A metalanguage for multi-phase modularity

JONATHAN STERLING and ROBERT HARPER, Carnegie Mellon University, USA

Type abstraction, the phase distinction, and computational effects all play an important role in the design and implementation of ML-style module systems. We propose a simple type theoretic metalanguage φML for multi-phase modularity in which these concepts are treated individually, supporting the definition of high-level modular constructs such as generative and applicative functors, as well as all extant forms of structure sharing.

In most accounts of ML modules, the phase distinction between static code and dynamic code is enforced pervasively throughout the language [12, 18]; for instance, in a functor signature of the form \((x : A) \to B(x)\), the signature \(B(x)\) is only allowed to depend on the “static part” of \(x : A\). The purpose of this restriction is to ensure that the judgmental equality of types and other static constructs can be decided independently of the existence of any notion of equality for programs.

Recently several authors have advanced a monadic presentation of ML modules in which both generativity and other effects are treated using a lax modality \(\lmod\) on signatures [5, 10, 24]. When effects are treated monadically, there is however no obstacle to formulating a (conservative and tractable) notion of judgmental equality for programs, hence it is appropriate to revisit the global restriction that types shall never depend on runtime code.

1 THE NEED FOR VALUE-DEPENDENCY

In order to preserve abstraction, it is often necessary for types to depend on runtime identity; generativity of ML functors is one way to achieve this in the context of effects, but the need for this kind of dependency also occurs even for applicative functors such as MkSet, as pointed out by Rossberg et al. [22]. This shows that one needs to depend on runtime value identity to achieve abstraction regardless of whether computational effects are in play; generative functors capture specifically the case where modules (potentially) exhibit dynamic initialization effects.

Static dependency on runtime identity can be approximated using phantom types as in the elaboration of Rossberg et al. [22, § 8.1], a logical version of the stamps of SML ’90 [16]. While phantom types have a definite role to play, providing the most conservative possible static approximation of value identity, experience implementing and compiling full-spectrum dependently typed programming languages (e.g. Idris 2 and Lean 4 [4, 6]) suggests that there is no longer any reason to make this the only way that types can depend on values.

2 LET A HUNDRED PHASE DISTINCTIONS BLOOM!

The venerable static–dynamic phase distinction is not the only phase distinction that can be considered. For instance, logical relations arguments can be reformulated à la Sterling and Harper [24] in terms of a syntactic–semantic phase distinction; type refinements in the sense of Melliès and Zeilberger [15] evince a phase distinction between computation (extraction) and logic (specification); security typing and information flow can be seen to exhibit a lattice of phase distinctions.

Because these are surely not the only phase distinctions that will play a role in future programming languages, we propose an adequate type theoretic metalanguage φML that can accommodate any number of phase distinctions simultaneously. φML starts with ordinary Martin-Löf type theory [19] and adds to it enough constructs to express modularity relative to a lattice of phase distinctions, denoted \(\varphi : \Omega\).

Authors’ address: Jonathan Sterling, jmsterli@cs.cmu.edu; Robert Harper, rwh@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.
Each phase $\phi : \emptyset$ induces a context extension $(\Gamma, \varphi)$; types and terms in such a context are restricted to their $\varphi$-visible components. For instance if $\varphi := \phi_{\text{st}}$ is the static phase, the dynamic parts of a type $\Gamma, \phi_{\text{st}} \vdash A$ type are collapsed. In this sense, the weakening substitution along $\Gamma, \phi_{\text{st}} \rightarrow \Gamma$ implements the static projection operation $\text{Fst}(\cdot)$ from prior type theoretic accounts of module languages [7], and judgmental equality $\Gamma, \phi_{\text{st}} \vdash A \equiv B$ type in the extended context reconstructs the static equivalence judgment of Dreyer et al. [8].

### 2.1 Modal type structure of $\varphi$ML

#### 2.1.1 The phase modality

The context extension $\Gamma, \varphi$ is internalized as the phase modality $(\varphi \Rightarrow \emptyset)$; semantically, $(\varphi \Rightarrow \emptyset)$ behaves like a function space whose domain is the (subsingleton) collection of witnesses that we are “in” the $\varphi$ phase.

#### 2.1.2 The sealing modality

A type $A$ is called sealed at $\varphi : \emptyset$, written $\Gamma \vdash A$ sealed @ $\varphi$, when it is equivalent to the unit type in phase $\varphi$. We include a sealing modality $[\varphi \setminus A]$ that seals a type $A$ at phase $\varphi$; the laws for this modality are similar to those of the protection modality in the dependency core calculus of Abadi et al. [1], but they actually come from those of the local operator induced by a closed subspace in the topological semantics of intuitionistic logic. Indeed, the relationship between the phase and sealing modalities is essentially that of open subspace (e.g. static fragment) and closed complement (e.g. dynamic fragment).

