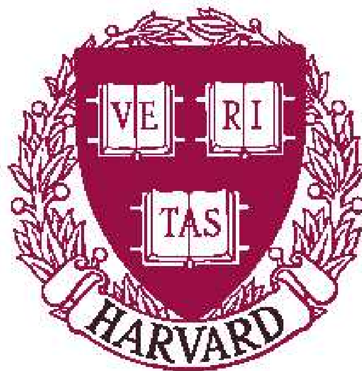
Type Generativity in Higher-Order Module Systems

Paul Govereau

TR-05-05



Computer Science Group
Harvard University
Cambridge, Massachusetts

Type Generativity in Higher-Order Module Systems

Paul Govereau

Harvard University

govereau@eecs.harvard.edu

Abstract

We present a higher-order module system similar to those found in Standard ML and Objective Caml. Our system allows both generative and non-generative types. Unlike other systems, the generativity of a type is reflected directly in the signature of the module in which it is declared, allowing a more direct analysis of type abstraction and generativity. Our module system can express both generative and applicative functors, and allows mixing of generative and non-generative types within a single module. This gives the programmer greater control over type generativity, and brings a new perspective to the relationship between the two styles of abstraction.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Modules

General Terms Modules, Type Systems, Abstraction

Keywords ML Modules, Generativity, Standard ML, Objective Caml

1. Introduction

Arguably, one of the most important jobs of a module system is to provide a means of type abstraction. A module system should allow a module writer to define a new *abstract type* whose implementation can be hidden from clients of the module. The ability to hide the implementation of a type allows large software systems to be split up into self-contained modules with well-defined interfaces. Type abstraction, together with separate compilation, make it possible to change the implementation of one module without affecting other code in the system.

In the ML family of languages, there are two different styles of abstract type: generative and non-generative. A non-generative, abstract type (or simply, abstract type) is a type whose implementation is hidden. A generative, abstract type (or generative type) also has the property that each instance of the type is distinct from every other instance. Given two identical modules declared with an abstract type, one module may manipulate data of that type from the other module. However, declaring the type generative prohibits this sharing. Both styles of abstraction have been extensively studied. However, only more recently have both styles been studied together within the same framework [2, 13].

This paper describes a higher-order module system such as those found in the Standard ML and OCaml programming languages. While such systems have been carefully studied in the past, our formulation makes several additional contributions. Our system treats generative and non-generative types in a new way: the two styles of abstraction are represented as different kinds of signatures. In addition, our calculus allows both generative and non-generative types to be mixed within a single module. This flexibility exposes a new language design issue in that it is possible for the programmer to specify generative or non-generative for each abstract type independently.

Our system is based on the module system described by Dreyer et al. [2]. However, we do not have their effects system, and our language of signatures differs. Our system is a variant of the higher-order polymorphic lambda calculus (F_ω) extended with singleton kinds [18]. In the following section, we give an informal description of the system. A more complete description of the language and type system can be found in Section 3.

1.1 Informal Development

A *module* is a collection of type definitions and values. Shown below is a module that defines the type `t` to be a list of integers, and three values: `x`, a simple value of type `t`, `new` a function that returns a value of type `t`, and `size`, a function that takes a value of type `t` and produces an integer (the length of the list)[1].

```
module M = struct
  type t = int list
  val  x = [1,2,3]
  fun new () = x
  fun size x = length x
end
```

The module `M` itself can be assigned a type. The type of module is referred to as its *signature*. Every module has one or more signatures that can be assigned to it. The *principal signature* of a module is the most specific, or descriptive, signature. The principal signature for the module `M` (shown below) gives the definition of the type `t` and the types of the three values.

```
signature S = sig
  type t = int list
  val x : t
  val new : unit → t
  val size : t → int
end
```

The signature `S` also could have been written with `int list` in place of `t` in the types of the values. This is valid, in this case, because `t` is defined to be `int list` in the signature. Therefore, whenever the declarations of the values are available, the definition of `t` is also available. It is possible to write a signature similar to `S` in which `t` is declared but not defined. In this case, the type `t` is *abstract*, and the declaration `val x:t` is not the same as `val x:int list`.

Type Abstraction The principal signature of `M` gives the definition of all of the types, and the types of all of the values in `M`. We may wish to write down a restricted signature for `M` in which some type definitions or values are not specified. Using the restricted signature, we can then *seal* `M` to create a new module in which some types or values are hidden. The sealing operation is written as $M :: S$. If the principal signature of module M is a subtype

[1] The examples given in this paper use an *ML-like* syntax. However, the code in this paper is not meant to be read as ML code, but rather as a convenient notation for our module calculus.

	Abstract	Generative
`type t = τ`	×	×
`type t`	√	×
`newtype t`	√	√

Table 1. Three forms of type declaration in signatures.

of signature S, then $M :: S$ is a new module with the same implementation as M, but with signature S. The sealing operation is one mechanism for type abstraction[2]. For example, we can make the type `t` in module `M` abstract using the signature:

```
signature SA = sig
  type t
  val new : unit → t
  val size : t → int
end
```

If we seal the module `M` with `SA`, then the resulting module will differ from the original in that:

1. The type `t` is abstract.
2. the value `size` now requires a value of the abstract type `t` as its first argument instead of any integer list.

By "the type `t` is abstract" we mean any code that uses a module with the signature `SA` cannot rely on the definition of `t`. Furthermore, since all of the types in `SA` are abstract, it is possible to exchange one implementation of this signature for another. That is, if we have two modules, M and N, that have been sealed with signature `SA`, then M can be used in place of N and vice-versa, without affecting clients of the modules.

Type Generativity In some situations, it is convenient to have a slightly stronger form of abstraction. In the previous example, we sealed the module `M` with a signature that declared the type `t` abstract. We can also write a signature, that when used with the sealing operation, creates a fresh abstract type that is not equal to any other type. This form of declaration is called a *generative* type declaration. In our system, we declare a type generative using `newtype`. For example, the signature below can be used to create a generative type.

```
signature SG = sig
  newtype t
  val new : unit → t
  val size : t → int
end
```

Just as with abstraction, any code that uses a module with the signature `SG` cannot rely on the definition of `t`. In addition, if we seal the module M with the signature `SG` multiple times, then all of the generated types are unique: $(M :: SG).t \neq (M :: SG).t$. Contrast this with the abstract type declaration in signature `SA`: $(M :: SA).t = (M :: SA).t$. This is surprising, because it seems that only sealing with generative declarations would be useful. However, sealing with non-generative declaration plays an important role in the definition of functors.

Table 1 shows the three forms of type declaration that may appear in a signature. The first form, `type t = τ`, declares a simple type alias that is neither abstract nor generative. The second form, `type t`, declares an abstract type that is not generative. The third form, `newtype t`, declares a generative type; generative types are always abstract. There is no declaration that produces a generative type that is not abstract.

[2] The other mechanism is functors; the arguments of a functor may have types that appear abstract within the body of the functor.

SML-style generative functor	OCaml-style applicative functor

`funsig FS(A:S)= sig`
`  newtype t`
`  newtype u`
`  type v = int`
`end`
 	
`module F :: FS`
 	
$F(A).t \neq F(A).t$
	`funsig FS(A:S)= sig`
	`  type t`
	`  type u`
	`  type v = int`
	`end`
	
	`module F :: FS`
	
	$F(A).t = F(A).t$

Figure 1. Two styles of functor.

A *functor* is a function from modules to modules. A functor can take a module (or another functor) as an argument and produce a new module (or functor). Our two forms of abstract type are useful for declaring two different styles of functor: generative [4, 6] and applicative [7]. Generative functors, when applied to an argument module, always generate fresh types for each abstract type in their result. When an applicative functor is applied, the abstract types in the result are equal to the types from any other application of the functor to the same argument. In the Standard ML language, all functors are generative, whereas in the OCaml language, all functors are applicative. Figure 1 shows an encoding for an SML-style, generative functor, and an OCaml-style, applicative functor. In general, SML-style modules can be encoded using only the `newtype` declaration, and OCaml-style modules can be encoded using both `type` and `newtype`.

2. Background

In order to provide a formal description of our module system, we must first develop a language for representing modules and signatures. We might represent a module as a list of type and value definitions. For instance, the module `M` from the previous section might be represented as the list:

$$\langle t = [\texttt{int list}], l = [1,2,3], \quad new = \lambda x : unit.l, size = length\rangle$$

In order to simplify our formal system, we will encode module using pairs instead of lists. Our encoding is the usual encoding of lists as pairs in which the first component of the pair contains an element of the list, and the second component contains another encoded list. In order to terminate the list, we use the value $\langle\rangle$. Thus, we represent the module `M` as the following sequence of nested pairs:

$$\langle t = [\texttt{int list}], \langle l = [1,2,3], \langle new = \lambda x : unit.l, \langle size = length, \langle\rangle\rangle\rangle\rangle$$

Note that the name l, defined as the second element, appears in later definitions. In general, a name defined in the first component of a pair can appear in the second component. We write pairs using the notation $\langle s = a, b\rangle$. A name is always given to the first component of the pair (here the expression a is named s) so that the second component of the pair can mention it.

Functors are encoded using normal lambda terms. We write functors using a capital lambda (Λ) in order to distinguish them from term-level functions. The notation $\Lambda s : \sigma.M$, represents a

```
module F(X : sig type t end) =
struct
  type t = X.t * int
  type u = t
  module M = struct
    type v = t * u
  end
end :: sig
  type t
  type u = t
end
```

$$\Pi X{:}(\Sigma t{:}[\![T]\!].1)\\ .\langle t = [\texttt{Typ}\ (\pi_1 X) * \texttt{int}]\\ ,\langle u = t\\ ,\langle M =\\ \langle v = [\texttt{Typ}\ t * \texttt{Typ}\ u], 1\rangle\\ ,1\\ \rangle\rangle\rangle :: \Sigma t{:}[\![T]\!]\\ .\Sigma u{:}\mathbb{S}(t).1$$

Figure 2. A larger module and its encoding.

functor that takes as an argument any module with signature σ and returns the module expression M.

Dependent Types The principal signature of a module describes the module's interface. Since a module contains both type definitions and values, its principal signature provides the definitions of all types, and the types of all values defined in the module. The two primary signatures forms are for describing module pairs and functors.

Consider a simpler version of the module above that only has the type t and the value l

$$\langle t = [\texttt{int list}], \langle l = [1, 2, 3], \langle\rangle\rangle\rangle$$

The signature of this module may be written using a normal pair type, but we add the names t and l

$$(t = [\texttt{int list}] \times (l : \texttt{int list} \times 1))$$

Where 1 is the type of the unit value $\langle\rangle$. However, we may wish to assign l the type t:

$$(t = [\texttt{int list}] \times (l : \texttt{Typ}\ t \times 1))$$

The notation $\texttt{Typ}\ t$ indicates the type of t[3]. In this last signature, the t that appears in the type of l is meant to be the same t that is defined to be `int list`. In order to emphasize that we are binding the name t and then using it later on, we use the notation:

$$\Sigma t{:}[\texttt{int list}].(l : \texttt{Typ}\ t \times 1))$$

The notation $\Sigma t : a.b$, indicates a pair where the first component is a, and the second component is b. Furthermore, the first component is given the name t, and the second component may mention t. This kind of pair is called a *dependent pair*.

A similar situation occurs with functors; the result signature of a functor may depend on the argument signature. Therefore, we use a *dependent function* type for functor signatures. The signature $\Pi s : \sigma_a.\sigma$ describes a functor that takes an argument with signature σ_a and return a module with σ. Note that the signature σ may mention s.

Dependent types were used by MacQueen [9] for representing second-class module systems; and later, Sheldon and Gifford [14] used dependent types for a first class module system. Many of the subsequent formalisms (including this one) have built upon this basic framework.

