A Type System for Higher-Order Modules

Derek Dreyer Karl Crary Robert Harper

Carnegie Mellon University

Abstract

We present a type theory for higher-order modules that accounts for most current issues in modular programming languages, including translucency, applicativity, generativity, and modules as first-class values. Our theory harmonizes design elements from various previous work, resulting in a simple, economical, and elegant language. This language is useful as a framework for comparing alternative designs, and is the first language to provide all of these features simultaneously and still support a practical type checking algorithm.

1 Introduction

The design of languages for modular programming is surprisingly delicate and complex. There is a fundamental tension between the desire to separate program components into relatively independent parts and the need to integrate these parts to form a coherent whole. To some extent the design of modularity mechanisms is independent of the underlying language [16], but to a large extent the two are inseparable. For example, languages with polymorphism, generics, or type abstraction require far more complex module mechanisms than those that do not.

Much work has been devoted to the design of modular programming languages. Early work on CLU [18] and the Modula family of languages [32, 2] has been particularly influential. Much effort has gone into the design of modular programming mechanisms for the ML family of languages, notably Standard ML [21] and Objective Caml [25]. Numerous extensions and variations of these designs have been considered in the literature [19, 17, 27, 29, 5].

Despite (or perhaps because of) these substantial efforts, the field has remained somewhat fragmented, with no clear unifying theory of modularity having yet emerged. Several competing designs have been proposed, often seemingly at odds with one another. These decisions are as often motivated by pragmatic considerations, such as engineering a useful implementation, as by more fundamental considerations, such as the semantics of type abstraction. The relationship between these design decisions is not completely clear, nor is there a clear account of the trade-offs between them, or whether they can be coherently combined into a single design.

The goal of this paper is to provide a simple, unified formalism for modular programming that consolidates and elucidates much of the work mentioned above. Building on a substantial and growing body of work on type-theoretic accounts of language structure, we propose a type theory for higher-order program modules that harmonizes and enriches these designs and that would be suitable as a foundation for the next generation of modular languages.

1.1 Design Issues

Before describing the main technical features of our language, it is useful to review some of the central issues in the design of module systems for ML. These issues extend to any language of similar expressive power, though some of the trade-offs may be different for different languages.

Controlled Abstraction Modularity is achieved by using signatures (interfaces) to mediate access between program components. The role of a signature is to allow the programmer selectively to "hide" type information. The mechanism for controlling type propagation is *translucency* [10, 13], with transparency and opacity as limiting cases.

Phase Separation ML-like module systems enjoy a *phase separation* property [11] stating that every module is separable into a static part, consisting of type information, and a dynamic part, consisting of executable code. To obtain fully expressive higher-order modules and to support abstraction, it is essential to build this phase separation principle into the definition of type equivalence.

Generativity MacQueen coined the term *generativity* for the creation of "new" types corresponding to run-time instances of an abstraction. For example, we may wish to define a functor `SymbolTable` that, given some parameters, creates a new symbol table. It is natural for the symbol table module to export an abstract type of symbols that are dynamically created by insertion and used for subsequent retrieval. To preclude using the symbols from one symbol table to index another, generativity is essential—each instance of the hash table must yield a "new" type, distinct from all others, even when applied twice to the same parameters.

Separate Compilation One goal of module system design is to support separate compilation [13]. This is achieved by ensuring that all interactions among modules are mediated by interfaces that capture all of the information known to the clients of separately-compiled modules.

Principal Signatures The *principal*, or most expressive, signature for a module captures all that is known about that module during type checking. It may be used as a proxy for that module for purposes of separate compilation. Many type checking algorithms, including the one given in this paper, compute principal signatures for modules.

Hidden Types Introducing a local, or "hidden", abstract type within a scope requires that the types of the externally visible components avoid mention of the abstract type. This *avoidance problem* is often a stumbling block for module

system design, since in most expressive languages there is no "best" way to avoid a type variable [7, 17].

1.2 A Type System for Modules

The type system proposed here takes into account all of these design issues. It consolidates and harmonizes design elements that were previously seen as disparate into a single framework. For example, rather than regard generativity of abstract types as an alternative to non-generative types, we make both mechanisms available in the language. We support both generative and applicative functors, admit translucent signatures, support separate compilation, and are able to accommodate modules as first-class values [22, 28].

Generality is achieved not by a simple accumulation of features, but rather by isolating a few key mechanisms that, when combined, yield a flexible, expressive, and implementable type system for modules. Specifically, the following mechanisms are crucial.

Singletons Propagation of type sharing is handled by *singleton signatures*, a variant of Aspinall's and Stone and Harper's *singleton kinds* [31, 30, 1]. Singletons provide a simple, orthogonal treatment of sharing that captures the full equational theory of types in a higher-order module system with subtyping. No previous module system has provided both abstraction and the full equational theory supported by singletons,[1] and consequently none has provided optimal propagation of type information.

Static Module Equivalence The semantics of singleton signatures is dependent on a (compile-time) notion of equivalence of modules. To ensure that the phase distinction is respected, we define module equivalence to mean "equivalence of static components," ignoring all run-time aspects.

Subtyping Subtyping is used at the signature level to model "forgetting" type sharing information, which is essential for signature matching. The coercive aspects of signature matching (*e.g.,* dropping of fields and specialization of polymorphic values) are omitted here, since the coercions required are definable in the language.

Determinacy To ensure the proper implementation of abstraction, our type system forbids some modules from being tested for equivalence to any other modules. Modules that are licensed for equality testing are termed *determinate.* Determinacy also provides the license for a module to appear in a signature (via the construction of a singleton signature) or in a type (via projection of one of the module's type components), as each such appearance implicitly requires the ability to test for equality.

Static and Dynamic Effects The "sealing" [10] of a module with a signature, which is used to model abstract and/or generative types, is deemed by our type system to induce a pro forma computational effect. This is backed up by the intuition that generativity involves the generation of new types at run time. (Of course, no such run-time generation actually occurs, but the pro forma effect nevertheless provides the intended behavior.) This notion of effects allows us to provide generativity without resorting to the use of "generative stamps" [21, 19].

