1 Introduction

The two main issues in the design of program modules are

1. Abstraction. To limit the interdependence among the modules in a system it is essential to be restrict the flow of type information among them.

2. Structure. To support multiple levels of modularity and to support reuse of modules it is essential to permit their hierarchical and parameterized construction.

These aspects of modularity are at once separable, complementary, and opposed to each other.

Abstraction alone is well-handled by existential types. The interface of an abstract type is given by an existential type. Its introductory form for an existential type packages a representation type with the implementation of the operations in terms of that representation. Its eliminatory form ensures that the representation type is hidden from the client, which ensures that the client is insulated from changes of its representation.

Existential types provide no help with structuring a program. The elimination form encapsulates the entire scope of an abstract type within which a package is opened. This means, in particular, that the lowest-level (most widely used) abstractions in a program must be given the largest scope—these abstractions lie at the root, rather than the leaves, of a larger program! Moreover, existential types do not address the structure of programs, providing no support for hierarchy or parameterization.

The structure of programs is well-managed by dependent product and function types for modules, as described in Chapter 45 of PFPL. Dependent products support the free flow of type information in a module hierarchy, with those components lower in the hierarchy accessing those higher up using projections. Dependent functions support parameterization, and permit the flow of type information from argument to result of an instance, also by using projections in the result type to refer to the argument module.

Dependent product and function types provide no help with abstraction, except insofar as one may consider the body of a parameterized module to be the “client” of its parameters. As with existentials, using dependent functions in this manner would invert program structure and obstruct the very forms of module composition it is intended to support.

Thus, an expressive type system for modules must support both abstraction and structuring. Neither suffices for the other, and both concepts are required. There is, however, a fundamental tension between the two aspects of modularity that must be resolved in any design. The core issue is the phase distinction between the statics and dynamics of a language. The clear separation between
compile-time and run-time featured in almost any programming language is threatened by modules, which combine statics and dynamics in a single entity. For example, an existential package consists of a representation type, its static component, with an implementation, its dynamic component. In a two-level hierarchy the lower module can access the static component of the entire upper module, appearing to create a type expression that involves both static and dynamic aspects. Similarly, in a parameterized module, the static component of the result depends on the entire parameter module, by projection, again appearing to violate phase separation.

It is natural to ask, so what? Why is it important to separate the static from the dynamic components of a module? Fundamentally, it is a matter of equality. Given two modules $M_1$ and $M_2$, when are their type components equal? Answering this is essential, because, for example, an application is type correct only when the argument type is equal to the domain type of the function. In special cases it may be plain to see that $M_1$ and $M_2$ have the same static components, and thus present no difficulties for checking their equality (beyond what is already required for checking type equality in the absence of modules).

Where the question becomes interesting (that is, difficult) is when the equality is not self-evident. For example, suppose that $M_1$ is of the form

$$\text{if the moon is full then } M_{1,1} \text{ else } M_{1,2}.$$ 

The static component of such a module is undefined during type checking, at least without provision of further information, and hence cannot be deemed equal to any given static component. Less extremely, $M_1$ might have an ascribed signature that renders its static components opaque in all contexts, precluding access to the “underlying truth.” For even if the ascribed module has a well-defined static component, it maximizes flexibility to allow it to be changed to one (such as in the preceding example) that does not, without affecting the type correctness of its clients.

To allow for this it is essential to distinguish module expressions from module computations, which is achieved using a modality similar to that used for PCF Harper (2019). The key property of a module value (a fully evaluated module expression) is that it always has a well-defined static component, accessible by projection, that can be compared for equality with any other. Module computations, however, must be bound to a module variable, which then stands as a placeholder for its value, before being accessed. Encapsulated module computations are ascribed with a signature that determines their public interface; such modules must be evaluated prior to their use, which has the effect of “generating” new abstract types.

2 A Language for Modularity

The syntax of the (revised) language Mod is given in Figure 1. Briefly,

1. Kinds and constructors are not affected by the presence of modularity constructs. This is achieved by ensuring that it is possible to determine the static significance of a module expression using a technical device called variable twinning.

