# 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 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 values* from *module computations*, which is achieved using a modality similar to that used for PCF. The key property of a module value 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 value using a technical device called *variable twinning*.

2. Expression values include the extraction of the value part of a module given by a module path, a composition of projections from a module variable. Expression computations include the extraction of the dynamic part of a module value.

3. Module expressions are separated into two categories, module values and module computations. 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. The static part of a module value is statically well-defined, but module computations need not have a well-defined static part.

5. There are two forms of atomic module value, a static module consisting of a constructor of a kind, and a dynamic module consisting of a value 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.
Typing contexts are defined by the following grammar:

\[ \Gamma ::= \bullet | \Gamma, u :: \kappa | \Gamma, x : \tau | \Gamma, u/X : \sigma \]

Declarations of constructor variables of a specified kind, and value variables of a specified type, are consolidated “twinned” declarations of a constructor variable with a module variable of a specified signature. Ordinary and module variables range over values, not computations. There is no loss of generality, because of the computation modality, which is anyway critical to managing the interplay between abstraction and effects. Contexts that bind only constructor variables are often denoted by \( \Delta \).

In a given context, \( \Gamma \), every module value, \( V \), of signature, \( \sigma \), has a statically defined static part, \( \text{st}(V) \), which is a constructor of kind \( \text{kd}(\sigma) \). These are both defined by the equations in Figure 2. For the sake of notational simplicity, the context is left implicit, and is extended parenthetically in the style of natural deduction.

**Lemma 2.1** (Phase Separation).

1. If \( \Gamma \vdash \text{ctx} \), then there exists unique \( \Delta \) such that \( \text{kd}(\Gamma) \triangleq \Delta \) and \( \Delta \vdash \text{ctx} \).
2. If \( \Delta \vdash \sigma \vdash \text{sig} \), then there exists unique \( \kappa \) such that \( \text{kd}(\sigma) \triangleq \kappa \) and \( \Delta \vdash \kappa \vdash \text{kind} \).
3. If \( \Gamma \vdash V : \sigma \), with \( \text{kd}(\Gamma) = \Delta \) and \( \text{kd}(\sigma) = \kappa \), then there exists unique \( c \) such that \( \text{st}(V) \triangleq c \) and \( \Delta \vdash c :: \kappa \).

For the sake of brevity \( \text{kd}(\Gamma) \) is abbreviated by \( |\Gamma| \), \( \text{kd}(\sigma) \) is abbreviated by \( |\sigma| \), and \( \text{st}(V) \) is abbreviated by \( |V| \), whenever the translations are well-defined in the sense of Lemma 2.1.

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

1. If \( \Gamma \vdash V : \sigma \), then \( \Gamma \vdash V : \text{S}_\sigma(V) \).
2. If \( \Gamma \vdash V : \sigma \) and \( |\Gamma| \vdash \sigma < : \sigma' \), then \( \Gamma \vdash V : \sigma' \).
3. If \( \Gamma \vdash M : \sigma \) and \( |\Gamma| \vdash \sigma < : \sigma' \), then \( \Gamma \vdash M : \sigma' \).

Substitution is defined consistently with the intended interpretation of variables. Any constructor of the kind of a variable may be substituted into any other phrase. An expression value may be substituted for an expression variable in any expression or module. A module value, and its static part, may be simultaneously substituted for a twinned constructor variable and module variable in a module or expression. The forms of substitution, and their action on generic forms \( \varepsilon \), are defined in Figure 10.