We isolate two distinct sorts of computational effects, which we call *static* and *dynamic.*[2] We then break modules into four categories, depending on what sort of effects they may induce: *pure* modules involve no effects, *dynamically pure* modules involve only static effects (but no dynamic ones), *statically pure* modules involve only dynamic effects, and *general* modules may involve any effect.

For our type system, we set the class of determinate modules to be precisely the pure modules. Consequently, since modules involving sealing are impure, they are also indeterminate, and therefore may not appear within a type or signature. However, dynamic or static purity alone does provide some privileges not enjoyed by arbitrary modules. For example, dynamically pure modules (which may contain opaque but non-generative types) may appear within the body of an applicative functor.

Applicative and Generative Functors An *applicative* functor [14] is one that respects static equivalence of its arguments. This models the behavior of functors in Objective Caml. A *generative*, or *non-applicative*, functor does not respect equivalence—the abstract types in the result differ on each application.

Existential Signatures In a manner similar to Shao [29], our type system is carefully crafted to circumvent the avoidance problem, so that every module enjoys a principal signature. However, this requires imposing restrictions on the programmer. To lift these restrictions, we follow Russo [27] (generalizing Harper and Stone [12]) and employ existential signatures to provide principal signatures where none would otherwise exist. We show that these existential signatures are type-theoretically ill-behaved in general, so, like Russo and like Harper and Stone, we restrict their use to a well-behaved setting. In particular, we define an elaboration algorithm from an external language that may incur the avoidance problem, into our type system for higher-order modules, which does not.

First-Class Polymorphism Both ML-style polymorphic typing and the ability to treat modules as first-class values arise from a single polymorphic type constructor that abstracts user-level code over a module.

While these features combine naturally to form a very general language for modular programming, they would be of little use in the absence of a practical implementation strategy. Some previous attempts have encountered difficulties with undecidability [10] or incompleteness of type checking [25]. In contrast, our formalism leads to a practical, implementable programming language. First, we provide a sound, complete, and effective type checking algorithm. The algorithm consists of two parts: computation of principal signatures for modules, which we show exist, and checking of subsignature relationships. The latter aspect of the algorithm reduces to checking module equivalence, for which we

[1] Typically the omitted equations are not missed because restrictions to named form or valuability prevent programmers from writing code whose typeability would depend on those equations in the first place [3].

[2] Note that even dynamic effects are still static, in the sense they are a fiction of the type system, and would not be reflected in the operational semantics.

types	τ	::=	$\mathsf{Typ}\, M \mid \Pi s{:}\sigma.\tau \mid \tau_1 \times \tau_2$
terms	e	::=	$\mathsf{Val}\, M \mid \langle e_1, e_2\rangle \mid \pi_i e \mid e\, M \mid$
			$\mathsf{fix}\, f(s{:}\sigma){:}\tau.e \mid \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau)$
signatures	σ	::=	$1 \mid [\![T]\!] \mid [\![\tau]\!] \mid \Pi s{:}\sigma_1.\sigma_2 \mid \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2 \mid$
			$\Sigma s{:}\sigma_1.\sigma_2 \mid \mathfrak{S}(M)$
modules	M	::=	$s \mid \langle\rangle \mid [\tau] \mid [e : \tau] \mid \lambda s{:}\sigma.M \mid M_1 M_2 \mid$
			$\langle s = M_1, M_2\rangle \mid \pi_i M \mid$
			$\mathsf{let}\ s = M_1\ \mathsf{in}\ (M_2 : \sigma) \mid$
			$M :\!> \sigma \mid M :: \sigma$
contexts	Γ	::=	$\epsilon \mid \Gamma, s{:}\sigma$

Figure 1: Syntax

rely on an extension of Stone and Harper's algorithm [31]. Second, we provide an effective elaboration algorithm from a general external language (with hidden types and the resulting avoidance problem) into our type system.

2 Technical Development

We begin our technical development by presenting the syntax of our language in Figure 1. Our language consists of four syntactic classes: terms, types, modules, and signatures (which serve as the types of modules). The language does not explicitly include higher-order type constructors or kinds (which ordinarily serve as constructors' types); in our language the roles of constructors and kinds are subsumed by modules and signatures. Contexts bind module variables (s) to signatures.

As usual, we consider alpha-equivalent expressions to be identical. We write the capture-avoiding substitution of M for s in an expression E as $E[M/s]$.

Types and Terms There are three basic types in our language. The product type ($\tau_1 \times \tau_2$) is standard. The function type, $\Pi s{:}\sigma.\tau$, is the type of functions that accept a module argument s of signature σ and return a value of type τ (possibly containing s). As usual, if s does not appear free in τ, we write $\Pi s{:}\sigma.\tau$ as $\sigma \to \tau$. (This convention is used for the dependent products in the signature class as well.) Finally, when M is a module containing exactly one type (which is to say that M has the signature $[\![T]\!]$), that type is extracted by $\mathsf{Typ}\, M$. A full-featured language would support a variety of additional types as well.

The term language contains the natural introduction and elimination constructs for recursive functions and products. In addition, when M is a module containing exactly one value (which is to say that M has the signature $[\![\tau]\!]$, for some type τ), that value is extracted by $\mathsf{Val}\, M$. When f does not appear free in e, we write $\mathsf{fix}\, f(s{:}\sigma){:}\tau.e$ as $\Lambda s{:}\sigma.e$.

The conventional forms of functions and polymorphic function are built from module functions. Ordinary functions are built using modules containing a single value:

$$\begin{array}{lcl} \tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & [\tau_1] \to \tau_2 \\ \lambda x{:}\tau.e(x) & \stackrel{\text{def}}{=} & \Lambda s{:}[\tau].e(\mathsf{Val}\, s) \\ e_1 e_2 & \stackrel{\text{def}}{=} & e_1[e_2] \end{array}$$

and polymorphic functions are built using modules containing a single type:

$$\begin{array}{lcl} \forall \alpha.\tau(\alpha) & \stackrel{\text{def}}{=} & \Pi s{:}[\![T]\!].\tau(\mathsf{Typ}\, s) \\ \Lambda \alpha.e(\alpha) & \stackrel{\text{def}}{=} & \Lambda s{:}[\![T]\!].e(\mathsf{Typ}\, s) \\ e\,\tau & \stackrel{\text{def}}{=} & e[\tau] \end{array}$$

Signatures and Modules There are seven basic signatures in our language. The atomic signature $[\![T]\!]$ is the type of an atomic module containing a single type, and the atomic signature $[\![\tau]\!]$ is the type of an atomic module containing a single term. The atomic modules are written $[\tau]$ and $[e : \tau]$, respectively. (We omit the type label ": τ" from atomic term modules when it is clear from context.) The trivial atomic signature 1 is the type of the trivial atomic module $\langle\rangle$.

