Transparent Modules with Fully Syntactic Signatures[*]

Zhong Shao
Dept. of Computer Science
Yale University
New Haven, CT 06520
shao@cs.yale.edu

Abstract

ML-style modules are valuable in the development and maintenance of large software systems, unfortunately, none of the existing languages support them in a fully satisfactory manner. The official SML'97 Definition does not allow higher-order functors, so a module that refers to externally defined functors cannot accurately describe its import interface. MacQueen and Tofte [25] extended SML'97 with fully transparent higher-order functors, but their system does not have a type-theoretic semantics thus fails to support fully syntactic signatures. The systems of manifest types [19, 20] and translucent sums [12] support fully syntactic signatures but they may propagate fewer type equalities than fully transparent functors. This paper presents a module calculus that supports both fully transparent higher-order functors and fully syntactic signatures (and thus true separate compilation). We give a simple type-theoretic semantics to our calculus and show how to compile it into an F_ω-like λ-calculus extended with existential types.

1 Introduction

Modular programming is one of the most commonly used techniques in the development and maintenance of large software systems. Using modularization, we can decompose a large software project into smaller pieces (modules) and then develop and understand each of them in isolation. The key ingredients in modularization are the explicit interfaces used to model inter-module dependencies. Good interfaces not only make separate compilation type-safe but also allow us to think about large systems without holding the whole system in our head at once. A powerful module language must support equally expressive interface specifications in order to achieve the optimal results.

1.1 Why higher-order functors?

Standard ML [26, 27] provides a powerful module system. The main innovation of the ML module language is its support of parameterized modules, also known as *functors*. Unlike Modula-3 generics [28] or C++ templates [35], ML functors can be type-checked and compiled independently at its definition site; furthermore, different applications of the same functor can share a single copy of the implementation (i.e., object code), even though each application may produce modules with different interfaces.

Functors have proven to be valuable in the modeling and organization of extensible systems [1, 10, 6, 29]. The Fox project at CMU [1] uses ML functors to represent the TCP/IP protocol layers; through functor applications, different protocol layers can be mixed and matched to generate new protocol stacks with application-specific requirements. Also, a standard C++ template library written using the ML functors would not require nasty cascading recompilations when the library is updated, simply because ML functors can be compiled separately before even being applied.

Unfortunately, any use of functors and nested modules also implies that the underlying module language must support higher-order functors (i.e., functors passed as arguments or returned as results by other functors), because otherwise, there is no way to accurately specify the import signature of a module that refers to externally defined functors. For example, if we decompose the following ML program into two smaller pieces, one for FOO and another for BAR:

```
functor FOO (A : SIG) = ...
......
structure BAR = struct structure B = ...
                       structure C = FOO(B)
                end
```

the fragment for BAR must treat FOO as its import argument. This essentially turns BAR into a higher-order functor since it must take another functor as its argument. Without higher-order functors, we cannot fully specify the interfaces[1] of arbitrary ML programs. The lack of fully syntactic (i.e., explicit) signatures also violates the fundamental principles of modularization and makes it impossible to support Modula-2 style true separate compilation [19].

1.2 Main challenges

Supporting higher-order functors with fully syntactic signatures turns out to be a very hard problem. Standard ML (SML) [27] only supports first-order functors. MacQueen and Tofte [25, 36] extended SML with fully transparent higher-order functors but their scheme does not provide fully syntactic signatures. Independently, Harper and Lillibridge [12] and Leroy [19] proposed to use translucent sums and manifest types to model type sharing; their scheme

[*] This research was sponsored in part by the Defense Advanced Research Projects Agency ITO under the title "Software Evolution using HOT Language Technology," DARPA Order No. D888, issued under Contract No. F30602-96-2-0232, and in part by an NSF CAREER Award CCR-9501624, and NSF Grant CCR-9633390. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

To appear in ICFP'99: Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming, September 1999, Paris, France.

[1] We only need to write the signatures for first-order functors if we use a special "compilation unit" construct with import and export statements, but reasoning such construct would likely require similar formalism as reasoning higher-order modules.

supports fully syntactic signatures but fails to propagate as much sharing as in the MacQueen-Tofte system. Leroy [20] proposed to use applicative semantics to model full transparency, but his signature calculus is not fully syntactic since it only handles limited forms of functor expressions; this limitation was lifted in Courant's recent proposal [7], but only at the expense of putting arbitrary module implementation code into the interfaces, which in turn compromises the very benefits of modularization and makes interface checking much harder.

The main challenge is thus to design a module language that satisfies all of the following properties:

- It must have *fully syntactic signatures*: if we split a program at an arbitrary point, the corresponding interface must be expressible using the underlying signature calculus.
- It must have *simple type-theoretical semantics*: a clean semantics makes formal reasoning easier; it is also a prerequisite for a simple signature calculus.
- It should support *fully transparent higher-order functors*: higher-order functors should be a *natural* extension of first-order ones; simple ML functors can propagate type sharing from the argument to the result; higher-order functors should propagate sharing in the same way.
- It should support *opaque types and signatures*: type abstraction is the standard method of hiding implementation details from the clients of a module; the same mechanism should be applicable to higher-order functors as well.
- It should support *efficient elaboration and implementation*: a module system will not be practical if it cannot be typechecked and compiled efficiently; compilation of module programs should also be compatible with the standard type-directed compilation techniques [18, 15, 32, 33].

1.3 Our contributions

This paper presents a higher-order module calculus that satisfies all of the above properties. We show that fully transparent higher-order functors can also have simple type-theoretic semantics so they can be added into ML-like languages while still supporting true separate compilation. Our key idea is to adapt and incorporate the phase-splitting interpretation of higher-order modules [14, 33] into a surface module calculus—the result is a new method that propagates more sharing information (across functor application) than the system based on translucent sums [12] and manifest types [19]. More specifically, given a signature or a functor signature S, we extract all the *flexible* components in S into a single higher-order "type-constructor" variable u; here, by flexible, we mean those undefined type or module components inside S. We call such u as the **flexroot** constructor of signature S. We use K to denote the kind of u and S' to denote the instantiation of S whose flexible components are redirected to the corresponding entries in u. An opaque view of signature S can be modeled as an existential type $\exists u : K.S'$. A transparent view of S can be obtained by substituting the flexroot of S with the actual constructor information. Full transparency is then achieved by propagating the flexroot information through functor application.

Our new phase-splitting interpretation also leads to a simpler type theory for the system based on translucent sums and manifest types. Recent work on phase-splitting transformation [14, 33, 8] has shown that ML-like module languages are better understood by translating them into an F_ω-like polymorphic λ-calculus. These translations, however, do not support opaque modules very well because abstract types must be made concrete during the translation. The translation of translucent sums is even more problematic: Crary et al [8] have to extend F_ω with singleton and dependent kinds to capture the sharing information in the surface language. The translation based on our new interpretation (given in the companion technical report [34]) rightly turns opaque modules and abstract types into simple existential types. Furthermore, it does not need to use singleton and dependent kinds. This is significant because typechecking singleton and dependent kinds is notoriously difficult [8].

2 Informal Development

2.1 Fully transparent higher-order functors

We first use a series of examples to show how the MacQueen-Tofte system [25] supports fully transparent higher-order functors. We start by defining a signature SIG and a functor signature FSIG:

```
signature SIG = sig type t  val x : t end
funsig FSIG = fsig (X: SIG): SIG
```

MacQueen and Tofte use strong sum Σ to express the module type, so signature SIG is equivalent to a dependent sum type $SIG = \Sigma t.t$ and signature FSIG is same as the dependent product type $\Pi X : SIG.SIG$. We also define a structure S with signature SIG, and two functors F1 and F2, both with signature FSIG:

```
structure S = struct type t=int val x=1 end

functor F1 (X: SIG) =
  struct type t=X.t val x=X.x end
functor F2 (X: SIG) =
  struct type t=int val x=1 end
```

Although SIG does not define the actual type for t, functor applications such as F1(S) will always re-elaborate the body of F1 with X bound to S, so the type identity of X.t (which is int) is faithfully propagated into the result F1(S). Now suppose we define the following higher-order functor which takes a functor F as argument and applies it to the previously defined structure S:

```
functor APPS (F: FSIG) = F(S)
```

We can then apply APPS to functors F1 and F2:

```
structure R =
  struct structure R1 = APPS(F1)
         structure R2 = APPS(F2)
         val res = (R1.x = R2.x)
         ............
  end
```

In the MacQueen-Tofte system, both APPS(F1) and APPS(F2) will re-elaborate the body of APPS which in turn re-elaborates the functor body in F1 and F2; it successfully infers that R1.x and R2.x all have type int, so the equality test (R1.x = R2.x) will typecheck.

MacQueen and Tofte [25] call functors such as APPS as *fully transparent* modules since they faithfully propagate all sharing information in the actual argument (e.g., F1 and F2) into the result (e.g., R1 and R2). Unfortunately, their scheme does not support fully syntactic signatures. If we want to turn module R into a separate compilation unit, we have no way to completely specify its import interface. More specifically, we cannot write a signature for APPS so that all sharing information in the argument is propagated into the result. The closest we get is to assign APPS with signature:

```
funsig BADSIG =
  fsig (F: FSIG): sig type t=int val x : t end
```

But this would not work if R also contains the following code:

```
functor F3(X: SIG) =
  struct type t=real val x=3.0 end
structure R3 = APPS(F3)
```

Signature BADSIG clearly does not capture the sharing information propagated during the application of APPS(F3). The actual implementation of the MacQueen-Tofte system [33] memoizes a "skeleton" for each functor body to support re-elaboration, but this is clearly too complex to be used in a surface signature calculus.

2.2 Translucent sums and manifest types

A more severe problem of the MacQueen-Tofte system is that it lacks a clean type-theoretic semantics: its typechecker must use an operational stamp generator to model abstract types; this makes it impossible to express the typing property in the surface signature calculus. In 1994, Harper and Lillibridge [12] and Leroy [19] proposed (independently) to use translucent sums and manifest types to model ML modules; the resulting framework—which we call it *the abstract approach*—has a clean type-theoretic equational theory on types; furthermore, both systems support fully syntactic signatures. Leroy [21] and Harper [16] have also shown that their systems are sufficiently expressive that it can type the entire module language in the official SML'97 Definition [27].

Unfortunately, in the case of higher-order functors, the abstract approach does not propagate as much sharing as one would normally expect in the MacQueen-Tofte system. For example, the previous equality test (R1.x = R2.x) would not typecheck in Harper and Leroy's systems [12, 19]. In fact, the abstract approach treats the signature SIG as an existential type $SIG = \exists t.t$ and the signature FSIG as a dependent product $\Pi X : SIG.SIG$. The functor APPS is assigned with the following signature type:

$$\Pi F{:}(\Pi X{:}SIG.SIG).SIG$$

Applying APPS to F1 or F2 always yields a new existential package $\exists t.t$ so R1.t and R2.t are two distinct abstract types.