Type Abstraction In the above examples, all type definitions are *transparent*. That is, the definitions of types are represented in the signatures. As discussed in the introduction, we can also create modules with abstract types through the sealing operation. In order to create a module with an abstract type, we seal the module with a signature in which the type is not specified. We use the special signature $[\![T]\!]$ to indicate that a type should be made abstract when the sealing operation is used. For example, consider the simple module with a single type component `t` defined to be `int`.

$$\langle t = [\texttt{int}], \langle\rangle\rangle$$

We can make a new module where the type t is abstract by sealing the above module with the signature

$$\Sigma t{:}[\![T]\!].1$$

This signature describes a module with a single type component named t that is abstract.

Singleton Signatures Thus far, we have not described how to specify that one type is equal to another type in our system. In order to specify that one type is equal to another, we use *singleton signatures* [2, 17]. A singleton signature, written $\mathbb{S}(M)$, is a signature that can be given to any module that is equivalent to M. We can use this signature to specify that one type is equal to another type. For example, consider the signature S given below.

```
signature S = sig
  type t
  type u = t
end
```

We represent this signature in our system as:

$$S = \Sigma t{:}[\![T]\!].\Sigma u{:}\mathbb{S}(t).1 \quad .$$

The singleton signature $\mathbb{S}(t)$ that is given to the u component indicates that u must be equivalent to t.

Similar mechanisms have been used in other module systems for the same purpose. In Leroy [6], two kinds of sum types are used: a strong sum, and a manifest sum. The manifest sum, $\exists\alpha = \tau.M$, is used to constrain the quantified type variable α to be τ. Harper and Lillibridge [4] introduced the translucent sum. The translucent sum has two kinds of type components: opaque components, written $b \triangleright \alpha$, and transparent components, written $b \triangleright \alpha = \tau$. A translucent sum allows some of the type variables to be constrained to a particular type. Singleton signatures are a bit more general than either of these mechanisms because a they can be used anywhere a normal signature can be used, not just at quantified type variables.

Figure 2 shows a larger example of a module and its encoding in our language. The functor `F` takes as an argument a module with a single type component `t`. The functor body defines two types and a submodule with a single type. The body of the functor is sealed

[3] Technically, t must be a module expression, so t is a module with a single component: the type `int list`. The expression $\texttt{Typ}\ t$ projects the type component out of the module t.

types	$\tau ::=$	$\texttt{Typ}\ M \mid \widetilde{\Pi} s : \sigma.\tau$
terms	$e ::=$	$\texttt{Val}\ M \mid \lambda s : \sigma.e \mid eM$
signatures	$\sigma ::=$	$1 \mid [\![T]\!] \mid [\![\nu T]\!] \mid [\![\tau]\!] \mid \mathbb{S}(M)$
	$\mid$	$\Sigma s{:}\sigma_1.\sigma_2 \mid \Pi s{:}\sigma_1.\sigma_2$
modules	$M ::=$	$s \mid \langle\rangle \mid [\tau] \mid [e{:}\tau]$
	$\mid$	$\langle s = M_1, M_2\rangle \mid \pi_i M$
	$\mid$	$\Lambda s{:}\sigma.M \mid M_1 M_2$
	$\mid$	$\texttt{let}\ s = M_1\ \texttt{in}\ M_2{:}\sigma$
	$\mid$	$M :: \sigma$
contexts	$\Gamma ::=$	$\cdot \mid \Gamma, s : \sigma$

Figure 3. Syntax

with a signature that hides the submodule and the definition of the type `t`. The functor is sealed with a signature which specifies that `u` is equal to `t`.

3. Formal Development

3.1 Syntax

Our calculus is based on $\lambda_{\leq}^{\Pi\Sigma S}$, a dependent lambda calculus with singleton types described by Stone and Harper [18, 19]. The syntax of our language is given in Figure 3. The syntax is divided into a term language and a module language. The term language has been kept as small as possible, however we believe it can be extended without difficulty. We begin with a description of the module language.

The signature 1 describes the the empty module $\langle\rangle$. A module may contain a single type component, written $[\tau]$, and may be described using three possible signatures. The first two signatures indicate that the type is abstract $[\![T]\!]$, or abstract and generative $[\![\nu T]\!]$. The third possibility is that the type is not abstract, but equal to a type defined in another module $\mathbb{S}(M)$. The last basic module contains a single term e of type τ. Such a module is written as $[e{:}\tau]$, and its signature is $[\![\tau]\!]$.

In order to construct more complex modules, we have a dependent sum type for modules $\Sigma s : \sigma_1.\sigma_2$. The dependent sum describes a pair of modules; the first element of the pair has the signature σ_1, and the second element of the pair has the signature σ_2 where the variable s may appear free in σ_2 and refers to the first element of the pair. If s does not appear free in σ_2, we will sometimes write $\sigma_1 \times \sigma_2$ to emphasize the fact that the pair is not dependent. Module pairs are written $\langle s = M_1, M_2\rangle$. The elements of a module pair can be accessed using the usual projections $\pi_i M$.

The dependent function type, or functor type, is written $\Pi s : \sigma_1.\sigma_2$, where σ_1 is the argument signature and σ_2 is the result signature which may depend on the argument. Module-level functions, or functors, are written as $\Lambda s{:}\sigma.M$ to distinguish them from term-level functions.

The last two module forms are let bindings and sealing. The let binding is standard; it binds M_1 to s within the body of M_2. Sealing of a module M with a signature σ produces a new module where the signature has been restricted to σ. The sealing operation is the only way to create abstract and generative types. In order for a module to be sealed with a signature, the module must be compatible with the signature. Compatibility is enforced through our typing judgments.

The term language has been kept as simple as possible, and derives most of its expressiveness from the module language. The term language has two types. There is a dependent function type $\widetilde{\Pi} s : \sigma.\tau$, that describes functions from modules with signature σ to terms of type τ. In addition, if a module contains a single type component, then that type can projected out of the module using $\texttt{Typ}\ M$.

Module Purity	:	$\Gamma \vdash M\ pure$
Well-formed types	:	$\Gamma \vdash \tau \Rightarrow \sigma$
Signature subtyping	:	$\Gamma \vdash \sigma_1 \leq \sigma_2$
Well-formed modules	:	$\Gamma \vdash M : \sigma$
Module equivalence	:	$\Gamma \vdash M_1 \cong M_2 : \sigma$

Table 2. The primary typing judgments.

The term language has three forms of expression: lambda abstraction, application and a value projection. A term-level lambda abstraction takes a module as an argument and produces a term; application applies a term-level lambda abstraction to a module. There is also a value projection operation that allows a term to be extracted from a module. If a module contains a single value, that value can be lowered to the term language using $\texttt{Val}\ M$. A notable omission from our syntax is term-level variables. In place of term-level variables, we will use module-level variables lowered with `Val` to the term-level. The term language could be easily enriched with base types, pairs and fix. However, this would not add to the discussion here, so we leave them out.

3.2 Typing

In order to enforce type abstraction and generativity, we have designed a type system that rules out invalid module expressions. The type system consists of five primary judgments. The primary judgment forms are listed in Table 2. The context Γ is a mapping from variables to signatures. We adopt the usual bound-variable convention that requires all bound variables to be assigned unique names via α-conversion. Furthermore, we require that all names appearing in Γ be unique, and we may therefore treat Γ as a partial function from variable names to signatures. We use the notation $M[N/s]$ to indicate the capture-avoiding substitution of N for s within M.

Because we have a dependent calculus, the context Γ is ordered. Signatures appearing in Γ may mention variables that appear earlier in Γ. In addition, for reasons that will be discussed below, the context Γ is split into two halves. The split in Γ is indicated with a semicolon: $\Gamma_1; \Gamma_2$. Often, we will just write Γ instead of $\Gamma_1; \Gamma_2$. In such cases we adopt the following notational conventions:

- When adding variables to the context, $\Gamma, x : \sigma$ is defined to mean:

$$(\Gamma_1; \Gamma_2), x : \sigma \overset{\text{def}}{\iff} \begin{cases} \Gamma_1, x : \sigma; \Gamma_2 & \text{if } \Gamma_2 = \emptyset \\ \Gamma_1; \Gamma_2, x : \sigma & \text{if } \Gamma_2 \neq \emptyset \end{cases}$$

- When reading variables from the context, $\Gamma(s)$ is defined to mean:

$$(\Gamma_1; \Gamma_2)(s) \overset{\text{def}}{\iff} \begin{cases} \Gamma_1(s) & \text{if } s \in \texttt{dom}(\Gamma_1) \\ \Gamma_2(s) & \text{if } s \in \texttt{dom}(\Gamma_2) \end{cases}$$

When the right-hand side of Γ is empty, then new variables are added to the left-hand side. To add a variable to the right-hand side, we will explicitly write $\Gamma_1; \Gamma_2, x : \sigma$. Once a variable is added to the right-hand side of Γ, all future variables will be added to the right-hand side. The split in Γ capture a simple notion of lexical scope: all of the variables contained in the right-hand side of Γ are contained within some enclosing lexical scope.

The first two judgments listed in Table 2 are unusual. Module purity indicates that a module will not generate any types. That is, a module expression M is pure if evaluating M will not result in any new types being generated. The well-formed type judgment checks that a type is well formed. In addition, the judgment computes an associated signature (either $[\![T]\!]$ or $[\![\nu T]\!]$) that a module containing the type would have.

$$\frac{\Gamma \vdash ok}{\Gamma \vdash 1\ pure}\ (3) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!]\ pure}\ (4) \qquad \frac{\Gamma \vdash \tau \Rightarrow \sigma}{\Gamma \vdash [\![\tau]\!]\ pure}\ (5) \qquad \frac{\Gamma, s : \sigma_1 \vdash \sigma_2\ pure}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2\ pure}\ (6) \qquad \frac{\Gamma \vdash \sigma_1\ pure \qquad \Gamma, s : \sigma_1 \vdash \sigma_2\ pure}{\Gamma \vdash \Sigma s{:}\sigma_1.\sigma_2\ pure}\ (7)$$

$$\frac{\Gamma \vdash M\ pure}{\Gamma \vdash \mathbb{S}(M)\ pure}\ (8)$$

$$\frac{\Gamma \vdash s : \sigma}{\Gamma \vdash s\ pure}\ (9) \qquad \frac{\Gamma \vdash M\ pure}{\Gamma \vdash \pi_i M\ pure}\ (10) \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma\ pure}{\Gamma \vdash M\ pure}\ (11)$$

Figure 4. Purity judgments for modules and signatures.

Signature equivalence is defined in terms of the signature subtyping judgment. That is, two signatures σ_1 and σ_2 are equivalent if and only if they are subtypes of each other:

$$\Gamma \vdash \sigma_1 \equiv \sigma_2 \overset{\text{def}}{\iff} \Gamma \vdash \sigma_1 \leq \sigma_2 \wedge \Gamma \vdash \sigma_2 \leq \sigma_1 \quad .$$

Type equivalence can also be defined in terms of the judgments above. Two base types are equivalent if the modules containing those types are equivalent:

$$\Gamma \vdash \tau_1 \equiv \tau_2 \overset{\text{def}}{\iff} \Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!] \quad .$$

In the remainder of this section, we describe the type system and its properties. A complete listing of the typing rules can be found in Appendix A.

3.2.1 Substitutions and Purity

In our calculus, the substitution of one signature into another is not always sound. Suppose we have a functor F with result signature F_{sig}. Now suppose we apply F to a module M; in this case, we might expect the type of the application to just be $F_{sig}[M/s]$. However, this is not necessarily correct, because the substitution may be unsound in the presence of effects and generative types.