The functor signatures $\Pi s{:}\sigma_1.\sigma_2$ and $\Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2$ express the type of functors that accept an argument of signature σ_1 and return a result of signature σ_2 (possibly containing s). We discuss the difference between Π and Π^{gen} in detail below.

The structure signature $\Sigma s{:}\sigma_1.\sigma_2$ is the type of a pair of modules where the left-hand component has signature σ_1 and the right-hand component has signature σ_2, in which s refers to the left-hand component. They are introduced by the pairing construct $\langle s = M_1, M_2\rangle$ in which s stands for M_1 and may appear free in M_2. As usual, when s does not appear free in σ_2, we write $\Sigma s{:}\sigma_1.\sigma_2$ as $\sigma_1 \times \sigma_2$.

The singleton signature $\mathfrak{S}(M)$ is used to express type sharing information. It classifies modules that have signature $[\![T]\!]$ and are statically equivalent to M. Two modules are considered statically equivalent if they are equal modulo term components; that is, type fields must agree but term fields may differ. Singletons at signatures other than $[\![T]\!]$ are not provided primitively because they can be defined using the basic singleton, as described by Stone and Harper [31]. The definition of $\mathfrak{S}_\sigma(M)$ (the signature containing only modules equal to M at signature σ) is given in Figure 5.

The module syntax contains module variables (s), the atomic modules, and the usual introduction and elimination constructs for Π and Σ signatures, except that Σ modules are introduced by $\langle s = M_1, M_2\rangle$, in which s stands for M_1 in M_2. (When s does not appear free in M_2, the "$s =$" is omitted.) No introduction or elimination constructs are provided for singleton signatures. Singletons are introduced and eliminated by rules in the static semantics; if M is judged equivalent to M' in σ, then M belongs to $\mathfrak{S}_\sigma(M')$, and vice versa.

The remaining module constructs are strong sealing, written $M :\!> \sigma$, and weak sealing, written $M :: \sigma$. When a module M is strongly sealed with a signature σ, the result is *opaque* and *generative*. By opaque we mean that no client of the module may depend on any details of the implementation of M other than what is exposed by the signature σ. By generative we mean that dynamic instance of $M :\!> \sigma$ produces types that are judged unequal to those of any other. A weakly sealed module is opaque but *not* generative; we discuss the utility of weak sealing in Section 3.

Although higher-order type constructors do not appear explicitly in our language, they are faithfully represented in our language by unsealed modules containing only type components. For example, the kind $(T \to T) \to T$ is represented by the signature $([\![T]\!] \to [\![T]\!]) \to [\![T]\!]$; and the constructor $\lambda\alpha{:}(T \to T).(\mathtt{int} * \alpha\, \mathtt{int})$ is represented by the module $\lambda s{:}([\![T]\!] \to [\![T]\!]).[\mathtt{int} * \mathsf{Typ}(s\, [\mathtt{int}])]$.

```
signature SIG =
  sig
    type s
    type t = s * int

    structure S : sig
                    type u
                    val f : u -> s
                  end

    val g : t -> S.u
  end
```

... is compiled as ...

$$\begin{array}{l}\Sigma s{:}[\![T]\!].\\ \Sigma t{:}\mathfrak{S}([\mathsf{Typ}\, s * \mathtt{int}]).\\ \Sigma S{:}(\Sigma u{:}[\![T]\!].\Sigma f{:}[\![\mathsf{Typ}\, u \to \mathsf{Typ}\, s]\!].1).\\ \Sigma g{:}[\![\mathsf{Typ}\, t \to \mathsf{Typ}(\pi_1 S)]\!].1\end{array}$$

Figure 2: ML Signature Example

```
structure S1 =
  struct
    type s = bool
    type t = bool * int

    structure S = struct
                    type u = string
                    val f = (fn y:u => true)
                  end

    val g = (fn y:t => "hello world")
  end
```

... is compiled as ...

$$\begin{array}{l}\langle[\mathtt{bool}],\\ \quad\langle[\mathtt{bool} * \mathtt{int}],\\ \quad\quad\langle\langle[\mathtt{string}], \langle[\lambda y{:}\mathtt{string}.\mathtt{true}], \langle\rangle\rangle\rangle,\\ \quad\quad\quad\langle[\lambda y{:}\mathtt{bool} * \mathtt{int}.\mathtt{"hello\ world"}], \langle\rangle\rangle\rangle\rangle\rangle\end{array}$$

Figure 3: ML Structure Example

Examples of how ML-style signatures and structures may be expressed in our language appear in Figures 2 and 3.

2.1 Module Equivalence and Implications

The key issue in the design of our module calculus lies in two related questions: When can modules be compared for equivalence, and, given two comparable modules, when are they deemed equivalent? We say that a module is *determinate* if it is eligible to be compared for equivalence.

With regard to comparability, we seek to provide the largest class of determinate modules possible while still providing for programmer-specified abstraction and generativity. With regard to equivalence of comparable modules, we will rule that two determinate modules with the same signature are equivalent if and only if their static (*i.e.*, type-related) components are equal. This provides the most permissive equality while still complying with the phase distinction [11].

We will look first at the topic of determinacy, dealing with equivalence at a strictly informal level. Then we will look at the specifics of equivalence and formalize our equivalence judgement.

2.1.1 Determinacy

In the literature different accounts of higher-order modules provide different classes of determinate modules. For example, in Harper and Lillibridge's first-class module system [10], only values are considered determinate. This is necessary in their setting for type soundness because, in the presence of side-effects, non-values could compute different modules containing different types each time they are evaluated. Thus, the type components of a non-value are not well determined (hence the phrase "indeterminate") and cannot meaningfully be compared for equivalence.

