PFPL Supplement: Types for Program Modules[*]

Robert Harper

Summer, 2020

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

[*]Copyright © Robert Harper. All Rights Reserved.

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

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

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.

Acknowledgements: Thanks to Jon Sterling and Haoxuan Yue for their comments and corrections to earlier drafts of this supplement.

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 component 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 computation for use within a module. The encapsulated computation might well allocate mutable storage, which gives rise to the terminology.

8. *Sealing* of a module with a signature arises as the introductory form for the type of encapsulated (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.

Contexs are defined by the following grammar:

$$\Gamma \quad ::= \quad \bullet \mid \Gamma, u :: \kappa \mid \Gamma, x : \tau \mid \Gamma, X{\downarrow}u : \sigma$$

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*, Δ, are those contexts 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 judgments governing the dynamic aspects are parameterized by a context Σ that is used to associate types to locations and other dynamically-generated constants; it plays no explicit role here, but is required when stating and proving safety for a full language.

The static aspect of a signature and a module is specified in Figure 3, which defines simultaneously (1) the static context, $\mathtt{kd}(\,\Gamma\,)$, associated with a context, Γ; (2) the kind part, $\mathtt{kd}_{\Delta}(\,\sigma\,)$, of a signature, σ, relative to a static context, Δ; and (3) the constructor part, $\mathtt{con}_{\Delta}(\,E\,)$, of a module expression E, relative to a static context, Δ. For notational convenience $\mathtt{con}_{\Gamma}(\,E\,)$ is defined to mean $\mathtt{con}_{\Delta}(\,E\,)$, where $\mathtt{kd}(\,\Gamma\,) \triangleq \Delta$. Similarly,

These operations are well-defined when they are suitably well-formed:

Sig's	σ	$::=$	$\texttt{con}(\,k\,)$	$[\,k\,]$	static
			$\texttt{val}(\,\tau\,)$	$[\,\tau\,]$	dynamic
			$\texttt{comp}(\,\sigma\,)$	$\{\sigma\}$	capsule
			$\Sigma(\,\sigma_1\,;u\,.\,\sigma_2\,)$	$u :: \sigma_1 \times \sigma_2$	hierarchy
			$\Pi(\,\sigma_1\,;u\,.\,\sigma_2\,)$	$u :: \sigma_1 \rightarrow \sigma_2$	family
Mod Exp's	E	$::=$	X	X	variable
			$\texttt{con}(\,c\,)$	$[\,c\,]$	constructor
			$\texttt{val}(\,e\,)$	$[\,e\,]$	expression
			$\texttt{comp}\{\sigma\}(\,M\,)$	$M :> \sigma$	capsule
			$\texttt{hier}(\,E_1\,;E_2\,)$	$\langle E_1\,;E_2\rangle$	hierarchy
			$\texttt{fam}\{\sigma_1\}(\,X\,.\,u\,.\,E_2\,)$	$\lambda X{\downarrow}u : \sigma_1.E_2$	family
			$\texttt{fst}(\,E\,)$	$E \cdot \texttt{1}$	upper
			$\texttt{snd}(\,E\,)$	$E \cdot \texttt{2}$	lower
			$\texttt{inst}(\,E_1\,;E_2\,)$	$E_1(\,E_2\,)$	instance
Mod Comp's	M	$::=$	$\texttt{ret}(\,E\,)$	$\texttt{ret}\,E$	value
			$\texttt{bnd}(\,E_1\,;X\,.\,u\,.\,M_2\,)$	$\texttt{bnd}\,X{\downarrow}u \leftarrow E_1\,;M_2$	sequence
			$\texttt{use}(\,e\,)$	$\texttt{use}\,e$	use dynamic module
Kinds	κ	$::=$	$\texttt{Ty}$	$\texttt{Ty}$	type
			$\texttt{Unit}$	1	unit
			$\texttt{S}(\,c\,)$	$\texttt{S}(\,c\,)$	singleton
			$\Sigma(\,\kappa_1\,;u\,.\,\kappa_2\,)$	$u :: \kappa_1 \times \kappa_2$	product
			$\Pi(\,\kappa_1\,;u\,.\,\kappa_2\,)$	$u :: \kappa_1 \rightarrow \kappa_2$	function
Constr's	c, τ	$::=$	u	u	variable
			$\texttt{triv}$	$\langle\rangle$	null tuple
			$\texttt{comp}(\,\tau\,)$	$\tau\,\texttt{comp}$	computation
			$\texttt{mod}(\,\sigma\,)$	$\sigma\,\texttt{mod}$	dynamic module
			$\texttt{pair}(\,c_1\,;c_2\,)$	$\langle c_1, c_2\rangle$	pair
			$\texttt{proj}[\,i\,](\,c\,)$	$c \cdot i$	projection $(i = 1, 2)$
			$\lambda\{\kappa_1\}(\,u\,.\,c_2\,)$	$\lambda\,(\,u :: \kappa_1\,)\,c_2$	abstraction
			$\texttt{app}(\,c_1\,;c_2\,)$	$c_1[\,c_2\,]$	application
Exp's	e	$::=$	x	x	variable
			$\texttt{val}(\,E\,)$	$E \cdot \texttt{val}$	value part
			$\texttt{comp}(\,m\,)$	$\texttt{comp}(\,m\,)$	computation
			$\texttt{mod}\{\sigma\}(\,M\,)$	$M :> \sigma$	dynamic module
Comp's	m	$::=$	$\texttt{ret}(\,e\,)$	$\texttt{ret}\,e$	expression
			$\texttt{bnd}(\,e_1\,;x\,.\,m_2\,)$	$\texttt{bnd}\,x \leftarrow e_1\,;m_2$	sequence
			$\texttt{loc}(\,E\,;X\,.\,u\,.\,m\,)$	$\texttt{loc}\,X{\downarrow}u \leftarrow E\,;m$	local

