can solve this problem in a type safe and concise way has been regarded as one measure of the expressive power of the language. Many researchers have addressed this problem, using different programming languages [53, 61, 58].

Our aim here is not to draw a conclusion that our solution is better than other solutions. It is not easy to compare the quality of different solutions, without deep understandings of each implementation language that is used to express each solution. Instead, we aim to give a useful example of recursive modules, in order to show that by combining recursive modules with other constructs of the core and the module languages we can obtain more expressive power in a modular way.

The example we use here extends an example presented in [25]. It is a variation on the expression problem, where we only insist on the addition of new constructors. Adding new processors is easy in this setting of .

We shall assume that we have extended *Traviata* with polymorphic variants [24], private row types [25] and some usual module language constructions. Adding polymorphic variants and private row types is straightforward. We add typing rules for them to our language. Allowing any structure to contain module type definitions may not be easy, but having module type definitions in the top-level is easy.

We define our first expression language in Figure 76, using the functor `PF`. The module type `E` specifies the signature of the expression languages we are to define. They contain a type component named `exp` and two operations `eval` and `simp` of the specified types. The type `exp` defined in the body of `PF` indicates that the first language supports expressions composed

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

Figure 76: A first language

of integer constants and addition. The function `eval` is for evaluating the expressions into integers. The function `simp` is for simplifying the expressions, by reducing the `‘Plus` constructor into the `‘Num` constructor when possible.

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

The intuition of the example is that `PF` takes as argument an expression language which is built by extending the addition language that `PF` defines. This is exactly what the signature of `X` expresses; here is the key of the example. The type specification `type t = private [> PF(X).exp]` specifies an abstract type into which the type `PF(X).exp` can be coerced, or, informally, an abstract type which is a supertype of `PF(X).exp`. The type `PF(X).exp` refers to the type `exp` defined inside `PF`'s body. Hence `X`'s signature specifies that `PF` can only be applied to a module whose defining expression language supports both integer constant and addition. This recursive use of `PF(X).exp` to constrain `PF`'s argument is the main difference with the solution in [25]. By avoiding the need to define types outside of the functor, it allows for a

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

Figure 77: A second language

more concise and scalable solution. Observe that if we do not have all of applicative functors, private row types and flexible path references, we could not write `X`'s signature in this way.

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

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

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

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

Next, we define our second expression language using the functor `MF` in Figure 77. The second language supports expressions composed of multipli-

cation and addition on integer constants.

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

Note that we instantiate the first addition language inside `MF`, and use it when defining the type `exp` with variant inheritance and defining functions `eval` and `simp` to delegate known cases by variant dispatch. In this way we avoid duplication of program codes.

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

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

Finally, we demonstrate in Figure 78 that it is easy to compose independently developed extensions into a single expression language.

Having seen examples here and in Section 1 and 8, we confirm that recursive modules are useful in several situations. Moreover, when combined with other language constructs, they give us the highly expressive power in a modular way. We believe that recursive modules are a promising candidate for supporting robust extensible software.

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

Figure 78: To merge independantly developed extensions

Part IV
Discussions

14 Related work

Much work has been devoted to investigating recursive module extensions of the ML module system. Notably, type systems and initialization of recursive modules pose non-trivial issues, and have been the main subjects of study. Here we first examine previous work on these issues, then overview work on *mixin modules*, another proposal for introducing recursion to ML-like module systems.

14.1 Type systems

To the best of our knowledge, no previous work has proposed a type system for recursive modules with applicative functors, except for the experimental implementation in O'Caml [41], or examined type inference for recursive modules whether functors are applicative or generative. *Traviata* has the ability to take fix-points of functors, which is not formalized or even explored in previous work by others.

The experimental implementation of recursive modules in O'Caml is most related to our work. Indeed, we followed it in large part when designing *Traviata*. O'Caml supports a highly expressive core language and a strong type inference algorithm, which are one of our motivations for the effort to enable type inference.

In O'Caml, a programmer can write signatures of recursive modules with rather concise syntax. However, it allows to write problematic modules whose type checking diverges due to cyclic type specifications in signatures. The potential for divergence when typing O'Caml modules is well-known, but is assumed to be a rare phenomenon in practice. Recursive modules seem to make the problem much more acute. This motivated us to insist on decidable type checking for *Traviata*. Of course we obtain it through restrictions, and a less expressive signature language. We put the first-order restriction on functors to detect cycles; we do not support module type definitions inside arbitrary structures, avoiding the avoidance problem [43, 26]. Yet, this may be a price for safety.

```
module F = functor(X : sig type t end) →
  struct datatype t = A of X.t end
module Int = struct type t = int end

module AofInt1 = F(Int)
module AofInt2 = F(Int)
module I = Int
module AofInt3 = F(I)
```

