Parameterized Signatures and Higher-Order Modules*

Zhong Shao
Dept. of Computer Science
Yale University
P.O.Box 208285
New Haven, CT 06520-8285
shao-zhong@cs.yale.edu

Technical Report YALEU/DCS/TR-1161, Revised
August 1998

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 language does not allow higher-order functors, so there is no way to accurately specify the import signature of a module that refers to externally defined functors. This lack of fully syntactic signatures makes it impossible to support Modula-2 style true separate compilation. Recently, MacQueen and Toft proposed an extension to support fully transparent higher-order modules, but their scheme still does not provide fully syntactic signatures, partly because they use a rather operational stamp-based semantics to model abstract types. In this paper, we present a module calculus that supports both fully transparent higher-order modules and fully syntactic signatures (and thus true separate compilation). We give a simple type-theoretic semantics to our calculus, and show how it can be compiled into an Fω-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.

*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-9601624, and NSF Grant CCR-8633390. 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.

1.1 Why higher-order modules?

Standard ML [24, 25] 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 [34] which are more or less macros, 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 can produce modules with completely different interfaces.

Functors have proven to be extremely valuable in the modeling and organization of extensible systems [1, 8, 5, 29]. For example, 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 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); for 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:

```ml
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 have no hope of fully specifying the interfaces 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 [16].
1.2 Main challenges

Supporting higher-order functors with fully syntactic signatures turns out to be a very hard problem. Standard ML [25] only supports first-order functors. MacQueen and Tofte [23, 35] recently proposed to add fully transparent higher-order modules to ML but their scheme does not provide fully syntactic signatures. Independently, Harper and Lillibrige [10] and Leroy [16] proposed to use translucent signatures to model abstract types and type sharing; their scheme supports fully syntactic signatures, but fails to propagate as much sharing as in the MacQueen-Tofte system. Leroy [17] proposed to use applicative semantics to model full transparency, but his signature calculus only handles limited forms of functor arguments; this limitation was lifted in Courant's recent proposal [6], 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 it easier to carry out formal reasonings; it is also a prerequisite for a simple signature calculus.

- It should provide supports to **fully transparent higher-order modules** [23]. 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 the implementation details from the clients of a module. The same abstraction mechanism should be applicable to higher-order modules as well.

- It should support **efficient elaboration and implementation**. A higher-order module system will not be practical if it cannot be type-checked and compiled efficiently. Compilation of module programs should also be compatible with the standard type-based compilation techniques [15, 13, 31, 33].

1.3 Our contributions

This paper presents a higher-order module calculus that satisfies all of the above properties. Previous approaches have used either strong sums [22, 11, 23] or translucent sums [10, 16, 17] to express module types. We argue that the essence of higher-order modules would be better captured by using a combination of universal and existential quantifiers (i.e., ∀ and ∃). Our key innovation is to view each ML signature not directly as a module type but as a **module type template**; this allows us to assign different interpretations to signatures occurred at different places of a program. For example, consider the following ML program:

```ml
signature SIG = sig type t val x : t end
functor F(A : SIG) = struct val x = A.x end
structure B :> SIG = struct type t = int val x = 3 end.
```

Here, signature SIG is used both as a specification for functor argument A and for opaque module B. All strong-sum-based approaches [22, 11, 23] view SIG as a strong sum type ∃t.t; this does not capture abstract types in B well, so strong-sum-based approaches must use global type stamps to account for abstract types. Translucent-sum-based approaches [10, 16, 17], on the other hand, always view SIG as a weak sum type ∀t.t, this captures opaque module B well, but functor argument A is now limited to have type ∃t.t. This makes it hard to propagate type sharing through functor applications, especially when signature SIG contains higher-order functor components.

We still want opaque module B to have a type similar to ∃t.t, but we want to express functor F as a “polymorphic” type ∀t.t → t. In other words, we think a functor is not merely a module-to-module function, but rather a module-level “polymorphic” function. This view is consistent with recent work on phase-distinction and typed cross-module compilation [12, 33]. In fact, both Harper et al [12] and Shao [33] have shown that using universal quantifications to model functors would propagate the same amount of type sharing as in the strong-sum-based approaches.

To achieve this new interpretation, we view each signature declaration as defining a parameterized type template of form As:K.M where s is a higher-order constructor variable, K is a kind, and M is a module type; we borrow from Mark Jones [14] and name such templates as **parameterized signatures**.

When we use the signature to type an opaque module, we assign the module with type ∃s:K.M. When we use the signature to constrain a functor argument X, we consider that the functor not only binds a module X but also binds a higher-order constructor s of kind K; we can assign the functor with type ∀s:K.M → M’ where M’ is the module type of the functor body.

In the rest of this paper, we first use a series of examples to informally explain our main ideas. We then present a higher-order Kernel Module Calculus (KMC). Instead of following the usual ML syntax, the KMC calculus only uses the basic typing constructs such as universal quantification (∀), existential quantification (∃), dependent product (Π), and transparent module record (i.e., a record containing an ordered list of type, module, and value components; but unlike translucent sums, all type components are fully defined). We use KMC to illustrate the underlying type-theoretical semantics of our higher-order modules. From the typing rules for KMC, we show that all KMC module expressions have unique type signatures. The KMC calculus can also be easily integrated with languages other than ML where module expressions do not follow the ML-like signature syntax [14].

Based on the semantics of KMC, we then present an **External Module Calculus (EMC)** designed for the ML-like

---

1 Although our notion of parameterized signatures is not precisely the same as the one used by Jones [14], we do share the same philosophy of using parameterized type constructors to model inter-module type dependencies. In fact, our work demonstrates that Jones’s original proposal can indeed be extended to model the full-scale ML-like module constructs.
languages. EMC can be viewed as a special subset of KMC, but encoded using the ML-like syntax. EMC can also be viewed as an extension of Leroy’s abstract module calculus [16, 19] but with support to fully transparent higher-order modules. The signature calculus of EMC has a nice pay-as-you-go complexity property since simple modules in EMC will still get same signatures as in Leroy’s abstract calculus [16, 10] and only higher-order modules will use more complex type specifications.

Finally, we demonstrate the expressiveness of KMC and EMC by translating a version of the Abstract Module Calculus (AMC) and a version of the Transparent Module Calculus (TMC) into the EMC calculus. We show how KMC and EMC can be compiled into an Fω-like Target Calculus (FTC) [9, 30] extended with existential types and dot notations [27, 4, 3], so our module languages are still compatible with all the standard type-based compilation techniques [15, 13, 31, 33]. The relationship among these five calculus is depicted in Figure 1.

2 Informal Development

2.1 Fully-transparent higher-order modules

We first use a series of examples to show how the MacQueen-Tofte system [23] supports fully transparent higher-order modules. We start with the definitions of a signature S and a functor signature FS:

signature S = sig type t val x : t end
funsig FS = fsig (X: S): S

The MacQueen-Tofte system uses strong sum (Σ) to express the module type. So signature S is expressed as a dependent sum type \( \pi S = \Sigma t. S \) and signature FS is expressed as dependent product type \( \Pi X : S. S \). We define a structure Y with signature S, and two functors G and H both with signature FS:

structure Y = struct type t=int val x=1 end
functor G(X: S) = X
functor H(X: S) = struct type t=int val x=1 end

Here, though signature S does not include the actual type information (i.e., t=int) in structure Y, functor application \( G(Y) \) always re-elaborate the body of G with X bound to Y, so the type identity of X.t (which is int) is faithfully propagated into the result of G(Y). Now suppose we define the following higher-order functor:

functor FA (F: FS) = F(Y)

where FA takes a functor F as argument and applies it to the previously defined structure Y. We can then apply FA to functors G and H.

Under the MacQueen-Tofte semantics, functor application such as \( F(A) \) and \( FA(B) \) will both re-elaborate the body of \( F \) which in turn re-elaborates the bodies of \( A \) and \( B \); it successfully infers that both \( R_1.x \) and \( R_2.x \) have type int, so the equality test \( (R_1.x = R_2.x) \) will type-check.

MacQueen and Tofte [23] call functors such as \( FA \) as fully transparent modules since they propagate the type sharing in the actual argument (e.g., G and H) into the result (e.g., R1 and R2). Unfortunately, their scheme does not support fully syntactic signatures. If we want to turn module B into a separate compilation unit, we have no way to completely specify its import interface. More specifically, we cannot write down the signature for functor \( FA \) so that all sharing information in the argument is propagated into the result. The closest we can do is to assign \( FA \) with signature:

funsig Z1 = ffsig (F: FS): sig type t=int val x : t end

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

functor J(X: S) = struct type t=real val x=3.0 end
structure R3 = FA(J)

The actual implementation of the MacQueen-Tofte system [33] memoizes a skeleton of each functor body to support re-elaboration, but this is clearly too complex to export to the surface programmers.

2.2 Opaque signatures and abstract types

The abstract approach by Harper and Lillibridge [10] and Leroy [10, 16] simply does not type-check the above equality test \( R_1.x = R_2.x \). In fact, the abstract approach treats the signature S as an existential type \( \exists t.t \) and the signature FS as a dependent product \( \Pi z : S. S \). The higher-order functor \( FA \) is assigned with the following signature:

funsig Z2 = ffsig (F: FS) : S

so applying \( FA \) to G and H always yield a new package \( \exists t.t \) and \( R_1.t \) and \( R_2.t \) are two distinct abstract types.

The abstract approach uses the module subtyping relation to propagate type information from the functor argument to the result module. But the subtyping rules are not powerful enough to support fully transparent higher-order functors. Nevertheless, the abstract approach does have fully syntactic signatures; and having abstract functor parameter itself is sometimes still a useful thing:

functor FA (F: FS) = F(Y)

there are occasions when we indeed want to limit F to be those that always generate new types at each functor application.
2.3 Parameterized signatures and module constructors

We would like to extend the abstract approach to support fully-transparent higher-order modules. Our first observation is to view each signature as a module-type template. More specifically, we view each signature definition of $S$ as defining a parameterized signature of form $\lambda u:K. M$ where we extract all flexible type components in $S$ into a module constructor variable $u$, and we use $K$ to denote its kind, and $M$ for the transparent-sum-based module type\(^3\) with all of its flexible components referring to the corresponding entries in $u$. For example, the definition:

```
signature $S = \text{sig type } t \text{ val } x : t \text{ end}
```

is considered as defining a parameterized signature of form:

$$S = \lambda u:K. \text{(sig type } t = #t(u) \text{ val } x : t \text{ end)}$$

where kind $K_s$ is equal to $\{t:0\}$. We use $#t(C)$ to denote\(^3\) the selection of the $t$ component from a constructor record $C$.

Given a module constructor $C$ of kind $K$, we can apply a parameterized signature $T = \lambda u:K. M$ to $C$—denoted as $T[C]$—to get a new module type of form $[C[u]]M$, i.e., replacing all occurrences of $u$ in $M$ with $C$. For example, applying $S$ to constructor $\{t=\text{int}\}$ yields a module type of form:

```
sig type $t = \text{int} \text{ val } x : t \text{ end}
```

With the parameterized view, different uses of signatures inside a module program can be assigned with different meanings. For example, the following SML'97 code:

```
structure A : $\Rightarrow S = \text{struct type } t=\text{int} \text{ val } x=1 \text{ end}
```

creates an opaque module with type $\exists u:K_s. (S[u])$ or written in the expanded format as:

```
$\exists u:K_s. (\text{sig type } t = #t(u) \text{ val } x : t \text{ end})
```

On the other hand, the signature used to model functor argument has a different meaning; the following ML code:

```
functor G($X : S = \text{struct type } t=X.t \text{ val } x=X.x \text{ end}
```

creates a functor with type:

```
$\forall u:K_s. \Pi X.(S[u]). (\text{sig type } t = X.t \text{ val } x : t \text{ end})$
```

In other words, we consider each functor as a "polymorphic function" that not only binds an argument module $X$ but also binds a module constructor $u$.

This idea can be extended to the higher-order case. Because we want to extend upon the abstract approach, we introduce a new keyword named vis as a modifier to the result signature of each functor signature. Informally speaking, marking a result signature of an $\text{fsig}$ with vis means that we want this functor’s sharing information to be propagated through all functor application. We can also add another modifier $\text{pr}$ to only make part of the return result transparent (see Section 4). So in the following,

```
funsig ES = $\text{fsig (X : S) : S}$
funsig FS = $\text{fsig (X : S) : vis(S)}$
```

signature ES denotes a functor that always returns fresh new types, while FS denotes a fully transparent functor.

We can view the definition of ES as defining the following parameterized signature:

$$E = \lambda f:K_E. \forall u:K_s. \Pi X.(S[u]).(\exists \text{w}. S[w])$$

where kind $K_E$ is just $K_s \rightarrow \{\}$. Notice ES does not propagate any sharing information (f) into its result signature; instead, each functor application always returns an existential package. For example, the abstract functor:

```
functor EA ($F : ES) = F(Y)
```

is assigned with the following interface:

```
$\forall f:K_E. \Pi X.(E[f]).((\exists \text{w}. S[w])$
```

Similarly, the definition of FS can be viewed as defining the following parameterized signature:

$$F = \lambda f:K_F. \forall u:K_s. \Pi X.(S[u]).(S[f[u]])$$

where kind $K_F$ is simply $K_s \rightarrow K_s$. Notice here, inspired by recent work on phase splitting [12, 33], we consider each functor signature as being parameterized by a higher-order constructor $f$ which captures the sharing information in the actual functor. The fully transparent functor:

```
functor FA ($F : FS) = F(Y)
```

can now be assigned with an accurate interface:

```
$\forall f:K_F. \Pi X.(E[f]).(S[f[(\text{t=Y}.\text{t})]])$
```

This in turn can be re-written in an extended ML-like signature calculus:

```
funsig Z3 = $\text{fsig (F : FS) : sig type } t = \#t(F[\{\text{t=Y}.\text{t}\}])$
```

```
val x : t
```

end
```

Here, we assume that every formal functor parameter such as $F$ also binds a module constructor such as $\overline{F}$. With proper syntactic sugar, the signature can even be written as:

```
funsig Z3 = $\text{fsig (F : FS) : sig type } t = \#t(F(Y))$
```

```
val x : t
```

end
```

as long as we know that any module identifiers (e.g., $F$ and $Y$) referred inside the constructor context $\#t(\cdot)$ are really referring to their module constructor counterparts.
Module expression and declaration:

| path  p  ::=  x | p,lₚ | πₘ(p) |
| mexp m  ::=  p \{d₁, \ldots, dₙ\} | \lambda x:M.m | m(p) |
|          | | \Lambda s:K.m | m[C] | \{s:K=C,m,M\} |
|          | | (m:M) | let d in m |
| mdec d  ::=  \lambda t ⋅ cones | \lambda v t = \tau | \lambda v v = e |

Module type and constructor:

| mtyp M  ::=  \{D₁, \ldots, Dₙ\} | \Pi x:M.M' |
|          | | \forall s:K.M | \exists s:K.M |
| mtd f D  ::=  \lambda t ⋅ cones | \lambda v t = \tau | \lambda v v = \tau |
| mcon C  ::=  s | \pi (p) | \{F₁, \ldots, Fₙ\} | \#l₁(C) |
|          | | \lambda s:K.C | C₁[C₂] |
| mcf d F  ::=  \lambda t ⋅ cones | \lambda v t = \tau | \lambda v v = \tau |
| mknd K  ::=  \{Q₁, \ldots, Qₙ\} | K₁ \rightarrow K₂ |
| mkfd Q  ::=  \lambda t : \Omega | \lambda t : \Omega |

Core language:

| ctyp τ  ::=  t | p,lₚ | \#l₁(C) | \ldots |
| cexp e  ::=  v | p,lₚ | \ldots |

Elaboration context:

| ctx \Gamma  ::=  e | \Gamma; t = \tau | \Gamma; v : \tau | \Gamma; x : M | \Gamma; s : K |

Figure 2: Syntax of the kernel module calculus KMC

2.4 Relationship with applicative functors

Some may wonder that our syntactic signature looks particularly similar to Leroy's applicative-functor approach [17] where he can also assign functor FA with a syntactic signature:

\[ \text{fun sig } FA = \]
\[ \text{fsig (F; FS): sig type t=F(\tau).t val x: t end} \]

This similarity, however, stays only at the surface; the underlying interpretations of the two are completely different. Applicative functors only allow application of primitive paths in the signature while we allow arbitrary module constructor expressions. An applicative functor of signature ES will generate the equivalent abstract type if it is applied to the same module, while our scheme would still generate a new type at each functor application (regardless of what the argument structure is like). It is possible to simulate the effect of applicative functors in our scheme though, by doing an opaque signature matching of a functor against a transparent functor signature, e.g.,

\[ \text{functor NF ::= FS = FA} \]

Here, functor NF would have type

\[ N ::= \exists f:Kp.\forall u:Kp.\Pi x:(S[u]).(S[f[u]]) \]

Because what we are hiding here is the constructor f, application of f to equivalent constructors will yield equivalent types (e.g., \#(NF,f[{t=int}])),

| ctx formation \( \vdash \Gamma \) valid |
| ctyp formation \( \Gamma \vdash \tau \) |
| cexp formation \( \Gamma \vdash e : \tau \) |
| mcon formation \( \Gamma \vdash C : K \) |
| mcf d formation \( \Gamma \vdash F : Q \) |
| mknd formation \( \Gamma \vdash D : M \) |
| mkfd formation \( \Gamma \vdash D \) |
| mexp formation \( \Gamma \vdash m : M \) |
| mdec formation \( \Gamma \vdash d : D \) |
| ctyp equivalence \( \Gamma \vdash \tau \equiv \tau' \) |
| mcon equivalence \( \Gamma \vdash C \equiv C' : K \) |
| mcf d equivalence \( \Gamma \vdash F \equiv F' : Q \) |
| mknd subsumption \( \Gamma \vdash D \leq D' \) |
| mkfd subsumption \( \Gamma \vdash D \leq D' \) |

Figure 3: Static semantics for KMC: a summary

3 The Kernel Module Calculus KMC

This section presents a higher-order Kernel Module Calculus (KMC). Instead of following the ML-like syntax, we use simple and well-known typing constructs such as universal quantification (\( \forall \)), existential quantification (\( \exists \)), dependent product (\( \Pi \)), and transparent record (\( \{ \cdot \} \)) to model higher-order modules. The syntax of KMC is given in Figure 2. The static semantics for KMC is derived from the typing rules for each type construct mentioned above (i.e., \( \forall, \exists, \Pi, \) and \( \{ \cdot \} \)); it is summarized in Figure 3; the complete typing rules are given in Figure 4 and Appendix A.

As in any module calculus, KMC supports a form of simple modules that consist of an ordered list of type, module, and value declarations. Following Harper and Lillibridge [10], we assume that each declaration in KMC simultaneously defines an internal name (e.g., \( e, \tau, v \)) and an external label (e.g., \( l₁, l₂, lₙ \)). Given a module record \( S = \{d₁, \ldots, dₙ\} \), declarations defined later can refer to those defined earlier using the internal names. However, to access \( S \)'s components from outside, we must use the access paths such as \( p.l₁, p.l₂ \), and \( p.lₙ \) where \( p \) is another path and \( l₁, l₂, \) and \( lₙ \) are external labels. Simple module is typed using the transparent record type, e.g., \( M = \{D₁, \ldots, Dₙ\} \).

Transparent record (also named as transparent sum) is similar to transuse sum [10] except that all type components in a transparent record are fully defined.

The type structure of KMC resembles a typical predicative \( \Pi \)-like calculus [32]. Module kind \( (\text{mknd}) K \) characterizes module constructor (\( \text{mcon} \) C); module type (\( \text{mtyp} \) M) models module expressions (\( \text{mexp} \) m). An elaboration context \( \Gamma \) for KMC contains bindings for core variables \( (\cdot) \), core type variables \( (\cdot) \), module variables \( (\cdot) \), and module type variables \( (\cdot) \). We use \( D \) to denote the binding introduced by the module field \( D \) (i.e., simply stripping away the label part in \( D \)).

KMC supports universal and existential quantifications on module types, but type variables \( (\cdot) \) are only allowed to quantify over module constructors. Module constructors in KMC are similar to higher-order type constructors in the
mexp formation: \( \Gamma \vdash m : M \)

\[
\begin{array}{c}
\Gamma \vdash \text{valid} \quad x : M \in \Gamma \\
\Gamma \vdash x : M
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash p : \{D_1, \ldots, D_k, \ldots, D_n\} \\
\rho = \{t \mapsto p \cdot \lambda t, \lambda \cdot x \in \text{Dom}(X)\}
\end{array}
\]

where \( X = D_1, \ldots, D_{k-1} \) and \( D_k = \lambda t \cdot x : M \)

\[
\begin{array}{c}
\Gamma \vdash p : \exists s : K.M \\
\rho = \{s \mapsto \pi_s(p)\}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \pi_m(p) : \rho(M)
\end{array}
\]

mknd and mkfd subsumption:

\[
\sigma : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}
\]

\[
\begin{array}{c}
\Gamma \vdash Q_{\sigma(i)} \leq Q_i \\
i = 1, \ldots, m
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \{Q_1, \ldots, Q_n\} \leq \{Q'_1, \ldots, Q'_m\}
\end{array}
\]

mtyp and mtfd subsumption:

\[
\begin{array}{c}
\Gamma \vdash M \leq M' \quad \text{and} \quad \Gamma \vdash D \leq D' \quad \text{(rules for reflexivity and transitivity are omitted)}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash D_1 \leq D_1' \\
\Gamma \vdash \{D_2, \ldots, D_n\} \leq \{D'_2, \ldots, D'_n\}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \{D_1, \ldots, D_n\} \leq \{D'_1, \ldots, D'_n\}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash M'_1 \leq M'_2 \\
\Gamma \vdash M_1 \leq M_2 \leq M'_2
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash M \leq M' \\
\Gamma \vdash \forall s : K.M \leq \forall s : K'.M'
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \tau \equiv \tau' \\
\Gamma \vdash l_\rho v : \tau \leq l_\rho v : \tau'
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash \exists s : K.M \leq \exists s : K'.M'
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash l_\rho v : \tau \leq l_\rho v : \tau'
\end{array}
\]

Figure 4: Selected typing rules for KMC

standard \( F_w \)-like calculus. In this paper, we assume that
kind \( \Omega \) always modifies core-language types, and record kind
\( \{Q_1, \ldots, Q_n\} \) and function kind \( K_1 \to K_2 \) modify module
constructors. A record constructor consists of a sequence of
core-language types (marked by label \( l_r \)) and module
constructors (marked by label \( l_m \)). Given a record constructor
\( C \), the selection form \( \# l_r(C) \) is a module constructor equiva-
lent to the \( l_r \) field of \( C \) while \#\( l_m(C) \) is a core-language
type equivalent to the \( l_m \) field of \( C \).

To simplify the presentation, we have intentionally left
most of the core language undefined. We only include core
variables (\( v \)) and value paths (\( p.l_u \)) as the representative
core terms, and type variables (\( t \)), type paths (\( p.l_u \)), and
constructor selections (\( \# l_m(C) \)) as the representative core
types. These simplifications does not affect our main
results because all the formalism in this paper can be easily
extended to handle more expressive core languages [18].

Opaque modules in KMC are modeled using existential
types [27] and dot notations [4,3]. Given a module path \( p \) of
type \( \exists s : K.M \), we use \( \pi_s(p) \) to denote \( p \)'s constructor compo-
nent (which should have kind \( K \)), and \( \pi_m(p) \) to denote the
module component (which should have type \( \pi_s(p)/s[M] \)). To
construct an opaque module, we use the module expression
of form \( \{ s : K = C, m : M \} \) where constructor \( C \) must
be of kind \( K \), module \( m \) must have type \( [C/s]M \), and the
resulting module has type \( \exists s : K.M \).

KMC supports two forms of parameterized modules: one
abstracted over module values (of type \( M \)), and another
over module constructors (of kind \( K \)). A module function
\( \lambda x : M.m \) has the dependent product type \( \Pi x : M.M' \).
Dependent product is necessary because we use dot notations
to access opaque modules so the return type of a function
might refer to the actual argument. Dot notations [3] also
require that functions in KMC be applied to module access
paths only, as in \( m(p) \). This restriction is not a problem
because we can always use the \( \lambda e \) expression to introduce
local declarations.

Polymorphic modules in KMC are parameterized over
module constructors. A module expression \( \lambda e : K.m \) has the
quantified type \( \forall s : K.M \). It can be applied to constructor