The abstract approach relies on signature subsumption and strengthening [12, 19] to propagate sharing information from the functor argument to the result. But the subsumption rules are not powerful enough to support fully transparent higher-order functors. Nevertheless, the abstract approach does have fully syntactic signatures; and having a functor parameter returning an abstract result is sometimes useful. Take the functor APPS as an example, sometimes we indeed want the parameter F to be a functor that always generate new types at each application.

2.3 Transparent modules with syntactic signatures

We would like to extend the abstract approach to support fully transparent higher-order functors. Our key idea is to adapt and incorporate the phase-splitting interpretation of higher-order modules [14, 33] into a surface module calculus; the result is a new method that propagates more sharing information (across functor application) than the system based on translucent sums and manifest types. Given a signature or a functor signature S, we extract all the *flexible* components in S into a single higher-order "type-constructor" variable u; here, by flexible, we mean those undefined type or module components inside S. We call such u as the **flexroot** constructor of signature S. We use K to denote the kind of u and S' to denote the instantiation of S with all of its flexible components referring to the corresponding entries in u. An opaque view of signature S can be modeled as an existential type $\exists u : K.S'$; a transparent view of S can be obtained by substituting all occurrences of u in S' by an actual flexroot constructor. For example, the previous signature declaration:

```
signature SIG = sig type t  val x : t end
```

can be viewed as a template of form:

$$SIG = \lambda u{:}K_{SIG}.(\texttt{sig type t = \#t}(u)\texttt{ val x : t end})$$

where kind K_{SIG} is equal to $\{\texttt{t}:\Omega\}$. We use $\#\texttt{t}(u)$ to denote the t component from a constructor record u—this is to emphasize its difference from the module access paths in ML (e.g., X.t).

Instantiating the flexroot of SIG with constructor $\{\texttt{t} = \texttt{int}\}$ yields a signature of form:

```
sig type t = int val x : t end
```

Meanwhile, the following SML code:

```
structure X :> SIG =
  struct type t=int val x=1 end
```

creates an opaque view of SIG so module X has type $\exists u : K_{SIG}.(SIG[u])$, or expanded to:

$$\exists u{:}K_{SIG}.(\texttt{sig type t = \#t}(u)\texttt{ val x : t end}).$$

In the rest of this paper, we follow the abstract approach to treat signature matching as opaque by default. Given a module identifier X and a signature S, we say that X has signature S if X has type $\exists u : K_S.(S[u])$. The abstract flexroot constructor in X can be retrieved using dot notation on existentials [5]—such notation is usually written as $X.\mathbf{typ}$, but in this paper we use a more concise notation; we will use the overlined identifier $\overline{X}$ to represent the flexroot of X.

It is informative to compare flexroot with the notion of access paths in the abstract approach [12, 19]. A type path $X.t$ in the system based on translucent sums and manifest types may denote an abstract type (as in dot notation). Under the flexroot notation, $X.t$ always refers to an actual type definition—the t component of module X—which in turn is defined as type $\#t(\overline{X})$; in other words, all the flexible components in X are now redirected to the abstract flexroot constructor $\overline{X}$.

Combining all the flexible components into a single flexroot constructor makes it easier to propagate sharing information through functor application. For example, the earlier ML code:

```
functor F1(X: SIG) =
  struct type t=X.t val x=X.x end
```

creates a functor with type:[2]

$$\Pi\texttt{X}{:}(\exists u.SIG[u]).(\texttt{sig type t = X.t val x : t end})$$

or written as signature:

```
fsig (X: SIG): sig type t = X.t val x : t end
```

Here, the type path X.t in the result signature really refers to type $\#\texttt{t}(\overline{\texttt{X}})$. During functor application, we create a transparent view of the actual argument following signature SIG; we instantiate the flexroot $\overline{\texttt{X}}$ into an actual constructor and then propagate this information into the result signature.

[2] we omitted the kind annotation for u to simplify the presentation; we will do the same in the rest of the paper if the kind is clear from the context.

The idea gets more interesting in the higher-order case. Because all functors are abstract under the abstract approach, we first need to find a way to introduce transparent higher-order functors. SML'97 uses the "`:`" and "`:>`" notation to distinguish between transparent and opaque signature matching; we borrow the same notation and use it to specify the abstract and transparent functors. In the following example,

```
funsig NFSIG = fsig (X: SIG):> SIG
funsig TFSIG = fsig (X: SIG): SIG
```

Signature NFSIG represents an abstract functor that always creates fresh new types at each application. Signature TFSIG represents a fully transparent functor that always propagates the sharing information from the actual argument (i.e., X) into the result.

The definition of NFSIG introduces a template of form:

$$NFSIG = \lambda u_1 : K_{NFSIG}.\Pi\mathtt{X}:(\exists u_2.SIG[u_2]).(\exists u_3.SIG[u_3])$$

where kind K_{NFSIG} is just $K_{SIG} \rightarrow \{\}$. Notice NFSIG does not propagate any sharing information (u_1) into the result signature; instead, each functor application always returns an existential package. For example, the abstract version of functor APPS:

```
functor APPS (F: NFSIG) = F(S)
```

is assigned with the following interface type:

$$\Pi\mathtt{F}:(\exists u_1.NFSIG[u_1]).(\exists u_2.SIG[u_2])$$

or written as signature:

```
fsig (F: NFSIG):> SIG
```

On the other hand, the definition of TFSIG introduces a template of form:

$$TFSIG = \lambda u_1 : K_{TFSIG}.\Pi\mathtt{X}:(\exists u_2.SIG[u_2]).(SIG[u_1[\overline{\mathtt{X}}]])$$

where kind K_{TFSIG} is equal to $K_{SIG} \rightarrow K_{SIG}$ (the algorithm calculating such kind is given later in Section 3.2). The flexroot of TFSIG has a different kind from that of NFSIG because functors with signature TFSIG propagate more sharing information (e.g., constructor of kind K_{TFSIG}) than those with signature NFSIG. Notice how functor application propagates sharing into the return result: the flexroot of the result is $u_1[\overline{\mathtt{X}}]$ where u_1 is the flexroot of the functor itself and $\overline{\mathtt{X}}$ is the flexroot of the actual argument.

We can now write the fully transparent version of APPS as:

```
functor APPS (F: TFSIG) = F(S)
```

and we can assign it with the following interface type:

$$\Pi\mathtt{F}:(\exists u_1 : K_{TFSIG}.TFSIG[u_1]).(SIG[\overline{\mathtt{F}}[\{\mathtt{t}=\mathtt{S.t}\}]])$$

or if we write it in an extended signature calculus:

```
fsig (F: TFSIG): sig type t= #t(F̄[{t=S.t}])
                     val x: t
                 end
```

With proper syntactic hacks, this signature can even be written as:

```
fsig (F: TFSIG): sig type t = #t(F(S))
                     val x : t
                 end
```

as long as we assume that all module identifiers (e.g., F and S) referred inside the constructor context $\mathtt{\#t}(\cdot)$ are always referring to their constructor counterparts.

Getting back to the earlier example in Section 2.1 where we apply APPS to functors F1 and F2, we see why both R1.t and R2.t are now equivalent to int. To apply APPS to F1 (or F2), we match F1 (or F2) against TFSIG and calculate the flexroot $\overline{\mathtt{F}}$ of the actual argument; $\overline{\mathtt{F}}$ is equal to $\lambda u.\{\mathtt{t} = \#t(u)\}$ for F1 or $\lambda u.\{\mathtt{t}=\mathtt{int}\}$ for F2; in both cases, the t component of the result is $\#t(\overline{\mathtt{F}}[\{t = \mathtt{int}\}])$ which ends up as int.

2.4 Relationship with Leroy's applicative functors

Our syntactic signature looks similar to Leroy's applicative-functor approach [20] where he can also assign APPS with a signature:

```
fsig (F: FSIG): sig type t=F(S).t
                    val x: t
                end
```

This similarity, however, stays only at the surface; the underlying interpretations of the two are completely different. Under the applicative approach, a functor with signature FSIG will always generate the same abstract type if applied to the same argument. Under our scheme, an abstract functor (with signature NFSIG) always generates a new type at each application while a transparent functor (with signature TFSIG) does not. We can simulate applicative functors by opaquely matching a functor against a transparent functor signature. For example,

```
functor F3 :> TFSIG = F1
```

Functor F3 would have type:

$$\exists u_1 : K_{TFSIG}.\Pi\mathtt{X}:(\exists u_2.SIG[u_2]).(SIG[u_1[\overline{\mathtt{X}}]])$$

Because F3 is abstracted over its flexroot information, applying $\overline{\mathtt{F3}}$ to equivalent constructors will still result in equivalent types (e.g., $\mathtt{\#t}(\overline{\mathtt{F3}}[\{\mathtt{t} = \mathtt{int}\}])$).

One problem of the applicative approach is that it solely relies on access paths to propagate sharing. Because access paths are not allowed to contain arbitrary module expressions (doing otherwise may break abstraction), the applicative approach cannot give an accurate signature to the following functor:

```
functor PAPP (F : FSIG) (X : SIG) =
  let structure Y =
        struct type t = X.t * X.t
               val x = X.x * X.x
        end
   in F(Y)
```

Leroy [20] did propose to type PAPP by "lambda-lifting" module Y out of PAPP, but this dramatically alters the program structure, making the module language impractical to program with.

Our approach uses the flexroot constructor to propagate sharing. We can easily give PAPP an accurate signature:

```
fsig (F : TFSIG) (X : SIG) :
  sig type t = #t (F̄({t= #t(X̄) * #t(X̄)}))
      val x : t
  end
```

Notice we use TFSIG rather than NFSIG to emphasize that F is a transparent functor.