Figure 1: Abstract Syntax

$\Delta\ \mathsf{ctx}$	Δ is a static context
$\Delta \vdash \kappa\ \mathsf{kind}$	κ is a kind in Δ
$\Delta \vdash c :: \kappa$	c is a constructor of kind κ in Δ
$\Delta \vdash \kappa \leq \kappa'$	k is a subkind of κ' in Δ
$\Gamma\ \mathsf{ctx}$	Γ is a context
$\Gamma \vdash_\Sigma e : \tau$	e is an expression of type τ in Γ
$\Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau$	m is a computation of type τ in Γ
$\Delta \vdash \sigma\ \mathsf{sig}$	σ is a signature in Γ
$\Gamma \vdash_\Sigma E : \sigma$	E is a module expression of signature σ in Γ
$\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma$	M is a module computation of signature σ in Γ
$\Delta \vdash \sigma \leq \sigma'\ \mathsf{sig}$	σ is a subsignature of σ' in Δ

Figure 2: Static Judgment Forms

Lemma 2.1 (Phase Separation)**.**

1. *If $\Gamma\ \mathsf{ctx}$, then there exists unique Δ such that $\mathtt{kd}(\Gamma) \triangleq \Delta$ and $\Delta\ \mathsf{ctx}$.*
2. *If $\Delta \vdash \sigma\ \mathsf{sig}$, then there exists unique κ such that $\mathtt{kd}_\Delta(\sigma) \triangleq \kappa$ and $\Delta \vdash \kappa\ \mathsf{kind}$.*
3. *If $\Gamma \vdash_\Sigma E : \sigma$, with $\mathtt{kd}(\Gamma) \triangleq \Delta$ and $\mathtt{kd}_\Delta(\sigma) \triangleq \kappa$, then there exists unique c such that $\mathtt{con}_\Delta(E) \triangleq c$ and $\Delta \vdash c :: \kappa$.*

Lemma 2.2 (Self-Recognition and Subsumption)**.**

1. *If $\Gamma \vdash_\Sigma E : \sigma$, then $\Gamma \vdash_\Sigma E : \mathtt{S}_\sigma(E)$.*
2. *If $\Gamma \vdash_\Sigma E : \sigma$ and $\Delta \vdash \sigma \leq \sigma'\ \mathsf{sig}$, where $\mathtt{kd}(\Gamma) \triangleq \Delta$, then $\Gamma \vdash_\Sigma E : \sigma'$.*
3. *If $\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma$ and $\Delta \vdash \sigma \leq \sigma'\ \mathsf{sig}$, where $\mathtt{kd}(\Gamma) \triangleq \Delta$, then $\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma'$.*

A *substitutions* is a finite map sending constructor variables to constructors, expression variables to expression, and module variables to module expressions. A *static substitution* is a finite map sending constructor variables to constructors. The statics of substitutions is given by the judgment $\Gamma \vdash_\Sigma \gamma : \Gamma$ defined in Figure 8. The general form restricts to the static case, written $\Delta' \vdash \delta :: \Delta'$, when the contexts and substitution govern only constructor variables.