For instance, suppose that F is a functor with signature

$$F : \Pi s{:}[\![\nu T]\!]\,.\mathtt{Typ}\ s \times \mathtt{Typ}\ s \quad .$$

The functor F is applicative because the body has not been sealed with any generative types. Therefore, it should be the case that for any module M, with signature $[\![\nu T]\!]$, if we apply F to M, then the first and second components of F should be equivalent. Now, suppose that M is a complex module expression such as the application of a generative functor to an argument: $M = GX$. With this module M, the first and second components of F cannot be equivalent:

$$\begin{gathered}\pi_1(F(GX)) \rightarrow \pi_1(GX \times GX) \rightarrow GX \\ \pi_2(F(GX)) \rightarrow \pi_2(GX \times GX) \rightarrow GX \\ GX \ncong GX\end{gathered}$$

We know that this last line must be correct because the functor G is generative, and two applications of a generative functor are not equivalent. Note that even though the above application of F is unsound, the following module expression *is* sound:

$$\mathtt{let}\ s = GX\ \mathtt{in}\ Fs{:}$$

In this case, the module s still has a generative type within its signature. However, s is a variable and variables are always equivalent to themselves. In this case, the interaction between generative types and type substitution has created a subtle problem with the typing of the functor application.

Harper and Lillibridge [4] solve this problem by requiring functions to be non-dependent at each application. Their application rule is (in our notation):

$$\text{HL-APP}\quad \frac{\Gamma \vdash F : \Pi s{:}\sigma_a.\sigma \qquad \Gamma \vdash A : \sigma_a \qquad s \notin FV(\sigma)}{\Gamma \vdash FA : \sigma}$$

While this rule is effective, it is difficult (if not impossible) to implement. In order to implement this rule, we must be able to find a minimal super-signature for the body of the functor that does not mention s. That is, given the dependent signature type $\Pi s:\sigma_a.\sigma'$, we must find a σ such that $\sigma' \leq \sigma$ and $s \notin \mathtt{fv}(\sigma)$. In general, there is no best choice for such a signature as this is an instance of the avoidance problem [3] which is know to complicate type checking.

An alternative solution proposed by Dreyer et al. [2] is to model the generation of new types as an (static) *effect*, and to require that the argument of a functor application be (statically) *pure*. A module expression is pure if its evaluation will not produce any effects. A module expression that contains a generative type may have the effect of generating a new type when it is evaluated, and therefore, is not pure. With this model in hand, the typing judgments must then track each module's type and any effects that the module may have. At an application node, if the argument is of the correct type, and cannot have any future static effects, then the application is sound. Dreyer et al. achieve this by layering an effect system over their typing judgments.

We adopt a similar solution in that we require functor arguments to be pure. However, unlike Dreyer et al., we do not use an effect system. Because we represent generativity in signatures, we can determine purity using a purely syntactic judgment. In particular, a module M is pure if either:

- M has a signature with no generative types
- M has a singleton signature over a pure module
- M is a projection from a pure module
- M is a variable

This first two conditions are checked by inspecting the signature of M. The last two conditions are safe because in both cases it is not possible for M to generate any new types: projecting a component of a pure module cannot generate any new types, and a variable expression cannot generate new types. These four conditions are captured by the judgments given in Figure 4. There are actually two judgment forms, one for signatures and one for modules. Note that these judgments are purely syntactic and syntax-directed. Therefore, they are quite easy to implement. A prototype implementation appears in Appendix C.

One consequence of our purity judgment is that impure modules may need to be assigned to variables before they can be used. This

(29)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!] \leq [\![T]\!]}$$

(30)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![\nu T]\!] \leq [\![\nu T]\!]}$$

(31)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!] \leq [\![\nu T]\!]}$$

(32)
$$\frac{\Gamma \vdash M \cong M' : [\![T]\!]}{\Gamma \vdash [\![\mathtt{Typ}\ M]\!] \leq [\![\mathtt{Typ}\ M']\!]}$$

(33)
$$\frac{\Gamma \vdash \sigma' \equiv \sigma \qquad \Gamma, s : \sigma \vdash [\tau] \cong [\tau'] : [\![T]\!]}{\Gamma \vdash \left[\!\!\left[\widetilde{\Pi} s : \sigma.\tau\right]\!\!\right] \leq \left[\!\!\left[\widetilde{\Pi} s : \sigma'.\tau'\right]\!\!\right]}$$

(34)
$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \leq [\![T]\!]}{\Gamma \vdash \mathbb{S}(M) \leq [\![T]\!]}$$

(35)
$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \leq [\![\nu T]\!]}{\Gamma \vdash \mathbb{S}(M) \leq [\![\nu T]\!]}$$

(36)
$$\frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathbb{S}(M_1) \leq \mathbb{S}(M_2)}$$

Figure 5. Signature Subtyping Rules (Abbreviated)

requirement may be syntactically burdensome, but most of these bindings can be handled by an elaborator from a more human-friendly syntax. Our system is less restrictive than the calculus of Leroy [8] which requires that dependent functor arguments only be applied to paths. However, our system requires more bindings than the calculus of Dreyer et al. [2] where purity information is propagated through an effect system.

3.2.2 Signatures

Signatures can be formed according to the seven syntactic forms described earlier. The only restriction is that a singleton signature must refer to a module that contains a type.

(24)
$$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash \mathbb{S}(M)\ sig}$$

(25)
$$\frac{\Gamma \vdash M : [\![\nu T]\!]}{\Gamma \vdash \mathbb{S}(M)\ sig}$$

Notice that we can form a singleton for a module that contains a generative type as well as a module that contains a non-generative type.

The subtyping relation for signatures is shown in Figure 5. There are three rules for singletons (34-36). The first rule says that a singleton signature $\mathbb{S}(M)$ is a subtype of the signature $[\![T]\!]$ if the module M contains a single non-generative type; there is a similar rule for $[\![\nu T]\!]$. The third rule says that two singleton signatures, $\mathbb{S}(M_1)$ and $\mathbb{S}(M_2)$, are in the subtype relation if the modules M_1 and M_2 are equivalent.

The subtyping relation allows a non-generative type to be coerced into a generative type (Rule 31). This rule is important for functor application. If a functor does not care whether its argument is generative or non-generative, then the functor can specify generative, and this rule will allow an argument with non-generative types to be applied.

3.2.3 Labeled Singletons

The signature formation rules described in the previous section only allow the formation of singleton types for modules with a single type component. Forming singletons of higher modules (such as functors and dependent sums) requires that we specify the signature at which the singleton is formed. The singleton of module M at the higher signature σ is written $\mathbb{S}_\sigma(M)$. This form of singleton is called a *labeled singleton*.

The signature $\mathbb{S}_\sigma(M)$ is not necessarily the same as the signature $\mathbb{S}_{\sigma'}(M)$. A singleton signature, $\mathbb{S}(M)$, represents the set of modules that are equivalent to M. In our calculus, equivalence between modules depends on the signature at which they are compared. Therefore, the set of modules equivalent to M at signature σ is not necessarily the same as the set of modules equivalent to M at signature σ'. For example, suppose that we have the two functors

$$
\begin{aligned}
\mathbb{S}_1(M) &= 1 \\
\mathbb{S}_{[\![T]\!]}(M) &= \mathbb{S}(M) \\
\mathbb{S}_{[\![\nu T]\!]}(M) &= \mathbb{S}(M) \\
\mathbb{S}_{[\![\tau]\!]}(M) &= [\![\tau]\!] \\
\mathbb{S}_{\mathbb{S}(M')}(M) &= \mathbb{S}(M) \\
\mathbb{S}_{\Pi s \sigma_1 . \sigma_2}(M) &= \Pi s \!:\! \sigma_1 . \mathbb{S}_{\sigma_2}(M s) \\
\mathbb{S}_{\Sigma s \sigma_1 . \sigma_2}(M) &= \Sigma s \!:\! \mathbb{S}_{\sigma_1}(\pi_1 M) . \mathbb{S}_{\sigma_2[\pi_1 M / s]}(\pi_2 M)
\end{aligned}
$$

Figure 6. Encoding Singletons at Higher Signatures

M and N defined below.

$$
\begin{aligned}
M &= \Lambda s \!:\! [\![T]\!] . s \\
N &= \Lambda s \!:\! [\![T]\!] . [\tau]
\end{aligned}
$$

Initially, it may seem that these two functors are not equivalent; one is an identity function, and the other is a constant function. However, equivalence depends on the signature at which we consider these two functors. At the most natural signature of $\Pi s \!:\! [\![T]\!] . [\![T]\!]$, M and N are not equivalent just as expected. At the signature $\Pi s : \mathbb{S}([\tau]) . [\![T]\!]$, M and N are equivalent. The reason is that this second signature restricts the argument type of M to only those modules that N can return as a result. Thus, for any valid argument, both M and N return the same result, and are therefore (extensionally) equivalent.

Figure 6 shows an encoding of labeled singletons in our existing calculus. For example, if M is a functor, then the encoding takes an η-expansion of M and forms the singleton signature of the body $M s$. The singleton of the functor M is the set of all modules that take take the same arguments as M, and return the same result that M would return. This encoding corresponds with our notion of equivalence. A similar encoding is used for dependent sum types, but we use the first and second projections of M to form singletons of the first and second components.

3.2.4 Types and Terms

There are two forms of term-level types: a dependent function type, and the projection of a type from a module, $\mathtt{Typ}\ M$. A type can be projected from a module if the module has the signature $[\![T]\!]$ or the signature $[\![\nu T]\!]$. The well-formed type judgment ensures that types are projected only from modules with one of these two signatures.

Recall that the well-formed type judgement also computes a signature that a module containing the type would have; this signature will be either $[\![T]\!]$ or $[\![\nu T]\!]$. For example, the projection of a non-generative type from a module is well-formed, and a module containing the projected type would have the signature $[\![T]\!]$:

(12)
$$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash \texttt{Typ}\ M \Rightarrow [\![T]\!]}$$

The projection of a non-generative type could also be given the signature $[\![\nu T]\!]$.

When a generative type is projected from a module, we may associate it with the signature $[\![\nu T]\!]$; the projected type carries along the generative signature of the original type. However, this may be too restrictive, and there are some cases in which it is safe to give a projected, generative type the signature $[\![T]\!]$. We can think of the well-formed type judgment as deciding when it is safe to drop a generative signature in favor of a non-generative one. For example, suppose that G is a functor with the following signature.

```
G : funsig() = sig newtype t end
```

Now, consider the definition of the *non-generative* functor defined below.

```
M = G()
functor F() = struct
  type t = M.t
end :: sig type t end
```

In the example above, the module `M` contains a new type generated by the application of the generative functor `G`. However, each time the functor `F` is applied, the same type is returned in the body of `F` — namely, the type generated by the application of `G` immediately before the definition of `F`. In this case, the projection of the generative type `t` can be given the non-generative signature $[\![T]\!]$, and therefore `F` can be given a non-generative signature.

If we change the above example slightly, by moving the application of `G` inside the body of the functor `F`, then it is no longer safe to give the projection of `t` a non-generative signature.

```
functor F() = struct
  M = G()
  type t = M.t
end :: sig newtype t end
```

In this case, because the application of `G` occurs within the body of `F`, each application of `F` will generate a new type. Therefore, `F` must have a non-generative signature.

In the last example, the functor `F` would also have a generative signature if `G` had been given as a argument to `F`. Russo describes this problem in an epilogue to his thesis [12]. Russo notes that if a non-generative functor is given a generative functor as an argument, then the generative types in the argument can be converted to non-generative types and an unsoundness can occur. If a functor can only be completely generative or completely non-generative, then it is enough to require non-generative functors to have non-generative arguments. However, in our system, a module or functor can have a mix of generative and non-generative components.

