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 [10, 16]; 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 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* $\bigcirc$ on signatures [3, 8, 22]. 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. [20]. 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. [20, § 8.1], a logical version of the *stamps* of SML '90 [14]. 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 [2, 4]) 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 [22] in terms of a *syntactic–semantic* phase distinction; type refinements in the sense of Melliès and Zeilberger [13] 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 [17] and adds to it enough constructs to express modularity relative to a lattice of phase distinctions, denoted $\boxed{\varphi : \mathcal{O}}$. (The rules of **φML** are summarized in Fig. 1.)

Authors' address: Jonathan Sterling, jmsterli@cs.cmu.edu; Robert Harper, rwh@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

$$\boxed{\Gamma\ ctx} \qquad \boxed{\Gamma \vdash A\ type} \qquad \boxed{\Gamma \vdash A \equiv B\ type} \qquad \boxed{\Gamma \vdash M \equiv N : A} \qquad \boxed{\varphi : \mathbb{O}} \qquad \boxed{\Gamma \vdash A\ \mathsf{sealed}\ @\ \varphi}$$

$$\frac{\Gamma\ ctx \qquad \varphi : \mathbb{O}}{\Gamma, \varphi\ ctx} \qquad \frac{\Gamma, \varphi \vdash A \equiv \mathsf{unit}\ type}{\Gamma \vdash A\ \mathsf{sealed}\ @\ \varphi} \qquad \frac{\Gamma, \varphi \vdash A\ type}{\Gamma \vdash \varphi \Rightarrow A\ type} \qquad \frac{\Gamma, \varphi \vdash M : A}{\Gamma \vdash \langle\varphi\rangle M : \varphi \Rightarrow A} \qquad \frac{\Gamma \vdash M : \varphi \Rightarrow A}{\Gamma, \varphi \vdash M\ @\ \varphi : A}$$

$$\frac{\Gamma \vdash A\ type}{\Gamma \vdash [\varphi \setminus A]\ type} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{seal}_\varphi(M) : [\varphi \setminus A]} \qquad \frac{\Gamma \vdash M : [\varphi \setminus A] \qquad \Gamma, x : A \vdash N : C \qquad \Gamma \vdash C\ \mathsf{sealed}\ @\ \varphi}{\Gamma \vdash x \leftarrow \mathsf{unseal}_\varphi(M); N : C}$$

$$\frac{\Gamma \vdash A\ type \qquad \Gamma, \varphi \vdash M : A}{\Gamma \vdash \{A \mid \varphi \hookrightarrow M\}\ type} \qquad \frac{\Gamma \vdash N : A \qquad \Gamma, \varphi \vdash N \equiv M : A}{\Gamma \vdash N : \{A \mid \varphi \hookrightarrow M\}}$$

$$\Gamma \vdash [\varphi \setminus A]\ \mathsf{sealed}\ @\ \varphi \qquad \Gamma \vdash \{A \mid \varphi \hookrightarrow M\}\ \mathsf{sealed}\ @\ \varphi \qquad \Gamma, \varphi \vdash (\langle\varphi\rangle M)\ @\ \varphi \equiv M : A$$

$$\Gamma \vdash \langle\varphi\rangle M\ @\ \varphi \equiv M : \varphi \Rightarrow A \qquad \Gamma \vdash x \leftarrow \mathsf{unseal}_\varphi(\mathsf{seal}_\varphi(M)); N \equiv [M/x]N : C$$

Fig. 1. Selected rules for the modal account of phase distinctions.

Each phase $\phi : \mathbb{O}$ induces a context extension (Γ, φ); types and terms in such a context are *restricted* to their φ-visible components. For instance if $\varphi := \phi_{st}$ is the *static* phase, the dynamic parts of a type $\Gamma, \phi_{st} \vdash A\ type$ are collapsed. In this sense, the weakening substitution along $\Gamma, \phi_{st} \longrightarrow \Gamma$ implements the *static projection* operation $\mathsf{Fst}(-)$ from prior type theoretic accounts of modules [5], and judgmental equality $\Gamma, \phi_{st} \vdash A \equiv B\ type$ in the extended context reconstructs the *static equivalence* judgment of Dreyer et al. [6].

2.1 Modal type structure of φML

2.1.1 The phase modality. The context extension Γ, φ is internalized as the **phase modality** $(\varphi \Rightarrow -)$; semantically, $(\varphi \Rightarrow -)$ behaves like a function space whose domain is the collection of proofs that we are "in" the φ phase. Such a modality corresponds in the topological frame semantics of intuitionistic logic to the local operator induced by an open subspace.