2. Expressions include the extraction of the dynamic part of a module expression.

3. Module computations may be encapsulated as module values, with specified signature, and evaluated using sequencing, the associated elimination form.
4. Module values are mixed-phase entities with a static part consisting of constructors and a
dynamic part consisting of both constructors and values. Module values have well-defined
static parts, but module computations, such as the one in the introduction, do not.

5. There are two atomic forms of module value, a static module consisting of a constructor of a
kind, and a dynamic module consisting of an expression of a type. The static part of a static
module is itself; the static part of a dynamic module is trivial.

6. Module values include hierarchies, which are pairs in which the signature of the second com-
ponent may depend on the static part of the first component, and families, which are functions
whose result signature may depend on the instance module.

7. Module computations include initialization, which executes an encapsulated expression com-
putation for use within a module. The encapsulated computation might well allocate mutable
storage, which gives rise to the terminology.

8. Scaling of a module with a signature arises as the introductory form for the type of encap-
sulated (suspended) module computations, which must be active with a binding construct,
thereby enforcing abstraction. Abstract type proxy for the (implied) underlying state of the
module computation; for example, any two instances of a hash table have distinct underlying
state and hence induce distinct abstract types. Even pure modules can be sealed, and hence
be regarded as having a pro forma effect, so as to maximize future opportunities for replacing
a pure module by an impure one that relies on state.

Contexts are defined by the following grammar:

$$\Gamma ::= \epsilon | \Gamma, u :: \kappa | \Gamma, x : \tau | \Gamma, \Delta \downarrow : \sigma$$

Contexts may declare constructor variables of a kind, value variables of a type, and
twinned module variables with constructor variables of a signature. Static contexts, $\Delta$, are those that bind only
constructor variables.

The judgment forms that comprise the statics of the language are summarized in Figure 2. The
definitions of these forms make essential use of the computation of the static aspect of a pure module
and of its signature, those aspects that that govern its static behavior. Only pure modules have well-
deﬁned static aspects; this motivates the modal separation between expressions and computations.
The dynamic aspect of a module is its run-time significance, which in general encompasses its static
aspect, and also includes the runnable code of the module.

The static aspect of a signature and a module is speciﬁed in Figure 3, which deﬁnes simultane-
ously (1) the static context, $kd(\Gamma)$, associated with a context, $\Gamma$; (2) the kind part, $kd_\Delta(\sigma)$, of
a signature, $\sigma$, relative to a static context, $\Delta$; and (3) the constructor part, $con_\Delta(E)$, of a mod-
ule expression $E$, relative to a static context, $\Delta$. For notational convenience $con(\Gamma)E$ is deﬁned
to mean $con_\Delta(E)$, where $kd(\Gamma) = \Delta$. These operations are well-deﬁned when they are suitably
well-formed:

Lemma 2.1 (Phase Separation).

1. If $\Gamma \text{ ctx}$, then there exists unique $\Delta$ such that $kd(\Gamma) \triangleq \Delta$ and $\Delta \text{ ctx}$.

2. If $\Delta \vdash \sigma \text{ sig}$, then there exists unique $\kappa$ such that $kd_\Delta(\sigma) \triangleq \kappa$ and $\Delta \vdash \kappa \text{ kind}$.  

3
Figure 1: Abstract Syntax
\[ \Delta \text{ ctx} \quad \text{\(\Delta\) is a static context} \\
\Delta \vdash \kappa \text{ kind} \quad \kappa \text{ is a kind in } \Delta \\
\Delta \vdash c :: \kappa \quad c \text{ is a constructor of kind } \kappa \text{ in } \Delta \\
\Delta \vdash \kappa \leq \kappa' \quad \kappa \text{ is a subkind of } \kappa' \text{ in } \Delta \\
\Gamma \text{ ctx} \quad \Gamma \text{ is a context} \\
\Gamma \vdash e : \tau \quad e \text{ is an expression of type } \tau \text{ in } \Gamma \\
\Gamma \vdash m \triangleleft \tau \quad m \text{ is a computation of type } \tau \text{ in } \Gamma \\
\Delta \vdash \sigma \text{ sig} \quad \sigma \text{ is a signature in } \Gamma \\
\Gamma \vdash E : \sigma \quad E \text{ is a module expression of signature } \sigma \text{ in } \Gamma \\
\Gamma \vdash M \triangleleft \sigma \quad M \text{ is a module computation of signature } \sigma \text{ in } \Gamma \\
\Delta \vdash \sigma \leq \sigma' \text{ sig} \quad \sigma \text{ is a subsignature of } \sigma' \text{ in } \Delta \\
\]