Module expression and declaration:

$$\begin{array}{llll}
\textit{path} & p & ::= & x_i \mid p.x \\
\textit{mexp} & m & ::= & p \mid \texttt{str}\ d_1, \ldots, d_n\ \texttt{end} \\
 & & \mid & \texttt{fct}(x_i : S)m \mid p_1(p_2) \\
 & & \mid & (p :> S) \mid \texttt{let}\ d\ \texttt{in}\ m \\
\textit{mdec} & d & ::= & x_i = m \mid t_i = \tau \mid v_i = e
\end{array}$$

Module signature and specification:

$$\begin{array}{llll}
\textit{sig} & S & ::= & \texttt{sig}\ H_1, \ldots, H_n\ \texttt{end} \\
 & & \mid & \texttt{fsig}(x_i : S) :> S' \\
\textit{spec} & H & ::= & x_i : S \mid t_i \mid t_i = \tau \mid v_i : \tau
\end{array}$$

Core language:

$$\begin{array}{llll}
\textit{ctyp} & \tau & ::= & t_i \mid p.t \mid \ldots \\
\textit{cexp} & e & ::= & v_i \mid p.v \mid \ldots
\end{array}$$

Elaboration context:

$$\begin{array}{llll}
\textit{ctxt} & \Gamma & ::= & \varepsilon \mid \Gamma; H
\end{array}$$

Figure 1: Syntax of the abstract module calculus AMC

3 Formalization

In this section we present an Extended Module Calculus (EMC) that supports both fully transparent higher-order functors and fully syntactic signatures. EMC is an extension of Leroy and Harper's abstract module calculus [19, 21, 12] but with support for fully transparent functors. To make the presentation easier to follow, we first define an abstract module calculus that reviews the main ideas behind translucent sums and manifest types. We then present our new EMC calculus and show how it propagates more sharing than the abstract approach.

3.1 The abstract module calculus AMC

We use the Abstract Module Calculus (AMC) [19] as a representative of the system based on translucent sums [12] and manifest types [19, 21]. The syntax of AMC is given in Figure 1. The static semantics for AMC is summarized in Figure 2. The complete typing rules are given in Figures 3 to 5 and in the companion TR [34].

AMC is a typical ML-style module calculus containing constructs such as module expressions (*mexp*), module declarations (*mdec*), module access paths (*path*), signatures (*sig*, specifications (*spec*), core-language types (*ctyp*) and expressions (*cexp*). Following Leroy [21], we use x_i, t_i, and v_i to denote module, type, and value identifiers, and x, t, and v for module, type, and value labels. We assume that each declaration or specification in AMC simultaneously defines an *internal name* (e.g., i) and an *external label* (e.g., x, t, v). Given a structure $\texttt{str}\ d_1, \ldots, d_n\ \texttt{end}$ or a signature $\texttt{sig}\ H_1, \ldots, H_n\ \texttt{end}$, declarations and specifications defined later can refer to those defined earlier using the internal names. However, to access the module components from outside, we must use the access paths such as $p.x$, $p.v$, and $p.t$ where p is another path and x, v, and t are external labels.

Signatures are used to type module expressions. An AMC signature can be either a functor signature or a regular signature that contains an ordered list of module, type, and value specifications. A functor signature is written as $\texttt{fsig}(x_i : S) :> S'$ where S denotes the *argument* signature and S' the *result* signature. We borrow the SML'97 notation "`:`" and "`:>`" for signature matching and use it to specify the abstract and transparent functors. Because AMC only supports abstract functors, a functor signature in AMC always uses `:>` to specify its result. Later in Section 3.2, we'll extend AMC with transparent functor signatures in the form of $\texttt{fsig}(x_i : S) : S'$.

ctxt formation	$\vdash\ \Gamma\ \mathit{ok}$
ctyp formation	$\Gamma\ \vdash\ \tau$
cexp formation	$\Gamma\ \vdash\ e : \tau$
spec formation	$\Gamma\ \vdash\ H$
sig formation	$\Gamma\ \vdash\ S$
mexp formation	$\Gamma\ \vdash\ m : S$
mdec formation	$\Gamma\ \vdash\ d : H$
ctyp equivalence	$\Gamma\ \vdash\ \tau \equiv \tau'$
spec subsumption	$\Gamma\ \vdash\ H \leq H'$
sig subsumption	$\Gamma\ \vdash\ S \leq S'$

Figure 2: Static semantics for AMC: a summary

$$\frac{H_j/p \Rightarrow H'_j \quad \text{for } j = 1, \ldots, n}{(\texttt{sig}\ H_1, \ldots, H_n\ \texttt{end})/p \Rightarrow (\texttt{sig}\ H'_1, \ldots, H'_n\ \texttt{end})}$$

$$(\texttt{fsig}(x_i : S) :> S')/p \Rightarrow (\texttt{fsig}(x_i : S) :> S')$$

$$\frac{S/p.x \Rightarrow S'}{(x_i : S)/p \Rightarrow (x_i : S')} \qquad (v_i : \tau)/p \Rightarrow (v_i : \tau)$$

$$(t_i = \tau)/p \Rightarrow (t_i = \tau) \qquad (t_i)/p \Rightarrow (t_i = p.t)$$

Figure 3: Signature strengthening in AMC

AMC allows two kinds of type specifications: *flexible* type specification (t_i) and *manifest* type specification ($t_i = \tau$). Figure 5 lists the standard signature subsumption rules. Manifest types can be made opaque when matched against flexible specifications. Subsumption on functor signatures is contra-variant on the argument but covariant on the result.

Figure 4 lists the formation rules for the AMC module expressions and declarations. AMC supports the usual set of module constructs such as module access path (p), structure definition ($\texttt{str}\ d_1, \ldots, d_n\ \texttt{end}$), functor definition ($\texttt{fct}(x_i : S)m$), functor application ($p_1(p_2)$), signature matching ($p :> S$), and the `let` expression. Most of the typing rules for AMC are straightforward: signature matching in AMC is done opaquely; to type a `let` expression, the result signature must not contain any references to locally defined module variables (i.e., S is well formed in context Γ but not $\Gamma; H$; see Figure 4).

Type sharing in AMC is propagated through signature strengthening and functor application. Signature strengthening, which is defined in Figure 3, is a variation of dot notation [4]; a module identifier x_i of signature S is strengthened to have signature S/x_i. Functor application (e.g., $p_1(p_2)$) can propagate the sharing information in the argument (p_2) into the result signature—this is achieved by substituting the formal parameter x_i with the actual argument p_2 (see Figure 4).

Unfortunately, this strengthening procedure has no effect on its functor components. In the higher-order case, functor application in AMC does not propagate as much sharing as one would normally expect in the MacQueen-Tofte system. In the following, we show how to extend AMC to support fully transparent functors.

$$\frac{\vdash \Gamma\ ok \quad x_i : S \in \Gamma}{\Gamma \vdash x_i : S/x_i} \qquad \frac{\Gamma \vdash p : \mathtt{sig}\ H_1 \ldots H_k, x'_i : S', \ldots\ \mathtt{end} \quad \rho = \{t_i \mapsto p.t, x_i \mapsto p.x \mid t_i, x_i \in \mathrm{Dom}(H_1 \ldots H_k)\}}{\Gamma \vdash p.x' : \rho(S')}$$

$$\frac{\Gamma; x_i : S \vdash m : S'}{\Gamma \vdash \mathtt{fct}(x_i : S)m : \mathtt{fsig}(x_i : S) :> S'} \qquad \frac{\Gamma \vdash p_1 : \mathtt{fsig}(x_i : S) :> S' \quad \Gamma \vdash p_2 : S'' \quad \Gamma \vdash S'' \leq S}{\Gamma \vdash p_1(p_2) : \{x_i \mapsto p_2\}(S')}$$

$$\frac{\vdash \Gamma\ ok}{\Gamma \vdash \mathtt{str\ end} : \mathtt{sig\ end}} \qquad \frac{\Gamma; H_1; \ldots; H_{k-1} \vdash d_k : H_k \quad k = 1, \ldots, n}{\Gamma \vdash \mathtt{str}\ d_1, \ldots, d_n\ \mathtt{end} : \mathtt{sig}\ H_1, \ldots, H_n\ \mathtt{end}} \qquad \frac{\Gamma \vdash p : S' \quad \Gamma \vdash S' \leq S}{\Gamma \vdash (p :> S) : S}$$

$$\frac{\Gamma \vdash S \quad \Gamma \vdash d : H \quad \Gamma; H \vdash m : S}{\Gamma \vdash \mathtt{let}\ d\ \mathtt{in}\ m : S} \qquad \frac{\Gamma \vdash m : S}{\Gamma \vdash (x_i = m) : (x_i : S)} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash (t_i = \tau) : (t_i = \tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (v_i = e) : (v_i : \tau)}$$

Figure 4: Selected typing rules for AMC: $\Gamma \vdash m : S$ and $\Gamma \vdash d : H$

Rules for reflexivity and transitivity are omitted:

$$\frac{\begin{array}{c} X = H_2, \ldots, H_n \quad \text{and} \quad X' = H'_2, \ldots, H'_m \\ \Gamma \vdash H_1 \leq H'_1 \quad \Gamma; H_1 \vdash \mathtt{sig}\ X\ \mathtt{end} \leq \mathtt{sig}\ X'\ \mathtt{end} \end{array}}{\Gamma \vdash \mathtt{sig}\ H_1, X\ \mathtt{end} \leq \mathtt{sig}\ H'_1, X'\ \mathtt{end}}$$

$$\frac{\begin{array}{c} X = H_1, \ldots, H_n \quad \text{and} \quad X' = H'_1, \ldots, H'_m \\ \Gamma; H_0 \vdash \mathtt{sig}\ X\ \mathtt{end} \leq \mathtt{sig}\ X'\ \mathtt{end} \end{array}}{\Gamma \vdash \mathtt{sig}\ H_0, X\ \mathtt{end} \leq \mathtt{sig}\ X'\ \mathtt{end}}$$

$$\frac{\Gamma \vdash S'_1 \leq S_1 \quad \Gamma; x_i : S'_1 \vdash S_2 \leq S'_2}{\Gamma \vdash \mathtt{fsig}(x_i : S_1) :> S_2 \leq \mathtt{fsig}(x_i : S'_1) :> S'_2}$$

$$\frac{\Gamma \vdash S \leq S'}{\Gamma \vdash (x_i : S) \leq (x_i : S')} \qquad \frac{\Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash (t_i = \tau) \leq (t_i = \tau')}$$

$$\Gamma \vdash (t_i = \tau) \leq t_i \qquad \frac{\Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash (v_i : \tau) \leq (v_i : \tau')}$$

Figure 5: Signature subsumption in AMC

All in AMC (Figure 1) plus:

sig	S	$::=$	$\ldots \mid \mathtt{fsig}(x_i : S) : S'$
ctyp	τ	$::=$	$\ldots \mid \#t(C)$
ctxt	Γ	$::=$	$\ldots \mid \Gamma; u : K$

Module constructor and kind

mcon	C	$::=$	$\overline{x_i} \mid u \mid \lambda u : K.C \mid C_1[C_2]$
		$\mid$	$\{F_1, \ldots, F_n\} \mid \#x(C)$
mcfd	F	$::=$	$t = \tau \mid x = C$
mknd	K	$::=$	$\{Q_1, \ldots, Q_n\} \mid K_1 \rightarrow K_2$
mkfd	Q	$::=$	$t : \Omega \mid x : K$

Figure 6: Syntax of the extended module calculus EMC

3.2 The extended module calculus EMC

The extended module calculus EMC contains the same set of module expressions and declarations as those in AMC. However, EMC uses a different method to propagate sharing information; this allows EMC to support fully transparent higher-order functors. EMC also has a more expressive signature calculus so that all functors in EMC have fully syntactic signatures.