2.1.2 The sealing modality. A type A is called *sealed* at $\varphi : \mathbb{O}$, written $\boxed{\Gamma \vdash A\ \mathsf{sealed}\ @\ \varphi}$, when it is equivalent to the *unit* type in phase φ. We include a **sealing modality** $[\varphi \setminus A]$ that seals a type A at phase φ; 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 intuitionsitic 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 $\varphi \vdash M : A$, we may form the **structure sharing** type $\{A \mid \varphi \hookrightarrow M\} \subseteq A$ that classifies all the elements of A equal to M at phase φ. In case $\varphi := \phi_{st}$, the structure sharing type $\{A \mid \phi_{st} \hookrightarrow M\} \leq A$ classifies the elements of A that are statically equivalent to M in the sense of Dreyer et al. [6] and therefore captures the *weak structure sharing* of SML '97 [15]. On the other hand, if $\varphi := \top$ is the "top" phase distinction, then $\{A \mid \top \hookrightarrow M\}$ is the true singleton type that is approximated by SML '90's *strong* structure sharing [14] via stamps, and by the F-ing Modules calculi via phantom types [20].

2.2 Applications of φML

We briefly survey a few applications of **φML**'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 $\phi_{\mathsf{st}} : \mathcal{O}$ together with a lax modality $\bigcirc A$ for effects that is always statically sealed, in the sense that $\boxed{\Gamma \vdash \bigcirc A \text{ sealed @ } \phi_{\mathsf{st}}}$ holds. Given another modality T that is not sealed, one could define the effect modality by $\bigcirc A :=$ $[\phi_{\mathsf{st}} \setminus T(A)]$ in terms of the sealing modality. ML-style generative and applicative functors may then be defined like so:

$$\begin{aligned}
&\Pi^{\mathrm{gen}}, \Pi^{\mathrm{app}} : (A : \mathbf{Sig})\,(B : (\phi_{\mathsf{st}} \Rightarrow A) \to \mathbf{Sig}) \to \mathbf{Sig} \\
&\Pi^{\mathrm{gen}}(A, B) = (x : A) \to \bigcirc B(\langle \phi_{\mathsf{st}} \rangle x) \\
&\Pi^{\mathrm{app}}(A, B) = (x : A) \to B(\langle \phi_{\mathsf{st}} \rangle x)
\end{aligned}$$

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

2.2.2 Compile-time inlining without breaking abstraction. Under a separate compilation discipline, *e.g.* that of Swasey et al. [26], 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 [23, § 1.5.3] and Leroy [11, § 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 $\phi_{\mathsf{cmpl}} : \mathcal{O}$ between *compile-time* and *runtime.*[1] Value identities are exposed for inlining by means of the the structure sharing type $\{A \mid \phi_{\mathsf{cmpl}} \hookrightarrow M\}$; programmers will not be able to rely on the identities so-exposed, but the compiler will be executed in the ϕ_{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 [7].

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 $\phi_{\mathsf{dbg}} : \mathcal{O}$ and, by default, expose the identities of all modules within the debug phase by means of the structure sharing type $\{A \mid \phi_{\mathsf{dbg}} \hookrightarrow M\}$; then we may add a primitive operation to the standard basis library that allows a ϕ_{dbg}-phase string to be printed, $\mathsf{debug} : (\phi_{\mathsf{dbg}} \Rightarrow \mathsf{string}) \to \bigcirc\mathsf{unit}$. Then in the presence of an element $a : M.\mathsf{t}$ whose (hidden) representation type is int, we may freely debug by executing the side effect $\mathsf{debug}(\langle \phi_{\mathsf{dbg}} \rangle \mathsf{Int.toString}(a))$.

2.2.4 Representation independence. Following the **Logical Relations As Types** principle of Sterling and Harper [22], we may capture *binary parametricity* [18] by adding two phases $\phi^{\mathsf{L}}_{\mathsf{syn}}, \phi^{\mathsf{R}}_{\mathsf{syn}} : \mathcal{O}$ with $\phi^{\mathsf{L}}_{\mathsf{syn}} \sqcap \phi^{\mathsf{R}}_{\mathsf{syn}} \equiv \bot$ and defining $\phi_{\mathsf{syn}} := \phi^{\mathsf{L}}_{\mathsf{syn}} \sqcup \phi^{\mathsf{R}}_{\mathsf{syn}}$. Then representation independence results can be proved: a simulation between queue implementations $M, N : \mathsf{QUEUE}$ is given by a third implementation $O : \{\mathsf{QUEUE} \mid \phi_{\mathsf{syn}} \hookrightarrow [\phi^{\mathsf{L}}_{\mathsf{syn}} \hookrightarrow M, \phi^{\mathsf{R}}_{\mathsf{syn}} \hookrightarrow N]\}$. This method is used by *op. cit.* to prove a generalized Reynolds Abstraction Theorem for a module calculus, and by Sterling and Angiuli [21] to prove normalization and decidability of judgmental equality for cubical type theory.