Definition 2.3. *Suppose that $\Gamma' \vdash_\Sigma \gamma : \Gamma$ for some $\Gamma\ \mathsf{ctx}$. The* action $\widehat{\gamma}(\varepsilon)$ *of γ on a syntactic object ε is defined as follows:*

$$\begin{aligned}
\widehat{\emptyset}(\varepsilon) &\triangleq \varepsilon \\
\widehat{\gamma \otimes u \hookrightarrow c}(\varepsilon) &\triangleq [c/u]\widehat{\gamma}(\varepsilon) \\
\widehat{\gamma \otimes x \hookrightarrow e}(\varepsilon) &\triangleq [e/x]\widehat{\gamma}(\varepsilon) \\
\widehat{\gamma \otimes X{\downarrow}u \hookrightarrow E}(\varepsilon) &\triangleq [E/X][c/u]\widehat{\gamma}(\varepsilon), \ \textit{where } \mathtt{con}_{\Gamma'}(E) \triangleq c
\end{aligned}$$

$$
\begin{array}{rcll}
\mathtt{kd}_\Delta(\,[\,\kappa\,]\,) & \triangleq & \kappa & \\
\mathtt{kd}_\Delta(\,[\,\tau\,]\,) & \triangleq & 1 & \\
\mathtt{kd}_\Delta(\,\{\sigma\}\,) & \triangleq & 1 & \\
\mathtt{kd}_\Delta(\,u :: \sigma_1 \times \sigma_2\,) & \triangleq & u :: \kappa_1 \times \kappa_2 & \text{if } \mathtt{kd}_\Delta(\,\sigma_1\,) \triangleq \kappa_1 \text{ and } \mathtt{kd}_{\Delta,u::\kappa_1}(\,\sigma_2\,) \triangleq \kappa_2 \\
\mathtt{kd}_\Delta(\,u :: \sigma_1 \to \sigma_2\,) & \triangleq & u :: \kappa_1 \to \kappa_2 & \text{if } \mathtt{kd}_\Delta(\,\sigma_1\,) \triangleq \kappa_1 \text{ and } \mathtt{kd}_{\Delta,u::\kappa_1}(\,\sigma_2\,) \triangleq \kappa_2 \\
 & & & \\
\mathtt{kd}(\,\bullet\,) & \triangleq & \bullet & \\
\mathtt{kd}(\,\Gamma, u :: \kappa\,) & \triangleq & \Delta, u :: \kappa & \text{if } \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \\
\mathtt{kd}(\,\Gamma, x : \tau\,) & \triangleq & \Delta & \text{if } \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \\
\mathtt{kd}(\,\Gamma, X{\downarrow}u : \sigma\,) & \triangleq & \Delta, u :: \kappa & \text{if } \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \text{ and } \mathtt{kd}_\Delta(\,\sigma\,) \triangleq \kappa \\
 & & & \\
\mathtt{con}_\Delta(\,u\,) & \triangleq & u & \text{if } u :: \kappa \in \Delta \\
\mathtt{con}_\Delta(\,[\,c\,]\,) & \triangleq & c & \\
\mathtt{con}_\Delta(\,[\,e\,]\,) & \triangleq & \langle\rangle & \\
\mathtt{con}_\Delta(\,M :> \sigma\,) & \triangleq & \langle\rangle & \\
\mathtt{con}_\Delta(\,\langle E_1\,;E_2\rangle\,) & \triangleq & \langle c_1, c_2\rangle & \text{if } \mathtt{con}_\Delta(\,E_1\,) \triangleq c_1 \text{ and } \mathtt{con}_\Delta(\,E_2\,) \triangleq c_2 \\
\mathtt{con}_\Delta(\,\lambda X{\downarrow}u : \sigma_1.E_2\,) & \triangleq & \lambda\,(\,u :: \kappa_1\,)\,c_2 & \text{if } \mathtt{kd}_\Delta(\,\sigma_1\,) \triangleq \kappa_1 \text{ and } \mathtt{con}_{\Delta,u::\kappa_1}(\,E_2\,) \triangleq c_2 \\
\mathtt{con}_\Delta(\,E \cdot \mathtt{1}\,) & \triangleq & c \cdot \mathtt{1} & \text{if } \mathtt{con}_\Delta(\,E\,) \triangleq c \\
\mathtt{con}_\Delta(\,E \cdot \mathtt{2}\,) & \triangleq & c \cdot \mathtt{2} & \text{if } \mathtt{con}_\Delta(\,E\,) \triangleq c \\
\mathtt{con}_\Delta(\,E(\,E_1\,)\,) & \triangleq & c[\,c_1\,] & \text{if } \mathtt{con}_\Delta(\,E\,) \triangleq c \text{ and } \mathtt{con}_\Delta(\,E_1\,) \triangleq c_1
\end{array}
$$