Figure 2: Static Judgment Forms

3. If \( \Gamma \vdash \Sigma \ E : \sigma \), with \( \text{kd}(\Gamma) = \Delta \) and \( \text{kd}_\Delta(\sigma) = \kappa \), then there exists unique \( c \) such that \( \text{con}_\Delta(E) \triangleq c \) and \( \Delta \vdash c :: \kappa \).

Lemma 2.2 (Self-Recognition and Subsumption).

1. If \( \Gamma \vdash \Sigma \ E : \sigma \), then \( \Gamma \vdash \Sigma \ E : \Sigma_{\sigma}(E) \).

2. If \( \Gamma \vdash \Sigma \ E : \sigma \) and \( \Delta \vdash \sigma \leq \sigma' \text{ sig} \), where \( \text{kd}(\Gamma) = \Delta \), then \( \Gamma \vdash \Sigma \ E : \sigma' \).

3. If \( \Gamma \vdash \Sigma \ M : \sigma \) and \( \Delta \vdash \sigma \leq \sigma' \text{ sig} \), where \( \text{kd}(\Gamma) = \Delta \), then \( \Gamma \vdash \Sigma \ M : \sigma' \).

Substitutions are finite maps sending constructor variables to constructors, expression variables to expressions, and module variables to module expressions. The statics of substitutions is given in Figure 8.

Definition 2.3. Suppose that \( \Gamma' \vdash \gamma : \Gamma \) for some \( \Gamma \text{ ctx} \). The action \( \gamma(\varepsilon) \) of \( \gamma \) on a syntactic object \( \varepsilon \) is defined as follows:

\[
\begin{align*}
\gamma(\varepsilon) & \triangleq \varepsilon \\
\gamma \otimes \{u \mapsto c\}(\varepsilon) & \triangleq [c/u]\gamma(\varepsilon) \\
\gamma \otimes \{x \mapsto e\}(\varepsilon) & \triangleq [e/x]\gamma(\varepsilon) \\
\gamma \otimes \{X \downarrow u \mapsto E\}(\varepsilon) & \triangleq [E/X][c/u]\gamma(\varepsilon), \text{ where } \text{con}_{\Gamma'}(E) = c
\end{align*}
\]

Lemma 2.4 (Substitution). Suppose that \( \Gamma' \vdash \Sigma \gamma : \Gamma \) and let \( \delta = \text{con}_{\Gamma'}(\gamma) \).

1. If \( \Delta \vdash \kappa \text{ kind} \), then \( \Delta' \vdash \delta(\kappa) \text{ kind} \), and if \( \Delta \vdash \kappa \leq \kappa' \), then \( \Delta' \vdash \delta(\kappa) \leq \delta(\kappa') \).