The syntax of EMC is given in Figure 6. The static semantics for EMC is summarized in Figure 7. The complete typing rules are given in Figures 8 to 13 and in Appendix A. Our typing rules can be directly turned into a type-checking algorithm because the signature subsumption rules are only used at functor application and opaque signature matching (the same is true for AMC).

The EMC signature calculus contains two new features that are not present in AMC: one is the new functor signature $\mathtt{fsig}(x_i : S) : S'$ used to specify *transparent* higher-order functors; another is a simple constructor calculus that captures the sharing information (using constructor C and kind K) and a new type expression $\#t(C)$ that selects the type field t from constructor C.

Transparent functors can propagate more sharing than abstract functors. For example, suppose S is defined as $\mathtt{sig}\ t_i\ ,\ x_i : t_i\ \mathtt{end}$, a functor with signature $\mathtt{fsig}(x_i : S) :> S$ corresponds to an *abstract* functor whose application always produces a module with a new abstract type component t. On the other hand, a functor with signature $\mathtt{fsig}(x_i : S) : S$ corresponds to a *fully transparent* functor whose application always propagates the type information from its argument into its result.

The constructor calculus itself (see Figure 6) is similar to those used in the F_ω-like polymorphic λ-calculi. In this paper, we assume all types in the core language have kind Ω; we use u to denote constructor variables; and we use the record kind $\{Q_1, \ldots, Q_n\}$ and function kind $K_1 \rightarrow K_2$ to *type* module constructors. A record constructor consists of a sequence of core-language types (marked by label t) and module constructors (marked by label x). Given a record constructor C, the selection form $\#x(C)$ is a module constructor equivalent to the x field of C while $\#t(C)$ is a core-language type expression equivalent to the t field of C. Figure 8 gives the formation rules for the constructor calculus; other typing rules summarized in Figure 7 are given in Appendix A.

The constructor calculus is designed to faithfully capture the sharing information inside all EMC module constructs. More specifically, given a signature (or a functor signature) S, we extract all the *flexible* components in S into a single constructor variable u; we call such u as the **flexroot** constructor of signature S. We use K to denote the kind of u and S' to denote the instantiation of S whose flexible components are redirected to the corresponding entries in u. An opaque view of signature S can be modeled as an existential type $\exists u : K.S'$. A transparent view of S can be obtained by substituting the flexroot of S with the actual constructor information. Full transparency is then achieved by propagating the

Similar forms as those for AMC (Figure 2) plus:

mcon formation	$\Gamma \vdash C : K$
mcfd formation	$\Gamma \vdash F : Q$
mcon equivalence	$\Gamma \vdash C \equiv C' : K$
mcfd equivalence	$\Gamma \vdash F \equiv F' : Q$
mknd subsumption	$\vdash K \leq K'$
mkfd subsumption	$\vdash Q \leq Q'$

Figure 7: Static semantics for EMC: a summary

$$\frac{\vdash \Gamma \; ok \quad u{:}K \in \Gamma}{\Gamma \vdash u : K} \qquad \frac{\vdash \Gamma \; ok \quad x_i{:}S \in \Gamma}{\Gamma \vdash \overline{x_i} : \mathbf{knd}(S)}$$

$$\frac{\Gamma \vdash F_j : Q_j \quad j = 1, \ldots, n}{\Gamma \vdash \{F_1, \ldots, F_n\} : \{Q_1, \ldots, Q_n\}}$$

$$\frac{\Gamma \vdash C : K' \quad K' = \{\ldots, x{:}K, \ldots\}}{\Gamma \vdash \#x(C) : K}$$

$$\frac{\Gamma; u{:}K \vdash C : K'}{\Gamma \vdash \lambda u{:}K.C : K \to K'}$$

$$\frac{\Gamma \vdash C_1 : K \to K' \quad \Gamma \vdash C_2 : K}{\Gamma \vdash C_1[C_2] : K'}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash (t{=}\tau) : (t{:}\Omega)} \qquad \frac{\Gamma \vdash C : K}{\Gamma \vdash (x{=}C) : (x{:}K)}$$

Figure 8: EMC constructor formation

flexroot information through functor application.

Both K and S' can be calculated easily. Figure 9 shows how to deduce $\mathbf{knd}(S)$— the kind of the flexroot constructor of a module with signature S. Here, $\mathbf{knd}(S) \Rightarrow K$ means that the flexible constructor part of signature S is of kind K and $\mathbf{knd}(H) \Rightarrow Q^*$ means that the flexible part in specification H is of kind field Q^* (which denotes either Q or empty field ε). Notice in addition to the flexible type specifications (t_i), functor specifications are also considered as the flexible components. A *transparent* functor with signature $\texttt{fsig}(x_i{:}S){:}S'$ is treated as a higher-order constructor of kind $K \to K'$ where K and K' are the kinds for S and S'. An *abstract* functor with signature $\texttt{fsig}(x_i : S) : S'$ is treated as a dummy constructor that returns an empty record kind.

Signature S' is calculated using a procedure similar to the idea of signature strengthening, but signature strengthening in EMC is very different from that in AMC: instead of relying on the access path p to propagate sharing, EMC uses the flexroot constructor to strengthen a signature. Given a signature S and a constructor C of kind $\mathbf{knd}(S)$, signature strengthening S/C returns the result of substituting the flexroot constructor in S with C. We use the auxiliary procedures given in Figure 10 to deduce S/C. Here, $S/(C : K) \Rightarrow S'$ means that instantiating S by constructor C of kind K yields signature S', and $H/(C : K) \Rightarrow H'$ means that strengthening specification H by constructor C of kind K yields specification H'. The additional kind parameter is used to identify the flexible components in a signature.

Signature strengthening produces a special form of signature whose type components are fully defined and whose functor components have abstract result signatures. This special form, which we call it *instantiated signature*, can be accurately defined using the following grammar:

$$\mathbf{knd}(\texttt{sig}\; H_1 \ldots H_n \;\texttt{end}) \Rightarrow \{\; \mathbf{knd}(H_1) \ldots \mathbf{knd}(H_n) \;\}$$

$$\frac{\mathbf{knd}(S) \Rightarrow K}{\mathbf{knd}(\texttt{fsig}(x_i{:}S){:}{>}S') \Rightarrow K \to \{\}}$$

$$\frac{\mathbf{knd}(S) \Rightarrow K \qquad \mathbf{knd}(S') \Rightarrow K'}{\mathbf{knd}(\texttt{fsig}(x_i{:}S){:}S') \Rightarrow K \to K'}$$

$$\mathbf{knd}(t_i) \Rightarrow t{:}\Omega \qquad \mathbf{knd}(t_i{=}\tau) \Rightarrow \varepsilon$$

$$\mathbf{knd}(x_i{:}S) \Rightarrow x{:}\mathbf{knd}(S) \qquad \mathbf{knd}(v_i{:}\tau) \Rightarrow \varepsilon$$

Figure 9: EMC kind calculation $\texttt{knd}(S)$

$$S/C \;\text{ is a shorthand of }\; S/(C : \mathbf{knd}(S))$$

$$\frac{H_j/(C : K) \Rightarrow H'_j \qquad j = 1, \ldots, n}{(\texttt{sig}\; H_1 \ldots H_n \;\texttt{end})/(C : K) \Rightarrow \texttt{sig}\; H'_1 \ldots H'_n \;\texttt{end}}$$

$$(\texttt{fsig}(x_i{:}S){:}{>}S')/(C : K) \Rightarrow \texttt{fsig}(x_i{:}S){:}{>}S'$$

$$\frac{K = K' \to K'' \quad S'/(C[\overline{x_i}] : K'') \Rightarrow S''}{(\texttt{fsig}(x_i{:}S){:}S')/(C : K) \Rightarrow \texttt{fsig}(x_i{:}S){:}{>}S''}$$

$$\frac{S/(\#x(C) : K) \Rightarrow S'}{(x_i{:}S)/(C : \{\ldots, x{:}K, \ldots\}) \Rightarrow (x_i{:}S')}$$

$$(t_i)/(C : \{\ldots, t{:}\Omega, \ldots\}) \Rightarrow (t_i{=}\#t(C))$$

$$(t_i{=}\tau)/(C : K) \Rightarrow (t_i{=}\tau) \qquad (v_i{:}\tau)/(C : K) \Rightarrow (v_i{:}\tau)$$

Figure 10: Signature strengthening in EMC

$$\begin{array}{rcl} S^I & ::= & \texttt{sig}\; H^I_1, \ldots, H^I_n \;\texttt{end} \\ & \mid & \texttt{fsig}(x_i{:}S){:}{>}S \\ H^I & ::= & x_i{:}S^I \mid t_i{=}\tau \mid v_i{:}\tau \end{array}$$

Notice under this special form, the argument of a functor signature could still be an arbitrary EMC signature, but the result must always be abstract. The following lemma can be proved by structural induction on the EMC signatures:

Lemma 3.1 *Given an EMC context Γ, a signature S, a kind K, and a constructor C, if $\Gamma \vdash S$ and $\Gamma \vdash C : K$ and $\vdash K \leq \mathbf{knd}(S)$ then S/C is an instantiated signature and $\Gamma \vdash S/C$.*

Figure 11 gives the typing rules for the EMC module expressions and declarations. Intuitively, we say a module expression m has signature S if m has type equal to $\exists u{:}\mathbf{knd}(S).(S/u)$. Given a module x_i of signature S, we use the overlined identifier $\overline{x_i}$ to refer to the flexroot constructor hidden inside x_i. This is a form of dot notation [5] where $\overline{x_i}$ represents the abstract type defined by the existential package x_i. In AMC, signature strengthening is applied to the access identifier (x_i) itself and hidden type components are represented using access paths (p). EMC generalizes this idea so it can propagates more sharing than AMC does.