**Lemma 2.3** (Substitution). Suppose that \( \Gamma' \vdash \gamma : \Gamma \).

1. If \( |\Gamma| \vdash \kappa \vdash \text{kind} \), then \( |\Gamma'| \vdash \gamma(\kappa) \vdash \text{kind} \), and if \( |\Gamma| \vdash \kappa < : \kappa' \), then \( |\Gamma'| \vdash \gamma(\kappa) < : \gamma(\kappa') \).
2. If \( |\Gamma| \vdash \kappa :: \kappa \), then \( |\Gamma'| \vdash \gamma(\kappa) :: \gamma(\kappa) \).
3. If \( |\Gamma| \vdash \sigma \vdash \text{sig} \), then \( |\Gamma'| \vdash \gamma(\sigma) \vdash \text{sig} \), and if \( |\Gamma| \vdash \sigma < : \sigma' \), then \( |\Gamma'| \vdash \gamma(\sigma) < : \gamma(\sigma') \).
4. If \( \Gamma \vdash v : \tau \), then \( \Gamma' \vdash \gamma(v) : \gamma(\tau) \).
5. If \( \Gamma \vdash e \triangleright \tau \), then \( \Gamma' \vdash \gamma(e) \triangleright \gamma(\tau) \).
6. If \( \Gamma \vdash V : \sigma \), then \( \Gamma' \vdash \gamma(V) : \gamma(\sigma) \).
7. If \( \Gamma \vdash M \triangleright \sigma \), then \( \Gamma' \vdash \gamma(M) \triangleright \gamma(\sigma) \).
\[
\begin{align*}
\text{kd}(\kappa) & \triangleq \kappa \\
\text{kd}(\tau) & \triangleq 1 \\
\text{kd}(\sigma) & \triangleq 1 \\
\text{kd}(u :: \sigma_1 \times \sigma_2) & \triangleq u :: \kappa_1 \times \kappa_2 \quad \text{if } \text{kd}(\sigma_1) \triangleq \kappa_1 \text{ and } \text{kd}(\sigma_2) \triangleq \kappa_2 \quad (u :: \kappa_1) \\
\text{kd}(u :: \sigma_1 \rightarrow \sigma_2) & \triangleq u :: \kappa_1 \rightarrow \kappa_2 \quad \text{if } \text{kd}(\sigma_1) \triangleq \kappa_1 \text{ and } \text{kd}(\sigma_2) \triangleq \kappa_2 \quad (u :: \kappa_2)
\end{align*}
\]

\[
\begin{align*}
st(X) & \triangleq u \quad \text{if } u/X : \sigma \in \Gamma \\
st(P \cdot 1) & \triangleq c \cdot 1 \quad \text{if } st(P) \triangleq c \\
st(P \cdot r) & \triangleq c \cdot 1 \quad \text{if } st(P) \triangleq c \\
st(P(V_1)) & \triangleq c[c_1] \quad \text{if } st(P) \triangleq c \text{ and } st(V_1) \triangleq c_1 \\
st([c]) & \triangleq c \\
st([v]) & \triangleq \emptyset \\
st(M :> \sigma) & \triangleq \emptyset \\
st(V_1 ; V_2) & \triangleq \langle c_1, c_2 \rangle \quad \text{if } st(V_1) \triangleq c_1 \text{ and } st(V_2) \triangleq c_2 \\
st(\lambda u/X : \sigma_1, V_2) & \triangleq \lambda (u :: \kappa_1) c_2 \quad \text{if } \text{kd}(\sigma_1) \triangleq \kappa_1 \text{ and } st(V_2) \triangleq c_2 \quad (u/X : \sigma_1)
\end{align*}
\]

Figure 2: Static Part of Signatures, Module Values, and Contexts

\[
\begin{align*}
\text{SIG-CON} & \quad \text{SIG-VAL} & \quad \text{SIG-COMP} \\
\Delta \vdash \kappa & \quad \Delta \vdash \tau :: Ty & \quad \Delta \vdash \sigma & \quad \Delta \vdash \sigma & \quad \Delta \vdash \sigma & \quad \Delta \vdash \sigma \\
\Delta \vdash \kappa & \quad \Delta \vdash \tau & \quad \Delta \vdash \sigma & \quad \Delta \vdash \sigma
\end{align*}
\]

\[
\begin{align*}
\text{SIG-SUB} & \quad \text{SIG-FUN} \\
\Delta \vdash \sigma_1 & \quad \Delta, u :: \sigma_1 \vdash \sigma_2 & \quad \Delta \vdash \sigma_1 & \quad \Delta, u :: \sigma_1 \vdash \sigma_2 & \quad \Gamma, u :: \sigma_1 \vdash \sigma_2 & \quad \Delta \vdash \sigma_1, V_2
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash u :: \sigma_1 \times \sigma_2 & \quad \Delta \vdash u :: \sigma_1 \rightarrow \sigma_2 & \quad \Delta \vdash u :: \sigma_1 \rightarrow \sigma_2
\end{align*}
\]