2. If \( \Delta \vdash c :: \kappa \), then \( \Delta' \vdash \delta(c) :: \delta(\kappa) \).

3. If \( \Gamma \vdash \Sigma e : \tau \), then \( \Gamma' \vdash \Sigma \gamma(e) : \delta(\tau) \).
\[ \begin{align*}
kd_{\Delta}(\kappa) & \triangleq \kappa \\
kd_{\Delta}(\tau) & \triangleq 1 \\
kd_{\Delta}(\{\sigma\}) & \triangleq 1 \\
kd_{\Delta}(u :: \sigma_1 \times \sigma_2) & \triangleq u :: \kappa_1 \times \kappa_2 \text{ if } \kd_{\Delta}(\sigma_1) \triangleq \kappa_1 \text{ and } \kd_{\Delta,\kappa_1}(\sigma_2) \triangleq \kappa_2 \\
kd_{\Delta}(u :: \sigma_1 \rightarrow \sigma_2) & \triangleq u :: \kappa_1 \rightarrow \kappa_2 \text{ if } \kd_{\Delta}(\sigma_1) \triangleq \kappa_1 \text{ and } \kd_{\Delta,\kappa_1}(\sigma_2) \triangleq \kappa_2 \\
kd(\bullet) & \triangleq \bullet \\
kd(\Gamma, u :: \kappa) & \triangleq \Delta, u :: \kappa \text{ if } \kd(\Gamma) \triangleq \Delta \\
kd(\Gamma, x :: \tau) & \triangleq \Delta \text{ if } \kd(\Gamma) \triangleq \Delta \\
k\Gamma{\downarrow}u :: \sigma & \triangleq \Delta, u :: \kappa \text{ if } \kd(\Gamma) = \Delta \text{ and } \kd_{\Delta}(\sigma) \triangleq \kappa \\
con_{\Delta}(u) & \triangleq u \text{ if } u :: \kappa \in \Delta \\
con_{\Delta}(\{c\}) & \triangleq c \\
con_{\Delta}([c]) & \triangleq \langle \rangle \\
con_{\Delta}(M :: \sigma) & \triangleq \langle \rangle \\
con_{\Delta}(\langle E_1; E_2 \rangle) & \triangleq \langle c_1, c_2 \rangle \text{ if } \con_{\Delta}(E_1) \triangleq c_1 \text{ and } \con_{\Delta}(E_2) \triangleq c_2 \\
con_{\Delta}(\lambda X \downarrow u :: \sigma_1, E_2) & \triangleq \lambda (u :: \kappa_1) c_2 \text{ if } \kd_{\Delta}(\sigma_1) \triangleq \kappa_1 \text{ and } \kd_{\Delta,\kappa_1}(E_2) \triangleq c_2 \\
con_{\Delta}(E \cdot 1) & \triangleq c \cdot 1 \text{ if } \kd_{\Delta}(E) \triangleq c \\
con_{\Delta}(E \cdot 2) & \triangleq c \cdot 1 \text{ if } \kd_{\Delta}(E) \triangleq c \\
con_{\Delta}(E(E_1)) & \triangleq c[c_1] \text{ if } \kd_{\Delta}(E) \triangleq c \text{ and } \kd_{\Delta}(E_1) \triangleq c_1 \\
\end{align*} \]

Figure 3: Static Parts of Signatures, Modules, and Contexts

\[ \begin{align*}
et{val}_{\Sigma} & \text{ expression } e \text{ is fully evaluated} \\
e & \rightarrow_{\Sigma} e' \text{ expression } e \text{ steps to } e' \\
\nu \Sigma\{\mu \parallel m\} \text{ final} & \text{ computation } m \text{ is finished} \\
\nu \Sigma\{\mu \parallel m\} & \rightarrow_{\Sigma} \nu \Sigma'\{\mu' \parallel m'\} \text{ computation } m \text{ steps to } m', \text{ with effect} \\
\text{E}\val_{\Sigma} & \text{ module expression } E \text{ is fully evaluated} \\
E & \rightarrow_{\Sigma} E' \text{ module expression } E \text{ steps to } E' \\
\nu \Sigma\{\mu \parallel M\} \text{ final} & \text{ module computation } M \text{ is finished} \\
\nu \Sigma\{\mu \parallel M\} & \rightarrow_{\Sigma} \nu \Sigma\{\mu \parallel M'\} \text{ module computation } M \text{ steps to } M', \text{ with effect} \\
\end{align*} \]

Figure 4: Dynamic Judgment Forms
4. If $\Gamma \vdash_{\Sigma} m \sim \tau$, then $\Gamma' \vdash_{\Sigma} \hat{\gamma}(m) \sim \hat{\delta}(\tau)$.