Figure 12 gives the additional signature subsumption rules for the EMC signatures. Subsumption on transparent functor signa-

Only two of these rules—module identifier and functor application—are different from those for AMC (Figure 4):

$$\frac{\vdash \Gamma\ ok \quad x_i\!:\!S \in \Gamma}{\Gamma \vdash x_i : S/\overline{x_i}} \qquad \frac{\Gamma \vdash p : \texttt{sig}\ H_1\ldots H_k, x_i'\!:\!S', \ldots\ \texttt{end} \quad \rho = \{t_i \mapsto p.t, x_i \mapsto p.x \mid t_i, x_i \in \mathrm{Dom}(H_1\ldots H_k)\}}{\Gamma \vdash p.x' : \rho(S')}$$

$$\frac{\Gamma; x_i\!:\!S \vdash m : S'}{\Gamma \vdash \texttt{fct}(x_i\!:\!S)m : \texttt{fsig}(x_i\!:\!S)\!:>S'} \qquad \frac{\Gamma \vdash p_1 : \texttt{fsig}(x_i\!:\!S)\!:>S' \quad \Gamma \vdash p_2 : S'' \quad \Gamma \vdash S'' \leq S \quad \Gamma \vdash S''\!\downarrow\!\mathbf{knd}(S) \Rightarrow C}{\Gamma \vdash p_1(p_2) : \{\overline{x_i} \mapsto C, x_i \mapsto p_2\}(S')}$$

$$\frac{\vdash \Gamma\ ok}{\Gamma \vdash \texttt{str end} : \texttt{sig end}} \qquad \frac{\Gamma; H_1; \ldots; H_{k-1} \vdash d_k : H_k \quad k = 1, \ldots, n}{\Gamma \vdash \texttt{str}\ d_1, \ldots, d_n\ \texttt{end} : \texttt{sig}\ H_1, \ldots, H_n\ \texttt{end}} \qquad \frac{\Gamma \vdash p : S' \quad \Gamma \vdash S' \leq S}{\Gamma \vdash (p :> S) : S}$$

$$\frac{\Gamma \vdash S \quad \Gamma \vdash d : H \quad \Gamma; H \vdash m : S}{\Gamma \vdash \texttt{let}\ d\ \texttt{in}\ m : S} \qquad \frac{\Gamma \vdash m : S}{\Gamma \vdash (x_i\!=\!m) : (x_i\!:\!S)} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash (t_i\!=\!\tau) : (t_i\!=\!\tau)} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash (v_i\!=\!e) : (v_i\!:\!\tau)}$$

Figure 11: Selected typing rules for EMC: $\Gamma \vdash m : S$ and $\Gamma \vdash d : H$

All subsumption rules in AMC (Figure 5) plus:

$$\frac{\Gamma \vdash S_1' \leq S_1 \quad \Gamma; x_i\!:\!S_1' \vdash S_2 \leq S_2'}{\Gamma \vdash \texttt{fsig}(x_i\!:\!S_1)\!:\!S_2 \leq \texttt{fsig}(x_i\!:\!S_1')\!:\!S_2'}$$

$$\Gamma \vdash \texttt{fsig}(x_i\!:\!S_1)\!:\!S_2 \leq \texttt{fsig}(x_i\!:\!S_1)\!:>S_2$$

$$\frac{S_2 \text{ is an instantiated signature}}{\Gamma \vdash \texttt{fsig}(x_i\!:\!S_1)\!:>S_2 \leq \texttt{fsig}(x_i\!:\!S_1)\!:\!S_2}$$

Figure 12: Signature subsumption in EMC

$$\frac{\Gamma \vdash H_j\!\downarrow\!K \Rightarrow F_j^* \quad j = 1, \ldots, n}{\Gamma \vdash (\texttt{sig}\ H_1 \ldots H_n\ \texttt{end})\!\downarrow\!K \Rightarrow \{F_1^* \ldots F_n^*\}}$$

$$\frac{\vdash K \leq \mathbf{knd}(S) \quad \Gamma; x_i\!:\!S \vdash S'\!\downarrow\!K' \Rightarrow C' \\ C = \lambda u\!:\!K.\{\overline{x_i} \mapsto u\}(C') \quad \Gamma \vdash C : K \rightarrow K'}{\Gamma \vdash (\texttt{fsig}(x_i\!:\!S)\!:>S')\!\downarrow\!(K \rightarrow K') \Rightarrow C}$$

$$\frac{\Gamma \vdash S\!\downarrow\!K \Rightarrow C}{\Gamma \vdash (x_i\!:\!S)\!\downarrow\!\{\ldots, x\!:\!K, \ldots\} \Rightarrow (x\!=\!C)}$$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash (t_i\!=\!\tau)\!\downarrow\!\{\ldots, t\!:\!\Omega, \ldots\} \Rightarrow (t\!=\!\tau)}$$

For all other cases, $\Gamma \vdash H\!\downarrow\!K \Rightarrow \varepsilon$

Figure 13: Narrowing instantiated signatures in EMC

tures is also contra-variant on the argument and covariant on the result. More interestingly, a transparent signature $\texttt{fsig}(x_i\!:\!S_1)\!:\!S_2$ is a subtype of its abstract counterpart $\texttt{fsig}(x_i\!:\!S_1)\!:>S_2$; this is because we can always coerce a transparent functor into an abstract one by blocking all of its sharing information. Finally, an abstract signature $\texttt{fsig}(x_i\!:\!S_1)\!:>S_2$ is a subtype of its transparent counterpart *if* the result S_2 is an instantiated signature; this corresponds to the special case where the abstract version only hides a *dummy* constructor so it should be equivalent to the transparent version.

More specifically, a kind K is a dummy kind if it is $\{\}$, or $K_1 \rightarrow K_2$ where K_2 is a dummy kind, or $\{Q_1, \ldots, Q_n\}$ where all fields Q_i have dummy kinds. Given a context Γ and a constructor C, we say C is a dummy constructor if $\Gamma \vdash C : K$ and K is a dummy kind. A dummy constructor conveys no useful information thus it can be safely eliminated. It is easy to show that if S is an instantiated signature then $\mathbf{knd}(S)$ is a dummy kind.

Only two of the typing rules in Figure 11 are different from those for AMC (in Figure 4): one for module identifier and another for functor application. To access a module identifier x_i, we always strengthen it with its flexroot constructor $\overline{x_i}$. To type functor application $p_1(p_2)$, we first notice that the typing rules for access paths (in Figure 11) satisfies the following property: if $\Gamma \vdash p : S$, then S is an instantiated signature. This observation can be easily established via Lemma 3.1. So we can assume p_1 has signature $\texttt{fsig}(x_i\!:\!S)\!:>S'$ and p_2 has signature S''; furthermore, S'' is an instantiated signature. Typing $p_1(p_2)$ then involves checking if S'' subsumes S, extracting the actual flexroot information in p_2 (let's call it C), and substituting all instances of $\overline{x_i}$ in S' with constructor C and all instances of x_i (not counting $\overline{x_i}$) with access path p_2. Here, the substitution on $\overline{x_i}$ is the key on why can propagate more sharing and support fully transparent higher-order functors.

Constructor C can be extracted from the actual argument signature S'' of p_2 using the signature-narrowing procedure defined in Figure 13. This procedure is called upon instantiated signatures only. Given a context Γ, the deduction $\Gamma \vdash S \downarrow K \Rightarrow C$ extracts the type components from an instantiated signature S and produces a constructor C of kind K; the specification counterpart $\Gamma \vdash H \downarrow K \Rightarrow F^*$ extracts the type components in H and produces either F or empty field ε. We can prove the following lemma using structural induction on the EMC signatures:

Lemma 3.2 *Given an EMC context Γ, a signature S, and an instantiated signature S'', let $K = \mathbf{knd}(S)$, if $\Gamma \vdash S'' \leq S$ and $\Gamma \vdash S''\!\downarrow\!K \Rightarrow C$, then $\Gamma \vdash C : K$.*

Given an EMC context Γ, we say two signatures S and S' are equivalent, denoted as $\Gamma \vdash S \equiv S'$, if and only if both $\Gamma \vdash S \leq S'$ and $\Gamma \vdash S' \leq S$ are true. The following propositions show why the typing rules for EMC can hold together:

Lemma 3.3 *Given an EMC context Γ, a signature S, and an instantiated signature S'', assume $\Gamma \vdash S'' \leq S$ and $\Gamma \vdash p_2 : S''$ and $\Gamma \vdash S''\!\downarrow\!\mathbf{knd}(S) \Rightarrow C$, and let $\rho = \{\overline{x_i} \mapsto C, x_i \mapsto p_2\}$, then (1) given two type expressions τ_1 and τ_2, if $\Gamma; x_i\!:\!S \vdash \tau_1$ and $\Gamma; x_i\!:\!S \vdash \tau_2$ and $\Gamma; x_i\!:\!S \vdash \tau_1 \equiv \tau_2$ then $\Gamma \vdash \rho(\tau_1) \equiv \rho(\tau_2)$; (2) given two instantiated signatures S_1' and S_2', if $\Gamma; x_i : S \vdash S_1'$ and $\Gamma; x_i : S \vdash S_2'$ and $\Gamma; x_i : S \vdash S_1' \equiv S_2'$ then $\Gamma \vdash \rho(S_1') \equiv \rho(S_2')$.*

Theorem 3.4 (unique typing) *Given an EMC context Γ, two signatures S and S', and a module expression m, if $\Gamma \vdash m : S$ and*

$\Gamma \vdash m : S'$ then $\Gamma \vdash S \equiv S'$.

Proof: Expand this theorem to cover module declarations and core language expressions; the generalized version of this theorem can be proved by structural induction on the derivation tree. □

4 Expressiveness

In this section, we show that both the translucent-sum-based calculus and the strong-sum-based calculus can be embedded into our EMC calculus. We also compare EMC with the stamp-based semantics of the MacQueen-Tofte system [25, 33].

4.1 The abstract module calculus AMC

We use the AMC calculus presented in Section 3.1 as a representative for the system based on translucent sums [12] and manifest types [19]. Because AMC is a subset of EMC, the translation from AMC to EMC (denoted as $\lfloor \cdot \rfloor_a$) is just an identity function, We can show that this translation $\lfloor \cdot \rfloor_a$ maps all well typed AMC programs into well-typed EMC programs.