We must be able to distinguish the two examples above. In the first example, the type projection can be given the signature $[\![T]\!]$, and in the second example, the type projection must carry the generative signature of the original type. The key difference between the two examples above is that in one case, the type is generated under a functor, and in the other it is not. If a type is generated under a functor, then the expression that generated the type may represent many different types since the functor may be applied many times. Each time the functor is applied, the types generated in its body must be unique. Therefore, if a generative type is projected out of a module, and this module is defined under a functor, then we must continue to carry the generative type. However, if a type is generated outside of a functor, then the expression that generated the type only represents one new type. Therefore, it is safe to treat the projection of this type as non-generative.

When checking the well-formedness of a type projected from a module with signature $[\![\nu T]\!]$, we have a simple condition for deciding what signature a module that contains the projected type should have. If a type is defined under a functor, and it depends on generative types that are also defined under the functor, then the type must be generative. However, if the type is not defined under a functor, or is defined under a functor but does not depend on any generative types defined under the functor, then the type can be non-generative. In order check this condition, we have split Γ into two halves such that the right-hand side of Γ will contain variables that are defined under a functor. Therefore, to check if a module does not depend on any variables defined under a functor, we check that the free variables of the module appear are contained in the left-hand side of Γ.

(13)
$$\frac{\Gamma_1;\Gamma_2 \vdash M : [\![\nu T]\!] \qquad \Gamma_1;\Gamma_2 \vdash M\ pure \qquad \texttt{fv}(M) \subseteq \Gamma_1}{\Gamma_1;\Gamma_2 \vdash \texttt{Typ}\ M \Rightarrow [\![T]\!]}$$

(14)
$$\frac{\Gamma_1;\Gamma_2 \vdash M : [\![\nu T]\!] \qquad \Gamma_1;\Gamma_2 \vdash M\ pure \qquad \texttt{fv}(M) \not\subseteq \Gamma_1}{\Gamma_1;\Gamma_2 \vdash \texttt{Typ}\ M \Rightarrow [\![\nu T]\!]}$$

Note that we also require the module `M` to be pure. This implies that `M` is either a variable or some number of projections from a variable. Since projections cannot generate new types, it is enough to just consider the free variables of `M`.

Terms are typed using the well-formed term judgment. Our term language has been kept very small, and most of the typing rules are standard. We expect that it is straight-forward to add new term-level types, and that the additional typing rules would also be standard. However, the term-level judgments must ensure that a module is pure if it is going to be substituted into another expression. For example, our term-level application requires that the argument module be pure:

(19)
$$\frac{\Gamma \vdash e : \widetilde{\Pi} s : \sigma.\tau \qquad \Gamma \vdash M : \sigma' \qquad \Gamma \vdash \sigma' \leq \sigma \qquad \Gamma \vdash M\ pure}{\Gamma \vdash eM : \tau[M/s]}$$

3.2.5 Modules

Having established the typing rules for terms and signatures, we can now describe the declarative rules for typing modules. The interesting rules are shown in Figure 7. We use the symbol $\Vdash$ to distinguish these rules from the syntax-directed rules that are presented in the next section.

The (FUNCTOR) rule is the one typing rule that creates a split Γ by explicitly inserting a variable into the right-hand side of the context. The functor parameter and all variables defined in the body of the functor are inserted into the right-hand of the context. The split in the context corresponds to the lexical scope of the functor. The well-formed type judgment, described earlier, can then use the split Γ to decide when generativity should be propagated for a projected type.

Just as with the term language, we must be careful to ensure that modules are pure when they are used in substitutions. The (APP) rule constrains the argument module M to be pure since it is substituted into the body of the functor F. To apply a functor to an impure argument, the argument must first be bound to a variable

$$\frac{\Gamma \vdash ok \qquad \sigma = \Gamma(s)}{\Gamma \Vdash s : \sigma}\ (\text{VAR}) \qquad \frac{\Gamma_1; \Gamma_2, s : \sigma_1 \Vdash M : \sigma_2}{\Gamma_1; \Gamma_2 \Vdash \Lambda s{:}\sigma_1.M : \Pi s{:}\sigma_1.\sigma_2}\ (\text{FUNCTOR}) \qquad \frac{\Gamma \Vdash F : \Pi s{:}\sigma_1.\sigma_2 \qquad \Gamma \Vdash M : \sigma_1 \qquad \Gamma \vdash M\ pure}{\Gamma \Vdash FM : \sigma_2[M/s]}\ (\text{APP})$$

$$\frac{\Gamma \Vdash M_1 : \sigma_1 \qquad \Gamma, s : \sigma_1 \Vdash M_2 : \sigma \qquad \Gamma \vdash \sigma\ sig}{\Gamma \Vdash (\texttt{let}\ s = M_1\ \texttt{in}\ M_2{:}\sigma) : \sigma}\ (\text{LET}) \qquad \frac{\Gamma \Vdash M : \sigma}{\Gamma \Vdash (M :: \sigma) : \sigma}\ (\text{SEAL}) \qquad \frac{\Gamma \Vdash M : \sigma' \qquad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \Vdash M : \sigma}\ (\text{SUBSUME}) \qquad \frac{\Gamma \Vdash M : [\![T]\!]}{\Gamma \Vdash M : \mathbb{S}(M)}\ (\text{SGL-ABS})$$

$$\frac{\Gamma \Vdash M : [\![\nu T]\!]}{\Gamma \Vdash M : \mathbb{S}(M)}\ (\text{SGL-GEN}) \qquad \frac{\Gamma \Vdash M : \Pi s{:}\sigma_1.\sigma \qquad \Gamma, s : \sigma_1 \Vdash Ms : \sigma_2}{\Gamma \Vdash M : \Pi s{:}\sigma_1.\sigma_2}\ (\text{EXT-PI}) \qquad \frac{\Gamma \Vdash \pi_1 M : \sigma_1 \qquad \Gamma \Vdash \pi_2 M : \sigma_2}{\Gamma \Vdash M : \Sigma_{_}{:}\sigma_1.\sigma_2}\ (\text{EXT-SUM})$$

Figure 7. Well-formed Module Judgments (Abbreviated)

with a let expression. The (LET) rule does not need to constrain M_1 to only pure modules because M_1 is immediately bound to a variable which is pure.

Dependent products pose a similar problem, but only for their second components. The first component is not required to be pure when it is projected. However, the first component is required to be pure when the second component is projected. The reason for requiring purity in the first argument is the same as for dependent functions. When we substitute the first component, M_1 for the variable s in the body of the second component, this may produce an invalid signature if M_1 is not pure.

As noted above, the let expression and the pair projection expression have slightly different static semantics; it is worth noting the subtle differences between them here. Since the let expression $\texttt{let}\ s = M\ \texttt{in}\ N : \sigma$ and the pair projection $\pi_2\langle s = M, N\rangle$ have the same dynamic semantics, it is tempting to treat the let expression as syntactic sugar for the projection. However, the two expression behave differently from a typing perspective. For example, suppose that $M = [\tau] :: [\![\nu T]\!]$, and that N has the signature σ_n. Consider the the following projection:

$$\pi_2(\langle t = M, N\rangle)$$

This expression is *not* allowed by our type system. If it were allowed, then the type of the expression would have to be $\sigma_n[\pi_1 M/t]$. However, the module M is not pure, and this signature is not equal to the signature for a second instance of the same expression. This is because the sealing operation in the expression M will produce a new type each time the expression is evaluated. In order to project out the second component of this module pair it must first be bound to a variable.

The let expression does not suffer from the same restriction as the projection expression above. The following let expression has the same dynamic semantics as the projection above.

$$\texttt{let}\ t = M\ \texttt{in}\ N{:}\sigma$$

This expression *is* allowed by our type system. However, the signature of the module N may depend on the bound variable t, and the variable t cannot be allowed to escape the scope of the let expression. Therefore, we require that the signature of the let expression be specified; in the above example the signature is specified to be σ, and σ must be a super type of σ_n.

The (SEAL) rule allows a module to be sealed with a signature if the module can be typed with that signature. The (SUBSUME) rule allows a module M, that can be typed with signature σ', to be typed with any valid super-signature of σ'. The (SEAL) and (SUBSUME) rules work together to create abstract types. For instance, in a typical derivation, the (SUBSUME) rule would be applied immediately above the (SEAL) rule, as shown below.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \Vdash M : \sigma'} \qquad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \Vdash M : \sigma}}{\Gamma \Vdash (M :: \sigma) : \sigma}$$

In order to provide an efficient implementation, the derivation $\Gamma \vdash M : \sigma'$ should produce the most specific signature for M so that any valid super type of M can be used in the sealing operation. That is, we would like the typing derivation to produce the principal signature of M. We will return to the question of principal signatures shortly.

Singleton signatures are typed using the singleton introduction rules (SGL-ABS) and (SGL-GEN). The (SGL-ABS) rule allows a singleton signature to be formed over a module if the module contains a single abstract type. The (SGL-GEN) rule allows a singleton signature to be formed over modules containing generative types.

The two remaining rules (EXT-PI) and (EXT-SUM) can be thought of as introduction rules for labeled singletons. We require the extensionality rules (EXT-PI) and (EXT-SUM) because labeled singletons are not primitive in our calculus. If we consider a simple functor, F, with signature $\Pi s : \sigma'.\sigma$, then it should certainly be the case that F can be given the signature $\mathbb{S}_{\Pi s\sigma'.\sigma}(F)$. That is, F should be a member of the signature which only contains the functor F at F's signature. However, as noted by Stone and Harper [19], F is not the canonical member of this signature, but rather its η-expansion. The singleton signature of F is:

$$\mathbb{S}_{\Pi s\sigma'.\sigma}(F) = \Pi s{:}\sigma'.\mathbb{S}(Fs) \qquad .$$

Thus, it is easy to see that the η-expansion of F is in this signature. However, without the extensionality rules, we may not be able to prove that F has this signature. For instance, if F is a variable, then we cannot syntactically decompose F into an argument and a body and show that the body has the right signature. With the (EXT-PI) rule, we are able to prove that F has this signature.

The extansionality rules (EXT-PI) and (EXT-SUM) have a long history in the module-system literature. A version of the (EXT-SUM) rule appears in Harper et al. [5] under the name "non-standard typing rule for structures". A comparison can also be made with the "strengthening" operation of Leroy [6]. However, the strengthening operation is more similar to our labeled singleton encoding, and is introduced with the following rule (in our notation):

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \mathbb{S}_\sigma(M)}\ (\text{STRENGTHENING})$$

(39)
$$\frac{\Gamma \vdash ok \qquad \sigma = \Gamma(s)}{\Gamma \vdash s : \mathbb{S}_\sigma(s)}$$

(40)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash \langle\rangle : 1}$$

(41)
$$\frac{\Gamma \vdash \tau \Rightarrow \sigma}{\Gamma \vdash [\tau] : [\tau]}$$

(42)
$$\frac{\Gamma \vdash \tau \Rightarrow \sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash [e{:}\tau] : [\![\tau]\!]}$$

(43)
$$\frac{\Gamma_1; \Gamma_2, s : \sigma_1 \vdash M : \sigma_2}{\Gamma_1; \Gamma_2 \vdash \Lambda s{:}\sigma_1.M : \Pi s{:}\sigma_1.\sigma_2}$$

(44)
$$\frac{\Gamma \vdash F : \Pi s{:}\sigma_1.\sigma_2 \qquad \Gamma \vdash M : \sigma_1' \qquad \Gamma \vdash \sigma_1' \leq \sigma_1 \qquad \Gamma \vdash M\ pure}{\Gamma \vdash FM : \sigma_2[M/s]}$$

(45)
$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, s : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \langle s = M_1, M_2\rangle : \Sigma s{:}\sigma_1.\sigma_2}$$

(46)
$$\frac{\Gamma \vdash M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_1 M : \sigma_1}$$

(47)
$$\frac{\Gamma \vdash M : \Sigma s{:}\sigma_1.\sigma_2 \qquad \Gamma \vdash M\ pure}{\Gamma \vdash \pi_2 M : \sigma_2[\pi_1 M/s]}$$

(48)
$$\frac{\Gamma \vdash M : \sigma' \qquad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash (M :: \sigma) : \sigma}$$

(49)
$$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, s : \sigma_1 \vdash M_2 : \sigma' \qquad \Gamma, s : \sigma_1 \vdash \sigma' \leq \sigma \qquad \Gamma \vdash \sigma\ sig}{\Gamma \vdash (\texttt{let}\ s = M_1\ \texttt{in}\ M_2{:}\sigma) : \sigma}$$

Figure 8. Well-formed Module Judgments (Syntax-directed)

The same rule also appears in Leroy [8], and in Leroy's account of applicative functors [7] where it is referred to as the "self" rule. Also, in Harper and Lillibridge [4], the (VALUE-O) rule corresponds roughly to our (EXT-SUM) rule. The (VALUE-O) rule is essentially the same as the (STRENGTHENING) rule discussed above; it allows the singleton constructor to be immediately applied to any component of a dependent sum.