Figure 79: Example of O'Caml applicative functors

```
module Forest =
 functor(X : sig type t val compare : t → t →bool end) →
  functor(T : sig type t val labels : t → MakeSet(X).t end) →
   struct
    module Elm = X
    module ElmSet = MakeSet(Elm)
    type t = T.t * T.t
    val labels = λx.let (t1, t2) = x in
      ElmSet.union (T.labels t1) (T.labels t2)
   end
```

Figure 80: Weakness of applicative functors in O'Caml

Compared to O'Caml, *Traviata* has stronger notion of type equality in the sense that functors are fully applicative in *Traviata*. For instance in Figure 79, thanks to applicative functors, the two types `AofInt1.t` and `AofInt2.t` are equivalent. Yet the types `AofInt1.t` and `AofInt3.t` are not equivalent in O'Caml, since functors are not fully applicative. This is occasionally inconvenient for the use of module abbreviations. For instance, a program in Figure 80 is not typable in O'Caml. Since two types `MakeSet(X).t` and `ElmSet.t` are not equivalent, the body of the function `labels` cannot be typed. *Traviata* can type Figure 80, since it supports fully applicative functors.

Crary, Harper and Puri [11] (revisited later in [19]) gave a foundational type theoretic account of recursive modules. They analyzed recursive modules in terms of a phase-distinction formalism [28]. They introduced a fixed-point operator for structures and *recursively dependent signatures*, which can represent signatures of structures defined by the fixed-point operator. Then they interpreted these new constructs into primitive constructs of the structure calculus in [28]. The interpretation requires fully transparent signatures for recursive structures and contractiveness [2] of these signatures.

Russo designed a recursive module extension of the ML modules system in [56], which is implemented in Moscow ML [55]. He introduced explicitly typed declarations of self variables inside structures and signatures to enable forward references between structure components and between signature components, respectively. Self variables are a familiar construct in (class-based) object-oriented languages, where recursive definitions across class boundaries are a fundamental ingredient. We think the use of self variable in the context of recursive modules is intuitive to programmers and useful in practice. We extended his approach when designing *Traviata* by introducing implicitly typed declarations of self variables.

Dreyer [16] gave a theoretical account for type abstraction inside recursive modules. He investigated generative functors in the context of recursive modules, by interpreting type generativity in a destination passing style [59]. He gave a solution to the double vision problem, a typing difficulty involved in type abstraction inside recursive modules observed in [11], but in the process he sacrificed some flexibility in using structural types.

14.2 Initialization

Boudol [6], Hirschowitz and Leroy [31, 32, 33, 34], and Dreyer [15] have proposed type systems which ensure that initialization of recursive modules does not try to access components of modules that are not yet evaluated, under a call-by-value evaluation strategy of recursive modules. They are interested in the safety of initialization, hence their modules do not have type components.

Their type systems judge both the two programs:

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

and

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

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

14.3 Mixin modules

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

Duggan and Sourelis [20, 21] proposed mixin modules specifically for SML. Their mixin modules can split individual definitions of a datatype and a function into separate mixins: constructors of a datatype can be defined in several mixins; a function defined by cases on a datatype can be defined in several mixins, each mixin defining only certain cases. An operator for linking mixins is provided, to stitch together these constructors and cases to form a single datatype definition and a single function definition. Although we share the same motivation in principle, the ways we address are rather different.

Ancona and Zucca [3, 4] developed a theory for mixin modules in a call-by-name setting. Their work focuses on value level recursion of mixin modules, and is closely related to work on initialization of recursive modules.

Odersky et al. designed a calculus, named νObj [49], for classes and objects with dependent types, which is implemented as the Scala programming language [1]. Although the concrete syntax is rather different, νObj supports most mechanisms of the ML module system, including higher-order functors and nested structures with type components. Intuitively, νObj classes cor-

respond to ML functors and νObj objects to ML structures. νObj allows liberal recursion between classes and objects, which implies that it can express recursive ML modules.

The type system of νObj is undecidable. It traces type abbreviations in the intuitive way, which is one reason for the undecidability since there is the potential of cyclic type abbreviations.

Scala type system is kept decidable [12]. To avoid divergence in abbreviation expansion, it does not trace the same type abbreviation twice during expansion. As we examined in Example 2 of Section 4, this strategy sacrifices some flexibility of functors.

Recent work by S. Owens and M. Flatt [50] designed a module language which extends their previous work [23] on a MzScheme's module language with translucency and sharing of type information. Although their formalization does not include datatype definitions, which are a vital constituent of the ML core language, their language appears to be as expressive as the SML module system extended with recursion. Similarly to previous work on recursive modules, they do not examine support for type inference.

The operational semantics of their modules is different from that of ML modules. In their system modules are first-class values and can be dynamically composed and invoked. This semantics gives us insights into other possible design choices of recursive modules.