In Leroy's second-class module calculi [13, 14], determinacy is limited to the syntactic category of paths. In Leroy's case this was not necessary for type soundness, since modules were never produced by run-time computations, but it served to provide a certain degree of abstraction. In Harper, *et al.*'s "phase-distinction" calculus [11], an early higher-order, second-class module system, all modules were deemed to be determinate, but no means of abstraction was provided.

In our language we wish to admit as large a class of determinate modules as possible. First, we specify determinacy using a semantic condition (formalized by a judgement in the type theory), rather than as a syntactic condition. In addition to being more elegant, this semantic treatment of determinacy turns out to be necessary for a correct account of generativity. Second, we deem all modules to be determinate unless indeterminacy is imposed by programmer-specified abstraction and/or generativity through the language's sealing mechanisms.

Projection of Type Components Often the literature on module systems has focused on which modules can appear in type projections, rather than on which modules are equivalent to one another. For example, Harper and Lillibridge [10] and Leroy [13, 14] emphasize projection, but on the other hand Harper, *et al.* [11] stress module equivalence. Here we stress equivalence as the primary notion, since the choice of which modules may be compared for equality determines those from which types may be projected.

To see this, suppose that M is an indeterminate module with signature $[\![T]\!]$. By definition, M cannot be compared for equivalence with any other module. Suppose, however, we are permitted to form the type $\mathsf{Typ}\, M$. All types are comparable for equality, so $\mathsf{Typ}\, M$ can be compared for equality with any other type, including $\mathsf{Typ}\, M'$ (for some other M' with signature $[\![T]\!]$). Since $\mathsf{Typ}\, M$ returns the entire content of M, and likewise for M', this gives us a means by which we may compare M for equivalence with M', which cannot be permitted. Thus, the type system must permit the projection of type components only from *determinate* structures.

Strong Sealing and Generativity In our type system, determinacy is limited by the imposition of abstraction by the programmer. The principal means for doing so is *strong sealing* written $M :> \sigma$, which generatively seals the module M with the signature σ. By "generatively," we mean that multiple dynamic instances of the expression $M :> \sigma$ create

modules with unequal type components. Each dynamic instance of a sealed module is thereby said to "generate" an abstract type distinct to that occurrence.

When considered from the point of view of module equivalence, generativity means that a strongly sealed module should not even be equivalent to itself! Since module equivalence must surely be symmetric and transitive, any module expression that can be compared for equivalence will surely be equivalent to itself, for if M is equivalent to N, then, by symmetry and transitivity, M is equivalent to itself. Thus to ensure generativity, strongly sealed modules cannot be comparable to other modules, which is to say that they should be ruled indeterminate.

Strong sealing is not the only form of generativity in our language. We also support *generative functors*, which yield distinct abstract types for each application. Thus, if F is a generative functor, then the application $F(M)$ should behave generatively, and hence be considered indeterminate, even if F is a functor variable and M is determinate. In our system, generative functors are given signatures using the Π^{gen} construct; functors whose signatures use an ordinary Π are *applicative* [14]. Since functors can be simple variables, no syntactic condition alone can determine if a functor is generative.

2.1.2 Effects and Purity

The irreflexivity of generative module expressions is strongly reminiscent of the irreflexivity of expressions in languages with effects. Indeed, a guiding intuition in the development is to regard generativity as a *pro forma* computational effect. In a first-class module system such as Harper and Lillibridge's [10], an effectful module expression must be ruled generative, since it could yield a distinct type each time it is evaluated. In a second-class module system no such behavior is, in fact, possible, but it is useful to regard a strong-sealed module as "hiding" an arbitrary computational effect. Thus, a strongly sealed module behaves as if its opaque types were generated at run-time, even though they are not.

This brings about the familiar notions of *pure* (effect-free) and *impure* (effectful) expressions, namely by considering sealing to induce an effect.[3] With this in mind, we may define the set of determinate modules provided by our type system: a module is determinate if and only if it is pure. This follows our intuition, as the meaning of an effectful expression is not "well-determined," and it is necessary to provide the desired opacity and generativity properties.

Weak Sealing and Static Effects The purpose of strong sealing is to induce opacity and generativity. However, it is also useful to be able to induce opacity *without* generativity. That is, a programmer may wish to seal a module so that no client can depend on implementation details not reflected in the signature, but not to generate unequal type components at each dynamic instance.

We provide such a facility through *weak sealing* (written $M :: \sigma$). To ensure opacity, weakly sealed modules must not be determinate; otherwise selfification (Section 2.2) could be used to propagate information not given by σ. Therefore, we adopt the view that weak sealing (like strong sealing) creates new type components, and consequently is impure and cannot meaningfully be compared for equivalence. However, since weak sealing is not intended to be generative, we wish for the new type components to be the same at each dynamic instance.

In short, while $M :> \sigma$ is viewed as generating new type components at run time, $M :: \sigma$ is viewed as generating new type components at *compile time*. As a result, it is reasonable to distinguish between the sort of effects induced by weak and strong sealing. We say that weak sealing induces a *static* effect, while strong sealing induces a static and a *dynamic* effect.

We discuss the importance of supporting both weak and strong sealing in Section 3. One could also contemplate a third form of sealing ("co-weak" sealing), that provides generativity but not opacity, and which consequently induces a dynamic but not a static effect, but it is unclear what the utility of such a mechanism would be.

Dynamic vs. Static Effects In our type system, as usual, dynamic effects are suspended when they appear within a lambda. Thus, a lambda is always dynamically pure (free of dynamic effects). When such a lambda is applied, whatever dynamic effects were suspended within it are released, and so a functor application may or may not have dynamic effects. In order that our type system can track dynamic effects, we use signatures to distinguish between functors that may or may not induce dynamic effects when applied. An applicative functor always has a dynamically pure body, and thus its application (to a dynamically pure argument) is dynamically pure. However, a generative functor may have a dynamically impure body, and thus may generate dynamic effects when applied. One consequence is that the application of a generative functor is never determinate (even when the functor and its argument both are).