3.2.6 A Syntax-Directed System

The extensionality rules discussed in the previous section are only required so that we can introduce more specific signatures for higher modules. If we have a type system that always produces the most specific or principal signature of a module, then the extensionality rules are not necessary. Such a system was developed by Stone and Harper [19] for the $\lambda_{\leq}^{\Pi\Sigma S}$ calculus, upon which our calculus is based. Following Stone and Harper, we have developed a syntax-directed type system that produces principal signatures. The full, syntax-directed typing judgment for modules in shown in Figure 8.

In the syntax-directed system, the (SUBSUME) rule has been folded in where necessary, and the extensionality rules have been replaced by a new rule for variables (rule 39). The new variable rule is remarkably similar to the (STRENGTHENING) rule from Leroy [8]. However, in our system, the labeled singleton translation is only applied to variables, which are guaranteed to be pure. The variable rule will always give a variable the most specific signature possible. This signature can then be generalized by subsumption when necessary (Rules 44, 48, and 49).

3.3 Module Equivalence

The last judgment form in our system is for deciding the equivalence of two modules. The complete set of equivalence rules can be found in Appendix A; they include the standard equivalence and congruence rules, one rule for subsumption, and the five rules shown in Figure 9. These five rules demonstrate the subtlety of the equivalence relation in the presence of generative types and singletons. The first rule (Rule 50) is the normal reflexivity rule, except for the additional requirement that the module M be pure. Thus, not all modules are equal to themselves. In particular, an impure module (such as a generative functor application) cannot be compared with any other module, including itself.

Rules 62 and 63 are extensional equivalence rules. Note that neither of these rules require the modules to be pure. This is a bit surprising for the case of functors since only pure functors should be comparable (two generative functors are never extensionally equivalent). However, in our system, generative functors will have generative types within their signatures. Therefore, it will not be possible to prove that $M_1 s \cong M_2 s$ since there are no equivalence rules for impure modules.

Rule 64 relates a let expression to its reduction, but we must ensure that both M_1 and M_2 are pure. The typing rules for the let expression do not guarantee the purity of either M_1 or M_2. However, we cannot compare M_2 if it is impure, and the substitution of M_1 into M_2 may be unsound if M_1 is impure. A let expression is only equivalent to its reduction if all of the modules in sight are pure. Therefore, we must explicitly check for purity in the equivalence rule for let expressions.

The last rule (Rule 64) relates two modules at a singleton signature. A module M_1 is equivalent to M_2 at signature $\mathbb{S}(M_2)$ if M_1 can be typed with the signature $\mathbb{S}(M_2)$. While this rule is straightforward, it is not immediately clear how to handle this rule in an implementation. However, there is a sound and complete algorithm (with respect to the equivalence judgment) for deciding equivalence. See Appendix B for additional details.

4. Related Work

DCH Unlike our calculus, the system described by Dreyer, Crary, and Harper [2] (DCH) does not differentiate between non-generative ($[\![T]\!]$) and generative types ($[\![\nu T]\!]$) in the signature language. Instead, there are two forms of sealing: *weak sealing* and *strong sealing*. Strong sealing induces generative types and weak sealing does not. As a result, their calculus also requires two forms of functor: a *total* functor that does not generate new types, and a *partial* functor that does. Therefore, it is not possible to mix generative and non-generative types within a single module or functor. Total and partial functors can be encoded in our system in the following way:

$$\Pi^{\mathrm{tot}} s{:}\sigma.\,[\![T]\!] \quad\rightsquigarrow\quad \Pi s{:}\sigma.\,[\![T]\!]$$
$$\Pi^{\mathrm{par}} s{:}\sigma.\,[\![T]\!] \quad\rightsquigarrow\quad \Pi s{:}\sigma.\,[\![\nu T]\!]$$

In the DCH system, an effect system is used to track the purity of module expressions. The purity information from the effect system is used in the same manner as we use our syntactic purity judgment (our notion of purity was inspired by DCH). For all-generative or all-non-generative modules, the DCH effect system is able to track purity information more accurately than our syntactic judgment. That is, our system may require the programmer to explicitly write more let bindings. However, we believe that our

(50)
$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash M\ pure}{\Gamma \vdash M \cong M : \sigma}$$

(62)
$$\frac{\begin{array}{c}\Gamma, s : \sigma_1 \vdash M_1 s \cong M_2 s : \sigma_2 \\ \Gamma \vdash M_1 : \Pi s{:}\sigma_1.\rho_1 \qquad \Gamma \vdash M_2 : \Pi s{:}\sigma_1.\rho_2\end{array}}{\Gamma \vdash M_1 \cong M_2 : \Pi s{:}\sigma_1.\sigma_2}$$

(63)
$$\frac{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma_1 \qquad \Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma_2}{\Gamma \vdash M_2 \cong M_2 : \Sigma_{_}{:}\sigma_1.\sigma_2}$$

(64)
$$\frac{\begin{array}{c}\Gamma \vdash M_1 : \sigma' \\ \Gamma, s : \sigma' \vdash M_2 : \sigma \qquad \Gamma \vdash M_1\ pure \qquad \Gamma \vdash M_2\ pure\end{array}}{\Gamma \vdash \texttt{let}\ s = M_1\ \texttt{in}\ M_2{:}\sigma \cong M_2[M_1/s] : \sigma}$$

(65)
$$\frac{\Gamma \vdash M_1 : \mathbb{S}(M_2) \qquad \Gamma \vdash \mathbb{S}(M_2)\ sig}{\Gamma \vdash M_1 \cong M_2 : \mathbb{S}(M_2)}$$

Figure 9. Module Equivalence (Abbreviated)

syntactic judgment is easier to work with, and we were able to extend it to mixed generative and non-generative modules.

Non-dependent Types for Modules Russo's PhD thesis describes a module system that allows both generative and non-generative modules [12]. Unlike our system and the DCH system, Russo's system does not use dependent types. Rather, modules are typed using normal universal and existential types. A generative functor is represented using a type similar to: $\forall\sigma.S \rightarrow \exists\alpha.M$. A non-generative functor has a type similar to: $\exists\alpha.\forall\sigma.S \rightarrow M$. Of course, this last type only makes sense if the existentially quantified types (α) do not depend on the argument types (σ). Therefore, the argument to a non-generative functor cannot have any generative types that are used in the body of the functor. We have a similar restriction in our system, but we must also take into account any generative types produced in the body of a functor that are used to produce non-generative types.

ν-calculus The ν-calculus is an extension of the lambda calculus with explicit names [11, 15]. The generation of names in the ν-calculus is similar to the generation of types in our system. In the ν-calculus, a new name is introduced with the ν-binder. The expression $\nu n.e$ generates the fresh name n for use within the expression e. When deciding the equivalence of ν terms, the generation of new names inside of lambda terms must be taken into account [16]. For instance, the following "non-generative" function returns the same name each time it is applied: $\nu n.\lambda x.n$. This function is not equivalent to the "generative" function $\lambda x.\nu n.n$ which generates a different name each time the function is applied. The important difference between these two functions is position of the ν-binder relative to the λ-binder. In our system, we track the generation of new types relative to the functor binder (Λ) using a split context.

5. Conclusion

In this paper, we have presented a type system for higher-order modules. Using our system, it is possible to declare modules with both generative and non-generative types. We have tried to keep the system as simple as possible, hoping that the formalism can easily lead to an implementation. To this end, we have presented a syntax-directed type system, and used a purely syntactic judgment for deciding the purity of module expressions.

Acknowledgments

I would like to thank my PhD qualifying committee: Greg Morrisett, Norman Ramsey and Mitch Wand for their feedback on an earlier version of this work. Matthew Fluet also helped me greatly with many valuable conversations and feedback. In addition, I would like to thank Claudio Russo for helping me understand his thesis, and Derek Dreyer for pointing out some of the examples used in Section 3.

References

[1] Karl Crary. *Advanced Topics in Types and Programming Languages, Chapter 6*, volume Chapter 6. The MIT Press, 2005.

[2] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *POPL 2003*, pages 236–249. ACM Press, 2003. ISBN 1-58113-628-5.

[3] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theor. Comput. Sci.*, 193(1-2):75–96, 1998. ISSN 0304-3975.

[4] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL 1994*, pages 123–137, Portland, OR, January 1994.

[5] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[6] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[7] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd symposium Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[8] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[9] David B. MacQueen. Using dependent types to express modular structure. In *Proc. of 1986 ACM Symposium on Principles of Programming Languages*, pages 277–286, January 1986. St. Petersburg.

[10] Simon Peyton-Jones and Mark Shields. Practical type inference for arbitrary rank types. Unpublished tutorial on Haskell-style type inference., 2005.

[11] Andrew M. Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *Mathematical Foundations of Computer Science: Proceedings of the 18th International Symposium MFCS '93*, number 711 in Lecture Notes in Computer Science, pages 122–141. Springer-Verlag, 1993.

[12] Claudio V. Russo. Types for modules. In *Electronic Notes in Theoretical Computer Science*, volume 60. January 2003.

[13] Zhong Shao. Transparent modules with fully syntactic signatures. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, page (to appear). ACM Press, September 1999.

[14] Mark A. Sheldon and David K. Gifford. Static dependent types for first class modules. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*,

pages 20–29, New York, NY, USA, 1990. ACM Press. ISBN 0-89791-368-X.

[15] Ian Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, December 1994.

[16] Ian Stark. Names, equations, relations: Practical ways to reason about *new*. In *Typed Lambda Calculi and Applications: Proceedings of the Third International Conference TLCA '97*, number 1210 in Lecture Notes in Computer Science, pages 336–353. Springer-Verlag, 1997.

[17] Christopher A. Stone. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.

[18] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *POPL 2000*, pages 214–227, Boston, January 2000.

[19] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 2005.