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

$e\ \mathsf{val}_\Sigma$	expression e is fully evaluated
$e \underset{\Sigma}{\longmapsto} e'$	expression e steps to e'
$\nu\,\Sigma\{\mu \parallel m\}\ \mathsf{final}$	computation m is finished
$\nu\,\Sigma\{\mu \parallel m\} \longmapsto \nu\,\Sigma'\{\mu' \parallel m'\}$	computation m steps to m', with effect
$E\ \mathsf{val}_\Sigma$	module expression E is fully evaluated
$E \underset{\Sigma}{\longmapsto} E'$	module expression E steps to E'
$\nu\,\Sigma\{\mu \parallel M\}\ \mathsf{final}$	module computation M is finished
$\nu\,\Sigma\{\mu \parallel M\} \longmapsto \nu\,\Sigma\{\mu \parallel M'\}$	module computation M steps to M', with effect

Figure 4: Dynamic Judgment Forms

Lemma 2.4 (Substitution). *Suppose that* $\Gamma' \vdash_\Sigma \gamma : \Gamma$ *and let* $\Delta \triangleq \mathtt{kd}(\,\Gamma\,)$, $\Delta' \triangleq \mathtt{kd}(\,\Gamma'\,)$, *and* $\delta \triangleq \mathtt{con}_{\Delta'}(\,\gamma\,)$.

1. $\Delta' \vdash \delta :: \Delta$.
2. *If* $\Delta \vdash \kappa$ kind, *then* $\Delta' \vdash \hat{\delta}(\kappa)$ kind, *and if* $\Delta \vdash \kappa \leq \kappa'$, *then* $\Delta' \vdash \hat{\delta}(\kappa) \leq \hat{\delta}(\kappa')$.
3. *If* $\Delta \vdash c :: \kappa$, *then* $\Delta' \vdash \hat{\delta}(c) :: \hat{\delta}(\kappa)$.
4. *If* $\Gamma \vdash_\Sigma e : \tau$, *then* $\Gamma' \vdash_\Sigma \hat{\gamma}(e) : \hat{\delta}(\tau)$.
5. *If* $\Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau$, *then* $\Gamma' \vdash_\Sigma \hat{\gamma}(m) \mathrel{\dot{\sim}} \hat{\delta}(\tau)$.
6. *If* $\Delta \vdash \sigma$ sig, *then* $\Delta' \vdash \hat{\delta}(\sigma)$ sig, *and if* $\Delta \vdash \sigma \leq \sigma'$ sig, *then* $\Delta' \vdash \hat{\delta}(\sigma) <: \hat{\delta}(\sigma')$.
7. *If* $\Gamma \vdash_\Sigma E : \sigma$, *then* $\Gamma' \vdash_\Sigma \hat{\gamma}(E) : \hat{\delta}(\sigma)$.
8. *If* $\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma$, *then* $\Gamma' \vdash_\Sigma \hat{\gamma}(M) \mathrel{\dot{\sim}} \hat{\delta}(\sigma)$.