In contrast, static effects, being generated at compile time, may be resolved under a lambda. More generally, no construct can suspend a static effect. Therefore, any module containing any strong or weak sealing is statically impure, and therefore is indeterminate. This ensures opacity: a sealed module cannot be given any signature mentioning that module, so that module can always be replaced with another module having the same signature.

Although a dynamically pure, statically impure module is not determinate, it still enjoys privileges not enjoyed by modules in general. In particular, it can appear within the body of an applicative functor. Thus opaque sub-structures (such as ML-style datatypes [12]) can appear within applicative functors. Conversely, a statically pure, dynamically impure module may only appear in the body of a generative functor, but that functor then captures the dynamic effects, resulting in a determinate functor.

Formalization The rules of our type system follow from these definitions. We write our typing judgement $\Gamma \vdash_\kappa M : \sigma$, where κ indicates M's purity. The classifier κ is drawn from the lattice:

```
   W
  / \
 D   S
  \ /
   P
```

where P indicates that M is pure (and hence determinate), D indicates dynamic purity, S indicates static purity, and

[3] Since we are working with a second-class module system, this is the only source of impurity in the language. Effectful expressions in the core language do not introduce an impurity in our sense.

$$\frac{\Gamma \vdash_\kappa M : \sigma \quad \kappa \sqsubseteq \kappa'}{\Gamma \vdash_{\kappa'} M : \sigma} \ (1) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma} \ (2) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} (M :: \sigma) : \sigma} \ (3) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s)} \ (4)$$

$$\frac{\Gamma, s{:}\sigma_1 \vdash_\kappa M : \sigma_2 \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma_1.M : \Pi s{:}\sigma_1.\sigma_2} \ (5) \qquad \frac{\Gamma, s{:}\sigma_1 \vdash_\kappa M : \sigma_2}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma_1.M : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2} \ (6) \qquad \frac{\Gamma, s{:}\sigma_1 \vdash \sigma_2 \ \mathrm{sig}}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2 \leq \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2} \ (7)$$

$$\frac{\Gamma \vdash_\kappa M_1 : \Pi s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma_1}{\Gamma \vdash_\kappa M_1 M_2 : \sigma_2[M_2/s]} \ (8) \qquad \frac{\Gamma \vdash_\kappa M_1 : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma_1}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} M_1 M_2 : \sigma_2[M_2/s]} \ (9)$$

$$\frac{\Gamma \vdash_\kappa M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash_\kappa \pi_1 M : \sigma_1} \ (10) \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma_2[\pi_1 M/s]} \ (11) \qquad \frac{\Gamma \vdash_\kappa M : \sigma \quad \Gamma \vdash \sigma \leq \sigma'}{\Gamma \vdash_\kappa M : \sigma'} \ (12)$$

Figure 4: Key Typing Rules

$\mathtt{W}$ indicates well-formedness only (no purity information). Hence, $\Gamma \vdash_{\mathtt{P}} M : \sigma$ is our determinacy judgement. It will prove to be convenient in our typing rules to exploit the ordering (written $\sqsubseteq$), meets ($\sqcap$), and joins ($\sqcup$) of this lattice, where $\mathtt{P}$ is taken as the bottom and $\mathtt{W}$ is taken as the top. We also sometimes find it convenient to write functor signatures as $\Pi^\delta s{:}\sigma_1.\sigma_2$, where $\delta \in \{\epsilon, \mathtt{gen}\}$ and take $\epsilon \sqsubseteq \mathtt{gen}$.

Some key rules are summarized in Figure 4: Pure modules are dynamically pure and statically pure, and each of those are at least well-formed (rule 1). Strongly sealed modules are neither statically nor dynamically pure (2); weakly sealed modules are not statically pure, but are dynamically pure if their body is (3). Applicative functors must have dynamically pure bodies (5); generative functors have no restriction (6). Applicative functors may be used as generative ones (7). Variables are pure (4), and lambdas are dynamically pure (5 and 6). The application of an applicative functor is as pure as the functor itself (8), but the application of a generative functor is at best statically pure (9). Finally, the purity of a module is preserved by signature subsumption (12). The complete set of typing rules is given in Appendix A.

The rules for functor application (rules 8 and 9) require that the functor argument be determinate. This is because the functor argument is substituted into the functor's codomain to produce the result signature, and the substitution of indeterminate modules for variables (which are always determinate) can turn well-formed signatures into ill-formed ones (for example, $[\mathsf{Typ}\, s]$ is ill-formed when an indeterminate module is substituted for s). (The alternative rule proposed by Harper and Lillibridge [10] resolves this issue, but induces the avoidance problem, as we discuss in Section 4.) Therefore, when a functor is to be applied to an indeterminate argument, that argument must first be bound to a variable, thereby making it determinate. Similarly, projection of the second component of a pair is restricted to determinate pairs (rule 11), but no such restriction need be made for projection of the first component (rule 10).

2.1.3 Static Equivalence

The second key design issue is, when are determinate modules deemed equivalent? If a determinate module has signature $[\![T]\!]$, it is possible to extract types from it. Type checking depends essentially on the matter of which types are equal, so we must consider when $\mathsf{Typ}\, M$ is equal to $\mathsf{Typ}\, M'$.

The simplest answer to this question would be that $\mathsf{Typ}\, M = \mathsf{Typ}\, M'$ exactly when the modules M and M' are equal. This simple answer is naive because we cannot in general determine when two modules are equal. Suppose $F : [\![\mathtt{int}]\!] \to \sigma$ and $e, e' : \mathtt{int}$. Then $F[e] = F[e']$ if and only if $e = e'$, but the latter equality is undecidable in general. To preserve the phase distinction [11] between compile-time and run-time computation, we must ensure that type checking never requires the evaluation of program terms.

To resolve this problem, observe that our language provides no means by which a type component of a module can depend on a term component. (This is not happenstance, but the result of careful design. We will see in Section 5.1 that the matter is more subtle than it appears.) Consequently, we may ignore general equality and restrict our attention to the *static* equivalence of modules. Two modules are deemed to be equivalent if they agree on all type components.[4] In the sequel, we will sometimes refer to "static equivalence" to emphasize the static nature of our equivalence.