#### 2.1.3 Structure sharing

Given a type $A$ and an element $\varphi : M : A$, we may form the structure sharing type $\{A \mid \varphi \mapsto M\} \subseteq A$ that classifies all the elements of $A$ equal to $M$ at phase $\varphi$. In case $\varphi := \phi_{\text{st}}$, the structure sharing type $\{A \mid \text{\phi_{st} \mapsto M}\} \subseteq A$ classifies the elements of $A$ that are statically equivalent to $M$ in the sense of Dreyer et al. [8] and therefore captures the weak structure sharing of SML ’97 [17]. On the other hand, if $\varphi := T$ is the “top” phase distinction, then $\{A \mid T \mapsto M\}$ is the true singleton type that is approximated by SML ’90’s strong structure sharing [16] via stamps, and by the F-ing Modules calculi via phantom types [22].

### 2.2 Applications of $\varphi$ML

We briefly survey a few applications of $\varphi$ML’s perspective on multi-phase modularity.

#### 2.2.1 Reconstructing ML’s static–dynamic phase distinction

The classic static–dynamic phase distinction of SML and OCaml is recovered by adding a single phase distinction $\phi_{\text{st}} : \emptyset$ together with a lax modality $\bigcirc A$ for effects that is always statically sealed, in the sense that $\Gamma \vdash \bigcirc A$ sealed @ $\phi_{\text{st}}$ holds. Given another modality $T$ that is not sealed, one could define the effect modality by $\bigcirc A := [\phi_{\text{st}} \setminus T(A)]$ in terms of the sealing modality. ML-style generative and applicative functors may then be defined like so:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi^\text{gen}$</td>
<td>$(A : \text{Sig}) (B : (\phi_{\text{st}} \Rightarrow A) \rightarrow \text{Sig}) \rightarrow \text{Sig}$</td>
</tr>
<tr>
<td>$\Pi^\text{app}$</td>
<td>$(A, B) = (x : A) \mapsto \bigcirc B((\phi_{\text{st}})x)$</td>
</tr>
<tr>
<td>$\Pi^\text{app}$</td>
<td>$(A, B) = (x : A) \mapsto B((\phi_{\text{st}})x)$</td>
</tr>
</tbody>
</table>

We add a law to make the universe of kinds purely static in the sense that $(\phi_{\text{st}} \Rightarrow \text{Kind}) \equiv \text{Kind}$.

#### 2.2.2 Compile-time inlining without breaking abstraction

Under a separate compilation discipline, e.g. that of Swasey et al. [28], a module is compiled as a function of its dependencies; unless special arrangements are made, this can obstruct the inlining of functions whose identities are not exposed by the dependencies’ interfaces. To address the inlining problem, Stone [25, § 1.5.3] and Leroy [13, § 5.3] have suggested extending the module language to support sharing of non-static phrases in module signatures; then this interface can be used by the compiler to support inlining of the
exposed definitions. This is too naïve: users of module systems employ non-sharing in order to maintain abstraction and enforce their intention that a dependent module’s implementation is independent of some part of its dependency.

We propose to address the inlining problem by introducing a phase distinction $\phi_{\text{cmpl}} : \emptyset$ between compile-time and runtime.\(^1\) Value identities are exposed for inlining by means of the the structure sharing type $\{A \mid \phi_{\text{cmpl}} \rightarrow M\}$; programmers will not be able to rely on the identities so-exposed, but the compiler will be executed in the $\phi_{\text{cmpl}}$ phase and can therefore exploit exposed identities for inlining. This application provides essential theoretical support for the efficient implementation of Harper’s proposal to treat datatypes as abstract types with default implementations [9].

2.2.3 Reconciling debugging with abstraction. Debugging is a common source of frustration when developing code in the presence of abstract types; many engineers today still primarily rely on so-called “printf-debugging” to diagnose broken code, but this becomes a problem in the presence of abstract types whose representations are unknown. We propose to add a new “debug” phase $\phi_{\text{dbg}} : \emptyset$ and, by default, expose the identities of all modules within the debug phase by means of the structure sharing type $\{A \mid \phi_{\text{dbg}} \rightarrow M\}$; then we may add a primitive operation to the standard basis library that allows a $\phi_{\text{dbg}}$-phase string to be printed, $\text{debug} : (\phi_{\text{dbg}} \Rightarrow \text{string}) \rightarrow \text{unit}$. Then in the presence of an element $a : M.t$ whose (hidden) representation type is int, we may freely debug by executing the side effect $\text{debug}(\langle \phi_{\text{dbg}} \rangle \text{Int.toString}(a))$.