Theorem 4.1 *Given an AMC context Γ, we have*

- *if $\vdash \Gamma\ ok$ is a valid AMC deduction then $\vdash \lfloor\Gamma\rfloor_a\ ok$ is valid in EMC; similarly,*
- *if $\Gamma \vdash \tau$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor\tau\rfloor_a$;*
- *if $\Gamma \vdash e : \tau$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor e\rfloor_a : \lfloor\tau\rfloor_a$;*
- *if $\Gamma \vdash S$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor S\rfloor_a$;*
- *if $\Gamma \vdash m : S$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor m\rfloor_a : \lfloor S\rfloor_a$;*
- *if $\Gamma \vdash d : H$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor d\rfloor_a : \lfloor H\rfloor_a$;*
- *if $\Gamma \vdash \tau \equiv \tau'$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor\tau\rfloor_a \equiv \lfloor\tau'\rfloor_a$.*
- *if $\Gamma \vdash S \leq S'$ then $\lfloor\Gamma\rfloor_a \vdash \lfloor S\rfloor_a \leq \lfloor S'\rfloor_a$.*

Proof: By structural induction on the derivation tree. The main difference between EMC and AMC is the way how module identifiers and functor applications are typed. For the case of module identifiers, we use the following lemma (Lemma 4.2); for the case of functor application, notice the result of any AMC functor signature does not contain any reference to the flexroot constructor $\overline{x_i}$ so the typing rules for AMC and EMC have the same behavior. □

Lemma 4.2 *Given an AMC context Γ, suppose S is an AMC signature and $x_i : S \in \Gamma$, then $\lfloor\Gamma\rfloor_a \vdash \lfloor S/x_i\rfloor_a \equiv \lfloor S\rfloor_a/\overline{x_i}$ is a valid deduction in EMC.*

Proof: Notice S/x_i refers to the strengthening operation for AMC (as in Figure 3) while $\lfloor S\rfloor_a/\overline{x_i}$ refers to the strengthening operation for EMC (as in Figure 10). To prove this lemma, we need to show the following: given an EMC type path $p.t$, let x_i be the root identifier in p, and $F(p)$ denotes the EMC constructor $\overline{x_i}$ if $p = x_i$, and $\#x'(F(p'))$ if $p = p'.x'$, then the judgement $\Gamma \vdash p.t \equiv \#t(F(p))$ is valid in EMC. □

4.2 The transparent module calculus TMC

We use the Transparent Module Calculus (TMC) as a representative of the strong-sum-based approach. The syntax of TMC is given in Figure 14; the static semantics is summarized in Figure 15; the complete typing rules are given in the companion TR [34].

path	p	$::=$	$x \mid \pi_1(p) \mid \pi_2(p)$
mexp	m	$::=$	$p \mid \iota_v(e) \mid \iota_t(\mu) \mid \langle x = m_1, m_2\rangle$
		$\mid$	$\lambda x : S.m \mid p_1(p_2) \mid \texttt{let}\ x = m_1\ \texttt{in}\ m_2$
sig	S	$::=$	$\mathsf{V}(\mu) \mid \mathsf{TYP} \mid \Sigma x : S_1.S_2 \mid \Pi x : S_1.S_2$
ctsp	μ	$::=$	$\pi_t(p) \mid \ldots$
cexp	e	$::=$	$\pi_v(p) \mid \ldots$
mtyp	M	$::=$	$\mathsf{V}(\tau) \mid \mathsf{EQ}(\tau) \mid \Sigma x : M_1.M_2 \mid \Pi x : L.M$
	L	$::=$	$\mathsf{V}(\tau) \mid \mathsf{TYP} \mid \Sigma x : L_1.L_2 \mid \Pi x : L_1.L_2$
ctyp	τ	$::=$	$\pi_t(m') \mid \ldots$
ctme	m'	$::=$	$x \mid \iota_v(e') \mid \iota_t(\tau) \mid \lambda x : L.m'$
		$\mid$	$m'_1(m'_2) \mid \texttt{let}\ x = m'_1\ \texttt{in}\ m'_2$
		$\mid$	$\langle x = m'_1, m'_2\rangle \mid \pi_1(m') \mid \pi_2(m')$
ctce	e'	$::=$	$\pi_v(m') \mid \ldots$
ctxt	Γ	$::=$	$\varepsilon \mid \Gamma; x : M \mid \Gamma; x : L$

Figure 14: Syntax of the transparent module calculus TMC

ctxt formation	$\vdash \Gamma\ ok$
mtyp formation	$\Gamma \vdash M$ and $\Gamma \vdash L$
ctyp formation	$\Gamma \vdash \tau$
ctme formation	$\Gamma \vdash m' : M$
ctce formation	$\Gamma \vdash e' : \tau$
cexp formation	$\Gamma \vdash e : \tau$
mexp formation	$\Gamma \vdash m : M$
sig formation	$\Gamma \vdash S$
ctsp formation	$\Gamma \vdash \mu$
ctyp equivalence	$\Gamma \vdash \tau_1 \equiv \tau_2$
mtyp equivalence	$\Gamma \vdash M_1 \equiv M_2$ or $\Gamma \vdash L_1 \equiv L_2$
mtyp subsumption	$\Gamma \vdash M \leq L$
mtyp strengthening	$L/m' \Rightarrow M$

Figure 15: Static semantics for TMC: a summary

Following other strong-sum-based module systems [25, 27, 33], we distinguish module signatures (S) from module types (M and L): module signatures are source-level specifications while module types are semantic objects used for typechecking.

A module signature can either contain a single value specification ($\mathsf{V}(\mu)$), a single type specification (TYP), or a pair of two other module components ($\Sigma x : S_1.S_2$); it can also be a functor signature ($\Pi x : S_1.S_2$). Only simple access paths ($\pi_t(p)$) are allowed in a specification.[3] An L-shaped module type is like a module signature except that in its value specification $\mathsf{V}(\tau)$, core type τ can contain arbitrary module expressions (m'). M-shaped module types are slightly different from L-shaped ones: they allow *manifest types* (or type abbreviations) of form $\mathsf{EQ}(\tau)$ but no flexible type specification of form TYP. The module expression m' inside the core type τ helps achieve the fully transparent propagation of the sharing information in TMC.

A module expression in TMC can either be an access path (p), a single-value-component module ($\iota_v(e)$), a single-type-component module ($\iota_t(e)$), a strong sum of two module components ($\langle x = m_1, m_2\rangle$), a functor ($\lambda x : S.m$), a functor application ($p_1(p_2)$), or a `let` expression.

To simplify the presentation, we restrict the TMC functor application to work on simple access paths only (i.e., $p_1(p_2)$). Arbitrary functor applications (e.g., $m_1(m_2)$) can just be A-normalized into the restricted form using `let` expressions. We also do not support

[3] The Standard ML signature calculus [27, 26] enforces a similar restriction.

ctxt-to-ctxt translation	$\lfloor \Gamma \rfloor_n \mapsto \Gamma$
ctsp-to-ctyp translation	$\lfloor \mu \rfloor_n \mapsto \tau$
sig-to-sig translation	$\lfloor S \rfloor_n \mapsto S$
cexp-to-cexp translation	$\lfloor e \rfloor_n \mapsto e$
path-to-path translation	$\lfloor p \rfloor_n \mapsto p$
mexp-to-mexp translation	$\lfloor m \rfloor_n \mapsto m$
ctyp-to-ctyp translation	$\Gamma \vdash \tau \leadsto \tau'$
mtyp-to-sig translation	$\Gamma \vdash M \leadsto S$
mtyp-to-sig translation	$\Gamma \vdash L \leadsto S$
mtyp-to-kind translation	$\lfloor M \rfloor_c \mapsto K$
mtyp-to-kind translation	$\lfloor L \rfloor_c \mapsto K$
mtyp-to-mcon translation	$\Gamma \vdash M \leadsto C$
ctme-to-mcon translation	$\Gamma \vdash m' : M \leadsto C$

Figure 16: Translation from TMC to EMC: a summary

type abbreviations in signatures. We insist that M be a subtype of L if they have same number of components (see the subtyping rules $\Gamma \vdash M \leq L$ in the companion TR [34]). These restrictions do not affect the main result because it is easy (but tedious) to extend TMC and the TMC-to-EMC translation to support the additional features.

Figure 16 summarizes the translation from TMC to EMC; the actual definition is given in the companion TR [34]. Here, $\lfloor \cdot \rfloor_n$ maps TMC contexts, core types (in signatures), signatures, core expressions, access paths, and module expressions into their EMC counterparts; $\lfloor \cdot \rfloor_c$ maps TMC module types into EMC kinds. The translation from TMC types to EMC types is based on the type formation rules, so the judgement $\Gamma \vdash \tau \leadsto \tau'$ maps the TMC core type τ into an EMC core type τ'; the judgements $\Gamma \vdash M \leadsto S$ and $\Gamma \vdash L \leadsto S$ map the TMC module types M or L into an EMC signature S. We also use judgements $\Gamma \vdash M \leadsto C$ and $\Gamma \vdash m' : M \leadsto C$ to map TMC module types and expressions (embedded inside core types) into EMC constructors. We can prove the following type preservation theorem for the TMC-to-EMC translation:

Theorem 4.3 *Given a TMC context* Γ*, we have:*

- *if* $\vdash \Gamma\ ok$ *is a valid deduction in TMC, then* $\vdash \lfloor \Gamma \rfloor_n\ ok$ *is valid in EMC; similarly,*
- *if* $\Gamma \vdash \mu$ *then* $\lfloor \Gamma \rfloor_n \vdash \lfloor \mu \rfloor_n$*;*
- *if* $\Gamma \vdash S$ *then* $\lfloor \Gamma \rfloor_n \vdash \lfloor S \rfloor_n$*;*
- *if* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash \tau \leadsto \tau'$ *then* $\lfloor \Gamma \rfloor_n \vdash \lfloor e \rfloor_n : \tau'$*;*
- *if* $\Gamma \vdash p : M$ *and* $\Gamma \vdash M \leadsto S$ *then* $\lfloor \Gamma \rfloor_n \vdash \lfloor p \rfloor_n : S$*;*
- *if* $\Gamma \vdash m : M$ *and* $\Gamma \vdash M \leadsto S$ *then* $\lfloor \Gamma \rfloor_n \vdash \lfloor m \rfloor_n : S$*;*
- *if* $\Gamma \vdash \tau \leadsto \tau'$ *then* $\lfloor \Gamma \rfloor_n \vdash \tau'$*;*
- *if* $\Gamma \vdash M \leadsto S$ or $\Gamma \vdash L \leadsto S$ *then* $\lfloor \Gamma \rfloor_n \vdash S$*;*
- *if* $\Gamma \vdash M \leadsto S_1$ *and* $\Gamma \vdash L \leadsto S_2$ *and* $\Gamma \vdash M \leq L$ *and* $\lfloor \Gamma \rfloor_n \vdash S_1 {\downarrow} \lfloor L \rfloor_c \Rightarrow C$ *then* $\lfloor \Gamma \rfloor_n \vdash S_1 \leq S_2 / C$*;*
- *if* $\Gamma \vdash M_1 \equiv M_2$ *and* $\Gamma \vdash M_1 \leadsto C_1$ *and* $\Gamma \vdash M_2 \leadsto C_2$*, then* $\lfloor M_1 \rfloor_c \equiv \lfloor M_2 \rfloor_c$ *and* $\lfloor \Gamma \rfloor_n \vdash C_1 \equiv C_2 : \lfloor M_1 \rfloor_c$*.*
- *if* $\Gamma \vdash m' : M \leadsto C_1$ *and* $\Gamma \vdash M \leadsto C_2$ *then* $\lfloor \Gamma \rfloor_n \vdash C_1 \equiv C_2 : \lfloor M \rfloor_c$*.*