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. PCF-By-Value. Supplement to Harper (2016), Fall 2019. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/pcfv.pdf`.

$$\begin{aligned}
\mathtt{S}_{[\,\kappa\,]}\,(\,E\,) &\triangleq \;[\,\mathtt{S}_{\kappa}\,(\,c\,)\,] \quad \text{where } \mathtt{con}_{\Delta}(\,E\,) \triangleq c \\
\mathtt{S}_{[\,\tau\,]}\,(\,E\,) &\triangleq [\,\tau\,] \\
\mathtt{S}_{\{\sigma\}}\,(\,E\,) &\triangleq \{\sigma\} \\
\mathtt{S}_{u::\sigma_1\times\sigma_2}\,(\,E\,) &\triangleq u :: \mathtt{S}_{\sigma_1}\,(\,E\cdot\mathtt{1}\,) \times \mathtt{S}_{[c_1/u]\sigma_2}\,(\,E\cdot\mathtt{2}\,) \quad \text{where } \mathtt{con}_{\Delta}(\,E\cdot\mathtt{1}\,) \triangleq c_1 \\
\mathtt{S}_{u::\sigma_1\to\sigma_2}\,(\,E\,) &\triangleq \lambda X{\downarrow}u : \sigma_1.\mathtt{S}_{\sigma_2}\,(\,E(\,X\,)\,)
\end{aligned}$$

Figure 5: Singleton Signatures: $\Delta \vdash \mathtt{S}_{\sigma}\,(\,E\,) \triangleq \sigma'$

SIG-CON
$$\frac{\Delta \vdash \kappa \ \mathsf{kind}}{\Delta \vdash [\,\kappa\,]\ \mathsf{sig}}$$

SIG-VAL
$$\frac{\Delta \vdash \tau :: \mathtt{Ty}}{\Delta \vdash [\,\tau\,]\ \mathsf{sig}}$$

SIG-COMP
$$\frac{\Delta \vdash \sigma\ \mathsf{sig}}{\Delta \vdash \{\sigma\}\ \mathsf{sig}}$$

SIG-HIER
$$\frac{\Delta \vdash \sigma_1\ \mathsf{sig} \qquad \Delta, u :: \kappa_1 \vdash \sigma_2\ \mathsf{sig} \qquad \mathtt{kd}_{\Delta}(\,\sigma_1\,) \triangleq \kappa_1}{\Delta \vdash u :: \sigma_1 \times \sigma_2\ \mathsf{sig}}$$

SIG-FAM
$$\frac{\Delta \vdash \sigma_1\ \mathsf{sig} \qquad \Gamma, u :: \kappa_1 \vdash \sigma_2\ \mathsf{sig} \qquad \mathtt{kd}_{\Delta}(\,\sigma_1\,) \triangleq \kappa_1}{\Delta \vdash u :: \sigma_1 \to \sigma_2\ \mathsf{sig}}$$

Figure 6: Signature Statics: $\Delta \vdash \sigma\ \mathsf{sig}$

SUBSIG-CON
$$\frac{\Delta \vdash \kappa <:: \kappa'}{\Delta \vdash [\,\kappa\,] <: [\,\kappa'\,]}$$

SUBSIG-VAL
$$\frac{\Delta \vdash \tau :: \mathtt{Ty}}{\Delta \vdash [\,\tau\,] <: [\,\tau\,]}$$

SUBSIG-COMP
$$\frac{\Delta \vdash \sigma <: \sigma'}{\Delta \vdash \{\sigma\} <: \{\sigma'\}}$$

SUBSIG-HIER
$$\frac{\Delta \vdash \sigma_1 <: \sigma_1' \qquad \Delta, u :: \kappa_1 \vdash \sigma_2 <: \sigma_2' \qquad \mathtt{kd}_{\Delta}(\,\sigma_1\,) \triangleq \kappa_1}{\Delta \vdash u :: \sigma_1 \times \sigma_2 <: u :: \sigma_1' \times \sigma_2'}$$

SUBSIG-FAM
$$\frac{\Delta \vdash \sigma_1' <: \sigma_1 \qquad \Delta, u :: \kappa_1' \vdash \sigma_2 <: \sigma_2' \qquad \mathtt{kd}_{\Delta}(\,\sigma_1'\,) \triangleq \kappa_1'}{\Delta \vdash u :: \sigma_1 \to \sigma_2 <: u :: \sigma_1' \to \sigma_2'}$$