2.2.4 Representation independence. Following the Logical Relations As Types principle of Sterling and Harper [24], we may capture binary parametricity [20] by adding two phases $\phi^L_{\text{syn}}, \phi^R_{\text{syn}} : \emptyset$ with $\phi^L_{\text{syn}} \cap \phi^R_{\text{syn}} \equiv \bot$ and defining $\phi_{\text{syn}} := \phi^L_{\text{syn}} \cup \phi^R_{\text{syn}}$. Then representation independence results can be proved: a simulation between queue implementations $M, N : \text{QUEUE}$ is given by a third implementation $O : \{\text{QUEUE} \mid \phi_{\text{syn}} \rightarrow [\phi^L_{\text{syn}} \rightarrow M, \phi^R_{\text{syn}} \rightarrow N]\}$. This method is used by op. cit. to prove a generalized Reynolds Abstraction Theorem for a module calculus, and by Sterling and Angiuli [23] to prove normalization and decidability of judgmental equality for cubical type theory.

REFERENCES


\(^1\)Here compile-time refers to a stage subsequent to typechecking/elaboration, and is therefore semantically different from a static phase.


A SELECTED RULES

A.1 Judgments of $\phi_{\text{ML}}$

We specify $\phi_{\text{ML}}$ parametrically in a meet semilattice $\mathcal{O}$ of phases, writing $[\varphi : \mathcal{O}]$ to mean that $\varphi$ is an element of $\mathcal{O}$. We begin by recapitulating the ordinary judgments of type theory:

1. $\Gamma \text{ctx}$ means that $\Gamma$ is a context.
2. $\Gamma \vdash A \text{ type}$ presupposes $\Gamma \text{ctx}$ and means that $A$ is a type in context $\Gamma$.
3. $\Gamma \vdash A \equiv B \text{ type}$ presupposes $\Gamma \text{ctx}$ and $\Gamma \vdash A, B \text{ type}$, and means that $A$ and $B$ are equal types in context $\Gamma$.

To the above, $\phi_{\text{ML}}$ adds the following forms of judgment that pertain to the structure of phases:

1. $\Gamma \vdash \varphi$ presupposes $\Gamma \text{ctx}$ and $\varphi : \mathcal{O}$, and means that $\Gamma$ entails that the $\varphi$ phase is activated.
2. $\Gamma \vdash A \text{ sealed } @ \varphi$ presupposes $\Gamma \text{ctx}$, $\Gamma \vdash A \text{ type}$, and $\varphi : \mathcal{O}$, and means that $\Gamma$ entails that $A$ is sealed at phase $\varphi$. Intuitively this means that the type $A$ can expose no information to clients at phase $\varphi$, i.e. is a singleton type at phase $\varphi$.

Contexts in $\phi_{\text{ML}}$ are totally structural; all the judgments of $\phi_{\text{ML}}$ specified above are stable under weakening, contraction, and exchange.

A.2 Contexts and phases

To activate a given phase, $\phi_{\text{ML}}$ has a context extension $\Gamma, \varphi$ governed by the following rules:

\[
\begin{array}{cccc}
\text{cx/emp} & \text{cx/var} & \text{cx/ph} & \text{ph/var} \\
\Gamma \text{ctx} & \varphi : \mathcal{O} & \Gamma \text{ctx} & \Gamma \vdash A \text{ type} & \varphi \in \Gamma \\
\cdot \text{ctx} & \Gamma, \varphi \text{ctx} & \Gamma, x : A \text{ctx} & \Gamma \vdash \varphi \\
\end{array}
\]

We impose rules to make the judgment $\Gamma \vdash \varphi$ preserve meets:

\[
\begin{array}{c}
\text{top/intro} \\
\Gamma \vdash \top \\
\end{array}
\]

\[
\begin{array}{c}
\text{meet/intro} \\
\Gamma \vdash \varphi & \Gamma \vdash \psi \\
\Gamma \vdash \varphi \land \psi \\
\end{array}
\]

\[
\begin{array}{c}
\text{meet/elim} \\
\Gamma \vdash \varphi \land \psi \\
\Gamma \vdash \varphi & \Gamma \vdash \psi \\
\end{array}
\]

Observation A.1. The following monotonicity law is derivable:

\[
\begin{array}{c}
\text{ph/mod/mono} \\
\varphi \leq \psi \\
\Gamma \vdash \varphi \\
\Gamma \vdash \psi \\
\end{array}
\]

\begin{proof}
If $\varphi \leq \psi$, then $\varphi \land \psi = \psi$; to derive $\Gamma \vdash \psi$ we therefore may apply meet/elim.
\end{proof}