Proof: By structural induction on the derivation tree; along the process, we need to use the following two lemmas. □

Lemma 4.4 *Given a TMC context* Γ*, suppose* $\Gamma \vdash m_1' : M_1$*, let* $\rho = \{x \mapsto m_1'\}$*, then*

- *if* $\Gamma; x\!:\!M_1 \vdash M_2$ *then* $\Gamma; x\!:\!M_1 \vdash \rho(M_2) \equiv M_2$*.*
- *if* $\Gamma; x\!:\!M_1 \vdash L_2$ *then* $\Gamma; x\!:\!M_1 \vdash \rho(L_2) \equiv L_2$*.*
- *if* $\Gamma; x\!:\!M_1 \vdash \tau$ *then* $\Gamma; x\!:\!M_1 \vdash \rho(\tau) \equiv \tau$*.*
- *if* $\Gamma; x\!:\!M_1 \vdash m' : M_2$ *then* $\Gamma; x\!:\!M_1 \vdash \rho(m') : M_2$*.*

Lemma 4.5 *Given a TMC context* Γ*, a TMC module type* M*, an EMC constructor* C*, and an EMC kind* K*, if* $\lfloor \Gamma; x\!:\!M \rfloor_n \vdash C : K$ *is valid in EMC, then* $\lfloor \Gamma \rfloor_n \vdash C : K$ *is valid in EMC as well.*

4.3 Comparison with the stamp-based semantics

Compilers for the strong-sum-based calculus [25, 33] use stamps to support type generativity and abstract types (TMC did not include these features). There are still higher-order module programs that are supported by the stamp-based semantics but not by our type-theoretic semantics. Take the higher-order functor `APPS` in Section 2.1 as an example and consider applying it to the following functors:

```
functor G1(X: SIG) = X
functor G2(X: SIG) = struct abstype t = A
                            with val x = A
                            end
                     end
```

Both applications are legal under the stamp-based semantics: applying `APPS` to `G1` results in a module whose `t` component is equal to `int` while applying `APPS` to `G2` creates a module whose `t` component is a new abstract type. Under our scheme, the transparent version of `APPS` cannot be applied to `G2`; the abstract version works for both but it does not propagate sharing when applied to `G1`. We believe this lack of expressiveness is not a problem in practice.

5 Implementation

A module system will not be practical if it cannot be type-checked and compiled efficiently. Our EMC calculus can be checked efficiently following the typing rules given in Section 3.2; the only nontrivial aspect of the elaboration is on how to efficiently test the equivalence between two arbitrary EMC types; we plan to use the realization-based approach used in the SML/NJ compiler [33] to propagate type definitions.

EMC is also compatible with the standard type-directed compilation techniques [18, 15, 32, 33]. Most of these techniques are developed in the context of F_ω-like polymorphic lambda calculi [11, 30]. In the companion technical report [34], we define a Kernel Module Calculus (KMC) and show how to translate EMC into KMC and then translate KMC into an F_ω-like target calculus.

6 Related Work

Module systems have been an active research area in the past decade. The ML module system was first proposed by MacQueen [24] and later incorporated into Standard ML [26]. Harper and Mitchell [13] show that the SML'90 module language can be translated into a typed lambda calculus (XML) with dependent types. Together with Moggi, they later show that even in the presence of dependent types, type-checking of XML is still decidable [14], thanks to the phase-distinction property of ML-style modules. The SML'90 module language, however, contains several major problems; for example, type abbreviations are not allowed in signatures, opaque signature matching is not supported, and modules are first-order only. These problems were heavily researched [12, 19, 20, 23, 36, 25, 17] and mostly resolved in SML'97 [27]. The main remaining issue is to design a higher-order module calculus that satisfies all of the properties mentioned in the beginning of this paper (see Section 1.2).

Supporting higher-order functors with fully syntactic signatures turns out to be a very hard problem. In addition to the work discussed at the begining of Section 1.2, Biswas [2] gives a semantics for the MacQueen-Tofte modules based on simple polymorphic types. His formulation differs from the phase-splitting semantics [14, 33] in that he does not treat functors as higher-order type constructors. As a result, his scheme requires encoding certain type components of kind Ω using higher-order types—this significantly complicates the type-checking algorithm. Russo [31]'s recent work is an extension of Biswas's semantics to support opaque modules; he uses the existentials to model type generativity, but his type-checking algorithm still relies on the use of higher-order matching as in Biswas [2].

7 Conclusions

A long-standing open problem on ML-style module systems is to design a calculus that supports both fully transparent higher-order functors and fully syntactic signatures. In his Ph.D. thesis [23, page 310] Mark Lillibridge made the following assessment on the difficulty of this problem:

> In principle it should be possible to build a system with a rich enough type system so that both separate compilation and full transparency can be achieved at the same time. Because separate compilation requires that all information needed for type checking the uses of a functor be expressible in that functor's interface, this goal will require functor interfaces to (optionally) contain an idealized copy of the code for the functor whose behavior they specify, I expect such a system to be highly complicated and hard to reason about.

This paper shows that fully transparent higher-order functors can also have simple type-theoretic semantics, so they can be added to ML-like languages while still supporting true separate compilation. Our solution only involves a conservative extension over the system based on translucent sums and manifest types: modules that do not use transparent higher order functors can still have the same signature as before.

The new insight on full transparency also improves our understanding about other module constructs. Harper et al [14] and Shao [33] have given a type-preserving translation from ML-like module languages to polymorphic λ-calculus F_ω. Their phase-splitting translations, however, do not handle opaque modules well—abstract types must be made concrete during the translation. Our new translation (given in the companion TR [34]) rightly turns opaque modules and abstract types into simple existential types.

Higher-order functors and fully syntactic signatures allow us to accurately express the linking process of ML module programs inside the module language itself. In the future we plan to use the module calculus presented in this paper to formalize the configuration language used in the SML/NJ Compilation Manager [3]. We also plan to extend our module calculus to support dynamic linking [22] and mutually recursive compilation units [9, 8].

Acknowledgement

I would like to thank Karl Crary, Robert Harper, Xavier Leroy, David MacQueen, Claudio Russo, Valery Trifonov, and the ICFP program committee for their comments and suggestions on an early version of this paper.

References

[1] E. Biagioni, R. Harper, P. Lee, and B. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *1994 ACM Conference on Lisp and Functional Programming*, pages 55–64, New York, June 1994. ACM Press.

[2] S. K. Biswas. Higher-order functors with transparent signatures. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 154–163, New York, Jan 1995. ACM Press.

[3] M. Blume. A compilation manager for SML/NJ. as part of SML/NJ User's Guide, 1995.

[4] L. Cardelli and X. Leroy. Abstract types and the dot notation. In *Proc. Programming Concepts and Methods*, pages 479–504. North Holland, 1990.

[5] L. Cardelli and D. MacQueen. Persistence and type abstraction. In M. P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 31–41. Springer-Verlag, 1988.

[6] S. Corrico, B. Ewbank, T. Griffin, J. Meale, and H. Trickey. A tool for developing safe and efficient database transactions. In *15th International Switching Symposium of the World Telecommunications Congress*, pages 173–177, April 1995.

[7] J. Courant. An applicative module calculus. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development: LNCS Vol 1214*, pages 622–636, New York, 1997. Springer-Verlag.

[8] K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Proc. SIGPLAN '99 Symp. on Prog. Language Design and Implementation*, page (to appear). ACM Press, May 1999.

[9] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation*, pages 236–248. ACM Press, 1998.

[10] L. George. MLRISC: Customizable and reusable code generators. Technical memorandum, Lucent Bell Laboratories, Murray Hill, NJ, 1997.

[11] J. Y. Girard. *Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur.* PhD thesis, University of Paris VII, 1972.

[12] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 123–137, New York, Jan 1994. ACM Press.

[13] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Trans. on Programming Languages and Systems*, 15(2):211–252, April 1993.

[14] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Seventeenth Annual ACM Symp. on Principles of Prog. Languages*, pages 341–344, New York, Jan 1990. ACM Press.

[15] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 130–141, New York, Jan 1995. ACM Press.

[16] R. Harper and C. Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[17] M. P. Jones. Using parameterized signatures to express modular structure. In *Twenty-third Annual ACM Symp. on Principles of Prog. Languages*, pages 68–78, New York, Jan 1996. ACM Press.

[18] X. Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 177–188, New York, Jan 1992. ACM Press. Longer version available as INRIA Tech Report.

[19] X. Leroy. Manifest types, modules, and separate compilation. In *Twenty-first Annual ACM Symp. on Principles of Prog. Languages*, pages 109–122, New York, Jan 1994. ACM Press.

[20] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-second Annual ACM Symp. on Principles of Prog. Languages*, pages 142–153, New York, Jan 1995. ACM Press.

[21] X. Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):1–32, September 1996.

[22] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proc. ACM SIGPLAN '98 Conf. on Object-Oriented Programming Systems, Languages, and applications*, pages 36–44, New York, October 1998. ACM Press.

[23] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1997. Tech Report CMU-CS-97-122.

[24] D. MacQueen. Modules for Standard ML. In *1984 ACM Conference on Lisp and Functional Programming*, pages 198–207, New York, August 1984. ACM Press.

[25] D. MacQueen and M. Tofte. A semantics for higher order functors. In *The 5th European Symposium on Programming*, pages 409–423, Berlin, April 1994. Spinger-Verlag.

[26] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

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

[28] G. Nelson, editor. *Systems programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[29] J. H. Reppy and E. R. Gansner. The eXene library manual. SML/NJ documentations, March 1991.

[30] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation, Lecture Notes in Computer Science, volume 19*, pages 408–425. Springer-Verlag, Berlin, 1974.

[31] C. V. Russo. *Types For Modules*. PhD thesis, University of Edinburgh, Edinburgh, UK, June 1998.

[32] Z. Shao. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 85–98, New York, June 1997. ACM Press.

[33] Z. Shao. Typed cross-module compilation. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 141–152. ACM Press, September 1998.