Figure 3: Signature Formation: \( \Delta \vdash \sigma \)

\[
\begin{align*}
\text{SUBSIG-CON} & \quad \text{SUBSIG-VAL} & \quad \text{SUBSIG-COMP} \\
\Delta \vdash \kappa & \quad \Delta \vdash \tau & \quad \Delta \vdash \sigma & \quad \Delta \vdash \kappa & \quad \Delta \vdash \tau & \quad \Delta \vdash \sigma
\end{align*}
\]

\[
\begin{align*}
\text{SUBSIG-SUB} & \quad \text{SUBSIG-FUN} \\
\Delta \vdash \sigma_1 & \quad \Delta, u :: \sigma_1 \vdash \sigma_2 & \quad \Delta \vdash \sigma_1 & \quad \Delta, u :: \sigma_1 \vdash \sigma_2 & \quad \Delta \vdash \sigma_1 & \quad \Delta, u :: \sigma_1 \vdash \sigma_2
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash \sigma_1 \times \sigma_2 & \quad \Delta \vdash \sigma_1 \rightarrow \sigma_2 & \quad \Delta \vdash \sigma_1 \rightarrow \sigma_2
\end{align*}
\]

Figure 4: Subsignatures: \( \Delta \vdash \sigma \)
\[ S_{[\kappa]}(V) \triangleq [S_{\kappa}(\text{st}(V))] \]
\[ S_{[\tau]}(V) \triangleq [\tau] \]
\[ S_{\{\sigma\}}(V) \triangleq \{\sigma\} \]
\[ S_{u::\sigma_1 \times \sigma_2}(V) \triangleq \begin{cases} 
  u :: S_{\sigma_1}(V_1) \times S_{\text{st}(V_1)/u|\sigma_2}(V_2) & \text{if } V = \langle V_1 ; V_2 \rangle \\
  u :: S_{\sigma_1}(P \cdot 1) \times S_{\text{st}(P \cdot 1)/u|\sigma_2}(P \cdot \tau) & \text{if } V = P 
\end{cases} \]
\[ S_{u::\sigma_1 \rightarrow \sigma_2}(V) \triangleq \begin{cases} 
  \lambda u/X : \sigma_1.S_{\sigma_2}(V_2) & \text{if } V = \lambda u/X : \sigma_1.V_2 \\
  \lambda u/X : \sigma_1.S_{\sigma_2}(P(X)) & \text{if } V = P 
\end{cases} \]

Figure 5: Singleton Signatures: \( S_{\sigma}(V) \triangleq \sigma' \)

\[
\begin{align*}
\text{PATH-VAR} & \\
|\Gamma| \vdash \sigma <: \sigma' & \quad \text{PATH-FST} & \quad \text{PATH-SND} \\
\Gamma, X : \sigma \vdash X : S_{\sigma'}(X) & & \Gamma \vdash P : u :: \sigma_1 \times \sigma_2 \\
& & \Gamma \vdash P \cdot 1 : \sigma_1 \\
& & \Gamma \vdash P \cdot \tau : [\text{st}(P \cdot 1)/u|\sigma_2] \\
\text{PATH-INST} & \\
\Gamma \vdash P : u :: \sigma_1 \rightarrow \sigma_2 & \quad \Gamma \vdash V_1 : \sigma_1 \\
\Gamma \vdash P(V_1) : [\text{st}(V_1)/u|\sigma_2] & \\
\end{align*}
\]

Figure 6: Module Paths: \( \Gamma \vdash P : \sigma \)

\[
\begin{align*}
\text{MOD-CON} & \\
|\Gamma| \vdash c :: \kappa & \quad \text{MOD-VAL} & \quad \text{MOD-COMP} \\
\Gamma \vdash [c] : [S_{\kappa}(c)] & & \Gamma \vdash v : [\tau] \\
& & \Gamma \vdash M \bowtie \sigma \vdash [\sigma <: \sigma'] \\
\text{MOD-SUB} & \\
\Gamma \vdash V_1 : \sigma_1 & \quad \Gamma \vdash V_2 : [\text{st}(V_1)/u|\sigma_2] \\
\Gamma \vdash \langle V_1 ; V_2 \rangle : u :: \sigma_1 \times \sigma_2 & \\
\text{MOD-FUN} & \\
\Gamma, u/X : \sigma_1 \vdash V_2 : \sigma_2 & \quad \Gamma \vdash \lambda u/X : \sigma_1.V_2 : u :: \sigma_1 \rightarrow \sigma_2 \\
\end{align*}
\]