The generative nature of abstract types in their language is a notable difference from *Traviata*. It seems difficult to express applicative functors in their language. Hence *Traviata* and their language do offer distinct expressiveness. We would like to draw more thorough comparison between our proposals, which would be useful for even better design of recursive modules.

15 Future work

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

15.1 Separate type checking and compilation

Although we have not discussed in the thesis, *Traviata* is already prepared for separate type checking. We only have to extend the look-up judgment (Figure 39) so that the judgment informs the type system of signatures of modules which are type checked separately (i.e., to replace concrete module expressions with their signatures).

Indeed, we need not reconstruct a complete lazy program type from a given program P at once before checking type-correctness of P. When checking type correctness of a module expression, the type system only has to know signatures of modules that are visible from the module expression, but not signatures of modules hidden inside sealing. Hence a practical way of type checking programs would be to alternate reconstruction and type-correctness check in turn so that when type checking a module expression the type system only reconstructs signatures of visible modules. Once module expressions outside sealing have been type checked, the type system proceeds to reconstruct signatures of and type check module expressions inside sealing. For simplicity, we prefer to the current presentation of the type system.

Support for separate compilation [8] of recursive modules is another nontrivial issue, if one wants to ensure safe linking and evaluation of separately compiled recursive modules. We would like to investigate this issue, too.

15.2 Lazy modules

The operational semantics presented in the thesis adopts a lazy evaluation strategy for both modules and their value components, in the sense that only components of modules that are accessed are evaluated and the evaluation is triggered at access time. This semantics simplifies the soundness statements and their proof. It might not be natural for practical programming, however. Currently we are investigating lazy modules with eager value components, that is, to keep modules lazy but evaluate all the value components (but not module components) of a module at once, triggered by the first access to some component of the module. Lazy semantics of modules would allow

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

15.3 Relaxing the first-order structure restriction

It would be nice to relax the first-order structure restriction we put on functors. Support of higher-order functors does not seem urgent in practice. Yet, lack of nested functor arguments may be sever on occasion.

As we explained in Section 2, a programmer can pass sub-modules as independent parameters to a functor. Yet, if he wants to express type sharing constraint between these submodules, a typical situation is the coherence problem [29], he has to factor out the shared types in the sharing-by-construction style [7, 35]. This style is cumbersome compared to sharing-by-specification style [44], which requires functors to take nested arguments.

The reason of the restriction is for termination of the module and the type expansions. The module path expansion is based on ground term rewriting, where termination conditions are well-investigated [13]. We obtained ideas from recursive path ordering [14] when designing the type expansion. Although it is clear that these expansions are closely related to rewriting theory, we have not yet succeeded in formalizing them in the standard rewriting terminology. We think that such formalization will make clear the intrinsic difficulties in keeping expansions terminating and may open an avenue to apply known technique in rewriting theory for relaxing the restriction.

15.4 The double vision problem

K. Crary et al. observed a typing difficulty involved in type abstraction between recursive modules [11]. Dreyer named it the double vision problem and gave a detailed examination of this problem in his PhD thesis [17]. A typical situation of this problem occurs when a programmer attempts to cyclically import, inside a sealed module, a value that was exported by the same module as a value of an abstract type. Then a type system might not

```
struct (Z1)
module F = functor(X : sig type t end) →
  struct datatype t = A of X.t end
module M = (struct (Z3)
  type t = Z1.F(Z3).t end : sig (Z2) type t = Z1.F(Z2).t end)
end
```

Figure 81: Example on the double vision problem

regard the reimported value as of type of the underlying representation of the abstract type, with which the value was exported.

We do not solve this problem in a satisfactory way. In particular, the double vision problem can arise when a sealing signature involves type paths containing functor application where the self variable declared in the signature appears. For instance a program in Figure 81 is not typed in *Traviata*, since two self variables $\mathtt{Z}_2$ and $\mathtt{Z}_3$ are not equivalent even after the manifestation operation described in Section 11.

The double vision problem does not decrease the expressive power of the language; there is an encoding to avoid such a problematic situation. Yet this encoding is verbose. To give a fully satisfactory solution, we need 1) to sophisticate the manifestation operation and 2) to enrich the type environment so that it becomes aware of equivalence between self variables declared in different layers of sealing signatures which share the same implementation module. We are now undertaking formalization of this solution.

16 Conclusion

In this thesis, we designed and formalized a programming language, named *Traviata*, for a ML-like module system extended with liberal recursion between modules.

Traviata is strongly typed in the sense that the type system guarantees that well-typed programs never get stuck. We proved that the type system is sound for a call-by-value operational semantics. Moreover the type system is decidable, that is, whether or not a given program is well-typed is determined in a deterministic and terminating way.