Figure 7: Signature Weakening: $\Delta \vdash \sigma <: \sigma'$

$$\frac{}{\bullet \vdash_\Sigma \emptyset : \bullet}\ \textsc{subst-emp}$$

$$\frac{\Gamma' \vdash_\Sigma \gamma : \Gamma \qquad \mathtt{kd}(\,\Gamma'\,) \triangleq \Delta' \qquad \mathtt{con}_{\Delta'}(\,\gamma\,) \triangleq \delta \qquad \Delta' \vdash c :: \hat{\delta}(\kappa)}{\Gamma' \vdash_\Sigma \gamma \otimes u \hookrightarrow c : \Gamma, u :: \kappa}\ \textsc{subst-con}$$

$$\frac{\Gamma' \vdash_\Sigma \gamma : \Gamma \qquad \mathtt{con}_{\Gamma'}(\,\gamma\,) \triangleq \delta \qquad \Gamma' \vdash_\Sigma e : \hat{\delta}(\tau)}{\Gamma' \vdash_\Sigma \gamma \otimes x \hookrightarrow e : \Gamma, x : \tau}\ \textsc{subst-exp}$$

$$\frac{\Gamma' \vdash_\Sigma \gamma : \Gamma \qquad \mathtt{con}_{\Gamma'}(\,\gamma\,) \triangleq \delta \qquad \Gamma' \vdash_\Sigma E : \hat{\delta}(\sigma)}{\Gamma' \vdash_\Sigma \gamma \otimes X{\downarrow}u \hookrightarrow E : \Gamma, X{\downarrow}u : \sigma}\ \textsc{subst-Val}$$

Figure 8: Substitution Statics

$$\frac{\mathtt{kd}(\,\Gamma\,) \triangleq \Delta \qquad \Delta \vdash \sigma <: \sigma'}{\Gamma, X : \sigma\ \Gamma' \vdash_\Sigma X : \mathtt{S}_{\sigma'}\,(\,X\,)}\ \textsc{mod-var} \qquad \frac{\mathtt{kd}(\,\Gamma\,) \triangleq \Delta \qquad \Delta \vdash c :: \kappa}{\Gamma \vdash_\Sigma [\,c\,] : [\,\mathtt{S}_\kappa\,(\,c\,)\,]}\ \textsc{mod-con} \qquad \frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma [\,e\,] : [\,\tau\,]}\ \textsc{mod-exp}$$

$$\frac{\Gamma \vdash_\Sigma M \dot{\sim} \sigma \qquad \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \qquad \Delta \vdash \sigma <: \sigma'}{\Gamma \vdash_\Sigma M :> \sigma' : \{\sigma'\}}\ \textsc{mod-comp}$$

$$\frac{\Gamma \vdash_\Sigma E_1 : \sigma_1 \qquad \Gamma \vdash_\Sigma E_2 : [c_1/u]\sigma_2 \qquad \mathtt{con}_\Gamma(\,E_1\,) \triangleq c_1}{\Gamma \vdash_\Sigma \langle E_1\,;\,E_2\rangle : u :: \sigma_1 \times \sigma_2}\ \textsc{mod-hier} \qquad \frac{\Gamma, X{\downarrow}u : \sigma_1 \vdash_\Sigma E_2 : \sigma_2}{\Gamma \vdash_\Sigma \lambda X{\downarrow}u : \sigma_1.E_2 : u :: \sigma_1 \to \sigma_2}\ \textsc{mod-fam}$$

$$\frac{\Gamma \vdash_\Sigma E : u :: \sigma_1 \times \sigma_2}{\Gamma \vdash_\Sigma E \cdot \mathtt{1} : \sigma_1}\ \textsc{mod-fst} \qquad \frac{\Gamma \vdash_\Sigma E : u :: \sigma_1 \times \sigma_2 \qquad \mathtt{con}_\Gamma(\,E\,) \triangleq c}{\Gamma \vdash_\Sigma E \cdot \mathtt{2} : [c \cdot \mathtt{1}/u]\sigma_2}\ \textsc{mod-snd}$$

$$\frac{\Gamma \vdash_\Sigma E : u :: \sigma_1 \to \sigma_2 \qquad \Gamma \vdash_\Sigma E_1 : \sigma_1 \qquad \mathtt{con}_\Gamma(\,E_1\,) \triangleq c_1}{\Gamma \vdash_\Sigma E(\,E_1\,) : [c_1/u]\sigma_2}\ \textsc{mod-inst}$$