5. If $\Delta \vdash \sigma \text{ sig}$, then $\Delta' \vdash \hat{\delta}(\sigma) \text{ sig}$, and if $\Delta \vdash \sigma \leq \sigma' \text{ sig}$, then $\Delta' \vdash \hat{\delta}(\sigma) <: \hat{\delta}(\sigma')$.

6. If $\Gamma \vdash_{\Sigma} E : \sigma$, then $\Gamma' \vdash_{\Sigma} \hat{\gamma}(E) : \hat{\delta}(\sigma)$.

7. If $\Gamma \vdash_{\Sigma} M \sim_{\sigma}$, then $\Gamma' \vdash_{\Sigma} \hat{\gamma}(M) \sim \hat{\delta}(\sigma)$.

References


\[ S_{[\kappa]}(E) \triangleq [S_{\kappa}(c)] \] if \( \text{con}_\Delta(E) = c \)

\[ S_{[\tau]}(E) \triangleq [\tau] \]

\[ S_{\{\sigma\}}(E) \triangleq \{\sigma\} \]

\[ S_{u: \sigma_1 \times \sigma_2}(E) \triangleq u :: S_{\sigma_1}(E \cdot 1) \times S_{[c_1/u] \sigma_2}(E \cdot 2) \] where \( \text{con}_\Delta(E \cdot 1) = c_1 \)

\[ S_{u: \sigma_1 \rightarrow \sigma_2}(E) \triangleq \lambda X.\downarrow u :: \sigma_1. S_{\sigma_2}(E(X)) \]

Figure 5: Singleton Signatures: \( \Delta \vdash S_{\sigma}(E) \triangleq \sigma' \)

---

\[ \frac{\Delta \vdash \kappa \text{ kind}}{\Delta \vdash [\kappa] \text{ sig}} \]

\[ \frac{\Delta \vdash \tau :: \text{Ty}}{\Delta \vdash [\tau] \text{ sig}} \]

\[ \frac{\Delta \vdash \sigma \text{ sig}}{\Delta \vdash \{\sigma\} \text{ sig}} \]

**SIG-CON**

**SIG-VAL**

**SIG-COMP**

\[ \frac{\Delta \vdash \sigma_1 \text{ sig}}{\Delta \vdash \Delta, u :: \kappa_1 \vdash \sigma_2 \text{ sig}} \quad \text{kd}_\Delta(\sigma_1) = \kappa_1 \]

\[ \Delta \vdash \Delta, u :: \sigma_1 \times \sigma_2 \text{ sig} \]

**SIG-HIER**

**SIG-FAM**

\[ \frac{\Delta \vdash \sigma_1 \text{ sig}}{\Delta \vdash \Gamma, u :: \kappa_1 \vdash \sigma_2 \text{ sig}} \quad \text{kd}_\Delta(\sigma_1) = \kappa_1 \]

\[ \Delta \vdash \Delta, u :: \sigma_1 \rightarrow \sigma_2 \text{ sig} \]

**SIG-STAT**

\[ \frac{\Delta \vdash \sigma \text{ sig}}{\Delta \vdash \sigma \text{ sig}} \]

Figure 6: Signature Statics: \( \Delta \vdash \sigma \text{ sig} \)

---

\[ \frac{\Delta \vdash \kappa <:: \kappa'}{\Delta \vdash [\kappa] <:: [\kappa']} \]

\[ \frac{\Delta \vdash \tau :: \text{Ty}}{\Delta \vdash [\tau] <:: [\tau]} \]

\[ \frac{\Delta \vdash \sigma <:: \sigma'}{\Delta \vdash \{\sigma\} <:: \{\sigma\}'} \]

**SUBSIG-CON**

**SUBSIG-VAL**

**SUBSIG-COMP**

\[ \frac{\Delta \vdash \Delta, u :: \kappa_1 \vdash \sigma_2 <:: \sigma_2'}{\Delta \vdash \Delta, u :: \sigma_1 \times \sigma_2 <:: u :: \sigma_1' \times \sigma_2'} \]

\[ \frac{\Delta \vdash \text{kd}_\Delta(\sigma_1') = \kappa_1'}{\Delta \vdash \Delta, u :: \sigma_1 \vdash \sigma_2 <:: u :: \sigma_1' \rightarrow \sigma_2'} \]