[34] Z. Shao. Transparent modules with fully syntactic signatures (extended version). Technical Report YALEU/DCS/RR-1181, Dept. of Computer Science, Yale Univ., New Haven, CT, June 1999.

[35] B. Stroustrup, editor. *The C++ Programming Languages, Third Edition*. Addison Wesley, Reading, MA, 1998.

[36] M. Tofte. Principal signatures for high-order ML functors. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 189–199, New York, Jan 1992. ACM Press.

A Static Semantics for EMC

This appendix gives the rest of the typing rules for the external module calculus EMC. The formation rules for module constructors ($\Gamma \vdash C : K$), and module constructor fields ($\Gamma \vdash F : Q$) are given in Figure 8 in Section 3.2. The formation rules for module expressions ($\Gamma \vdash m : S$) and module declarations ($\Gamma \vdash d : H$) are given in Figure 11 in Section 3.2. The subsumption rules for signatures ($\Gamma \vdash S \leq S'$) and specifications ($\Gamma \vdash H \leq H'$) are given in Figure 12 in Section 3.2.

A.1 ctxt formation: $\vdash \Gamma \ ok$

$$\vdash \varepsilon \ ok \tag{1}$$

$$\frac{\Gamma \vdash H}{\vdash \Gamma; H \ ok} \tag{2}$$

$$\frac{\vdash \Gamma \ ok \quad u \notin dom(\Gamma)}{\vdash \Gamma; u\!:\!K \ ok} \tag{3}$$

A.2 ctyp formation: $\Gamma \vdash \tau$

$$\frac{\vdash \Gamma \ ok \quad t_i \in \Gamma \text{ or } t_i = \tau \in \Gamma}{\Gamma \vdash t_i} \tag{4}$$

$$\frac{\Gamma \vdash p : S \quad S = \{\ldots, t_i = \tau, \ldots\} \text{ or } \{\ldots, t_i, \ldots\}}{\Gamma \vdash p.t} \tag{5}$$

$$\frac{\Gamma \vdash C : K \quad K = \{\ldots, t\!:\!\Omega, \ldots\}}{\Gamma \vdash \#t(C)} \tag{6}$$

A.3 cexp formation: $\Gamma \vdash e : \tau$

$$\frac{\vdash \Gamma \ ok \quad v_i\!:\!\tau \in \Gamma}{\Gamma \vdash v_i : \tau} \tag{7}$$

$$\frac{\begin{array}{c}\Gamma \vdash p : \texttt{sig}\ H_1, \ldots, H_k, \ldots, H_n\ \texttt{end} \\ \rho = \{t_i \mapsto p.t, x_i \mapsto p.x \mid t_i, x_i \in \mathrm{Dom}(X)\} \\ \text{where } X = H_1, \ldots, H_{k-1} \text{ and } H_k = v_i\!:\!\tau\end{array}}{\Gamma \vdash p.v : \rho(\tau)} \tag{8}$$

A.4 sig formation: $\Gamma \vdash S$

$$\frac{\vdash \Gamma \ ok}{\Gamma \vdash \texttt{sig end}} \tag{9}$$

$$\frac{\begin{array}{c}\forall x_i \in \mathrm{Dom}(H_1 \ldots H_{k-1}), \overline{x_i} \text{ is not free in } H_k \\ \Gamma; H_1; \ldots; H_{k-1} \vdash H_k \qquad k = 1, \ldots, n\end{array}}{\Gamma \vdash \texttt{sig}\ H_1, \ldots, H_n\ \texttt{end}} \tag{10}$$

The side condition on H_k in this rule is not absolutely necessary. If we remove this requirement, we essentially allow free flexroot references such as $\overline{x_i}$ even when x_i is a locally declared structure component. To make this work, we need revise the EMC signature-strengthening operation so that all references to $\overline{x_i}$ are substituted by equivalent constructors that have no such references. The new routine is shown in Figure 17 where S/C is now implemented as $S/(C : \mathbf{knd}(S)); \emptyset$ with $\emptyset$ denoting the identity substitution. The auxiliary procedures $S/(C : K); \rho \Rightarrow S'$ means that instantiating S by constructor C of kind K under substitution ρ yields signature S', and $H/(C\!:\!K); \rho \Rightarrow H'; \rho'$ means that strengthening specification H by constructor C of kind K under substitution ρ yields specification H' and new substitution ρ'.

$$\frac{\Gamma \vdash S \quad \Gamma; x_i : S \vdash S'}{\Gamma \vdash \texttt{fsig}(x_i\!:\!S) : S'} \tag{11}$$

$$\frac{\Gamma \vdash S \quad \Gamma; x_i : S \vdash S'}{\Gamma \vdash \texttt{fsig}(x_i\!:\!S) :> S'} \tag{12}$$

$$S/C \text{ is a shorthand of } S/(C:\mathbf{knd}(S));\emptyset$$

$$\frac{\rho_0=\rho \quad H_j/(C{:}K);\rho_{j-1}\Rightarrow H'_j;\rho_j \quad j=1,\ldots,n}{(\texttt{sig}\ H_1\ldots H_n\ \texttt{end})/(C{:}K);\rho\Rightarrow \texttt{sig}\ H'_1\ldots H'_n\ \texttt{end}}$$

$$(\texttt{fsig}(x_i{:}S){:>}S')/(C{:}K);\rho\Rightarrow \texttt{fsig}(x_i{:}S){:>}S'$$

$$\frac{K=K'\to K'' \quad S'/(C[\overline{x_i}]{:}K'');\rho\Rightarrow S''}{(\texttt{fsig}(x_i{:}S){:}S')/(C{:}K);\rho\Rightarrow \texttt{fsig}(x_i{:}S){:>}S''}$$

$$\frac{S/(\#x(C){:}K);\rho\Rightarrow S' \quad \rho'=\rho\uplus\{\overline{x_i}\mapsto\#x(C)\}}{(x_i{:}S)/(C{:}\{\ldots,x{:}K,\ldots\});\rho\Rightarrow(x_i{:}S');\rho'}$$

$$(t_i)/(C{:}\{\ldots,t{:}\Omega,\ldots\});\rho\Rightarrow(t_i{=}\#t(C));\rho$$

$$(t_i{=}\tau)/(C{:}K);\rho\Rightarrow(t_i{=}\rho(\tau));\rho$$

$$(v_i{:}\tau)/(C{:}K);\rho\Rightarrow(v_i{:}\rho(\tau));\rho$$

Figure 17: Signature strengthening in EMC (revised)

A.5 spec formation: $\Gamma\ \vdash\ H$

$$\frac{\Gamma\ \vdash\ S \quad x_i\notin dom(\Gamma)}{\Gamma\ \vdash\ x_i{:}S} \tag{13}$$

$$\frac{\vdash\ \Gamma\ ok \quad t_i\notin dom(\Gamma)}{\Gamma\ \vdash\ t_i} \tag{14}$$

$$\frac{\Gamma\ \vdash\ \tau \quad t_i\notin dom(\Gamma)}{\Gamma\ \vdash\ t_i{=}\tau} \tag{15}$$

$$\frac{\Gamma\ \vdash\ \tau}{\Gamma\ \vdash\ v_i{:}\tau} \tag{16}$$

A.6 ctyp equivalence: $\Gamma\ \vdash\ \tau\equiv\tau'$

Rules for congruence, reflexivity, symmetry, and transitivity are omitted.

$$\frac{\vdash\ \Gamma\ ok \quad t_i{=}\tau\in\Gamma}{\Gamma\ \vdash\ t_i\equiv\tau} \tag{17}$$

$$\frac{\begin{array}{c}\Gamma\ \vdash\ p:\texttt{sig}\ H_1,\ldots,H_k,\ldots,H_n\ \texttt{end}\\ \rho=\{t_i\mapsto p.t, x_i\mapsto p.x \mid t_i,x_i\in\mathrm{Dom}(X)\}\\ \text{where } X{=}H_1,\ldots,H_{k-1} \text{ and } H_k=t'_i{=}\tau\end{array}}{\Gamma\ \vdash\ p.t'\equiv\rho(\tau)} \tag{18}$$

$$\frac{\Gamma\ \vdash\ C\equiv\{\ldots,t{=}\tau,\ldots\}}{\Gamma\ \vdash\ \#t(C)\equiv\tau} \tag{19}$$

A.7 mcon equivalence: $\Gamma\ \vdash\ C\equiv C':K$

Rules for congruence, reflexivity, symmetry, and transitivity are omitted.

$$\frac{\rho=\{u\mapsto C'\} \quad \Gamma\ \vdash\ C':K \quad \Gamma;u{:}K\ \vdash\ C:K'}{\Gamma\ \vdash\ (\lambda u{:}K.C)[C']\equiv\rho(C):K'} \tag{20}$$

$$\frac{\Gamma\ \vdash\ C:K'}{\Gamma\ \vdash\ (\lambda u{:}K.C[u])\equiv C:K\to K'} \tag{21}$$

$$\frac{\Gamma\ \vdash\ C\equiv\{\ldots,x{=}C',\ldots\}:\{\ldots,x{:}K',\ldots\}}{\Gamma\ \vdash\ \#x(C)\equiv C':K'} \tag{22}$$

$$\frac{\begin{array}{c}K=\{Q_1,\ldots,Q_n\} \qquad \Gamma\ \vdash\ C:K\\ \Gamma\ \vdash\ F_j\equiv(l{=}\#l(C)):Q_j \qquad j=1,\ldots,n; l=x,t;\end{array}}{\Gamma\ \vdash\ \{F_1,\ldots,F_n\}\equiv C:K} \tag{23}$$

A.8 mcfd equivalence: $\Gamma\ \vdash\ F\equiv F':Q$

Rules for congruence, reflexivity, symmetry, and transitivity are omitted.

A.9 mknd subsumption: $\vdash\ K\le K'$

Rules for reflexivity and transitivity are omitted.

$$\frac{\begin{array}{c}\sigma:\{1,\ldots,m\}\mapsto\{1,\ldots,n\}\\ \vdash\ Q_{\sigma(j)}\le Q'_j \qquad j=1,\ldots,m\end{array}}{\vdash\ \{Q_1,\ldots,Q_n\}\le\{Q'_1,\ldots,Q'_m\}} \tag{24}$$

$$\frac{\vdash\ K'_1\le K_1 \quad \vdash\ K_2\le K'_2}{\vdash\ K_1\to K_2\le K'_1\to K'_2} \tag{25}$$

A.10 mkfd subsumption: $\vdash\ Q\le Q'$

Rules for reflexivity and transitivity are omitted.

$$\frac{\vdash\ K\le K'}{\vdash\ x{:}K\le x{:}K'} \tag{26}$$