Figure 9: Module Expression Statics: $\Gamma \vdash_\Sigma E : \sigma$

$$\text{MOD-RET}\quad \frac{\Gamma \vdash_\Sigma E : \sigma}{\Gamma \vdash_\Sigma \mathtt{ret}\, E \mathrel{\dot{\sim}} \sigma} \qquad \text{MOD-USE}\quad \frac{\Gamma \vdash_\Sigma e : \mathtt{mod}(\,\sigma\,)}{\Gamma \vdash_\Sigma \mathtt{use}(\,e\,) \mathrel{\dot{\sim}} \sigma}$$

$$\text{MOD-BND}\quad \frac{\Gamma \vdash_\Sigma E_1 : \{\sigma_1\} \qquad \Gamma, X{\downarrow}u : \sigma_1 \vdash_\Sigma M_2 \mathrel{\dot{\sim}} \sigma_2 \qquad \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \qquad \Delta \vdash \sigma_2\ \mathsf{sig}}{\Gamma \vdash_\Sigma \mathtt{bnd}\, X{\downarrow}u \leftarrow E_1\,;\, M_2 \mathrel{\dot{\sim}} \sigma_2}$$

Figure 10: Module Computation Statics: $\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma$

$$\text{EXP-VAL}\quad \frac{\Gamma \vdash_\Sigma E : [\,\tau\,]}{\Gamma \vdash_\Sigma E \cdot \mathtt{val} : \tau} \qquad \text{EXP-COMP}\quad \frac{\Gamma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau}{\Gamma \vdash_\Sigma \mathtt{comp}(\,m\,) : \tau\, \mathtt{comp}} \qquad \text{EXP-DYNMOD}\quad \frac{\Gamma \vdash_\Sigma M \mathrel{\dot{\sim}} \sigma}{\Gamma \vdash_\Sigma M :> \sigma : \sigma\, \mathtt{mod}}$$

$$\text{COMP-RET}\quad \frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{ret}\, e \mathrel{\dot{\sim}} \tau} \qquad \text{COMP-BND}\quad \frac{\Gamma \vdash_\Sigma e_1 : \tau_1\, \mathtt{comp} \qquad \Gamma, x : \tau_1 \vdash m_2 \mathrel{\dot{\sim}} \tau_2}{\Gamma \vdash_\Sigma \mathtt{bnd}\, x \leftarrow e_1\,;\, m_2 \mathrel{\dot{\sim}} \tau_2}$$

$$\text{COMP-LOC}\quad \frac{\Gamma \vdash_\Sigma E : \sigma\, \mathtt{mod} \qquad \Gamma, X{\downarrow}u : \sigma \vdash_\Sigma m \mathrel{\dot{\sim}} \tau \qquad \mathtt{kd}(\,\Gamma\,) \triangleq \Delta \qquad \Delta \vdash \tau :: \mathtt{Ty}}{\Gamma \vdash_\Sigma \mathtt{loc}\, X{\downarrow}u \leftarrow E\,;\, m \mathrel{\dot{\sim}} \tau}$$

Figure 11: Expression and Computation Statics

$$\frac{}{[\,c\,]\ \mathsf{val}_\Sigma}\ \textsc{mod-val-con} \qquad \frac{e\ \mathsf{val}_\Sigma}{[\,e\,]\ \mathsf{val}_\Sigma}\ \textsc{mod-val-exp} \qquad \frac{}{M :> \sigma\ \mathsf{val}_\Sigma}\ \textsc{mod-val-comp}$$

$$\frac{E_1\ \mathsf{val}_\Sigma \qquad E_2\ \mathsf{val}_\Sigma}{\langle E_1\,;E_2\rangle\ \mathsf{val}_\Sigma}\ \textsc{mod−val-hier} \qquad \frac{}{\lambda X{\downarrow}u:\sigma_1.E\ \mathsf{val}_\Sigma}\ \textsc{mod−val-fam}$$