Another interesting aspect of KMC is that it supports a form of module subtyping during function application. Subtyping is also used when we constrain a module expression \( m \) with a module type \( M \). The subtyping rules for arbitrary module types are defined in Figure 4. We also introduce sub-kind relationship between record-kinds; universal quantification is contra-variant over the kind, while existential quantification is co-variant over the kind; dependent products are contra-variant over its argument types. This subtyping relationship is easily decidable because our kind calculus contains simple constant kinds only; it does not contain any arbitrary module constructors.

Finally, KMC supports a limited form of the \( \text{let} \) expression. Given a term \( \text{let} \ d \ in \ m \), the type \( M \) for \( m \) must not refer to any values defined in \( d \) (this is enforced via the type formation rule \( \Gamma \vdash M \) in the semantics in Figure 4).

**Lemma 3.1** The static semantics given in Figure 3 satisfies the following properties:

- if \( \Gamma \vdash \tau \ then \ \Gamma \ valid; \)
- if \( \Gamma \vdash e : \tau \ then \ \Gamma \ valid \ and \ \Gamma \vdash \tau; \)
- if \( \Gamma \vdash C : K \ then \ \Gamma \ valid; \)
- if \( \Gamma \vdash F : Q \ then \ \Gamma \ valid; \)
- if \( \Gamma \vdash M \ then \ \Gamma \ valid; \)
- if \( \Gamma \vdash D \ then \ \Gamma \ valid; \)
- if \( \Gamma \vdash m : M \ then \ \Gamma \ valid \ and \ \Gamma \vdash M; \)
- if \( \Gamma \vdash d : D \ then \ \Gamma \ valid \ and \ \Gamma \vdash D; \)
- if \( \Gamma \vdash \tau \equiv \tau' \ then \ \Gamma \vdash \tau \ and \ \Gamma \vdash \tau'; \)
- if \( \Gamma \vdash C \equiv C' : K \ then \ \Gamma \vdash C : K \ and \ \Gamma \vdash C' : K; \)
- if \( \Gamma \vdash F \equiv F' : Q \ then \ \Gamma \vdash F : Q \ and \ \Gamma \vdash F' : Q; \)
- if \( \Gamma \vdash M \leq M' \ then \ \Gamma \vdash M \ and \ \Gamma \vdash M'; \)
- if \( \Gamma \vdash D \leq D' \ then \ \Gamma \vdash D \ and \ \Gamma \vdash D'. \)

**Proof:** By structural induction on the derivation tree. \( \square \)

We define \( \Gamma \vdash M \equiv M' \) if both \( \Gamma \vdash M \leq M' \) and \( \Gamma \vdash M' \leq M \) are true; and \( \Gamma \vdash D \equiv D' \) if both \( \Gamma \vdash D \leq D' \) and \( \Gamma \vdash D' \leq D \) are true. We can show that all KMC program can be assigned a unique module type.

**Theorem 3.2 (unique typing)** Under the static semantics given in Figure 3, each KMC program is assigned a unique typing:

- if \( \Gamma \vdash m : M \) and \( \Gamma \vdash m : M' \) then \( \Gamma \vdash M \equiv M' \)
- if \( \Gamma \vdash d : D \) and \( \Gamma \vdash d : D' \) then \( \Gamma \vdash D \equiv D' \)

**Proof:** By structural induction on the derivation tree. \( \square \)

### 4 The External Module Calculus EMC

The KMC calculus is a very reasonable module language, however, it does not follow the usual ML convention where a signature can contain both flexible type specifications (i.e., \( \text{type} \ \tau \)) and type definitions (i.e., \( \text{type} \ \tau=\text{int} \)). KMC uses the module constructor parameter to introduce flexible type components, and only type definitions are allowed inside the simple module record.

This section presents a higher-order External Module Calculus (EMC) that follows the ML-style module syntax. EMC can be viewed as a special subset of KMC but encoded using ML-like signatures, structures, and functors. In the meantime, EMC is also an extension of Leroy's abstract module calculus [16, 19] but with support to fully transparent higher-order modules. The syntax of EMC is given in Figure 5. The static semantics for EMC is a straightforward adaption from that of KMC. It is summarized in Figure 6; the complete typing rules and utility functions are given in Figures 7 to 9 and in Appendix B.

EMC is a typical ML-style module language containing constructs such as structure definition and module access path, functor definition and application, and opaque structure matching. The signature calculus in EMC contains two new features that are not present in the usual ML-like signatures: one is the keywords \text{vis} \ and \text{pr} \ which are used to annotate the result of each functor signature to indicate whether a functor is abstract or transparent and if transparent how much sharing should be propagated during functor application; another is the constructor calculus \( C \) and \( K \) (both are almost same as those in KMC) plus the new core type \#\( \text{int} \)(C) that selects a type field from constructor \( C \).

A good way to understand an EMC construct is to relate it with its KMC counterpart and then use the KMC semantics to explain the EMC typing rule. Appendix C contains
a detailed translation algorithm that maps well-typed EMC programs into well-typed KMC programs. In the rest of this section, we’ll only discuss this relationship informally; instead, we concentrate on explaining the intuition behind each EMC typing rule.

We treat each signature \( S \) as if it is a parameterized type template of form \( \lambda \pi \cdot K.M \). We use \( K \) to denote the flexible kind of \( S \), i.e., the kind of the part containing all the flexible type components inside \( S \). We use \( M \) to denote the result of instantiating this flexible part of \( S \) with the constructor variable \( \pi \).

Both flexible kind \( K \) and signature instantiation \( M \) can be calculated using simple and straightforward procedures. In Figure 7, we show how to calculate the flexible kind \( \text{kind}(S) \) from a signature \( S \). Here, \( \text{kind}(S) = K \) means that the flexible constructor part of signature \( S \) is of kind \( K \), and \( \text{kind}(H) = Q^* \) means that the flexible part in specification \( H \) is of kind field \( Q^* \) (which denotes either \( Q \) or empty field \( \varepsilon \)). Notice in addition to flexible type specifications \( (lv \cdot t) \), functor specifications are also considered as parts of the flexible components. A fully transparent functor (of signature \( \text{fsig}(x:S) : \text{vis}(S') \)) is treated as a higher-order constructor of kind \( K \rightarrow K' \) where \( K \) and \( K' \) are the kinds for \( S \) and \( S' \). A partially transparent functor (of signature \( \text{fsig}(x:S) : \text{pr}(S',K') \))—with \( K' \) indicating which component in \( S' \) is considered as transparent—is treated as a constructor of kind \( K \rightarrow K' \) where \( K \) is the kind for \( S \), and the kind for \( S' \) must be a sub-kind of \( K' \). An abstract functor (of signature \( \text{fsig}(x:S) : S' \)) is treated as a dummy constructor that returns an empty record kind.

The signature instantiation procedure is defined in Figure 8. Given a signature \( S \) and a constructor \( C \) of kind \( \text{kind}(S) \), signature instantiation \( \text{app}(S,C) \) returns the result of substituting the flexible part of \( S \) with \( C \). We use two auxiliary procedures to implement \( \text{app}(S,C) \). In Figure 8, \( \text{app}(S,C,K) = S' \) means that instantiating \( S \) by constructor \( C \) of kind \( K \) yields signature \( S' \), and \( \text{app}(H,C,K) = H' \) means that instantiating specification \( H \) by constructor \( C \) of kind \( K \) yields specification \( H' \). The additional kind parameter in \( \text{app} \) is used to identify the flexible components. Signature instantiation always produces a signature whose type components are fully defined and whose functor components have abstract functor signatures (i.e., no \( \text{vis} \) or \( \text{pr} \) modifiers on their return signatures).

Each EMC signature plays two roles: in the module program, it is used to specify functor parameters and to create new opaque modules; in the static semantics, it denotes an actual module type: when we say that a module expression \( m \) has signature \( S \), we essentially mean that \( m \) has type equivalent to \( \exists \pi \cdot \text{kind}(S) \cdot \text{app}(S,\pi) \) (or if expressed as KMC types: \( \exists \pi \cdot \text{kind}(S) \cdot \text{[app(S,\pi)]}_v \), where \( [\cdot]_v \) is a function that maps EMC signatures into KMC module types, see Appendix C for details).

An EMC functor \( \text{fct}(x:S)m \) is roughly equivalent to a KMC polymorphic module function of form:
$$\Gamma \vdash H_i \downarrow K \Rightarrow F^i \quad i = 1, \ldots, n$$

$$\vdash S \downarrow K \Rightarrow C$$

$$\Gamma \vdash \text{valid } x : S \in \Gamma$$

$$\Gamma \vdash x : \text{app}(S, \pi, x)$$

$$\Gamma \vdash p : S'$$

$$\Gamma \vdash S' \downarrow \text{kn}(S) \Rightarrow C$$

$$\Gamma \vdash \text{valid } \Gamma \vdash \text{str end} : \text{sig end}$$

$$\Gamma \vdash \Pi \vdash d : H$$

$$\Gamma \vdash \text{let } d \in m : S$$

$$\Lambda \exists \Pi : \text{kn}(S), \lambda \alpha : [\text{app}(S, \Pi)]_\alpha, m'$$

where $\lfloor \cdot \rfloor$ maps EMC signatures to KMC module types and $m'$ is the corresponding KMC term for $m$. The typing rule for functor definition reflects this interpretation precisely (see Figure 9). Here we intentionally use $\Pi$ to denote the module type variable for each functor parameter $x$: in practice, we might even just use $x$ as long as we use different name space for constructor and module variables.

Functor application is typed following the same intuition. Given two module access paths $p_1$ and $p_2$, suppose $p_1$ has signature $\text{f} \text{sig}(x : S) : S'$ and $p_2$ has signature $S''$, then an EMC functor application $p_1[p_2]$ roughly corresponds to a KMC polymorphic instantiation followed by a function application $(p_1[C][p_2])$. Here construct $C$ is the flexible type components extracted from the actual argument signature ($S''$) of $p_2$ through a signature-narrowing function. Given a context $\Gamma$, the signature narrowing function $\Gamma \vdash S \downarrow K \Rightarrow C$ extracts the type components from a module type $S$ and produces a constructor $C$ of kind $K$. An invariant is that signature narrowing is only applied to instantiated signatures (i.e., signatures that returned by $\text{app}(\cdot, \cdot)$). The detailed definition of signature narrowing is given in Figure 9. Notice that we don’t cover the cases for flexible type specifications and transparent functor components since they won’t occur inside instantiated signatures.

An EMC structure resembles a KMC module record but it is really a KMC opaque module with existential types. In fact, to type a simple variable $x$, we always do a round of instantiation or “unpacking” (see Figure 9), so each use of $x$ has type $\text{app}(S, \pi, x)$ where $S$ is the signature bound to $x$ in the context and $\pi, x$ is the dot notation for the type part of $x$.

The rest of the rules in Figure 9 are straightforward. To type a opaque signature matching ($p : > S$), we first constrain $p$ with a restricted signature and then close it with the signature $S$ (e.g., the “pack” operation for creating opaque modules in KMC). To type a let expression, we also verify that the result signature does not contain any references to locally defined module variables (i.e., $S$ is well formed in context $\Gamma$ but not $\Gamma; H$).

Finally, EMC can also be extended to support other forms of module expressions in SML’97. For example, in SML’97, the let expression (at the module level) allows its body type to refer to the new type stamps generated in the let declarations. Also, SML’97 supports transparent signature matching such as:

```plaintext
structure A : sig type t val f : t end =
  struct abstype s = ...
  type t = s -> s
  fun f(x : s) = x
  end
```

Here, type $t$ is equivalent to $s -> s$, but the new type $s$ is not exported. Both of these features involve exporting values and types that make use of hidden abstract types. It is still in doubt whether they are really useful in practice. Nevertheless, we can extend the EMC signature calculus with the following specification for hidden types:

```
spec H ::= ... | l:t t is hidden
```

We can then write down the interface $S_A$ for $A$ as:

```
sig type s is hidden
  type t = s -> s
  val f : t
  end
```

which is in turn equivalent to the following module type:

```
\exists \Pi : \text{kn}(S), \lambda \alpha : [\text{app}(S, \Pi)]_\alpha, m' \text{val f : t end}
```

where kind $K_A$ is just $\{s : \Omega\}$. In other words, if we write each signature $S$ in parameterized form $\lambda \Pi : K_M$, then a hidden type specification will be included in the kind $K$ but not in the body type $M$. Notice in EMC, all components in $K$ are always present in $M$, but this is not necessarily true for KMC.
function, denoted as $\mathcal{[\cdot]}_\alpha$. A context in AMC may contain plain type-variable bindings such as $t$, however, such context is only used in elaborating AMC signatures. An AMC context is called type-free if it does not contain such type-variable binding. We can show that the translation $\mathcal{[\cdot]}_\alpha$ maps all well typed AMC programs into well-typed EMC programs. The main difference between AMC and EMC is the typing rules for signature subsumption and functor application.

**Theorem 5.1** Given a type-free AMC context $\Gamma$, we have:

- If $\Gamma$ is valid under AMC, then $\mathcal{[\Gamma]}_\alpha$ is valid under EMC;
- If $\Gamma \vdash t : \tau$ then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[t]}_\alpha]_\alpha$;
- If $\Gamma \vdash e : \tau$ then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[e]}_\alpha]_\alpha$;
- If $\Gamma \vdash S$ then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[S]}_\alpha]_\alpha$;
- If $\Gamma \vdash m : S$ then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[m]}_\alpha]_\alpha$;
- If $\Gamma \vdash d : H$ then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[d]}_\alpha]_\alpha$;
- If $\Gamma \vdash t \equiv \tau' \text{ then } \mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[t]}_\alpha]_\alpha \equiv [\mathcal{[\tau]}_\alpha]_\alpha$.

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

**Lemma 5.2** Suppose $S$ is an AMC signature resulted from signature strengthening, $S'$ is a regular AMC signature, then given a type-free AMC context $\Gamma$, if $\Gamma \vdash S \leq S'$ is valid under AMC and $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[S]}_\alpha]_\alpha \vdash \text{prd}(S') \Rightarrow C$ is valid under EMC, then $\mathcal{[\Gamma]}_\alpha \vdash [\mathcal{[S]}_\alpha]_\alpha \leq [\mathcal{[\text{prd}(S')]_\alpha}_\alpha \Rightarrow C$ is valid under EMC.

**Lemma 5.3** Given an EMC context $\Gamma$, suppose $m$ is an AMC module expression, $S$ and $S'$ are AMC signatures. If $\Gamma ; x : S \vdash m : S'$ is valid under EMC and $\overline{\mathcal{P}} \notin \text{Dom}(\Gamma)$, then $\Gamma ; \mathcal{P} ; \text{nd}(S); x : \mathcal{[\mathcal{[S]}_\alpha]_\alpha} \vdash m : S'$ is valid under EMC.

5.2 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 12, the static semantics for TMC is
summarized in Figure 13; the complete typing rules are given in Appendix E. Following the usual strong-sum-based module systems [23, 25, 33], we distinguish module signatures \((S)\) from module types \((M \text{ and } L)\): module signatures are used in the source programs for type specifications while module types are used in the static semantics for type checking.

A module signature can either contain a single value specification \((V(\mu))\), a single type specification \((\tau)\), or a pair of two 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 \((\pi_1(\mu))\) are allowed in the signature specifications.\(^4\) An L-shaped module type is like a module signature using \(\tau\) in its value specification \(V(\tau)\), and 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 \(E\theta(\tau)\) but no flexible type specification of form \(\tau\). The use of arbitrary module expressions \((m')\) inside core types \((\tau)\) helps achieve the fully transparent propagation of the type sharing information in EMC.

A module expression in TMC can either be an access path \((p)\), a single-value component \((i.e., \mu)\), a single-type-component module \((\alpha(e))\), a strong sum of two other module components \((\Sigma x: m_1, m_2)\), 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 application \((e.g., \alpha(e))\) can be \(\alpha\)-normalized into the restricted form using \(\lambda\) expressions. We also do not support type abbreviations in signatures. We insist that \(\alpha\) be a subtype of \(L\) if they have same number of components (see the subtyping rules \(\Gamma \leq L\) in Appendix E). These restrictions do not affect the main result since we can easily extend TMC and the TMC-to-EMC translation to handle the additional features.

The translation from TMC to EMC is summarized in Figure 14; the detailed translation rules are given in Appendix F. Here, \(\Gamma\) maps TMC contexts, core types (in signatures), signatures, core expressions, access paths, and module expressions into their EMC counterparts; \(\Gamma\) maps

\[\begin{align*}
ctxt \text{ formation} & : \Gamma \vdash \text{valid} \\
mtype \text{ formation} & : \Gamma \vdash M \text{ and } \Gamma \vdash L \\
cf \text{ formation} & : \Gamma \vdash \tau \\
cme \text{ formation} & : \Gamma \vdash m' : M \\
cce \text{ formation} & : \Gamma \vdash e' : \tau \\
cexp \text{ formation} & : \Gamma \vdash e : \tau \\
mexp \text{ formation} & : \Gamma \vdash m : M \\
sig \text{ formation} & : \Gamma \vdash S \\
cf \text{ formation} & : \Gamma \vdash \mu \\
cf \text{ equivalence} & : \Gamma \vdash \tau_1 \equiv \tau_2 \\
mexp \text{ equivalence} & : \Gamma \vdash M_1 \equiv M_2 \text{ or } \Gamma \vdash L_1 \equiv L_2 \\
mtpy \text{ subsumption} & : \Gamma \vdash M \leq L \\
mtpy \text{ strengthening} & : L/m' \Rightarrow M
\end{align*}\]

Figure 13: Static semantics for TMC: a summary

\[\begin{align*}
\text{ctxt-to-ctxt translation} & : [\Gamma]_n \Rightarrow [\Gamma]_n \\
\text{ctyp-to-ctyp translation} & : [\mu]_n \Rightarrow [\tau]_n \\
\text{sig-to-sig translation} & : [S]_n \Rightarrow [S]_n \\
\text{cecp-to-cecp translation} & : [e]_n \Rightarrow [e]_n \\
\text{path-to-path translation} & : [\Gamma]_n \Rightarrow [\Gamma]_n \\
\text{mexp-to-mexp translation} & : [m]_n \Rightarrow [m]_n \\
\text{ctyp-to-ctyp translation} & : \Gamma \vdash \tau \Rightarrow \tau' \\
\text{mtpy-to-sig translation} & : \Gamma \vdash M \Rightarrow S \\
\text{mtpy-to-sig translation} & : \Gamma \vdash L \Rightarrow S \\
\text{mtpy-to-kind translation} & : [M]_n \Rightarrow K \\
\text{mtpy-to-kind translation} & : [L]_n \Rightarrow K \\
\text{ctme-to-ctme translation} & : \Gamma \vdash m' \Rightarrow M \Rightarrow C
\end{align*}\]

Figure 14: Translation from TMC to EMC: a summary

TMC module types into EMC kinds. The translation from TMC types to EMC \(T\) is based on the type formation rules, so the judgements \(\Gamma \vdash \tau \Rightarrow \tau'\) maps the TMC core type \(\tau\) into an EMC core type \(\tau'\); the judgements \(\Gamma \vdash M \Rightarrow S\) and \(\Gamma \vdash L \Rightarrow S\) map the TMC module types \(M\) or \(L\) into a EMC signature \(S\). In addition, we use judgements \(\Gamma \vdash M \Rightarrow C\) and \(\Gamma \vdash m' \Rightarrow C\) to map TMC module types and expressions (embedded inside core types) into EMC constructors. We can prove a similar type preservation theorem for the TMC-to-EMC translation:

**Theorem 5.4** Given a TMC context \(\Gamma\), then we have:

- if \(\Gamma \vdash \text{valid}\) then \([\Gamma]_n \Rightarrow \text{valid}\);
- if \(\Gamma \vdash \mu\) then \([\Gamma]_n \Rightarrow [\mu]_n\);
- if \(\Gamma \vdash S\) then \([\Gamma]_n \Rightarrow [S]_n\);
- if \(\Gamma \vdash e : \tau\) and \(\Gamma \vdash \tau \Rightarrow \tau'\) then \([\Gamma]_n \Rightarrow [e]_n \Rightarrow \tau'\);
- if \(\Gamma \vdash p : M \text{ and } \Gamma \vdash M \Rightarrow S\) then \([\Gamma]_n \Rightarrow [p]_n \Rightarrow S\);
- if \(\Gamma \vdash m : M \text{ and } \Gamma \vdash M \Rightarrow S\) then \([\Gamma]_n \Rightarrow [m]_n \Rightarrow S\);
- if \(\Gamma \vdash \tau \Rightarrow \tau'\) then \([\Gamma]_n \Rightarrow \tau'\);
- if \(\Gamma \vdash M \Rightarrow S\) then \([\Gamma]_n \Rightarrow S\);
- if \(\Gamma \vdash L \Rightarrow S\) then \([\Gamma]_n \Rightarrow S\);
- if \(\Gamma \vdash M \Rightarrow S_1 \text{ and } \Gamma \vdash L \Rightarrow S_2\) then \([\Gamma]_n \Rightarrow [\Gamma]_n \Rightarrow [S_1 \uplus [L]_n \Rightarrow C \Rightarrow C\) then \([\Gamma]_n \Rightarrow S_1 \leq \text{app}(S_2, C)\);
- if \(\Gamma \vdash M_1 \equiv M_2 \text{ and } \Gamma \vdash M_1 \Rightarrow C_1 \text{ and } \Gamma \vdash M_2 \Rightarrow C_2\) then \([\Gamma]_n \Rightarrow [M_1]_n \equiv [M_2]_n\) and \([\Gamma]_n \Rightarrow C_1 \equiv C_2 : [M_1]_n\);
- if \(\Gamma \vdash m' \Rightarrow M \Rightarrow C_1 \text{ and } \Gamma \vdash M \Rightarrow C_2\) then \([\Gamma]_n \Rightarrow C_1 \equiv C_2 : [M]_n\).

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

**Lemma 5.5** Given a TMC context \(\Gamma\), suppose \(\Gamma \vdash m'_1 : M_1\), let \(p = \{x \mapsto m'_1\}\), 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 \(\Gamma, x : M_1 \vdash \rho(L_2) \equiv L_2\);
kind \( K \) ::= \( \Omega \setminus K_1 \to K_2 \mid \{ t_1: K_1, \ldots, t_n: K_n \} \)

con \( C \) ::= \ldots \mid s \mid \lambda s: K.C \mid C_1.C_2 \mid \{ t_1: C_1, \ldots, t_n: C_n \} \mid \#(C) \mid \pi_n(p)

type \( M \) ::= \forall x: (C) \mid \{ t_1: M_1, \ldots, t_n: M_n \} \mid \Pi x: M.M' \mid \forall s: K.M \mid \exists s: K.M

path \( p \) ::= x \mid p.l \mid \pi_m(p)

exp \( m \) ::= \ldots \mid \langle p \mid \{ t_1 = \ldots = t_n = \ldots \} \mid \lambda x: M.m \mid \Lambda x: M.m \mid \lambda p \mid \langle s: K = C, m: M \rangle \mid \langle (m: M) \rangle

let \( x = m_1 \) in \( m_2 \)

clet \( \Gamma \) ::= \epsilon \mid \Gamma; x: M \mid \Gamma; s: K

Figure 15: Syntax of the kernel module calculus FTC

context formation \( \vdash \Gamma \) valid
constructor formation \( \Gamma \vdash C::K \)
type formation \( \Gamma \vdash M \)
exp formation \( \Gamma \vdash m: M \)