We write our module equivalence judgement as $\Gamma \vdash M \cong M' : \sigma$. The rules for static equivalence of atomic modules are the expected ones. Atomic type components must be equal, but atomic term components need not be:

$$\frac{\Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash [\tau] \cong [\tau'] : [\![T]\!]} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![\tau]\!] \quad \Gamma \vdash_{\mathtt{P}} M' : [\![\tau]\!]}{\Gamma \vdash M \cong M' : [\![\tau]\!]}$$

Since the generative production of new types in a generative functor is notionally a dynamic operation, generative functors have no static components to compare. Thus, determinate generative functors are always statically equivalent, just as atomic term modules are:

$$\frac{\Gamma \vdash_{\mathtt{P}} M : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\mathtt{P}} M' : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2}{\Gamma \vdash M \cong M' : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2}$$

[4] The phase distinction calculus of Harper, *et al.* [11] includes "non-standard" equality rules for phase-splitting modules M into structures $\langle M_{stat}, M_{dyn} \rangle$ consisting of a static component M_{stat} and a dynamic component M_{dyn}. Our static equivalence $M \cong M'$ amounts to saying $M_{stat} = M'_{stat}$ in their system. However, we do not identify functors with structures, as they do.

The complete set of equivalence rules is given in Appendix A.

As an aside, the notion of static equivalence refutes the misconception that first-class modules are more general than second-class modules. In fact, first- and second-class modules are incomparable. First-class modules have the obvious advantage that they are first class (although this advantage can be mitigated somewhat by the existential packaging mechanism [22] we discuss in Section 5.1). However, since the type components of a first-class module can depend on run-time computations, it is impossible to get by with static module equivalence and must use dynamic equivalence instead (in other words, one cannot phase-split modules as in Harper, *et al.* [11]). Consequently, first-class modules can never propagate as much type information as second-class modules can.

2.2 Singleton Signatures

Type sharing information is expressed in our language using singleton signatures [31], a derivative of translucent sums [10, 13, 17]. (An illustration of the use of singleton signatures to express type sharing appears in Figure 2.) The type system allows the deduction of equivalences from membership in singleton signatures, and vice versa, and also allows the forgetting of singleton information using the subsignature relation:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \mathfrak{S}_\sigma(M') \quad \Gamma \vdash_{\mathsf{P}} M' : \sigma}{\Gamma \vdash M \cong M' : \sigma} \qquad \frac{\Gamma \vdash M \cong M' : \sigma}{\Gamma \vdash_{\mathsf{P}} M : \mathfrak{S}_\sigma(M')}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma}{\Gamma \vdash \mathfrak{S}_\sigma(M) \leq \sigma} \qquad \frac{\Gamma \vdash M \cong M' : \sigma}{\Gamma \vdash \mathfrak{S}_\sigma(M) \leq \mathfrak{S}_\sigma(M')}$$

When $\sigma = [\![T]\!]$, these deductions follow using primitive rules of the type system (since $\mathfrak{S}_{[\![T]\!]}(M) = \mathfrak{S}(M)$ is primitive). At other signatures, they follow from the definitions given in Figure 5.

Beyond expressing sharing, singletons are useful for "selfification" [10]. For instance, if s is a variable bound with the signature $[\![T]\!]$, s can be given the fully transparent signature $\mathfrak{S}(s)$. This fact is essential to the existence of principal signatures in our type checking algorithm. Note that since singleton signatures express static equivalence information, the formation of singleton signatures is restricted to determinate modules. Thus, only determinate modules can be selfified (as in Harper and Lillibridge [10] and Leroy [13]).

Singleton signatures complicate equivalence checking, since equivalence can depend on context. For example, $\lambda s{:}[\![T]\!].[\mathtt{int}]$ and $\lambda s{:}[\![T]\!].s$ are obviously inequivalent at signature $[\![T]\!] \to [\![T]\!]$. However, using subsignatures, they can also be given the signature $\mathfrak{S}([\mathtt{int}]) \to [\![T]\!]$ and at that signature they *are* equivalent, since they return the same result when given the only permissible argument, $[\mathtt{int}]$.

As this example illustrates, the context sensitivity of equivalence provides more type equalities than would hold if equivalence were strictly context insensitive, thereby allowing the propagation of additional type information. For example, if $F : (\mathfrak{S}([\mathtt{int}]) \to [\![T]\!]) \to [\![T]\!]$, then the types $\mathsf{Typ}(F(\lambda s{:}[\![T]\!].[\mathtt{int}]))$ and $\mathsf{Typ}(F(\lambda s{:}[\![T]\!].s))$ are equal, which could not be the case under a context-insensitive regime.

A subtle technical point arises in the use of the higher-order singletons defined in Figure 5. Suppose $F : [\![T]\!] \to [\![T]\!]$.

$$\begin{array}{rcl}
\mathfrak{S}_{[\![T]\!]}(M) & \stackrel{\text{def}}{=} & \mathfrak{S}(M) \\
\mathfrak{S}_{[\![\tau]\!]}(M) & \stackrel{\text{def}}{=} & [\![\tau]\!] \\
\mathfrak{S}_1(M) & \stackrel{\text{def}}{=} & 1 \\
\mathfrak{S}_{\Pi s:\sigma_1.\sigma_2}(M) & \stackrel{\text{def}}{=} & \Pi s{:}\sigma_1.\mathfrak{S}_{\sigma_2}(M\,s) \\
\mathfrak{S}_{\Pi^{\text{gen}} s:\sigma_1.\sigma_2}(M) & \stackrel{\text{def}}{=} & \Pi^{\text{gen}} s{:}\sigma_1.\sigma_2 \\
\mathfrak{S}_{\Sigma s:\sigma_1.\sigma_2}(M) & \stackrel{\text{def}}{=} & \mathfrak{S}_{\sigma_1}(\pi_1 M) \times \\
 & & \mathfrak{S}_{\sigma_2[\pi_1 M/s]}(\pi_2 M) \\
\mathfrak{S}_{\mathfrak{S}(M')}(M) & \stackrel{\text{def}}{=} & \mathfrak{S}(M)
\end{array}$$

Figure 5: Singletons at Higher Signatures

Then $\mathfrak{S}_{[\![T]\!]\to[\![T]\!]}(F) = \Pi s{:}[\![T]\!].\mathfrak{S}(F\,s)$, which intuitively contains the modules equivalent to F: those that take members of F's domain and return the same thing that F does. Formally speaking, however, the canonical member of this signature is not F but its eta-expansion $\lambda s{:}[\![T]\!].F s$. In fact, it is not obvious that F belongs to $\mathfrak{S}_{[\![T]\!]\to[\![T]\!]}(F)$.