$$\frac{E \underset{\Sigma}{\longmapsto} E'}{E\cdot\mathsf{1} \underset{\Sigma}{\longmapsto} E'\cdot\mathsf{1}}\ \textsc{mod-step-fst} \qquad \frac{}{\langle E_1\,;E_2\rangle\cdot\mathsf{1} \underset{\Sigma}{\longmapsto} E_1}\ \textsc{mod-step-fst-hier} \qquad \frac{E \underset{\Sigma}{\longmapsto} E'}{E\cdot\mathsf{2} \underset{\Sigma}{\longmapsto} E'\cdot\mathsf{2}}\ \textsc{mod-step-snd} \qquad \frac{}{\langle E_1\,;E_2\rangle\cdot\mathsf{2} \underset{\Sigma}{\longmapsto} E_2}\ \textsc{mod-step-snd-hier}$$

$$\frac{E \underset{\Sigma}{\longmapsto} E'}{E(\,E_1\,) \underset{\Sigma}{\longmapsto} E'(\,E_1\,)}\ \textsc{mod-step-inst} \qquad \frac{E\ \mathsf{val}_\Sigma \qquad E_1 \underset{\Sigma}{\longmapsto} E_1'}{E(\,E_1\,) \underset{\Sigma}{\longmapsto} E(\,E_1'\,)}\ \textsc{mod-step-inst-arg} \qquad \frac{E_1\ \mathsf{val}_\Sigma \qquad \mathtt{con}(\,E_1\,) \triangleq c_1}{(\,\lambda X{\downarrow}u:\sigma_1.E\,)(\,E_1\,) \underset{\Sigma}{\longmapsto} [E_1/X][c_1/u]E}\ \textsc{mod-step-inst-fam}$$

Figure 12: Module Expression Dynamics

$$\textsc{mod-initial}\quad \frac{\mu_0 : \Sigma_0}{\nu\,\Sigma_0\{\mu_0 \parallel M\}\ \mathsf{initial}} \qquad\qquad \textsc{mod–final}\quad \frac{E\ \mathsf{val}_\Sigma}{\nu\,\Sigma\{\mu \parallel \mathtt{ret}\,E\}\ \mathsf{final}}$$

$$\textsc{mod-step-ret}\quad \frac{E \underset{\Sigma}{\longmapsto} E'}{\nu\,\Sigma\{\mu \parallel \mathtt{ret}\,E\} \longmapsto \nu\,\Sigma\{\mu \parallel \mathtt{ret}\,E'\}}$$

$$\textsc{mod-step-bnd}\quad \frac{E_1 \underset{\Sigma}{\longmapsto} E_1'}{\nu\,\Sigma\{\mu \parallel \mathtt{bnd}\,X{\downarrow}u \leftarrow E_1\,;M_2\} \longmapsto \nu\,\Sigma\{\mu \parallel \mathtt{bnd}\,X{\downarrow}u \leftarrow E_1'\,;M_2\}}$$

$$\textsc{mod-step-bnd-ret}\quad \frac{E_1\ \mathsf{val}_\Sigma \qquad \mathtt{con}(\,E_1\,) \triangleq c_1}{\nu\,\Sigma\{\mu \parallel \mathtt{bnd}\,X{\downarrow}u \leftarrow (\mathtt{ret}\,E_1) :> \sigma_1\,;M_2\} \longmapsto \nu\,\Sigma'\{\mu' \parallel [E_1/X][c_1/u]M_2\}}$$

$$\textsc{mod-step-bnd-step}\quad \frac{\nu\,\Sigma\{\mu \parallel M_1\} \longmapsto \nu\,\Sigma'\{\mu' \parallel M_1'\}}{\nu\,\Sigma\{\mu \parallel \mathtt{bnd}\,X{\downarrow}u \leftarrow M_1 :> \sigma_1\,;M_2\} \longmapsto \nu\,\Sigma'\{\mu' \parallel \mathtt{bnd}\,X{\downarrow}u \leftarrow M_1' :> \sigma_1\,;M_2\}}$$

$$\textsc{mod-step-use}\quad \frac{e \underset{\Sigma}{\longmapsto} e'}{\nu\,\Sigma\{\mu \parallel \mathtt{use}(\,e\,)\} \longmapsto \nu\,\Sigma\{\mu \parallel \mathtt{use}(\,e'\,)\}} \qquad \textsc{mod-step-use-step}\quad \frac{}{\nu\,\Sigma\{\mu \parallel \mathtt{use}(\,M :> \sigma\,)\} \longmapsto \nu\,\Sigma\{\mu \parallel M\}}$$

Figure 13: Module Computation Dynamics