constructor equivalence \( \Gamma \vdash C \equiv C'::K \)
type equivalence \( \Gamma \vdash M \equiv M' \)
kind subsumption \( \vdash K \leq K' \)
type subsumption \( \Gamma \vdash M \leq M' \)

Figure 16: Static semantics for FTC: a summary

5.3 Comparison with the stamp-based semantics

Most compilers for the strong-sum-based calculus [23, 25, 33] use stamps to support type generativity and abstract types. There are higher-order modules that are supported by the stamp-based semantics, but not by our type-theoretic semantics, consider the following two functors \( G \) and \( H \):

functor \( G(X: S) = X \)
functor \( H(X: S) = \text{struct} \ a \text{ subtype} t = A \)
with \( \text{val} x = A \)
end

Under the stamp-based semantics, we can apply the following higher-order functor \( FA \):

fun sig \( F = \text{fsig}(X: S) = \text{vis}(S) \)
functor \( FA \) (\( F: FS \)) = \ldots

to both \( G \) and \( H \) while still propagate all the sharings. Under our semantics, we have no way to specify a single functor signature that can take both \( G \) and \( H \) while still propagate sharing, because in our semantics, we assign different types to functors that generates new abstract types (\( H \)) and those that does not (\( G \)). We believe this lack of expressiveness is not a problem in practice.

6 Implementation

One important advantage of KMC and EMC is that they can be easily compiled into the \( F_n \)-like polymorphic \( \Lambda \)-calculus. We can translate EMC into KMC and then drop all the type components in the KMC transparent records (after we inline all type definitions of course). The result is an \( F_n \)-based Target Calculus (FTC) as defined in Figure 15. FTC is essentially the standard predicative variant of \( F_n \) extended with existential types (\( \exists \)), dot notations (i.e., \( \pi_n(p) \) and \( \pi_m(p) \)), and dependent products (\( \Pi \)). Figure 16 and Appendix G gives the typing rules for FTC. The translation from KMC to FTC is omitted since it is quite trivial.

The fact that all the module languages discussed in this paper can be compiled into an \( F_n \)-based calculus is important because immediately all important type-based compilation techniques [15, 31, 13, 26, 36] become applicable to these module languages as well. In a previous paper [33], we presented a type-preserving translation from the MacQueen-Tofte higher-order modules [22] into an \( F_n \)-based calculus, however, that algorithm turns all abstract types into concrete ones; this makes it hard to reason about type-directed operations on values with abstract types. The translations presented in this paper, on the other hand, map all opaque modules into abstract types in FTC, so at least, two different types in the source calculus would not be considered as equivalent in the intermediate FTC calculus.

7 Related Work

Module systems have been an active research area in the past decade. The ML module system was first proposed by MacQueen [21] and later incorporated into Standard ML [24]. Harper and Mitchell [11] 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 [12], 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 [10, 16, 17, 20, 35, 23, 14] and mostly resolved in SML’97 [25]. The main remaining issue is on how to design a higher-order module calculus that satisfies all of the properties mentioned in our introduction.

Supporting higher-order functors with fully syntactic signatures turns out to be a very hard problem. The official definition of Standard ML [25] only supports first-order functors. MacQueen and Toft [23, 35] recently proposed to add fully transparent higher-order modules to ML but their scheme does not provide fully syntactic signatures. Independently, Harper and Lillibridge [10] and Leroy [16] proposed to use translucent signatures to model abstract types and type sharing; their scheme supports fully syntactic signatures, but fails to propagate as much sharing as in the MacQueen-Tofte system. Leroy [17] also proposed to use applicative semantics to model fully transparency, but his signature calculus only handles limited forms of functor arguments; this limitation was lifted in Courrant’s recent proposal [6], 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).

Parameterized signatures was first proposed by Jones [14] to capture the essence of various different modular structures. Our notion of parameterized signatures differs from
his in that we allow type components inside the module record. In fact, our module record is a transparent sum and it can contain an ordered list of type, value, and module declarations. Jones's parameterized signatures only allow value components. Both of our approaches do share the same philosophy of using parameterized type constructors to model inter-module type dependencies. In fact, our work demonstrates that Jones's original proposal can indeed be extended to model the full-scale ML-like module constructs.

8 Conclusions

A long-standing open problem on ML-style module systems is to find a calculus that supports both fully transparent higher-order modules and fully syntactic signatures. Mark Lillibridge made the following assessment on the difficulty of this problem in his Ph.D. thesis [20, page 310]:

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

This paper shows that with parameterized signatures, we can solve this problem using a rather simple module calculus. In fact, our kernel module calculus KMC only uses the standard typing constructs such as universal and existential quantifiers, dependent products, and transparent records, all of which have clean and well-understood type-theoretic semantics.

Our work on the external module calculus EMC shows that we can add fully transparent higher-order modules to ML-like languages while still supporting Module-2 style true separate compilation. The signature calculus in EMC is a conservative and pay-as-you-go extension of the one in SML'97 [25]: a program module that does not use higher-order functors can still have the same signature as that under the SML'97 module language. The only complexity of EMC is that signatures used to specify functor parameters are interpreted differently from those used to specify opaque modules.

The kernel module calculus KMC is another interesting candidate for future programming languages that intend to support ML-style parameterized modules. KMC has a very clean type-theoretic semantics. It supports both module functions and polymorphic modules. It can also be easily extended to support conditional modules or first-class modules. The only problem is that the same module typable under EMC might require a more verbose interface in KMC.

Higher-order modules and fully syntactic signatures make it possible to 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 (e.g., sources.cm file) used in Blume's Compilation Manager (CM) [2]. We also plan to extend our module calculus to support dynamic linking and mutually recursive compilation units [7].

Acknowledgement

I would like to thank David MacQueen and Valery Trifonov for discussions and comments on an early version of this paper.

References

A Static Semantics for KMC

This appendix gives the rest of the typing rules for the kernel module calculus KMC.

A.1 utility: field name stripping

$$\frac{}{\ell.v.x : M \equiv x : M} \quad (1)$$

$$\frac{}{l \vdash t : \tau} \quad (2)$$

$$\frac{}{l \vdash v : \tau \equiv v : \tau} \quad (3)$$

A.2 ctxf formation: $\vdash \Gamma \text{ valid}$

$$\frac{\vdash e \text{ valid}}{\vdash \Gamma} \quad (4)$$

$$\frac{\vdash \tau \quad t \notin \text{dom}(\Gamma)}{\vdash \Gamma ; t = \tau} \quad (5)$$

$$\frac{\vdash \tau}{\vdash \Gamma ; v : \tau} \quad \text{valid} \quad (6)$$

$$\frac{\vdash M \quad x \notin \text{dom}(\Gamma)}{\vdash \Gamma ; x : M} \quad \text{valid} \quad (7)$$

$$\frac{\vdash \Gamma \text{ valid} \quad s \notin \text{dom}(\Gamma)}{\vdash \Gamma ; s : K} \quad \text{valid} \quad (8)$$

A.3 typ formation: $\vdash \tau$

$$\frac{\vdash \Gamma \text{ valid}}{\vdash \Gamma ; t = t \in \Gamma} \quad \text{valid} \quad (9)$$

$$\frac{\vdash p : M \quad M = \{ \ldots , l_4 t = \tau , \ldots \}}{\vdash p.4} \quad (10)$$

$$\frac{\vdash C : K \quad K = \{ \ldots , l_4 : k , \ldots \}}{\vdash \#_{l_4}(C)} \quad (11)$$

A.4 expf formation: $\vdash e : \tau$

$$\frac{\vdash \Gamma \text{ valid}}{\vdash \Gamma ; v : \tau} \quad \text{valid} \quad (12)$$

$$\frac{\vdash p : \{ D_1 , \ldots , D_h , \ldots , D_n \}}{}$$

$$\begin{equation}
\rho = \{ \ell \mapsto p . l_4 , x \mapsto p . l_5 \mid l_4 t , t_4 v \equiv \text{Dom}(X) \}
\end{equation}$$

where $X = D_h , \ldots , D_{h-1}$ and $D_k = l_4 v : \tau$

$$\frac{\vdash \rho(\tau)}{\vdash p . l_5 : \rho(\tau)} \quad (13)$$
A.5 mcon formation: $\Gamma \vdash C : K$

$$\vdash \Gamma \text{ valid} \quad s : K \in \Gamma \quad \Gamma \vdash s : K$$ (14)

$$\Gamma \vdash \pi_e(p) : K$$ (15)

$$\Gamma \vdash F_i : Q, \quad i = 1, \ldots, n$$ (16)

$$\Gamma \vdash \{F_1, \ldots, F_n\} : \{Q_1, \ldots, Q_n\}$$ (17)

$$\Gamma \vdash C : K \quad \Gamma \vdash \#e_i(C) : K$$ (18)

$$\Gamma \vdash C_1 : K \rightarrow K' \quad \Gamma \vdash C_2 : K$$ (19)

A.6 mfield formation: $\Gamma \vdash F : Q$

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash \tau$$ (20)

$$\Gamma \vdash \lambda s : K.C : K'$$ (21)

$$\Gamma \vdash C : K \quad \Gamma \vdash \lambda s : K.C : K'$$ (22)

$$\Gamma \vdash \#e_i(C) \equiv \tau$$ (33)

A.7 mtyp formation: $\Gamma \vdash M$

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash \{\}$$ (23)

$$\Gamma \vdash \{D_1, \ldots, D_n\}$$ (24)

$$\Gamma \vdash \{D_1, \ldots, D_n\}$$ (25)

$$\Gamma \vdash \{D_1, \ldots, D_n\}$$ (26)

$$\Gamma \vdash \Pi x : M.M'M'$$ (34)

$$\Gamma \vdash \Pi x : M.M'M'$$ (35)

A.8 mtdf formation: $\Gamma \vdash D$

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash \tau$$ (27)

$$\Gamma \vdash \tau$$ (28)

$$\Gamma \vdash \tau$$ (29)

A.9 mexp formation: $\Gamma \vdash m : M$

See Figure 4 in Section 3.

A.10 mdec formation: $\Gamma \vdash d : D$

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash m : M$$ (30)

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash \tau$$ (31)

$$\vdash \Gamma \text{ valid} \quad \Gamma \vdash \tau$$ (32)

A.11 ctyp equivalence: $\Gamma \vdash \tau \equiv \tau'$

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

$$\vdash \Gamma \text{ valid} \quad \tau \equiv \tau$$ (33)

$$\vdash \Gamma \text{ valid} \quad \tau \equiv \tau$$ (34)

A.12 mcon equivalence: $\Gamma \vdash C \equiv C' : K$

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