The language design of *Traviata* is largely motivated by O'Caml. Typing of recursive modules in O'Caml is not formalized and is a rather liberal extension over previous proposals. It can handle practically useful examples. At the same time, it gives rise to several non-trivial issues, which include divergence in type expansion and lack of type inference for recursive modules.

We examined these issues in detail and gelled our proposal in *Traviata*. As we pointed out in Section 15, there is still a lot of work to be done to make *Traviata* a fully practical system. Yet we believe that *Traviata* can serve as a framework for formalizing a highly expressive module system with arbitrary nested structures, applicative functors and liberal recursion between modules.

References

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

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

[3] D. Ancona and E. Zucca. A primitive calculus for module systems. In *Proceedings of International Conference on Principles and Practice of Declarative Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

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

[5] M. Blume and A. Appel. Hierarchical Modularity. *ACM Transactions on Programming Languages and Systems*, 21(4), 1999.

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

[7] R. Burstall and B. Lampson. A kernel language for abstract datatypes and modules. In *Proc. International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer-Verlag, 1984.

[8] L. Cardelli. Program Fragments, Linking, and Modularization. In ACM Press, editor, *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, pages 266–277, 1997.

[9] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. IFIP TC2 working conference on programming concepts and methods*, pages 479–504, 1990.

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

[11] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proceedings of ACM SIGPLAN Conference on Programming Language Design an d Implementation*, pages 50–63, 1999.

[12] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A Core Calculus for Scala Type Checking. In *Proc. MFCS*, Springer LNCS, September 2006.

[13] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proceedings of Annual IEEE Symposium on Logic in Computer Science*, 1990.

[14] N. Dershowitz. Orderings For Term-Rewriting Systems. *Theoretical Computer Science*, 17(3):279–301, 1987.

[15] D. Dreyer. A Type System for Well-Founded Recursion. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, 2004.

[16] D. Dreyer. Recursive Type Generativity. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2005.

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

[18] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, pages 236–249, 2003.

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

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

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

[22] R. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 1998.

[23] M. Flatt and M. Felleisen. Units: Cool Modules for HOT Languages. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design an d Implementation*. ACM Press, 1998.

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

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

[26] G. Ghelli and B. Pierce. Bounded Existentials and Minimal Typing. *Theoretical Computer Science*, 193(1-2), 1998.

[27] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, 1994.

[28] R. Harper, J. C. Mitchell, and E. Moggi. Higher-Order Modules and the Phase Distinction. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, pages 341–354, 1990.

[29] R. Harper and B. Pierce. Design Considerations for ML-Style Module Systems. In *Advanced Topics in Types and Programming Languages*, chapter 8. The MIT Press, 2004.

[30] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15:253–289, 1993.

[31] T. Hirschowitz. Rigid Mixin Modules. In *International Symposium on Functional and Logic Programming*. ACM Press, 2004.

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

[33] T. Hirschowitz, X. Leroy, and J. B. Wells. Compilation of Extended Recursion in Call-by-Value Functional Languages. In *Principles and Practice of Declarative Programming*, pages 160–171. ACM Press, 2003.

[34] T. Hirschowitz, X. Leroy, and J. B. Wells. Call-by-Value Mixin Modules: Reduction Semantics, Side Effects, Types. In *European Symposium on Programming*, 2004.

[35] Mark P. Jones. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*. ACM Press, 1996.

[36] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical Report ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, 2000.

[37] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, pages 109–122. ACM Press, 1994.

[38] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Langu ages*, pages 142–153. ACM Press, 1995.

[39] X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

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

[41] X. Leroy. A proposal for recursive modules in Objective Caml. Available online at `http://caml.inria.fr/pub/papers/xleroy-recursive_modules-03.pdf`, May 2003.

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

[43] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

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

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

[46] R. Milner, M. Tofte, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, 1990.

[47] K. Nakata and J. Garrigue. Recursive Modules for Programming. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2006.

[48] K. Nakata, A. Ito, and J. Garrigue. Recursive Object-Oriented Modules. In *Proceedings of ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2005.

[49] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming*, 2003.

[50] S. Owens and M. Flatt. From Structures and Functors to Modules and Units. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2006.

[51] B. Pierce. *Types and Programming Languages*, chapter 9-11. MIT Press, 2002.

[52] N. Ramsey. ML Module Mania: A Type-Safe, Separately Compiled, Extensible Interpreter. In *Proceedings of ACM SIGPLAN Workshop on ML*, pages 172–202, 2005.

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

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

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

[56] C. Russo. Recursive Structures for Standard ML. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 50–61. ACM Press, 2001.

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

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

[59] P. Wadler. *Listlessness is Better than Laziness*. PhD thesis, Carnegie Mellon University, 1985.

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

[61] M. Zenger and M. Odersky. Independently Extensible Solutions to the Expression Problem. In *Proceedings of ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2005.