**SUBSIG-HIER**

**SUBSIG-FAM**

\[ \frac{\Delta \vdash \sigma_1 <:: \sigma_1'}{\Delta, u :: \kappa_1 \vdash \sigma_2 <:: \sigma_2'} \quad \text{kd}_\Delta(\sigma_1) = \kappa_1 \]

\[ \Delta \vdash \Delta, \frac{\Delta \vdash \sigma \vdash \sigma'}{\Delta \vdash \Delta, u :: \sigma_1 \rightarrow \sigma_2 <:: u :: \sigma_1' \rightarrow \sigma_2'} \]

**SUBSIG-WEAK**

\[ \frac{\Delta \vdash \sigma_1 <:: \sigma_1'}{\Delta, u :: \kappa_1 \vdash \sigma_2 <:: \sigma_2'} \quad \text{kd}_\Delta(\sigma_1') = \kappa_1' \]

\[ \Delta \vdash \Delta, \frac{\Delta \vdash \sigma \vdash \sigma'}{\Delta \vdash \Delta, u :: \sigma_1 \rightarrow \sigma_2 <:: u :: \sigma_1' \rightarrow \sigma_2'} \]

**SUBSIG-WEAK**

Figure 7: Signature Weakening: \( \Delta \vdash \sigma <:: \sigma' \)
Figure 8: Substitution Statics

\[
\begin{align*}
\text{SUBST-EMP} & : \quad \emptyset : \bullet \\
\text{SUBST-CON} & : \quad \Gamma' \vdash_{\Sigma} \gamma : \Gamma \quad \text{con}_\Gamma(\gamma) = \delta \quad \Delta' \vdash c :: \delta(\kappa) \\
& \quad \quad \quad \Gamma' \vdash_{\Sigma} \gamma \otimes \{u \leftarrow c\} : \Gamma, u :: \kappa \\
\text{SUBST-EXP} & : \quad \Gamma' \vdash_{\Sigma} \gamma : \Gamma \quad \text{con}_\Gamma(\gamma) = \delta \quad \Gamma' \vdash e : \delta(\tau) \\
& \quad \quad \quad \Gamma' \vdash_{\Sigma} \gamma \otimes \{x \leftarrow e\} : \Gamma, x : \tau \\
\text{SUBST-VAL} & : \quad \Gamma' \vdash_{\Sigma} \gamma : \Gamma \quad \text{con}_\Gamma(\gamma) = \delta \quad \Gamma' \vdash E : \delta(\sigma) \\
& \quad \quad \quad \Gamma' \vdash_{\Sigma} \gamma \otimes \{X \downarrow u \leftarrow E\} : \Gamma, X \downarrow u : \sigma
\end{align*}
\]

Figure 9: Module Expression Statics: \( \Gamma \vdash_{\Sigma} E : \sigma \)