$$\vdash \Gamma \text{ valid} \quad \tau \equiv \tau$$ (35)

A.13 mcfld equivalence: $\Gamma \vdash F \equiv F' : Q$

Rules for congruence, reflexivity, symmetry, and transitivity are omitted.
A.14 mknd and mtyp subsumption

See Figure 4 in Section 3.

B Static Semantics for EMC

This appendix gives the rest of the typing rules for the external module calculus EMC.

B.1 utility: field name stripping

\[ \text{let } x : S = x : S \]
\[ \text{let } t = t = t \]
\[ \text{let } v = v : t \]

B.2 ctxt formation: \( \vdash \Gamma \) valid

\[ \vdash e \text{ valid} \]
\[ \vdash \tau \text{ valid} \]
\[ \vdash x : S \text{ valid} \]
\[ \vdash \tau \text{ valid} \]

B.3 ctyp formation: \( \Gamma \vdash \tau \)

\[ \vdash \tau \text{ valid} \]
\[ \vdash p : S \text{ valid} \]
\[ \vdash C : K \text{ valid} \]

B.4 cexp formation: \( \Gamma \vdash e : \tau \)

\[ \vdash \tau \text{ valid} \]
\[ \vdash p : \rho (\tau) \]

B.5 mcon formation: \( \Gamma \vdash C : K \)

\[ \vdash \Gamma \text{ valid} \]
\[ \vdash \varphi : K \in \Gamma \]
\[ \vdash \varphi : K \]

B.6 mcfd formation: \( \Gamma \vdash F : Q \)

\[ \vdash \tau \text{ valid} \]
\[ \vdash \varphi : \text{knnd}(S) \text{ valid} \]

B.7 sig formation: \( \Gamma \vdash S \)

\[ \vdash \text{valid} \]
\[ \vdash \text{sig end} \]

B.8 rsig formation: \( \Gamma \vdash R \)

\[ \vdash \text{valid} \]
\[ \vdash S \]

B.9 pr formation: \( \Gamma \vdash \text{pr}(S, K) \)

\[ \vdash \text{valid} \]
\[ \vdash \text{pr}(S, K) \]

\[ \vdash \text{valid} \]
\[ \vdash \text{pr}(S, K) \]

\[ \vdash \text{valid} \]
\[ \vdash \text{pr}(S, K) \]
B.9 spec formation: $\Gamma \vdash H$

The specification $l : t$ is not directly allowed. Signatures with such specifications are admitted through Rule 64.

$$
\begin{align*}
\Gamma & \vdash S \\
\Gamma & \vdash l : t \vdash \tau
\end{align*}
$$

B.10 mexp formation: $\Gamma \vdash m : S$

See Figure 9 in Section 4.

B.11 mdec formation: $\Gamma \vdash d : H$

$$
\begin{align*}
\Gamma & \vdash m : S \\
\Gamma & \vdash l : t \vdash \tau : m : l : t \vdash \tau
\end{align*}
$$

B.12 ctyp equivalence: $\Gamma \vdash \tau \equiv \tau'$

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

$$
\begin{align*}
\Gamma & \vdash \tau \equiv \tau \\
\Gamma & \vdash t \equiv \tau
\end{align*}
$$

B.13 mcon equivalence: $\Gamma \vdash C \equiv C' : K$

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

$$
\begin{align*}
\Gamma & \vdash C \equiv C' : K \\
\Gamma & \vdash \tau : C \vdash \tau : C'
\end{align*}
$$

B.14 mcfd equivalence: $\Gamma \vdash F \equiv F' : Q$

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

B.15 mknd subsumption: $\vdash K \leq K'$

Rules for reflexivity and transitivity are omitted.

$$
\begin{align*}
\sigma : [1, \ldots, m] & \mapsto [1, \ldots, n] \\
\vdash Q_{\sigma(i)} & \leq Q_i' \\
\vdash \{Q_1, \ldots, Q_n\} & \leq \{Q_1', \ldots, Q_m'\}
\end{align*}
$$

B.16 mkfd subsumption: $\vdash Q \leq Q'$

Rules for reflexivity and transitivity are omitted.

$$
\begin{align*}
\vdash l : \Omega & \leq l : \Omega \\
\vdash K & \leq K'
\end{align*}
$$

B.17 sig subsumption: $\Gamma \vdash S \leq S'$

Rules for reflexivity and transitivity are omitted.

$$
\begin{align*}
X & = H_2, \ldots, H_n \\
\vdash H_i & \leq H_i' \\
\Gamma ; H_i \vdash \text{sig } X & \leq \text{sig } X' \text{' end}
\end{align*}
$$

$$
\begin{align*}
X & = H_1, \ldots, H_n \\
\Gamma ; H_i \vdash \text{sig } X & \leq \text{sig } X' \text{' end}
\end{align*}
$$

$$
\begin{align*}
\vdash \text{sig } S & \leq \text{sig } S' \\
\Gamma ; \text{sig } S & \vdash \text{app}(S, \tau) \leq \text{app}(S, \tau')
\end{align*}
$$

17
B.18 **spec subsumption**: \( \Gamma \vdash H \leq H' \)

Rules for reflexivity and transitivity are omitted.

\[
\begin{align*}
\Gamma \vdash S \leq S' \\
\Gamma \vdash L; x : S \leq L; x : S' \\
\Gamma \vdash \tau \equiv \tau' \\
\Gamma \vdash L; \tau = \tau \leq L; \tau = \tau' \\
\Gamma \vdash L; \tau = \tau \equiv L; \tau = \tau'
\end{align*}
\]

(89) (90) (91)

C **Translation from EMC to KMC**

This appendix gives a type-preserving translation algorithm from EMC to KMC. To make the presentation easier, we first modify the EMC syntax to distinguish different uses of module access paths:

\[
\begin{align*}
\text{path} & \quad p \ ::= \ x \mid p.\_ \\
\text{mexp} & \quad m \ ::= \ \langle p \rangle \mid \text{str} \ d_1, \ldots, d_n \end{align*}
\]

Here, we use \( \langle p \rangle \) to denote the places where a module path \( p \) is used as a stand-alone module expression. We then separate the formation rules for module paths from regular module expressions:

\[
\begin{align*}
\text{path formation} & \quad \Gamma \vdash p : S \\
\text{mexp formation} & \quad \Gamma \vdash m : S
\end{align*}
\]

As a result of this reorganization, we add the following rule to the mexp formation:

\[
\Gamma \vdash p : S \\
\Gamma \vdash \langle p \rangle : S
\]

The EMC-to-KMC translation is denoted as \([\ ]_\text{w} \). A special auxiliary function that translates EMC signature \( S \) into KMC existential module type \( M \) is denoted as \([\ ]_\text{b} \).

C.1 **ctx-to-ctx translation**: \( [\Gamma]_\text{w} \mapsto \Gamma \)

\[
\begin{align*}
[e]_\text{w} &= \varepsilon \\
[\Gamma; t = \tau]_\text{w} &= [\Gamma]_\text{w}; t = [\tau]_\text{w} \\
[\Gamma; v : \tau]_\text{w} &= [\Gamma]_\text{w}; v : [\tau]_\text{w} \\
[\Gamma; x : S]_\text{w} &= [\Gamma]_\text{w}; x : [S]_\text{b} \\
[\Gamma; \varpi : K]_\text{w} &= [\Gamma]_\text{w}; \varpi : K
\end{align*}
\]

C.2 **ctyp-to-ctyp translation**: \( [\tau]_\text{w} \mapsto \tau \)

\[
\begin{align*}
[t]_\text{w} &= t \\
[p.\_]_\text{w} &= [p]_\text{w}.\_ \\
[#\ell(C)]_\text{w} &= #\ell([C]_\text{w})
\end{align*}
\]

C.3 **cexp-to-cexp translation**: \( [e]_\text{w} \mapsto e \)

\[
\begin{align*}
[v]_\text{w} &= v \\
[p.\_]_\text{w} &= [p]_\text{w}.\_
\end{align*}
\]

C.4 **mcfd-to-mcfd translation**: \( [Q]_\text{w} \mapsto Q \)

\[
\begin{align*}
[l = \tau]_\text{w} &= l = [\tau]_\text{w} \\
[l = C]_\text{w} &= l = [C]_\text{w}
\end{align*}
\]

C.5 **mcon-to-mcon translation**: \( [C]_\text{w} \mapsto C \)

\[
\begin{align*}
[[F_1, \ldots, F_n]]_\text{w} &= \{ [F_1]_\text{w}, \ldots, [F_n]_\text{w} \} \\
[#\ell(C)]_\text{w} &= #\ell([C]_\text{w}) \\
[\lambda \varpi : K. C]_\text{w} &= \lambda \varpi : K.[C]_\text{w} \\
[C_1[C_2]]_\text{w} &= [C_1]_\text{w}[[C_2]_\text{w}] \\
[C]_\text{w} &= C
\end{align*}
\]

C.6 **path-to-path translation**: \( [p]_\text{w} \mapsto p \)

\[
\begin{align*}
[x]_\text{w} &= \pi_x(x) \\
[p.\_]_\text{w} &= [p]_\text{w}.\_
\end{align*}
\]

C.7 **sig-to-mtyp translation**: \( [S]_\text{b} \mapsto M \)

The translation from EMC signature to KMC module type is defined as follows:

\[
[S]_\text{b} = \exists \varpi : \text{kind}(S). [\text{app}(S, \varpi)]_\text{w}
\]

The internal translation \([\ ]_\text{w} \) is applied to a restricted set of KMC signatures:

\[
\begin{align*}
isig S' & ::= \text{sig} H_1', \ldots, H_n' \end{align*}
\]

\[
\text{fexpr}(x : S') : S
\]

In other words, the cases for fully or partially transparent functor signatures and flexible type specification \( l = \varpi \) cannot occur. The signature instantiation procedure \( \text{app}(S, \varpi) \) always produces a signature in such restricted form.

C.8 **isig-to-mtyp translation**: \( [S']_\text{w} \mapsto M \)

\[
\begin{align*}
isig H_1', \ldots, H_n' \text{ end} \_w &= \{ [H_1']_\text{w}, \ldots, [H_n']_\text{w} \} \\
[\text{fexpr}(x : S') : S']_\text{w} &= \Lambda \varpi : \text{kind}(S). \lambda x : [\text{app}(S, \varpi)]_\text{w}. [S']_\text{b}
\end{align*}
\]

C.9 **ispec-to-mtfd translation**: \( [H']_\text{w} \mapsto D \)

\[
\begin{align*}
[l = \varpi = \tau]_\text{w} &= l = \varpi = [\tau]_\text{w} \\
[l = \varpi = \tau]_\text{w} &= l = \varpi = [\tau]_\text{w}
\end{align*}
\]

18
C.10  mexp-to-mexp translation: \( \Gamma \vdash m: S \leadsto m' \)

The translation from EMC mexp to KMC mexp is conducted along the EMC typing rules. Given a context \( \Gamma \), an EMC module expression \( m \) is translated to a KMC expression \( m' \) if and only if \( \Gamma \vdash m: S \leadsto m' \).

\[
\begin{align*}
\Gamma \vdash p : S & \quad \Gamma \vdash S \downarrow \text{kn} \text{d}(S) \Rightarrow C \quad M = [\text{app}(S, \overline{x})]_u \\
\Gamma & \vdash (p) : S \leadsto \langle \overline{x} : \text{kn} \text{d}(S), [C]_u, [p]_u : M \rangle \\
\end{align*}
\]

\[\Gamma; \overline{x} : \text{kn} \text{d}(S), x : \text{app}(S, \overline{x}) \vdash m : S' \leadsto m' \]

\[
m' = \lambda \overline{x} : \text{kn} \text{d}(S), \lambda x : \text{app}(S, \overline{x}) : m' \\
\Gamma \vdash \text{fct}(x:S)m : \text{fsig}(x:S) : S' \leadsto m'
\]

\[\begin{align*}
\Gamma & \vdash p_1 : \text{fsig}(x:S) : S' \\
\Gamma & \vdash p_2 : S'' \\
\Gamma & \vdash S'' \downarrow \text{kn} \text{d}(S) \Rightarrow C \\
\Gamma & \vdash S'' \leq \text{app}(S, C) \\
\rho & = \{ \overline{x} \mapsto C, x \mapsto p \} \\
\Gamma & \vdash p_1(p_2) : \rho(S'') \leadsto ([p_1]_u[C]_u)[p_2]_u
\end{align*}\]

\[\begin{align*}
\Gamma & \vdash \text{str \ valid} \\
\Gamma & \vdash \text{end \ valid} : \text{sig \ end \leadsto} \langle \overline{x} : \{ \{ \}, \{ \}, \{ \} \rangle \\
\Gamma; \overline{H}_1, \ldots, \overline{H}_k \vdash d_k : H_k \leadsto d'_k, d''_k; H'_k & \quad k = 1, \ldots, n \\
S & = \text{sig } \overline{H}_1, \ldots, \overline{H}_n \quad \text{end} \\
K & = \text{kn} \text{d}(S) \\
S' & = \text{sig } \overline{H}'_1, \ldots, \overline{H}'_m \quad \text{end} \\
\Gamma & \vdash S' \downarrow K \Rightarrow C \\
m & = \langle \overline{x} : K = [C]_u, \{d'_1, \ldots, d'_m : \text{app}(S, \overline{x})\}_u \rangle
\end{align*}\]

\[\Gamma \vdash \text{str} \ d_1, \ldots, d_n \leadsto m : S \leadsto \text{let } d_1, \ldots, d_n \text{ in } m\]

\[\begin{align*}
\Gamma & \vdash p : S' \\
\Gamma & \vdash S' \downarrow \text{kn} \text{d}(S) \Rightarrow C \\
\Gamma & \vdash S' \leq \text{app}(S, C) \\
\rho & = \{ [p]_u : [\text{app}(S, C)]_u \} \\
\Gamma & \vdash (p : S') : S \leadsto \langle \overline{x} : \text{kn} \text{d}(S), [C]_u, m : [\text{app}(S, \overline{x})]_u \rangle \\
\end{align*}\]

\[\Gamma \vdash S \quad \Gamma \vdash d : H \leadsto d' ; \quad \Gamma ; \overline{H} \vdash m : S \leadsto m' \]

\[\Gamma \vdash \text{let } d \text{ in } m : S \leadsto \text{let } d' \text{ in } m' \]

\[\begin{align*}
\Gamma & \vdash m : S \leadsto m' \\
\alpha & \text{ is new} \\
H & = L_{vb} \alpha : \text{app} (S, \pi_\alpha (x)) \\
\Gamma & \vdash L_{vb} x = m : L_{vb} x : S \leadsto L_{vb} x = m'; L_{vb} x' = \pi_\alpha (x); H \\
\end{align*}\]