A.3 The phase modality

We include a modality $\varphi \Rightarrow A$ that governs programs that can be written at phase $\varphi$; semantically, this phase modality is just a dependent function space over the (subsingleton) collection of witnesses that the phase $\varphi$ is active.

\[
\begin{array}{c}
\text{ph/mod/formation} \\
\varphi : \mathcal{O} & \Gamma, \varphi \vdash A \text{ type} \\
\Gamma \vdash \varphi \Rightarrow A \text{ type} \\
\end{array}
\]

\[
\begin{array}{c}
\text{ph/mod/intro} \\
\Gamma, \varphi \vdash M : A \\
\Gamma \vdash (\varphi)M : \varphi \Rightarrow A \\
\end{array}
\]

\[
\begin{array}{c}
\text{ph/mod/elim} \\
\Gamma \vdash M : \varphi \Rightarrow A \\
\Gamma \vdash M @ \varphi : A \\
\end{array}
\]

\[
\begin{array}{c}
\text{ph/mod/beta} \\
\Gamma, \varphi \vdash M : A \\
\Gamma \vdash \varphi \\
\Gamma \vdash (\varphi)M @ \varphi \equiv M : A \\
\end{array}
\]

\[
\begin{array}{c}
\text{ph/mod/eta} \\
\Gamma \vdash (\varphi)(M @ \varphi) \equiv M : \varphi \Rightarrow A \\
\end{array}
\]
A.4 Structure sharing

To model structure sharing from ML languages, $\phi_{\text{ML}}$ includes a connective $\{A \mid \varphi \leftrightarrow M\}$ that classifies the elements of type $A$ that are equal to $M$ in phase $\varphi$.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sh/formation}$</td>
<td>$\varphi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma, \varphi \vdash M : A \quad \Gamma \vdash {A \mid \varphi \leftrightarrow M} \text{ type} $</td>
</tr>
<tr>
<td>$\text{sh/intro}$</td>
<td>$\Gamma \vdash M : A \quad \Gamma, \varphi \vdash M \equiv N : A \quad \Gamma \vdash {A \mid \varphi \leftrightarrow N} $</td>
</tr>
<tr>
<td>$\text{sh/elim}$</td>
<td>$\Gamma \vdash {A \mid \varphi \leftrightarrow N} \quad \Gamma \vdash \lfloor M \rfloor : {A \mid \varphi \leftrightarrow N} $</td>
</tr>
<tr>
<td>$\text{sh/elim/bdry}$</td>
<td>$\Gamma \vdash M : {A \mid \varphi \leftrightarrow N} \quad \Gamma, \varphi \vdash N : A \quad \Gamma \vdash \lceil M \rceil : {A \mid \varphi \leftrightarrow N} \equiv M : A $</td>
</tr>
<tr>
<td>$\text{sh/beta}$</td>
<td>$\Gamma \vdash M : {A \mid \varphi \leftrightarrow N} \quad \Gamma \vdash \lfloor \lceil M \rceil \rfloor : {A \mid \varphi \leftrightarrow N} \equiv M : A $</td>
</tr>
<tr>
<td>$\text{sh/eta}$</td>
<td>$\Gamma \vdash M : {A \mid \varphi \leftrightarrow N} \quad \Gamma, \varphi \vdash N : A \quad \Gamma \vdash \lfloor \lceil M \rceil \rfloor : {A \mid \varphi \leftrightarrow N} \equiv M : A $</td>
</tr>
</tbody>
</table>

**Example A.2.** Using the top element of the phase lattice, the structure sharing connective can express singleton types [2, 3, 25–27]. In particular, given $\Gamma \vdash A \text{ type}$ and $\Gamma \vdash M : A$, we define $S_A(M) := \{A \mid \top \leftrightarrow M\}$.