[1] Here compile-time refers to a stage subsequent to typechecking/elaboration, and is therefore semantically different from a static phase.

REFERENCES

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. Association for Computing Machinery, San Antonio, Texas, USA, 147–160. https://doi.org/10.1145/292540.292555

[2] Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. (2021). arXiv:2104.00480 [cs.PL] To appear in the proceedings of ECOOP 2021.

[3] Karl Crary. 2020. A focused solution to the avoidance problem. *Journal of Functional Programming* 30 (2020), e24. https://doi.org/10.1017/S0956796820000222 *Bob Harper Festschrift Collection*.

[4] Leonardo De Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language (System Description). (2021). To appear in the proceedings of the 28th International Conference on Automated Deduction.

[5] Derek Dreyer. 2005. *Understanding and Evolving the ML Module System*. Ph.D. Dissertation. Carnegie Mellon University, USA.

[6] Derek Dreyer, Karl Crary, and Robert Harper. 2003. A Type System for Higher-Order Modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New Orleans, Louisiana, USA, 236–249. https://doi.org/10.1145/604131.604151

[7] Robert Harper. 2013. The Future of Standard ML. (2013). https://www.cs.cmu.edu/~rwh/talks/mlw13.pdf Talk given at the ML Workshop.

[8] Robert Harper. 2020. ***PFPL*** *Supplement: Types for Program Modules*. http://www.cs.cmu.edu/~rwh/pfpl/supplements/modules.pdf

[9] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Portland, Oregon, USA, 123–137. https://doi.org/10.1145/174675.176927

[10] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1990. Higher-Order Modules and the Phase Distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, San Francisco, California, USA, 341–354. https://doi.org/10.1145/96709.96744

[11] Xavier Leroy. 2000. A Modular Module System. *Journal of Functional Programming* 10, 3 (May 2000), 269–303. https://doi.org/10.1017/S0956796800003683

[12] David MacQueen, Robert Harper, and John Reppy. 2020. The History of Standard ML. *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020). https://doi.org/10.1145/3386336

[13] Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In *POPL '15: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Mumbai, India. https://hal.inria.fr/hal-01096910

[14] Robin Milner, Mads Tofte, and Robert Harper. 1990. *The Definition of Standard ML*. MIT Press.

[15] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

[16] Eugenio Moggi. 1989. A Category-Theoretic Account of Program Modules. In *Category Theory and Computer Science*. Springer-Verlag, Berlin, Heidelberg, 101–117.

[17] Bengt Nordström, Kent Peterson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory*. International Series of Monographs on Computer Science, Vol. 7. Oxford University Press, NY.

[18] John C. Reynolds. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*.

[19] Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). https://doi.org/10.23638/LMCS-16(1:2)2020 arXiv:1706.07526 [math.CT]

[20] Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of Functional Programming* 24, 5 (2014), 529–607. https://doi.org/10.1017/S0956796814000264

[21] Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, New York, NY, USA. arXiv:2101.11479 [cs.LO] To appear.

[22] Jonathan Sterling and Robert Harper. 2021. Logical Relations As Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* (2021). arXiv:2010.08599 [cs.PL] To appear.

[23] Christopher Allen Stone. 2000. *Singleton Kinds and Singleton Types*. Ph.D. Dissertation. Carnegie Mellon University.

[24] Christopher A. Stone and Robert Harper. 2000. Deciding Type Equivalence in a Language with Singleton Kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, Boston, MA, USA, 214–227. https://doi.org/10.1145/325694.325724

[25] Christopher A. Stone and Robert Harper. 2006. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic* 7, 4 (2006), 676–722. https://doi.org/10.1145/1183278.1183281

[26] David Swasey, Tom Murphy, Karl Crary, and Robert Harper. 2006. A Separate Compilation Extension to Standard ML. In *Proceedings of the 2006 Workshop on ML (ML '06)*. Association for Computing Machinery, Portland, Oregon, USA, 32–42. https://doi.org/10.1145/1159876.1159883