\[\begin{align*}
\Gamma & \vdash \tau \quad \tau' \text{ is new} \\
H & = L_{vb} \tau \equiv \tau = L_{vb} \tau = \tau \equiv L_{vb} \tau \equiv \tau = t; H \\
\end{align*}\]

\[\begin{align*}
\Gamma & \vdash e : \tau \\
\Gamma & \text{ e : } \tau' \text{ is new} \\
H & = L_{vb} e : L_{vb} e : \tau \equiv L_{vb} e = [e]_u, L_{vb} e' = v; H
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad t \in \Gamma \\
\Gamma & \vdash t \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad t \notin \text{dom}(\Gamma) \\
\Gamma & \vdash t \quad \text{valid} \\
\end{align*}\]

\[\Gamma \vdash \tau \quad t \notin \text{dom}(\Gamma) \\
\Gamma \vdash \tau \quad \text{valid} \\
\]

\[\begin{align*}
\Gamma & \vdash \tau \\
\Gamma & \vdash \tau \quad \text{valid} \\
\end{align*}\]

\[\Gamma \vdash S \quad x \notin \text{dom}(\Gamma) \\
\Gamma ; x : S \quad \text{valid} \\
\]

D.3  ctyp formation: \( \Gamma \vdash \tau \)

\[\begin{align*}
\Gamma \vdash \text{valid} \\
\Gamma \vdash t \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad t \in \Gamma \\
\Gamma & \vdash t \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad t \equiv \tau \in \Gamma \\
\Gamma & \vdash \tau \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad \text{let } t_1, \ldots, t_n \quad \text{in } \tau \\
\Gamma & \vdash p : S_1 \equiv \{ \ldots, L_{vb} t_1, \ldots \} \\
\Gamma & \vdash p, L_{vb} t_1 \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid } \quad \text{let } t_1, \ldots, t_n \quad \text{in } \tau \\
\Gamma & \vdash p : S_1 \equiv \{ \ldots, L_{vb} t_1, \ldots \} \\
\Gamma & \vdash p, L_{vb} t_1 \\
\end{align*}\]

\[
\text{Proof: } \text{By structural induction on the derivations.} \quad \square
\]

D  Static Semantics for AMC

This appendix gives the complete typing rules for the abstract module calculus AMC.

D.1  utility: field name stripping

\[
\begin{align*}
\Gamma & \vdash x : S = x : S \\
\end{align*}\]

\[\Gamma \vdash \text{let } t = \tau \quad \text{valid} \]

\[\Gamma ; \tau \vdash \tau \quad \text{valid} \]

\[\begin{align*}
\Gamma & \vdash \tau \quad \text{valid} \\
\Gamma & \vdash \tau \quad \text{valid} \\
\end{align*}\]

\[\begin{align*}
\Gamma ; \tau \vdash \tau \quad \text{valid} \\
\Gamma & \vdash \tau \quad \text{valid} \\
\end{align*}\]

C.11 Properties of the translation

Theorem C.1  Given an EMC context \( \Gamma \), then

\[\begin{align*}
\text{If } \Gamma \vdash \text{valid then } \vdash [\Gamma]_u \quad \text{valid} \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash \tau \quad \text{then } [\Gamma]_u \vdash [\tau]_w \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash e : \tau \quad \text{then } [\Gamma]_u \vdash [e]_w : [\tau]_w \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash C : K \quad \text{then } [\Gamma]_u \vdash [C]_w : [K]_u \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash F : Q \quad \text{then } [\Gamma]_u \vdash [F]_w : [Q]_w \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash H^l \quad \text{then } [\Gamma]_u \vdash [H^l]_w \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash S^l \quad \text{then } [\Gamma]_u \vdash [S^l]_w \\
\end{align*}\]

\[\begin{align*}
\text{If } \Gamma \vdash S \quad \text{then } [\Gamma]_u \vdash [S]_w \\
\end{align*}\]
D.4 cexp formation: \( \Gamma \vdash e : \tau \)

\[
\Gamma \vdash \text{valid} \quad v : \tau \in \Gamma \\
\Gamma \vdash v : \tau
\] (105)

\( \Gamma \vdash p : \text{sig } H_1, \ldots, H_k, \ldots, H_n \text{ end} \rho = \{ t \mapsto p.l_t, x \mapsto p.l_x \mid l_t \triangleright t, l_x \triangleright x \in \text{Dom}(X) \} \)

where \( X = H_1, \ldots, H_{k-1} \) and \( H_k = l_x \triangleright v : \tau \)

\[
\Gamma \vdash p.l_v : \rho(\tau)
\] (106)

D.5 sig formation: \( \Gamma \vdash S \)

\[
\Gamma \vdash \text{valid} \\
\Gamma \vdash \text{sig end}
\] (107)

\[
\Gamma ; H_1, \ldots, H_{k-1} \vdash H_k \quad k = 1, \ldots, n \\
\Gamma \vdash \text{sig } H_1, \ldots, H_n \text{ end}
\] (108)

\[
\Gamma \vdash S \\
\Gamma ; x : S \vdash S' \\
\Gamma \vdash \text{fsig}(x : S) : S'
\] (109)

D.6 spec formation: \( \Gamma \vdash H \)

\[
\Gamma \vdash S \\
\Gamma \vdash l_x \triangleright x : S
\] (110)

\[
\Gamma \vdash \text{valid} \\
\Gamma \vdash l_x \triangleright t
\] (111)

\[
\Gamma \vdash \tau \\
\Gamma \vdash l_x \triangleright t \triangleright \tau
\] (112)

\[
\Gamma \vdash \tau \\
\Gamma \vdash l_x \triangleright v : \tau
\] (113)

D.7 mexp formation: \( \Gamma \vdash m : S \)

\[
\Gamma \vdash \text{valid} \\
\Gamma \vdash x : S \in \Gamma \\
S/x \Rightarrow S' \\
\Gamma \vdash x : S'
\] (114)

\[
\Gamma \vdash p : \text{sig } H_1, \ldots, H_k, \ldots, H_n \text{ end} \\
\rho = \{ t \mapsto p.l_t, x \mapsto p.l_x \mid l_t \triangleright t, l_x \triangleright x \in \text{Dom}(X) \} \\
\text{where } X = H_1, \ldots, H_{k-1} \text{ and } H_k = l_x \triangleright v : S'
\]

\[
\Gamma \vdash p.l_v : \rho(\tau)
\] (115)

\[
\Gamma \vdash \text{valid} \\
\Gamma \vdash \text{str end} : \text{sig end}
\] (116)

\[
\Gamma ; H_1, \ldots, H_{k-1} \vdash d_k : H_k \\
k = 1, \ldots, n \\
\Gamma \vdash \text{str } d_1, \ldots, d_n \text{ end} : \text{sig } H_1, \ldots, H_n \text{ end}
\] (117)

\[
\Gamma \vdash x : S \vdash m : S' \\
\Gamma \vdash \text{fct}(x : S)m : \text{fsig}(x : S) : S'
\] (118)

\[
\Gamma \vdash p_1 : \text{fsig}(x : S) : S' \\
\Gamma \vdash S'' \leq S \\
\rho = \{ x \mapsto p_2 \} \\
\Gamma \vdash p_1(p_2) : \rho(S')
\] (119)

\[
\Gamma \vdash p : S' \\
\Gamma \vdash S' \leq S \\
\Gamma \vdash (p > S) : S
\] (120)

\[
\Gamma \vdash d : H \\
\Gamma \vdash S \\
\Gamma ; \text{H} \vdash m : S \\
\Gamma \vdash \text{let } d \text{ in } m : S
\] (121)

D.8 mdec formation: \( \Gamma \vdash d : H \)

\[
\Gamma \vdash m : S \\
\Gamma \vdash l_x \triangleright x = m : l_x \triangleright x : S
\] (122)

\[
\Gamma \vdash \tau \\
\Gamma \vdash l_x \triangleright t = \tau : l_x \triangleright t = \tau
\] (123)

\[
\Gamma \vdash \tau \\
\Gamma \vdash l_x \triangleright v = \tau : l_x \triangleright v = \tau
\] (124)

D.9 ctyp equivalence: \( \Gamma \vdash \tau \equiv \tau' \)

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

\[
\Gamma \vdash \text{valid} \\
t = \tau \in \Gamma \\
\Gamma \vdash t \equiv \tau
\] (125)

\[
\Gamma \vdash p : \text{sig } H_1, \ldots, H_{k' - 1}, \ldots, H_n \text{ end} \\
\rho = \{ t \mapsto p.l_t, x \mapsto p.l_x \mid l_t \triangleright t, l_x \triangleright x \in \text{Dom}(X) \} \\
\text{where } X = H_1, \ldots, H_{k-1} \text{ and } H_k = l_x \triangleright v : S'
\]

\[
\Gamma \vdash p.l_v : \rho(\tau)
\] (126)

D.10 sig subsumption: \( \Gamma \vdash S \leq S' \)

Rules for reflexivity and transitivity are omitted.

\[
X = H_2, \ldots, H_n \text{ and } X' = H'_2, \ldots, H'_n \\
\Gamma ; H_1 \vdash \text{sig } X \text{ end} \leq \text{sig } X' \text{ end}
\] (127)

\[
X = H_1, \ldots, H_n \text{ and } X' = H'_1, \ldots, H'_m \\
\Gamma ; H_0 \vdash \text{sig } X \text{ end} \leq \text{sig } X' \text{ end}
\] (128)

\[
\Gamma ; S'_1 \leq S_1 \\
\Gamma ; x : S'_1 \vdash S_2 \leq S'_2 \\
\Gamma \vdash \text{fsig}(x : S_1) : S_2 \leq \text{fsig}(x : S'_1) : S'_2
\] (129)
D.11 spec subsumption: $\Gamma \vdash H \leq H'$

Rules for reflexivity and transitivity are omitted.

$\Gamma \vdash S \leq S'$

$\Gamma \vdash t_s = t \leq t_s = t'$

$\Gamma \vdash l_{s \cdot t} \leq l_{s \cdot t}$

$\Gamma \vdash t \leq l_{s \cdot t}$

$\Gamma \vdash t \leq l_{s \cdot t}$

$\Gamma \vdash l_{s \cdot t} = t \leq l_{s \cdot t}$

E.1 ctx formation: $\vdash \Gamma \ valid$

$\vdash e \ valid$ (141)

$\Gamma \vdash M \quad x \notin dom(\Gamma)$

$\vdash \Gamma ; x : M \ valid$ (142)

$\Gamma \vdash L \quad x \notin dom(\Gamma)$

$\vdash \Gamma ; x : L \ valid$ (143)

E.2 typ formation: $\Gamma \vdash \tau$

$\Gamma \vdash m' : EQ(\tau)$

$\Gamma \vdash \pi_i(m')$ (144)

E.3 cte formation: $\Gamma \vdash m' : M$

$\vdash \Gamma valid \quad x : L \in \Gamma \quad L/x \Rightarrow M$

$\Gamma \vdash x : M$ (145)

$\vdash \Gamma valid \quad x : M \in \Gamma$

$\Gamma \vdash x : M$ (146)

$\Gamma \vdash e' : \tau$

$\Gamma \vdash i_u(e') : \psi(\tau)$ (147)

$\Gamma \vdash \tau$

$\Gamma \vdash i_u(\tau) : \psi(\tau)$ (148)

$\Gamma \vdash m' : \Sigma x : M_1.M_2$

$\Gamma \vdash \pi_1(m') : M_1$ (149)

$\Gamma \vdash m' : \Sigma x : M_1.M_2 \quad \rho = \{x \mapsto \pi_1(m')\}$

$\Gamma \vdash \pi_2(m') : \rho(M_2)$ (150)

$\Gamma \vdash m' : \Sigma x : M_1.M_2 \quad \rho = \{x \mapsto \pi_1(m')\}$

$\Gamma \vdash \pi_2(m') : \rho(M_2)$ (151)

$\Gamma \vdash \lambda x : L. m' : \Pi x : L. M$

$\Gamma \vdash \lambda x : L. m' : \Pi x : L. M$ (152)

$\Gamma \vdash m' : \Pi x : L_1. M_2 \quad \Gamma \vdash \pi_2(m') : M_1$

$\Gamma \vdash M_1 \leq L_1 \quad \rho = \{x \mapsto \pi_2(m')\}$

$\Gamma \vdash m_1' : \rho(M_2)$ (153)

$\Gamma \vdash m_1' : M_1 \quad \Gamma \vdash m_2' : M_2$

$\Gamma \vdash m_1' : M_1 \quad \Gamma \vdash m_2' : M_2$

$\Gamma \vdash \lambda x : L. m' : \Pi x : L. M$

$\Gamma \vdash \lambda x : L. m' : \Pi x : L. M$ (154)

E.4 cte formation: $\Gamma \vdash e' : \tau$

$\Gamma \vdash m' : \psi(\tau)$

$\Gamma \vdash \pi_u(m') : \tau$ (155)

E.5 mty formation: $\Gamma \vdash M$ and $\Gamma \vdash L$

Rules for module types of form $M$:

$\Gamma \vdash \tau$

$\Gamma \vdash \psi(\tau)$ (156)
\[ \Gamma \vdash \tau \]
\[ \Gamma \vdash \text{EQ}(\tau) \]  
(157)

\[ \Gamma; x : M_1 \vdash M_2 \]
\[ \Gamma \vdash \Sigma x : M_1, M_2 \]  
(158)

\[ \Gamma; x : L \vdash M \]
\[ \Gamma \vdash \Pi x : L.M \]  
(159)

Rules for module types of form \( L \):

\[ \Gamma \vdash \tau \]
\[ \Gamma \vdash \text{W}(\tau) \]  
(160)

\[ \Gamma \vdash \text{TYP} \]  
(161)

\[ \Gamma; x : L_1 \vdash L_2 \]
\[ \Gamma \vdash \Sigma x : L_1, L_2 \]  
(162)

\[ \Gamma; x : L_1 \vdash L_2 \]
\[ \Gamma \vdash \Pi x : L_1, L_2 \]  
(163)

**E.6 exp form**: \( \Gamma \vdash e : \tau \)

\[ \Gamma \vdash \tau \]
\[ \Gamma \vdash i_*(e) : \text{W}(\tau) \]  
(164)

\[ \Gamma \vdash p : \Sigma \tau \]
\[ \Gamma \vdash \pi_*(p) : \tau \]  
(165)

**E.7 ctp form**: \( \Gamma \vdash \mu \)

\[ \Gamma \vdash p : \text{EQ}(\tau) \]
\[ \Gamma \vdash \pi_*(p) \]  
(166)

**E.8 sig form**: \( \Gamma \vdash S \)

Adding bindings such as "\( x : S \)" to the context \( \Gamma \) is fine because each signature \( S \) is also a module type of form \( L \).

\[ \Gamma \vdash \mu \]
\[ \Gamma \vdash \text{W}(\mu) \]  
(167)

\[ \Gamma \vdash \text{TYP} \]  
(168)

\[ \Gamma; x : S_1 \vdash S_2 \]
\[ \Gamma \vdash \Sigma x : S_1, S_2 \]  
(169)

**E.9 mexp form**: \( \Gamma \vdash m : M \)

\[ \vdash \Gamma \text{ valid } \quad x : L \in \Gamma \quad L / x \Rightarrow M \]