A. Type System

Syntax

$$
\begin{array}{llcl}
\text{types} & \tau & ::= & \mathtt{Typ}\ M \mid \widetilde{\Pi}s:\sigma.\tau \\
\text{terms} & e & ::= & \mathtt{Val}\ M \mid \lambda s:\sigma.e \mid eM \\
\text{signatures} & \sigma & ::= & 1 \mid [\![T]\!] \mid [\![\nu T]\!] \mid [\![\tau]\!] \mid \mathbb{S}(M) \\
 & & \mid & \Sigma s{:}\sigma_1.\sigma_2 \mid \Pi s{:}\sigma_1.\sigma_2 \\
\text{modules} & M & ::= & s \mid \langle\rangle \mid [\tau] \mid [e{:}\tau] \\
 & & \mid & \langle s = M_1, M_2\rangle \mid \pi_i M \\
 & & \mid & \Lambda s{:}\sigma.M \mid M_1 M_2 \\
 & & \mid & \mathtt{let}\ s = M_1\ \mathtt{in}\ M_2{:}\sigma \\
 & & \mid & M :: \sigma \\
\text{contexts} & \Gamma & ::= & \cdot \mid \Gamma, s:\sigma
\end{array}
$$

Judgments

Well-formed contexts: $\Gamma \vdash ok$

(1)
$$\frac{}{\cdot \vdash ok}$$

(2)
$$\frac{\Gamma \vdash \sigma\ sig \qquad s \notin \mathtt{dom}(\Gamma)}{\Gamma, s:\sigma \vdash ok}$$

Signature Purity: $\Gamma \vdash \sigma\ pure$

(3)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash 1\ pure}$$

(4)
$$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!]\ pure}$$

(5)
$$\frac{\Gamma \vdash \tau \Rightarrow \sigma}{\Gamma \vdash [\![\tau]\!]\ pure}$$

(6)
$$\frac{\Gamma, s:\sigma_1 \vdash \sigma_2\ pure}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2\ pure}$$

(7)
$$\frac{\Gamma \vdash \sigma_1\ pure \qquad \Gamma, s:\sigma_1 \vdash \sigma_2\ pure}{\Gamma \vdash \Sigma s{:}\sigma_1.\sigma_2\ pure}$$

(8)
$$\frac{\Gamma \vdash M\ pure}{\Gamma \vdash \mathbb{S}(M)\ pure}$$

Module Purity: $\Gamma \vdash M\ pure$

(9)
$$\frac{\Gamma \vdash s:\sigma}{\Gamma \vdash s\ pure}$$

(10)
$$\frac{\Gamma \vdash M\ pure}{\Gamma \vdash \pi_i M\ pure}$$

(11)
$$\frac{\Gamma \vdash M:\sigma \qquad \Gamma \vdash \sigma\ pure}{\Gamma \vdash M\ pure}$$

Well-formed types: $\Gamma \vdash \tau \Rightarrow \sigma$

(12)
$$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash \mathtt{Typ}\ M \Rightarrow [\![T]\!]}$$

(13)
$$\frac{\Gamma_1;\Gamma_2 \vdash M : [\![\nu T]\!] \qquad \Gamma_1;\Gamma_2 \vdash M\ pure \qquad \mathtt{fv}(M) \subseteq \Gamma_1}{\Gamma_1;\Gamma_2 \vdash \mathtt{Typ}\ M \Rightarrow [\![T]\!]}$$

(14)
$$\frac{\Gamma_1;\Gamma_2 \vdash M : [\![\nu T]\!] \qquad \Gamma_1;\Gamma_2 \vdash M\ pure \qquad \mathtt{fv}(M) \nsubseteq \Gamma_1}{\Gamma_1;\Gamma_2 \vdash \mathtt{Typ}\ M \Rightarrow [\![\nu T]\!]}$$

(15)
$$\frac{\Gamma \vdash M : \mathbb{S}(M') \qquad \Gamma \vdash \mathtt{Typ}\ M' \Rightarrow \sigma}{\Gamma \vdash \mathtt{Typ}\ M \Rightarrow \sigma}$$

(16)
$$\frac{\Gamma \vdash \sigma\ sig \qquad \Gamma, s:\sigma \vdash \tau \Rightarrow \sigma'}{\Gamma \vdash \widetilde{\Pi}s:\sigma.\tau \Rightarrow \sigma'}$$

Well-formed terms: $\Gamma \vdash e : \tau$

(17)
$$\frac{\Gamma \vdash M : [\![\tau]\!]}{\Gamma \vdash \mathtt{Val}\ M : \tau}$$

(18)
$$\frac{\Gamma \vdash \sigma\ sig \qquad \Gamma, s:\sigma \vdash e:\tau}{\Gamma \vdash \lambda s:\sigma.e : \widetilde{\Pi}s:\sigma.\tau}$$

(19)
$$\frac{\Gamma \vdash e : \widetilde{\Pi}s:\sigma.\tau \qquad \Gamma \vdash M:\sigma' \qquad \Gamma \vdash \sigma' \leq \sigma \qquad \Gamma \vdash M\ pure}{\Gamma \vdash eM : \tau[M/s]}$$

Well-formed signatures: $\Gamma \vdash \sigma\ sig$

(20) $$\frac{\Gamma \vdash ok}{\Gamma \vdash 1\ sig}$$

(21) $$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!]\ sig}$$

(22) $$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![\nu T]\!]\ sig}$$

(23) $$\frac{\Gamma \vdash \tau \Rightarrow \sigma}{\Gamma \vdash [\![\tau]\!]\ sig}$$

(24) $$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash \mathbb{S}(M)\ sig}$$

(25) $$\frac{\Gamma \vdash M : [\![\nu T]\!]}{\Gamma \vdash \mathbb{S}(M)\ sig}$$

(26) $$\frac{\Gamma \vdash \sigma_1\ sig \qquad \Gamma, s : \sigma_1 \vdash \sigma_2\ sig}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2}$$

(27) $$\frac{\Gamma \vdash \sigma_1\ sig \qquad \Gamma, s : \sigma_1 \vdash \sigma_2\ sig}{\Gamma \vdash \Sigma s{:}\sigma_1.\sigma_2}$$

Signature subtyping: $\Gamma \vdash \sigma_1 \leq \sigma_2$

(28) $$\frac{\Gamma \vdash ok}{\Gamma \vdash 1 \leq 1}$$

(29) $$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!] \leq [\![T]\!]}$$

(30) $$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![\nu T]\!] \leq [\![\nu T]\!]}$$

(31) $$\frac{\Gamma \vdash ok}{\Gamma \vdash [\![T]\!] \leq [\![\nu T]\!]}$$

(32) $$\frac{\Gamma \vdash M \cong M' : [\![T]\!]}{\Gamma \vdash [\![\texttt{Typ}\ M]\!] \leq [\![\texttt{Typ}\ M']\!]}$$

(33) $$\frac{\Gamma \vdash \sigma' \equiv \sigma \qquad \Gamma, s : \sigma \vdash [\tau] \cong [\tau'] : [\![T]\!]}{\Gamma \vdash \left[\!\!\left[\widetilde{\Pi} s : \sigma.\tau\right]\!\!\right] \leq \left[\!\!\left[\widetilde{\Pi} s : \sigma'.\tau'\right]\!\!\right]}$$

(34) $$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \leq [\![T]\!]}{\Gamma \vdash \mathbb{S}(M) \leq [\![T]\!]}$$

(35) $$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \leq [\![\nu T]\!]}{\Gamma \vdash \mathbb{S}(M) \leq [\![\nu T]\!]}$$

(36) $$\frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathbb{S}(M_1) \leq \mathbb{S}(M_2)}$$

(37) $$\frac{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2\ sig \qquad \Gamma \vdash \sigma_1' \leq \sigma_1 \qquad \Gamma, s : \sigma_1' \vdash \sigma_2 \leq \sigma_2'}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2 \leq \Pi s{:}\sigma_1'.\sigma_2'}$$

(38) $$\frac{\Gamma \vdash \Sigma s{:}\sigma_1.\sigma_2\ sig \qquad \Gamma \vdash \sigma_1 \leq \sigma_1' \qquad \Gamma, s : \sigma_1 \vdash \sigma_2 \leq \sigma_2'}{\Gamma \vdash \Sigma s{:}\sigma_1.\sigma_2 \leq \Sigma s{:}\sigma_1'.\sigma_2'}$$

Well-formed modules: $\Gamma \vdash M : \sigma$

(39) $$\frac{\Gamma \vdash ok \qquad \sigma = \Gamma(s)}{\Gamma \vdash s : \mathbb{S}_\sigma(s)}$$

(40) $$\frac{\Gamma \vdash ok}{\Gamma \vdash \langle\rangle : 1}$$

(41) $$\frac{\Gamma \vdash \tau \Rightarrow \sigma}{\Gamma \vdash [\tau] : [\tau]}$$

(42) $$\frac{\Gamma \vdash \tau \Rightarrow \sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash [e{:}\tau] : [\![\tau]\!]}$$

(43) $$\frac{\Gamma_1; \Gamma_2, s : \sigma_1 \vdash M : \sigma_2}{\Gamma_1; \Gamma_2 \vdash \Lambda s{:}\sigma_1.M : \Pi s{:}\sigma_1.\sigma_2}$$

(44) $$\frac{\Gamma \vdash F : \Pi s{:}\sigma_1.\sigma_2 \qquad \Gamma \vdash M : \sigma_1' \qquad \Gamma \vdash \sigma_1' \leq \sigma_1 \qquad \Gamma \vdash M\ pure}{\Gamma \vdash FM : \sigma_2[M/s]}$$

(45) $$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, s : \sigma_1 \vdash M_2 : \sigma_2}{\Gamma \vdash \langle s = M_1, M_2\rangle : \Sigma s{:}\sigma_1.\sigma_2}$$

(46) $$\frac{\Gamma \vdash M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_1 M : \sigma_1}$$

(47) $$\frac{\Gamma \vdash M : \Sigma s{:}\sigma_1.\sigma_2 \qquad \Gamma \vdash M\ pure}{\Gamma \vdash \pi_2 M : \sigma_2[\pi_1 M/s]}$$

(48) $$\frac{\Gamma \vdash M : \sigma' \qquad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash (M :: \sigma) : \sigma}$$

(49) $$\frac{\Gamma \vdash M_1 : \sigma_1 \qquad \Gamma, s : \sigma_1 \vdash M_2 : \sigma' \qquad \Gamma, s : \sigma_1 \vdash \sigma' \leq \sigma \qquad \Gamma \vdash \sigma\ sig}{\Gamma \vdash (\texttt{let}\ s = M_1\ \texttt{in}\ M_2{:}\sigma) : \sigma}$$

Module equivalence: $\Gamma \vdash M_1 \cong M_2 : \sigma$

(50)
$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash M \; pure}{\Gamma \vdash M \cong M : \sigma}$$

(51)
$$\frac{\Gamma \vdash M_2 \cong M_1 : \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma}$$

(52)
$$\frac{\Gamma \vdash M_1 \cong M_2 : \sigma \qquad \Gamma \vdash M_2 \cong M_3 : \sigma}{\Gamma \vdash M_1 \cong M_3 : \sigma}$$

(53)
$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \cong [\tau_2]}$$

(54)
$$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash [\texttt{Typ}\; M] \cong M : [\![T]\!]}$$

(55)
$$\frac{\Gamma \vdash M_1 : 1 \qquad \Gamma \vdash M_2 : 1}{\Gamma \vdash M_1 \cong M_2 : 1}$$

(56)
$$\frac{\Gamma \vdash M_1 : [\![\tau]\!] \qquad \Gamma \vdash M_2 : [\![\tau]\!]}{\Gamma \vdash M_1 \cong M_2 : [\![\tau]\!]}$$

(57)
$$\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 \qquad \Gamma, s : \sigma_1 \vdash M_1 \cong M_2 : \sigma}{\Gamma \vdash \Lambda s : \sigma_1 . M_1 \cong \Lambda s : \sigma_2 . M_2 : \Pi s{:}\sigma_1 . \sigma}$$

(58)
$$\frac{\Gamma \vdash F \cong F' : \Pi s{:}\sigma_1.\sigma_2 \qquad \Gamma \vdash M \cong M' : \sigma_1}{\Gamma \vdash FM \cong F'M' : \sigma_2[M/s]}$$