To ensure that F belongs to its singleton signature, our type system (following Stone and Harper [31]) includes the extensional typing rule:

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \Pi s{:}\sigma_1.\sigma_2' \quad \Gamma, s{:}\sigma_1 \vdash_{\mathsf{P}} M\,s : \sigma_2}{\Gamma \vdash_{\mathsf{P}} M : \Pi s{:}\sigma_1.\sigma_2}$$

Using this rule, F belongs to $\Pi s{:}[\![T]\!].\mathfrak{S}(F\,s)$ because it is a function and because $F s$ belongs to $\mathfrak{S}(F\,s)$. A similar extensional typing rule is provided for products.

It is possible that the need for these rules could be avoided by making higher-order singletons primitive, but we have not explored the metatheoretic implications of such a change.

2.3 Type Checking

Our type system enjoys a sound, complete, and effective type checking algorithm. Our algorithm comes in three main parts: first, an algorithm for synthesizing the principal (*i.e.,* minimal) signature of a module; second, an algorithm for checking subsignature relationships; and third, an algorithm for deciding equivalence of modules and of types.

Type checking modules then proceeds in the usual manner, by synthesizing the principal signature of a module and then checking that it is a subsignature of the intended signature. The signature synthesis algorithm is given in Appendix B, and its correctness theorems are stated below. The main judgement of signature synthesis is $\Gamma \vdash_\kappa M \Rightarrow \sigma$, which states that M's principal signature is σ and M's purity is inferred to be κ.

Subsignature checking is syntax-directed and easy to do, given an algorithm for checking module equivalence; module equivalence arises when two singleton signatures are compared for the subsignature relation. The equivalence algorithm is closely based on Stone and Harper's algorithm [31] for type constructor equivalence in the presence of singleton kinds. Space considerations preclude further discussion of this algorithm here. Full details of all these algorithms and proofs appear in the companion technical report [6].

Theorem 2.1 (Soundness) *If* $\Gamma \vdash_\kappa M \Rightarrow \sigma$ *then* $\Gamma \vdash_\kappa M : \sigma$.

Theorem 2.2 (Completeness) *If* $\Gamma \vdash_\kappa M : \sigma$ *then* $\Gamma \vdash_{\kappa'} M \Rightarrow \sigma'$ *and* $\Gamma \vdash \sigma' \leq \sigma$ *and* $\kappa' \sqsubseteq \kappa$.

Note that since the synthesis algorithm is deterministic, it follows from Theorem 2.2 that principal signatures exist. Finally, since our synthesis algorithm, for convenience, is presented in terms of inference rules, we require one more result stating that it really is an algorithm:

Theorem 2.3 (Effectiveness) *For any* Γ *and* M*, it is decidable whether there exist* σ *and* κ *such that* $\Gamma \vdash_\kappa M \Rightarrow \sigma$.

3 Strong versus Weak Sealing

Generativity is essential for providing the necessary degree of abstraction in the presence of effects. When a module has side-effects, such as the allocation of storage, abstraction may demand that types be generated in correspondence to storage allocation, in order to ensure that elements of those types relate to the local store and not the store of another instance.

Consider, for example, the symbol table example given in Figure 6. A symbol table contains a hidden type `symbol`, operations for interconverting symbols and strings, and an equality test (presumably faster than that available for strings). The implementation creates an internal hash table and defines symbols to be indices into that internal table.

The intention of this implementation is that the `Fail` exception never be raised. However, this depends on the generativity of the `symbol` type. If another instance, `SymbolTable2`, is created, and the types `SymbolTable.symbol` and `SymbolTable2.symbol` are considered equal, then `SymbolTable` could be asked to interpret indices into `SymbolTable2`'s table, thereby causing failure. Thus, it is essential that `SymbolTable.symbol` and `SymbolTable2.symbol` be considered unequal.

In our system, strong sealing (M `:>`σ) induces both opacity and generativity, thereby providing the necessary level of abstraction for stateful modules. However, in some cases opacity is desired but not generativity. For these purposes, weak sealing (M `::`σ) is provided.

The best examples of the need for weak sealing are provided by the interpretation of ML datatypes as abstract types [12]. In both Standard ML and Caml datatypes are *opaque* in the sense that their representation is not exposed.[5] Standard ML and Caml differ, however, on whether datatypes are *generative*. In the presence of applicative functors (which are absent from Standard ML) there is excellent reason for datatypes not to be generative, for otherwise datatypes could not be used within them. This would severely diminish the utility of applicative functors, particularly since in ML recursive types are provided only through the datatype mechanism.

For these reasons, strong (*i.e.,* generative) sealing is no substitute for weak (*i.e.,* applicative) sealing. Neither is weak sealing a substitute for strong. As Leroy [14] observed, in functor-free code, generativity can be simulated by what we call weak sealing. (This can be seen in our framework by observing that dynamic purity provides no extra privileges in the absence of functors.) Using functors, however, this "pseudo-generativity" provided by weak sealing can be defeated. Since a weakly sealed module is pure, it may be placed within an applicative functor and this may be bound to a variable. This functor can then be applied multiple times with no generative consequences:

[5]Indeed, the *transparent* interpretation of datatypes, which exposes their representations, presents severe typing difficulties [4].

```
signature SYMBOL_TABLE =
  sig
    type symbol
    val string_to_symbol : string -> symbol
    val symbol_to_string : symbol -> string
    val eq : symbol * symbol -> bool
  end

functor SymbolTableFun () :> SYMBOL_TABLE =
  struct
    type symbol = int

    val table : string array =
      (* allocate internal hash table *)
      Array.array (initial size, NONE)

    fun string_to_symbol x =
      (* lookup (or insert) x *) ...

    fun symbol_to_string n =
      (case Array.sub (table, n) of
         SOME x => x
       | NONE => raise (Fail "bad symbol"))

    fun eq (n1, n2) = (n1 = n2)
  end

structure SymbolTable = SymbolTableFun ()
```

Figure 6: Symbol Table Example

$$\begin{array}{l} \texttt{module}\ F : (1 \to \sigma) = \lambda\_{:}1.(M :: \sigma) \\ \texttt{module}\ st_1 = F\ \langle\rangle \\ \texttt{module}\ st_2 = F\ \langle\rangle \end{array}$$

If M is the `SymbolTable` implementation, then st_1 and st_2 provide equal `symbol` types but contain distinct hash tables, thereby breaking the implementation's abstraction requirements.