\[ \Gamma \vdash x : M \]  
(170)

\[ \vdash \Gamma \text{ valid } \quad x : M \in \Gamma \]
\[ \Gamma \vdash x : M \]  
(171)

\[ \Gamma \vdash p : \Sigma x : M_1, M_2 \]
\[ \Gamma \vdash \pi_*(p) : M_1 \]  
(172)

\[ \Gamma \vdash p : \Sigma x : M_1, M_2 \quad \rho = \{ x \mapsto \pi_1(p) \} \]
\[ \Gamma \vdash \pi_*(p) : \rho(M_2) \]  
(173)

**E.10 ctp equivalence**: \( \Gamma \vdash \tau_1 \equiv \tau_2 \)

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

\[ \vdash \Gamma \vdash \text{m}^* : \text{EQ}(\tau) \]
\[ \Gamma \vdash \pi_*(\text{m}^*) \equiv \tau \]  
(180)

**E.11 mtyp equivalence**: \( \Gamma \vdash M_1 \equiv M_2 \) and \( \Gamma \vdash L_1 \equiv L_2 \)

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

**E.12 mtyp subsumption**: \( \Gamma \vdash M \leq L \)

\[ \Gamma \vdash \text{EQ}(\tau) \leq \text{TYP} \]  
(181)

\[ \Gamma \vdash \pi_1 \equiv \pi_2 \]
\[ \Gamma \vdash \text{W}(\tau) \leq \text{W}(\tau_2) \]  
(182)

\[ \Gamma \vdash M_1 \leq L_1 \]
\[ \Gamma; x : M_1 \vdash M_2 \leq L_2 \]
\[ \Gamma \vdash \Sigma x : M_1, M_2 \leq \Sigma x : L_1, L_2 \]  
(183)

\[ \Gamma; x : L_1 \vdash M_2 \leq L_2 \]
\[ \Gamma \vdash \Pi x : L_1, M_2 \leq \Pi x : L_1, L_2 \]  
(184)
E.13 **mtyp strengthening**: $L/m' \Rightarrow M$

$$\forall \tau/m' \Rightarrow \forall \tau$$  
(185)

$$\text{TYP}/m' \Rightarrow \text{EQ}(\pi_t(m'))$$  
(186)

$$\frac{L_1/\pi_1(m') = M_1 \quad L_2/\pi_2(m') = M_2}{\Sigma x: L_1, L_2/m' \Rightarrow \Sigma x: M_1, M_2}$$  
(187)

$$\frac{L_2/m'(x) \Rightarrow M_2}{\Pi x: L_1, L_2/m' \Rightarrow \Pi x: M_1, M_2}$$  
(188)

F **Translation from TMC to EMC**

This appendix gives the complete translation algorithm from TMC to EMC; the translation is denoted as $[\_]_n$. An auxiliary function that maps from $M$ to EMC kind ($K$) and from $m'$ to EMC constructor ($\tau$) is denoted as $[\_]_c$.

F.1 **ctxt-to-ctxt translation**: $[\Gamma]_n \rightarrow \Gamma$

$$[\varepsilon]_n = \varepsilon$$

$$\Gamma \vdash L \sim S$$

$$[\Gamma; x: L]_n = [\Gamma]_n; x: \text{kn}(S); x: \text{app}(S, \text{tr})$$

$$\vdash \Gamma \Rightarrow M \sim S$$

$$[\Gamma; x: M]_n = [\Gamma]_n; x: S$$

F.2 **ctyp-to-ctyp translation**: $[\mu]_n \rightarrow \tau$

$$[\pi_t(p)]_n = [p]_n, \text{typ}$$

F.3 **sig-to-sig translation**: $[S]_n \rightarrow S$

$$[\forall \mu]_n = \text{sig op} \circ \text{v: } [\mu]_n, \text{end}$$

$$[\text{TYP}]_n = \text{sig typ} \circ \text{t: } \tau', \text{end}$$

$$[\Sigma x: S_1, S_2]_n = \text{sig fst: } [S_1]_n, \text{snd: } [S_2]_n$$

$$\text{end}$$

$$[\Pi x: S_1, S_2]_n = \text{fsig}(x: [S_1]_n; \text{vis}([S_2]_n))$$

F.4 **cexp-to-cexp translation**: $[e]_n \rightarrow e$

$$[\pi_t(p)]_n = [p]_n, \text{ops}$$

F.5 **path-to-path translation**: $[p]_n \rightarrow p$

$$[x]_n = x$$

$$[\pi_1(p)]_n = [p]_n, \text{fst}$$

$$[\pi_2(p)]_n = [p]_n, \text{snd}$$

F.6 **mexp-to-mexp translation**: $[m]_c \rightarrow m$

$$[p]_n = [p]_n$$

$$[\pi_t(e)]_n = \text{str op} \circ \text{v: } [e]_n, \text{end}$$

$$[\mu(e)]_n = \text{str typ} \circ \text{t: } [\mu]_n, \text{end}$$

$$[\langle x = m_1, m_2 \rangle]_n = \text{str fst: } [m_1]_n; \text{snd: } [m_2]_n$$

$$\text{end}$$

$$[\lambda x: S, m]_n = \text{fct}(x: [S]_n; [m]_n)$$

$$[p_1(p_2)]_n = [p_1]_n([p_2]_n)$$

$$[\text{let } x = m_1 \text{ in } m_2]_n = \text{let } x = [m_1]_n \text{ in } [m_2]_n$$

F.7 **ctyp-to-cotyp translation**: $\Gamma \vdash \tau \leadsto \tau'$

The translation of a core type in TMC is based on its formation rules. Given a well-formed core type $\tau$ in context $\Gamma$, it is translated into EMC type $\tau'$ if and only if the judgement $\Gamma \vdash \tau \leadsto \tau'$ is valid.

$$\Gamma \vdash m': M \sim C$$

$$\Gamma \vdash [\pi_t(m')]_c \leadsto \#\text{typ}(C)$$

F.8 **mtyp-to-sig translation**: $\Gamma \vdash M \sim S$

$$\Gamma \vdash \forall \tau \leadsto \tau'$$

$$\vdash \forall \tau \leadsto \tau'$$

$$\Gamma \vdash \text{EQ}(\tau) \sim \text{sig typ} \circ \text{t: } \tau', \text{end}$$

$$\Gamma \vdash M_1 \sim S_1$$

$$\Gamma; x: M_1 \vdash M_2 \sim S_2$$

$$\vdash \Sigma x: M_1, M_2 \sim \text{sig fst: } S_1$$

$$\text{snd: } x', S_2$$

$$\text{end}$$

$$\Gamma \vdash \Pi x: L_1, L_2 \sim \text{fsig}(x: S_1); S_2$$

F.9 **mtyp-to-sig translation**: $\Gamma \vdash L \sim S$

$$\Gamma \vdash \forall \tau \leadsto \tau'$$

$$\vdash \forall \tau \leadsto \tau'$$

$$\Gamma \vdash \text{TYP} \sim \text{sig typ} \circ \text{t: } \tau', \text{end}$$

$$\Gamma \vdash L_1 \sim S_1$$

$$\Gamma; x: L_1 \vdash L_2 \sim S_2$$

$$\Gamma \vdash \Sigma x: L_1, L_2 \sim \text{sig fst: } S_1$$

$$\text{snd: } x', S_2$$

$$\text{end}$$

$$\Gamma \vdash \Pi x: L_1, L_2 \sim \text{fsig}(x: S_1); S_2$$

F.10 **mtyp-to-kind translation**: $[M]_c \rightarrow K$

$$[\forall \tau]_c = \{\}$$

$$[\text{EQ}(\tau)]_c = \{\text{typ: } \Omega\}$$

$$[\Sigma x: M_1, M_2]_c = \{\text{fst: } [M_1]_c, \text{snd: } [M_2]_c\}$$

$$[\Pi x: L, M]_c = [L]_c \Rightarrow [M]_c$$
F.11 mtyp-to-kind translation: \([L] \rightarrow K\)

\[
\begin{align*}
&[\mathcal{V}(\tau)]_c = \{\} \\
&[\mathcal{TYP}]_c = \{\mathsf{typ} : \Omega\} \\
&S_\Sigma:M_1.M_2 \rightarrow \{\mathsf{fst}=C_1, \mathsf{snd}=C_2\} \\
&[\Pi:\Sigma:M_1.M_2]_c = [M_1]_c \rightarrow [M_2]_c
\end{align*}
\]

F.12 mtyp-to-mcon translation: \(\Gamma \vdash M \leadsto C\)

\(\Gamma\)-shaped TMC module types can be translated into EMC module constructors. The translation is based on the type formation rules for \(M\).

\[
\begin{prooftree}
\Gamma \vdash \mathcal{V}(\tau) \leadsto \{\}
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash \tau \leadsto \tau'
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash M \leadsto C_1
\quad \Gamma, x : M_1 \vdash M_2 \leadsto C_2
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash \Sigma x : M_1.M_2 \leadsto \{\mathsf{fst}=C_1, \mathsf{snd}=C_2\}
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash \Pi x : M_1.M_2 \leadsto \lambda \xi : [L_1]_c.C_2
\end{prooftree}
\]

F.13 ctmte-to-mcon translation: \(\Gamma \vdash m' : M \leadsto C\)

All TMC module expressions \((m')\) embedded inside the core types can be translated into EMC module constructors. The translation is based on the formation rules for \(m'\).

\[
\begin{prooftree}
\Gamma \vdash \mathcal{V}(\tau) \leadsto \{\}
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash \tau \leadsto \tau'
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash e' : \tau
\quad \Gamma \vdash \iota_e(e') : \mathcal{V}(\tau) \leadsto \{\}
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash \tau \leadsto \tau'
\quad \Gamma \vdash \mathcal{E}(\tau) \leadsto \{\mathsf{typ}=\tau'\}
\end{prooftree}
\]

\[
\begin{prooftree}
\Gamma \vdash m' : \Sigma x : M_1.M_2 \leadsto C
\quad \Gamma \vdash \pi_1(m') : M_1 \leadsto \#\mathsf{fst}(C)
\end{prooftree}
\]

G Static Semantics for FTC

This appendix gives the complete typing rules for the transparent module calculus FTC.

G.1 context formation: \(\vdash \Gamma \text{ valid}\)

\[
\begin{align*}
\vdash & \varepsilon \text{ valid} \\
\Gamma \vdash & \varepsilon \not\in \text{dom}(\Gamma) \\
\Gamma \vdash & \Gamma ; x : M \text{ valid}
\end{align*}
\]

G.2 constructor formation: \(\Gamma \vdash C : K\)

\[
\begin{align*}
\vdash & \Gamma \text{ valid} \quad s : K \in \Gamma \\
\Gamma \vdash & s : K
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & C_i : K_i \quad i = 1, \ldots, n \\
\Gamma \vdash & \{I_i : C_1, \ldots, I_n : C_n\} : \{I_1 : K_1, \ldots, I_n : K_n\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & C : K' \quad K' = \{I : K, \ldots\} \\
\Gamma \vdash & \#I(C) : K \\
\Gamma \vdash & \lambda s : K.C : K \rightarrow K'
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & C_1 : K \rightarrow K' \\
\Gamma \vdash & C_2 : K \\
\Gamma \vdash & C_1[C_2] : K'
\end{align*}
\]

G.3 type formation: \(\Gamma \vdash M\)

\[
\begin{align*}
\Gamma \vdash & C : \Omega \\
\Gamma \vdash & \mathcal{V}(C)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & M_i \quad i = 1, \ldots, n \\
\Gamma \vdash & \{I_i : M_1, \ldots, I_n : M_n\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & x : M \leadsto M' \\
\Gamma \vdash & \Pi x : M.M' \\
\Gamma \vdash & \lambda x : L.M' : \Pi x : L.M \leadsto \lambda \xi : [L]_c.C
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & m'_1 : M_1 \leadsto C_1 \\
\Gamma \vdash & m'_2 : M_2 \leadsto C_2 \\
\Gamma \vdash & \langle x = m'_1, m'_2 \rangle : \Sigma x : M_1.M_2 \leadsto \{\mathsf{fst}=C_1, \mathsf{snd}=C_2\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & \axis : L.M' : \Pi x : L.M \leadsto \lambda \xi : [L]_c.C
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & m'_1 : \Pi x : L_1.M_2 \leadsto C_1 \\
\Gamma \vdash & m'_2 : M_1 \leadsto C_2 \\
\Gamma \vdash & \langle x = m'_1, m'_2 \rangle : \Sigma x : M_1.M_2 \leadsto \{\mathsf{fst}=C_1, \mathsf{snd}=C_2\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash & m'_1 : M_1 \\
\Gamma \vdash & M_2 \\
\Gamma \vdash & \text{let } x = m'_1 \in m'_2 : M_2 \leadsto C
\end{align*}
\]
G.4 \textbf{exp formation:} $\Gamma \vdash m : M$

$\frac{}{\Gamma \vdash \text{valid} \ x : M \in \Gamma} $ \hspace{1cm} (203)

$\frac{}{\Gamma \vdash x : M} $ \hspace{1cm} (204)

$\frac{}{\Gamma \vdash \text{let} \ x = m \text{ in } m \in M} $ \hspace{1cm} (205)

$\frac{}{\Gamma \vdash p : \{\ldots, l ; M, \ldots\}} $ \hspace{1cm} (206)

$\frac{}{\Gamma \vdash \lambda x : M.m : \Pi x : M.M'} $ \hspace{1cm} (207)

$\frac{}{\Gamma \vdash \text{let} \ x = m \text{ in } m \in M} $ \hspace{1cm} (208)

G.6 \textbf{mtyp equivalence:} $\Gamma \vdash M \equiv M'$

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

G.7 \textbf{kind subsumption:} $\vdash K \leq K'$

Rules for reflexivity and transitivity are omitted.

$\frac{l_{\tau(i)} = l_i'}{\Gamma \vdash \{l_1 : K_1, \ldots, l_n : K_n\} \leq \{l'_1 : K_1', \ldots, l'_m : K'_m\}} $ \hspace{1cm} (218)

G.8 \textbf{type subsumption:} $\Gamma \vdash M \leq M'$

Rules for reflexivity and transitivity are omitted.

$\frac{l_{\tau(i)} = l_i'}{\Gamma \vdash \{l_1 : M_1, \ldots, l_n : M_n\} \leq \{l'_1 : M'_1, \ldots, l'_m : M'_m\}} $ \hspace{1cm} (219)

G.5 \textbf{constructor equivalence:} $\Gamma \vdash C \equiv C' : K$

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

$\frac{}{\Gamma \vdash \text{let} \ x = m_1 \text{ in } m_2 \in M_2} $ \hspace{1cm} (220)

$\frac{}{\Gamma \vdash \text{let} \ x = m_1 \text{ in } m_2 \in M_2} $ \hspace{1cm} (221)

$\frac{}{\Gamma \vdash \text{let} \ x = m_1 \text{ in } m_2 \in M_2} $ \hspace{1cm} (222)

$\frac{}{\Gamma \vdash \text{let} \ x = m_1 \text{ in } m_2 \in M_2} $ \hspace{1cm} (223)

\[\]