\[
\begin{align*}
\text{MOD-VAR} & : \quad \Delta \vdash \sigma <: \sigma' \quad \text{kd}(\Gamma) = \Delta \\
& \quad \quad \quad \Gamma, X : \sigma \Gamma' \vdash_{X} X : S_{\sigma'}(X) \\
\text{MOD-CON} & : \quad \Delta \vdash c :: \kappa \quad \text{kd}(\Gamma) = \Delta \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} \text{[c]} : [S_{\kappa}(c)] \\
\text{MOD-EXP} & : \quad \Delta \vdash \sigma' \quad \text{kd}(\Gamma) = \Delta \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} \text{[e]} : [\tau] \\
\text{MOD-COMP} & : \quad \Delta \vdash \sigma' \quad \Delta \vdash \sigma <: \sigma' \quad \text{kd}(\Gamma) = \Delta \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} M : \sigma \\
\text{MOD-FAM} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} \text{[c]} : [\sigma_1 \times \sigma_2] \\
\text{MOD-HIER} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E_1 : \sigma_1 \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E_2 : \text{[c_1/u]} \sigma_2 \\
& \quad \quad \quad \text{con}_\Gamma(E_1) = c_1 \\
\text{MOD-INST} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E_1 : \sigma_1 \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E_1 : \sigma_2 \\
& \quad \quad \quad \text{con}_\Gamma(E_1) = c_1 \\
\text{MOD-FST} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E : \sigma_1 \times \sigma_2 \\
\text{MOD-SND} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E : \sigma_1 \times \sigma_2 \\
& \quad \quad \quad \text{con}_\Gamma(E) = c \\
\text{MOD-RET} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma' \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E : \sigma \\
& \quad \quad \quad \text{mod-ret}(E) : \sigma \\
\text{MOD-BND} & : \quad \Delta \vdash \sigma \quad \Delta \vdash \sigma_2 \quad \text{sig} \\
& \quad \quad \quad \Gamma \vdash_{\Sigma} E_1 : \{\sigma_1\} \\
& \quad \quad \quad \Gamma, X \downarrow u : \sigma_1 \vdash_{\Sigma} M_2 : \sigma_2 \\
\text{Figure 10: Module Computation Statics: } \Gamma \vdash_{\Sigma} M : \sigma
\]
Figure 11: Expression and Computation Statics

Figure 12: Module Expression Dynamics
<table>
<thead>
<tr>
<th>MOD-INITIAL</th>
<th>MOD-FINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu_0 : \Sigma_0 )</td>
<td>( E \text{ val}_\Sigma )</td>
</tr>
<tr>
<td>( \nu \Sigma_0 { \mu_0 \parallel M \text{ initial} } )</td>
<td>( \nu \Sigma { \mu \parallel \text{ret } E \text{ final} } )</td>
</tr>
</tbody>
</table>

**MOD-STEP-RET**

\[
E \xrightarrow{\Sigma} E'
\]

\[
\nu \Sigma \{ \mu \parallel \text{ret } E \} \xrightarrow{\Sigma} \nu \Sigma \{ \mu \parallel \text{ret } E' \}
\]

**MOD-STEP-BND**

\[
E_1 \xrightarrow{\Sigma} E_1'
\]

\[
\nu \Sigma \{ \mu \parallel \text{bnd } X \downarrow u \leftarrow E_1 ; M_2 \} \xrightarrow{\Sigma} \nu \Sigma \{ \mu \parallel \text{bnd } X \downarrow u \leftarrow E_1' ; M_2 \}
\]

**MOD-STEP-BND-RET**

\[
E_1 \text{ val}_\Sigma \quad \text{con}(E_1) = c_1
\]

\[
\nu \Sigma \{ \mu \parallel \text{bnd } X \downarrow u \leftarrow (\text{ret } E_1) : \sigma_1 ; M_2 \} \xrightarrow{\Sigma} \nu \Sigma' \{ \mu' \parallel [E_1/X][c_1/u]M_2 \}
\]

**MOD-STEP-BND-STEP**

\[
\nu \Sigma \{ \mu \parallel M_1 \} \xrightarrow{\Sigma} \nu \Sigma' \{ \mu' \parallel M_1' \}
\]

\[
\nu \Sigma \{ \mu \parallel \text{bnd } X \downarrow u \leftarrow M_1 : \sigma_1 ; M_2 \} \xrightarrow{\Sigma} \nu \Sigma' \{ \mu' \parallel \text{bnd } X \downarrow u \leftarrow M_1' : \sigma_1 ; M_2 \}
\]

**MOD-STEP-USE**

\[
e \xrightarrow{\Sigma} e'
\]

\[
\nu \Sigma \{ \mu \parallel \text{use } (e) \} \xrightarrow{\Sigma} \nu \Sigma \{ \mu \parallel \text{use } (e') \}
\]

**MOD-STEP-USE-STEP**

\[
\nu \Sigma \{ \mu \parallel \text{use } (M : \sigma) \} \xrightarrow{\Sigma} \nu \Sigma \{ \mu \parallel M \}
\]

Figure 13: Module Computation Dynamics