(59)
$$\frac{\Gamma \vdash M_1 \cong M_1' : \sigma_1 \qquad \Gamma, s : \sigma_1 \vdash M_2 \cong M_2' : \sigma_2}{\Gamma \vdash \langle s = M_1, M_2 \rangle \cong \langle s = M_1', M_2' \rangle : \Sigma s{:}\sigma_1.\sigma_2}$$

(60)
$$\frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma_1}$$

(61)
$$\frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma_2[\pi_1 M/s]}$$

(62)
$$\frac{\Gamma, s : \sigma_1 \vdash M_1 s \cong M_2 s : \sigma_2 \qquad \Gamma \vdash M_1 : \Pi s{:}\sigma_1.\rho_1 \qquad \Gamma \vdash M_2 : \Pi s{:}\sigma_1.\rho_2}{\Gamma \vdash M_1 \cong M_2 : \Pi s{:}\sigma_1.\sigma_2}$$

(63)
$$\frac{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma_1 \qquad \Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma_2}{\Gamma \vdash M_2 \cong M_2 : \Sigma_{:}\sigma_1.\sigma_2}$$

(64)
$$\frac{\Gamma \vdash M_1 : \sigma' \qquad \Gamma, s : \sigma' \vdash M_2 : \sigma \qquad \Gamma \vdash M_1 \; pure \qquad \Gamma \vdash M_2 \; pure}{\Gamma \vdash \texttt{let}\; s = M_1 \;\texttt{in}\; M_2 {:} \sigma \cong M_2[M_1/s] : \sigma}$$

(65)
$$\frac{\Gamma \vdash M_1 : \mathbb{S}(M_2) \qquad \Gamma \vdash \mathbb{S}(M_2) \; sig}{\Gamma \vdash M_1 \cong M_2 : \mathbb{S}(M_2)}$$

(66)
$$\frac{\Gamma \vdash M_1 \cong M_2 : \sigma' \qquad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma}$$

B. Algorithmic Equivalence

The equivalence relation described in the last section is type sensitive. Because of this, we cannot use the typical "reduce and compare" strategy for deciding equivalence. The reduce and compare strategy requires the normal forms of two modules with the same signature to be equal. However, in the presence of unitary types (such as the empty signature or singleton signatures) this is not the case[4]. For example, two different variables that both have the empty signature should be equivalent even though their normal forms are not equal. In addition, if two modules are impure, then they are not equivalent even if they are syntactically equal. These considerations lead us to develop a type sensitive equivalence algorithm [1].

The equivalence algorithm is divided into two sub algorithms. These two algorithms rely on the weak head reduction relation shown in Figure B. The reduction relation reduces a module expression to a *path*. Paths are defined by the following grammar:

$$P ::= s \mid [\lambda s : \sigma.\tau] \mid PM \mid \pi_1 P \mid \pi_2 P$$

Figure B also shows the rules for computing the *natural signature* of a path. The natural signature rules are derived from the module typing rules by removing any rules that introduce singletons, and then restricting our attention only to paths. The reduction relation is standard except for the rule for reducing a path to a module if the path's natural signature is a singleton. If a path has a singleton signature, then the path should be equivalent to the module that the singleton is formed over. Therefore, we can reduce the path to the module and continue to reduce the expression. This one reduction rule is central to the equivalence algorithm; it allows us to reduce modules at a singleton signature to a weak head normal form that is unique for that signature.

The equivalence algorithm is shown in Figure 11. The primary algorithm is type directed; it is specified by rules 70-75. Modules compared at empty or singleton signatures are equivalent since these signatures are unitary. Rules 72 and 73 rely on the weak head reduction relation. For modules compared at the signatures $[\![T]\!]$ and $[\![\nu T]\!]$, we first reduce each module to a weak head normal form. If the modules are being compared at the signature $[\![\nu T]\!]$, then the normal forms must be variables and they must be equal. This corresponds with our definition of purity. If the modules are being compared at the signature $[\![T]\!]$, then we reduce them to paths, and then use the second sub algorithm for comparing paths. The only two other cases are for dependent functions and pairs. These are compared extensionally, thus reducing them to one of the other four cases.

The second sub algorithm compares modules that are in weak head normal form; it is specified by rules 76-80. Rule 77 relies on signature equivalence. Recall that signature equivalence is defined in terms of the signature subtyping judgment. This judgment is syntax directed, and can be easily implemented. Note that both Rule 77 and Rule 78 will use the primary algorithm when comparing modules that may not be paths.

The equivalence algorithm described above is essentially identical to the algorithm described by Stone and Harper [18]. Stone and Harper have proven this algorithm sound and complete for the $\lambda^{\Pi\Sigma S}_{\leq}$ calculus, upon which our system is based. Their proof has been extended to richer language for describing modules, similar to our own, by Dreyer et al. [2].

[4] Stone and Harper [19] have given a normalization algorithm for $\lambda^{\Pi\Sigma S}_{\leq}$ that can be used to decide equivalence in the presence of singleton types. However, our calculus has, in addition, a unit type (the empty signature), and generative types that we must still deal with in a type sensitive way.

Natural Signatures

$$\begin{array}{ll}
\Gamma \rhd [\tau] \uparrow [\![T]\!] & \\
\Gamma \rhd s \uparrow \Gamma(s) & \\
\Gamma \rhd PM \uparrow \sigma[M/s] & if\ \Gamma \rhd P \uparrow \Pi s{:}\sigma'.\sigma \\
\Gamma \rhd \pi_1 M \uparrow \sigma_1 & if\ \Gamma \rhd P \uparrow \Sigma s{:}\sigma_1.\sigma_2 \\
\Gamma \rhd \pi_2 M \uparrow \sigma_2[\pi_1 M/s] & if\ \Gamma \rhd P \uparrow \Sigma s{:}\sigma_1.\sigma_2
\end{array}$$

Weak Head Reduction

$$\begin{array}{lll}
\Gamma \rhd (\Lambda s{:}\sigma.M)M' & \rightsquigarrow M[M'/s] & \\
\Gamma \rhd \pi_1\langle s = M_1, M_2\rangle & \rightsquigarrow M_1 & \\
\Gamma \rhd \pi_2\langle s = M_1, M_2\rangle & \rightsquigarrow M_2[M_1/s] & \\
\Gamma \rhd \mathtt{let}\ s = M_1\ \mathtt{in}\ M_2{:}\sigma & \rightsquigarrow M_2[M_1/s] & \\
\Gamma \rhd [\mathtt{Typ}\ M] & \rightsquigarrow M & \\
\Gamma \rhd P & \rightsquigarrow M & if\ \Gamma \rhd P \uparrow \mathbb{S}(M) \\
\Gamma \rhd FM & \rightsquigarrow F'M & if\ \Gamma \rhd F \rightsquigarrow F' \\
\Gamma \rhd \pi_1 M & \rightsquigarrow \pi_1 M' & if\ \Gamma \rhd M \rightsquigarrow M' \\
\Gamma \rhd \pi_2 M & \rightsquigarrow \pi_2 M' & if\ \Gamma \rhd M \rightsquigarrow M'
\end{array}$$

Figure 10. Weak Head Reduction

(74) $$\frac{}{\Gamma \rhd M_1 \Leftrightarrow M_2 : 1}$$

(75) $$\frac{}{\Gamma \rhd M_1 \Leftrightarrow M_2 : \mathbb{S}(M)}$$

(76) $$\frac{\Gamma \rhd M_1 \rightsquigarrow^* s \qquad \Gamma \rhd M_1 \rightsquigarrow^* s}{\Gamma \rhd M_1 \Leftrightarrow M_2 : [\![\nu T]\!]}$$

(77) $$\frac{\Gamma \rhd M_1 \rightsquigarrow^* P_1 \qquad \Gamma \rhd M_2 \rightsquigarrow^* P_2 \qquad \Gamma \rhd P_1 \leftrightarrow P_2}{\Gamma \rhd M_1 \Leftrightarrow M_2 : [\![T]\!]}$$

(78) $$\frac{\Gamma \rhd \pi_1 M \Leftrightarrow \pi_1 M' : \sigma_1 \qquad \Gamma \rhd \pi_2 M \Leftrightarrow \pi_2 M' : \sigma_2[\pi_1 M/s]}{\Gamma \rhd M \Leftrightarrow M' : \Sigma s{:}\sigma_1.\sigma_2}$$

(79) $$\frac{\Gamma, s : \sigma_a \rhd Ms \Leftrightarrow M's : \sigma}{\Gamma \rhd M \Leftrightarrow M' : \Pi s{:}\sigma_a.\sigma}$$

....

(80) $$\frac{}{\Gamma \rhd s \leftrightarrow s : \Gamma(s)}$$

(81) $$\frac{\Gamma \vdash \sigma \equiv \sigma' \qquad \Gamma, s : \sigma \vdash [\tau] \Leftrightarrow [\tau'] : [\![T]\!]}{\Gamma \rhd [\lambda s : \sigma.\tau] \leftrightarrow [\lambda s : \sigma'.\tau'] : [\![T]\!]}$$

(82) $$\frac{\Gamma \rhd P \leftrightarrow P' : \Pi s{:}\sigma_a.\sigma \qquad \Gamma \rhd M \Leftrightarrow M' : \sigma_a}{\Gamma \rhd PM \leftrightarrow P'M' : \sigma[M/s]}$$

(83) $$\frac{\Gamma \rhd P \leftrightarrow P' : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \rhd \pi_1 P \leftrightarrow \pi_1 P' : \sigma_1}$$

(84) $$\frac{\Gamma \rhd P \leftrightarrow P' : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \rhd \pi_2 P \leftrightarrow \pi_2 P' : \sigma_2[\pi_1 P/s]}$$

Figure 11. Algorithmic Equivalence Rules

C. Prototype Implementation

C.1 Type Checker

```
data Entry t = E t | I t deriving(Eq,Show)
isE,isI :: Entry t → Bool
isE e = case e of { E _ → True; otherwise → False }
isI e = case e of { I _ → True; otherwise → False }


type Tc = TcMonad.TcM (Entry Sig)
trace = TcMonad.trace
failed = TcMonad.failed


lookupVar :: Var → Tc Sig
lookupVar v = do s ← TcMonad.lookupVar v
                 case s of (E t) → return t
                           (I t) → return t


allE :: [Var] → Tc Bool
allE vs = mapM TcMonad.lookupVar vs >>= return ∘ all isE


extendEnvI :: Var → Sig → Tc a → Tc a
extendEnvI name ty = TcMonad.extendEnv name (I ty)


hasIEntries :: Tc Bool
hasIEntries = TcMonad.foldEnv (\_ s b → b || isI s) False


extendEnv :: Var → Sig → Tc a → Tc a
extendEnv name ty tcm = do b ← hasIEntries
                           let t = if b then I ty else E ty
                           TcMonad.extendEnv name t tcm
```

C.1.1 Labeled Singletons

```
singleton :: Sig → Mod → Sig
singleton sig mod =
    case sig
    of Unit        → Unit
       AbsType s   → Singleton mod
       GenType s   → Singleton mod
       TermSig ty  → TermSig ty
       Singleton m → Singleton mod
       Pi  v s1 s2 → Pi  v s1 (singleton s2 (App mod (Var v)))
       Sum v s1 s2 → let m1 = Proj1 mod
                         m2 = Proj2 mod
                         s2' = subst m1 v s2
                     in Sum v (singleton s1 m1) (singleton s2' m2)
```

C.1.2 Purity

```
pureSig :: Sig → Tc ()
pureSig sig =
    trace ("pureSig ? " ++ show sig) >>
    case sig
    of Unit        → return ()
       AbsType T   → return ()
       GenType T   → fail "signature is not pure"
       TermSig t   → wfType t >> return ()
       Pi  s s1 s2 → extendEnv s s1 (pureSig s2)
       Sum s s1 s2 → pureSig s1 >> extendEnv s s1 (pureSig s2)
       Singleton m → pure m


pure :: Mod → Tc ()
pure (Var v)   = return ()
pure (Proj1 m) = pure m
pure (Proj2 m) = pure m
pure mod       = sigOf mod >>= pureSig
```