Figure 7: Module Values: \( \Gamma \vdash V : \sigma \)
Figure 8: Module Computations: \( \Gamma \vdash M \sim \sigma \)

INITIAL

\[ \nu \Sigma_0 \{ \mu_0 \| M \} \text{ initial} \]

FINAL

\[ \nu \Sigma \{ \mu \| \text{ret}(V) \} \text{ final} \]

STEP-BND-DONE

\[ \nu \Sigma \{ \mu \| \text{bnd } u/X \leftarrow \text{ret}(V_1) ;> \sigma_1 ; M_2 : \sigma_2 \} \mapsto \nu \Sigma' \{ \mu' \| [V_1/u/X]M_2 \} \]

STEP-BND-STEP

\[ \nu \Sigma \{ \mu \| M_1 \} \mapsto \nu \Sigma' \{ \mu' \| M'_1 \} \]

\[ \nu \Sigma \{ \mu \| \text{bnd } u/X \leftarrow M_1 ;> \sigma_1 ; M_2 : \sigma_2 \} \mapsto \nu \Sigma' \{ \mu' \| \text{bnd } u/X \leftarrow M'_1 ;> \sigma_1 ; M_2 : \sigma_2 \} \]

STEP-FST

\[ \nu \Sigma \{ \mu \| (V_1 \cdot V_2) \cdot 1 \} \mapsto \nu \Sigma \{ \mu \| \text{ret}(V_1) \} \]

STEP-INST

\[ \nu \Sigma \{ \mu \| (\lambda u/X : \sigma_1.V)(V_1) \} \mapsto \nu \Sigma \{ \mu \| [V_1/u/X]V \} \]

STEP-INST-DONE

\[ \nu \Sigma \{ \mu \| \text{ini } x \leftarrow \text{comp} \{ \text{ret}(v) \}; M \} \mapsto \nu \Sigma \{ \mu \| [v/x]M \} \]

STEP-INST-STEP

\[ \nu \Sigma \{ \mu \| e \} \mapsto \nu \Sigma' \{ \mu' \| e' \} \]

\[ \nu \Sigma \{ \mu \| \text{ini } x \leftarrow \text{comp} \{ e \}; M \} \mapsto \nu \Sigma' \{ \mu' \| \text{ini } x \leftarrow \text{comp} \{ e' \}; M \} \]

Figure 9: Module Dynamics: \( \nu \Sigma \{ \mu \| M \} \mapsto \nu \Sigma' \{ \mu' \| M' \} \)
\[
\begin{align*}
\text{SUBST-EMP} & \quad \emptyset : \cdot \\
\text{SUBST-CON} & \quad \dfrac{|\Gamma'| \vdash c :: \hat{\gamma} \langle \kappa \rangle \quad \Gamma' \vdash \gamma : \Gamma}{\Gamma' \vdash \gamma \otimes \{u \leftrightarrow c\} : \Gamma, u :: \kappa} \\
\text{SUBST-VAL} & \quad \dfrac{\Gamma' \vdash v :: \hat{\gamma} \langle \tau \rangle \quad \Gamma' \vdash \gamma : \Gamma}{\Gamma' \vdash \gamma \otimes \{x \leftrightarrow v\} : \Gamma, x :: \tau} \\
\text{SUBST-Val} & \quad \dfrac{\Gamma' \vdash V :: \hat{\gamma} \langle \sigma \rangle \quad \Gamma' \vdash \gamma : \Gamma}{\Gamma' \vdash \gamma \otimes \{X \leftrightarrow V\} : \Gamma, u/X :: \sigma}
\end{align*}
\]

\[\widehat{\emptyset}(\varepsilon) \triangleq \varepsilon\]
\[\gamma \otimes \{u \leftrightarrow c\}(\varepsilon) \triangleq [c/u]\varepsilon\]
\[\gamma \otimes \{x \leftrightarrow v\}(\varepsilon) \triangleq [v/x]\varepsilon\]
\[\gamma \otimes \{X \leftrightarrow V\}(\varepsilon) \triangleq [[V], V/u, X]\varepsilon\]

Figure 10: Substitution

References