A.5 Judgmental sealing

A type is sealed at phase $\varphi$ when it has exactly one element at that phase and hence can leak no information. This is expressed by the following rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sl/point}$</td>
<td>$\Gamma \vdash A \text{ sealed } @ \varphi \quad \Gamma \vdash \varphi \quad \Gamma \vdash \star_A : A $</td>
</tr>
<tr>
<td>$\text{sl/glue}$</td>
<td>$\Gamma \vdash A \text{ sealed } @ \varphi \quad \Gamma \vdash \varphi \quad \Gamma \vdash \Gamma \vdash M : A \quad \Gamma \vdash M \equiv \star_A : A $</td>
</tr>
</tbody>
</table>

Function types are sealed when their codomains are sealed; product types (including unit, the nullary product) are sealed when all their conjuncts are sealed; structure sharing types are sealed at the phase of their constraint:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{fun/sealed}$</td>
<td>$\Gamma \vdash A \text{ type} \quad \Gamma \vdash B \text{ sealed } @ \varphi \quad \Gamma \vdash A \rightarrow B \text{ sealed } @ \varphi $</td>
</tr>
<tr>
<td>$\text{unit/sealed}$</td>
<td>$\Gamma \vdash \text{ unit sealed } @ \varphi \quad \Gamma \vdash \Gamma \vdash A, B \text{ sealed } @ \varphi \quad \Gamma \vdash A \times B \text{ sealed } @ \varphi $</td>
</tr>
<tr>
<td>$\text{prod/sealed}$</td>
<td>$\Gamma \vdash \psi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma, \psi \vdash M : A \quad \Gamma, \varphi \vdash \psi \quad \Gamma \vdash {A \mid \psi \leftrightarrow M} \text{ sealed } @ \varphi $</td>
</tr>
<tr>
<td>$\text{sh/sealed/1}$</td>
<td>$\Gamma \vdash \psi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma, \psi \vdash M : A \quad \Gamma, \varphi \vdash \psi \quad \Gamma \vdash {A \mid \psi \leftrightarrow M} \text{ sealed } @ \varphi $</td>
</tr>
<tr>
<td>$\text{sh/sealed/2}$</td>
<td>$\Gamma \vdash \psi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma, \psi \vdash M : A \quad \Gamma \vdash A \text{ sealed } @ \varphi \quad \Gamma \vdash {A \mid \psi \leftrightarrow M} \text{ sealed } @ \varphi $</td>
</tr>
</tbody>
</table>

**Observation A.3.** The following rules are already derivable:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{product point}$</td>
<td>$\Gamma \vdash A, B \text{ sealed } @ \varphi \quad \Gamma \vdash \star_{A \times B} \equiv (\star_A, \star_B) : A \times B $</td>
</tr>
<tr>
<td>$\text{function point}$</td>
<td>$\Gamma \vdash A \text{ type} \quad \Gamma \vdash B \text{ sealed } @ \varphi \quad \Gamma \vdash \star_{A \rightarrow B} \equiv \lambda x : A. \star_B : A \rightarrow B $</td>
</tr>
</tbody>
</table>
A.6 The sealing modality

Not every type is sealed; for instance, the sum type $A + B$ is not sealed even if $A$ and $B$ are both sealed, because a single bit of information can be exposed by case analysis. To seal a non-sealed type, $\varphi_{\text{ML}}$ provides an idempotent modality $[\varphi \setminus A]$ governed by the following rules:

**SL/FORMATION**

\[ \varphi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma \vdash [\varphi \setminus A] \text{ type} \]

**SL/SEAL/1**

\[ \psi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma, \psi \vdash \varphi \quad \Gamma \vdash [\psi \setminus A] \text{ sealed } @ \varphi \]

**SL/SEAL/2**

\[ \psi : \emptyset \quad \Gamma \vdash A \text{ type} \quad \Gamma \vdash A \text{ sealed } @ \varphi \quad \Gamma \vdash [\psi \setminus A] \text{ sealed } @ \varphi \]

**SL/INTRO**

\[ \Gamma \vdash M : A \quad \Gamma \vdash \text{seal}_{\varphi}(M) : [\varphi \setminus A] \]

**SL/ELIM**

\[ \Gamma \vdash M : [\varphi \setminus A] \quad \Gamma \vdash B \text{ sealed } @ \varphi \quad \Gamma, x : A \vdash N(x) : B \]

\[ \Gamma \vdash x \leftarrow \text{unseal}_{\varphi}(M); N(x) : B \]

**SL/BETA**

\[ \Gamma \vdash M : A \quad \Gamma \vdash B \text{ sealed } @ \varphi \quad \Gamma, x : A \vdash N(x) : B \]

\[ \Gamma \vdash x \leftarrow \text{unseal}_{\varphi}(\text{seal}_{\varphi}(M)); N(x) \equiv N(M) : B \]

**SL/ETA**

\[ \Gamma \vdash M : [\varphi \setminus A] \quad \Gamma \vdash B \text{ sealed } @ \varphi \quad \Gamma, x : [\varphi \setminus A] \vdash N(x) : B \]

\[ \Gamma \vdash N(M) \equiv x \leftarrow \text{unseal}_{\varphi}(M); N(\text{seal}_{\varphi}(x)) : B \]