C.1.3 Well-formed Types

```
wfType :: Type → Tc Sig
wfType (Typ mod) =
    trace ("type projection from module "++ show mod) >>
    sigOf mod >>= \sig →
    case sig
    of AbsType T   → return (AbsType T)
       GenType T   → do pure mod
                        b ← allE (fv mod)
                        return (if b then AbsType T else GenType T)
       Singleton m → wfType (Typ m)
       otherwise   → fail ("Type projection not well formed from :"
                           ++ show mod ++ " with sig " ++ show sig)

wfType (TPi v s ty) = wfSig s >> extendEnv v s (wfType ty)
```

C.1.4 Well-formed Terms

```
typeOf :: Term → Tc Type
typeOf (Val m)      = do s ← sigOf m
                        case s of (TermSig ty) → return ty
                                  otherwise    → fail "invalid type"
typeOf (TLam v s e) = do wfSig s
                        ty ← extendEnv v s $ typeOf e
                        return $ TPi v s ty
typeOf (TApp e m)   = do (TPi v s ty) ← typeOf e
                        s' ← principal m
                        subSig s' s
                        pure m
                        return $ subst m v ty
```

C.1.5 Well-formed Signatures

```
wfSig :: Sig → Tc ()
wfSig  Unit           = return ()
wfSig (AbsType T)     = return ()
wfSig (GenType T)     = return ()
wfSig (TermSig t)     = wfType t >> return ()
wfSig (Singleton mod) = sigOf mod >>= \sig →
                        case sig
                        of AbsType T → return ()
                           GenType T → return ()
                           otherwise → fail ("Signature not well formed: "
                                             ++ show sig ++ " for mod "
                                             ++ show mod)
wfSig (Sum v s1 s2)   = wfSig s1 >> extendEnv v s1 (wfSig s2)
wfSig (Pi  v s1 s2)   = wfSig s1 >> extendEnv v s1 (wfSig s2)
```

C.1.6 Signature Subtyping

```
eqTy :: Type → Type → Tc ()
eqTy t1 t2 =
    trace ("eqTy ? " ++ show t1 ++ " and " ++ show t2) >>
    case (t1,t2)
    of (Typ m, Typ n)            → equiv m n (AbsType T)
       (TPi v s t, TPi v' s' t') → do subSig s  s'
                                      subSig s' s
                                      let t'' = subst (Var v) v' t'
                                      extendEnv v s $ equiv (Type t) (Type t'') (AbsType T)


subSig :: Sig → Sig → Tc ()
subSig sig1 sig2 =
    trace ("subSig ? " ++ show sig1 ++ " and " ++ show sig2) >>
    case (sig1,sig2)
    of (Unit,Unit)                  → return ()
       (AbsType T, AbsType T)       → return ()
       (GenType T, GenType T)       → return ()
       (AbsType T, GenType T)       → return ()
       (TermSig t1, TermSig t2)     → eqTy t1 t2
       (Singleton m, AbsType T)     → do s ← sigOf m
                                         subSig s (AbsType T)
       (Singleton m, GenType T)     → do s ← sigOf m
                                         subSig s (GenType T)
       (Singleton m, Singleton n)   → equiv m n (AbsType T)
```

```
       (Pi v s1 s2, Pi v' s1' s2')   → do wfSig sig1
                                          subSig s1' s1
                                          let s2'' = subst (Var v) v' s2'
                                          extendEnv v s1' (subSig s2 s2'')
       (Sum v s1 s2, Sum v' s1' s2') → do wfSig sig1
                                          subSig s1 s1'
                                          let s2'' = subst (Var v) v' s2'
                                          extendEnv v s1' (subSig s2 s2'')
       (sig1, sig2)                  → fail "not subtypes"
```

C.1.7 Well-formed Modules

```
sigOf :: Mod → Tc Sig
sigOf mod = principal_int (\s m → s) mod

principal :: Mod → Tc Sig
principal mod = principal_int singleton mod

principal_int :: (Sig → Mod → Sig) → Mod → Tc Sig
principal_int singleton mod =
    case mod
    of (Var v)         → do { s ← lookupVar v; return (singleton s mod) }
       (Empty)         → return Unit
       (Type ty)       → do { s ← wfType ty  ; return (singleton s mod) }
       (Term tm ty)    → do wfType ty
                            typeOf tm >>= eqTy ty
                            return (TermSig ty)
       (Lambda v s m)  → do s2 ← extendEnvI v s (principal m)
                            return (Pi v s s2)
       (App f m)       → do (Pi v s1 s2) ← principal f
                            s1' ← principal m
                            subSig s1' s1
                            pure m
                            return (subst m v s2)
       (Pair v m1 m2)  → do s1 ← principal m1
                            s2 ← extendEnv v s1 (principal m2)
                            return (Sum v s1 s2)
       (Proj1 m)       → do (Sum v s1 s2) ← principal_int singleton m
                            return s1
       (Proj2 m)       → do (Sum v s1 s2) ← principal_int singleton m
                            pure m
                            return $ subst (Proj1 m) v s2
       (Seal m s)      →  do s' ← principal m
                             subSig s' s
                             return s
       (Let v m1 m2 s) → do s1 ← principal m1
                            s' ← extendEnv v s1 (principal m2)
                            extendEnv v s1 (subSig s' s)
                            wfSig s
                            return s --(singleton s mod)  --- FIX ME
```

C.2 Module Equivalence

```
isPath :: Mod → Bool
isPath (Var s)      = True
isPath (Term tm ty) = True
isPath (App f m)    = isPath f
isPath (Proj1 m)    = isPath m
isPath (Proj2 m)    = isPath m
isPath otherwise    = False

natural :: Mod → Tc Sig
natural (Term tm ty) = return (TermSig ty)
natural (Var v)      = lookupVar v
natural (App f m)    = do (Pi v s1 s2) ← natural f
                          return (subst m v s2)
natural (Proj1 m)    = do (Sum v s1 s2) ← natural m
                          return s1
natural (Proj2 m)    = do (Sum v s1 s2) ← principal m
                          return (subst (Proj1 m) v s2)
```

```
whrStep :: Mod → Tc Mod
whrStep (App (Lambda v s m) m') = return (subst m' v m)
whrStep (Proj1 (Pair v m1 m2))  = return m1
whrStep (Proj2 (Pair v m1 m2))  = return (subst m1 v m2)
whrStep (Let v m1 m2 s)         = return (subst m1 v m2)
whrStep (Type (Typ m))          = return m
whrStep p | isPath p            = do trace ("whrStep "++ show p)
                                     nat ← natural p
                                     trace (" ◦. → nat is " ++ show nat)
                                     case nat of (Singleton m) → return m
                                                 otherwise     → return p

whrStep (App f m) = do { f ← whrStep f; return (App f m) }
whrStep (Proj1 m) = do { m ← whrStep m; return (Proj1 m) }
whrStep (Proj2 m) = do { m ← whrStep m; return (Proj2 m) }
whrStep  m        = return m

whr :: Mod → Tc Mod
whr m = do trace ("whr of "++ show m)
           m' ← whrStep m
           trace (" ◦.whr to "++ show m')
           if m = m' then return m else whr m'

eqPath :: Mod →  Mod → Sig → Tc ()
eqPath m1 m2 sig =
    trace ("eqPath ? " ++ show m1 ++ " and " ++ show m2) >>
    case (m1,m2)
    of (Type ty1, Type ty2)         → eqTy ty1 ty2
       (Var v1, Var v2) | v1 = v2   → return ()
                        | otherwise → fail ("eqPath failed " ++ v1 ++ " and " ++ v2)
       (App f1 m1, App f2 m2)       → eqPath f1 f2 sig >> equiv m1 m2 sig
       (Proj1 m, Proj1 n)           → eqPath m n sig
       (Proj2 m, Proj2 n)           → eqPath m n sig

equiv :: Mod → Mod → Sig → Tc ()
equiv m1 m2 sig =
    trace ("equiv ? " ++ show m1 ++ " and " ++
           show m2 ++ " at " ++ show sig) >>
    case sig
    of Unit        → return ()
       Singleton m → return ()
       TermSig ty  → fail "equiv"
       GenType T   → do m1 ← whr m1
                        m2 ← whr m2
                        case (m1,m2)
                             of (Var x, Var y) | x = y → return ()
                                otherwise → fail "equiv 2"
       AbsType T   → do m1 ← whr m1
                        m2 ← whr m2
                        eqPath m1 m2 sig
       Sum v s1 s2 → extendEnv v s1 $ equiv (App m1 (Var v)) (App m2 (Var v)) s2
       Pi  v s1 s2 → do equiv (Proj1 m1) (Proj1 m2) s1
                        extendEnv v s1 $ equiv (Proj2 m1) (Proj2 m2)
                                                (subst (Proj1 m1) v s2)
```

Type Checker Monad TcMonad Based on TcMonad from Peyton-Jones and Shields [10].

```
module TcMonad (
TcM,Gamma,
runTc,failed,trace,clearTrace,
newTyVar,
lookupVar,extendEnv,foldEnv,mapEnv
)
where
import qualified Data.Map as Map

type Name    = String
type ErrMsg  = String
type State   = (String,Int)
type Gamma t = Map.Map Name t

newtype TcM t a = TcM { unTcM :: Gamma t → State → (Either ErrMsg a, State) }

instance Functor (TcM t) where
    fmap f m = TcM $ \g s → let (a,s') = unTcM m g s
                             in (either Left (Right ∘ f) a, s')

instance Monad (TcM t) where
    return a = TcM $ \g s → (Right  a, s)
    fail msg = TcM $ \g s → (Left (fst s ++ msg), s)
    m >>= k  = TcM $ \g s → let (a,s') = unTcM m g s
                             in case a of Left msg → (Left msg, s')
                                          Right x  → unTcM (k x) g s'

runTc :: [(Name,t)] → TcM t a → Either ErrMsg a
runTc bindings tcm = fst $ unTcM tcm (Map.fromList bindings) ("", 1)

failed :: TcM t () → TcM t Bool
failed tcm = TcM $ \g s → (Right ∘left.fst $ unTcM tcm g s, s)
    where left = either (\_ → True) (\_ → False)

trace :: String → TcM t ()
trace msg = TcM $ \g (tr,i) → (Right (), (tr++msg++"\n",i))

clearTrace :: TcM t ()
clearTrace = TcM $ \g (_,i) → (Right (), ("",i))

—— State ——

getNextInt :: TcM t Int
getNextInt = TcM $ \g (tr,i) → (Right i, (tr,i+1))

getTrace :: TcM t String
getTrace = TcM $ \g s → (Right (fst s), s)

newTyVar :: Name → TcM t Name
newTyVar name = getNextInt >>= return ∘ (name ++) ∘ show

—— Env ——

getEnv :: TcM t (Gamma t)
getEnv = TcM $ \g s → (Right g, s)

lookupVar :: Name → TcM t t
lookupVar name = do env ← getEnv
                    case Map.lookup name env of
                     Just t  → return t
                     Nothing → fail $ "could not find name: " ++ name

extendEnv :: Name → t → TcM t a → TcM t a
extendEnv name t tcm = TcM $ \g s → unTcM tcm (Map.insert name t g) s

foldEnv :: (Name → t → a → a) → a → TcM t a
foldEnv f i = getEnv >>= return ∘ (Map.foldWithKey f i)

mapEnv :: (Name → t → t) → TcM t (Gamma t)
mapEnv f = getEnv >>= return ∘ (Map.mapWithKey f)
```