4 The Avoidance Problem

The rules of our type system (particularly rules 8, 9, and 11 from Figure 4) are careful to ensure that substituted modules are always determinate, at the expense of requiring that functor and second-projection arguments are determinate. This is necessary because the result of substituting an indeterminate module into a well-formed signature can be ill-formed. Thus, to apply a functor to an indeterminate argument, one must let-bind the argument and apply the functor to the resulting (determinate) variable.

A similar restriction is imposed by Shao [29], but Harper and Lillibridge [10] propose an alternative that softens the restriction. Harper and Lillibridge's proposal (expressed in our terms) is to include a non-dependent typing rule without a determinacy restriction:

$$\frac{\Gamma \vdash_\kappa M_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash_\kappa M_2 : \sigma_1}{\Gamma \vdash_\kappa M_1 M_2 : \sigma_2}$$

When M_2 is determinate, this rule carries the same force as our dependent rule, by exploiting singleton signatures and the contravariance of functor signatures:

$$\begin{array}{rcl} \Pi s{:}\sigma_1.\sigma_2 & \leq & \Pi s{:}\mathfrak{S}_{\sigma_1}(M_2).\sigma_2 \\ & \equiv & \Pi s{:}\mathfrak{S}_{\sigma_1}(M_2).\sigma_2[M_2/s] \\ & = & \mathfrak{S}_{\sigma_1}(M_2) \to \sigma_2[M_2/s] \end{array}$$

When M_2 is indeterminate, this rule is more expressive than our typing rule, because the application can still occur. However, to exploit this rule, the type checker must find a non-dependent supersignature that is suitable for application to M_2.

The *avoidance problem* [7, 17] is that there is no "best" way to do so. For example, consider the signature:

$$\sigma = (\llbracket T \rrbracket \to \mathfrak{S}(s)) \times \mathfrak{S}(s)$$

To obtain a supersignature of σ avoiding the variable s, we must forget that the first component is a constant function, and therefore we can only say that the second component is equal to the first component's result on some particular argument. Thus, for any type τ, we may promote σ to the supersignature:

$$\Sigma F{:}(\llbracket T \rrbracket \to \llbracket T \rrbracket).\, \mathfrak{S}(F[\tau])$$

This gives us an infinite array of choices. Any of these choices is superior to the obvious $(\llbracket T \rrbracket \to \llbracket T \rrbracket) \times \llbracket T \rrbracket$, but none of them is comparable to any other, since F is abstract. Thus, there is no minimal supersignature of σ avoiding s. The absence of minimal signatures is a problem, because it means that there is no obvious way to perform type checking.

In our type system, we circumvent the avoidance problem by requiring that the arguments of functor application and second-projection be determinate (thereby eliminating any need to find non-dependent supersignatures), and provide a let construct so that such operations can still be applied to indeterminate modules. We have shown that, as a result, our type theory does enjoy principal signatures. However, our let construct must be labelled with its result type (not mentioning the variable being bound), otherwise the avoidance problem re-arises.

This is awkward, as it essentially requires that every functor application or projection involving an indeterminate argument be labelled with its result signature. This seems likely to be unacceptable syntactic overhead in practice. Fortunately, programs can be systematically rewritten to avoid this problem, as we describe next.

4.1 The Elaborator

The avoidance problem arises when a variable is required to leave scope and there is no minimal way to do so. In our type system, we have ensured that there is always a minimal way to do so, using two means: First, in functor application and second-projection we require the argument to be determinate; then there *is* a minimal way for the variable to leave scope, because we may substitute the variable's actual value in its place. Second, in let binding we require that the programmer supply the resulting signature; that supplied signature then is trivially the minimal signature for the expression.

In practice, however, we wish to circumvent the avoidance problem without a determinacy restriction, and without requiring programmer-supplied signature annotations, as each of these lead to unacceptable awkwardness. Since we cannot provide a best signature not mentioning variables leaving scope, we instead follow Harper and Stone [12] and arrange that variables never do leave scope.

For example, consider the unannotated let expression $\mathsf{let}\ s = M_1\ \mathsf{in}\ M_2$, where $M_1 : \sigma_1$ and $M_2 : \sigma_2(s)$. If we assume that M_1 is indeterminate (otherwise the let expression can be given the minimal signature $\sigma_2(M_1)$), then we are left with the variable s leaving scope, but no minimal supersignature of $\sigma_2(s)$ not mentioning s. However, if we rewrite the let expression as the pair $\langle s = M_1, M_2 \rangle$, then we may give it the signature $\Sigma s{:}\sigma_1.\sigma_2(s)$ and no avoidance problem arises.

Similarly, the functor application $F(M)$ with $F : \Pi s{:}\sigma_1.\sigma_2$ and indeterminate $M : \sigma_1$ can be rewritten as $\langle s = M, F(s) \rangle$ and given signature $\Sigma s{:}\sigma_1.\sigma_2$.

Of course, no programmer is likely to enjoy writing code in this style any more than he or she would enjoy the restriction we are trying to avoid. Instead, we propose the use of an elaborator. This elaborator takes code written in an external language that supports unannotated lets and indeterminate functor application and second-projection, and produces code written in our type system. (For the purposes of this discussion, we refer to our type system as the "internal language.") The elaborator performs the above rewritings systematically, leaving the programmer all the convenience of the external language.

Existential Signatures Since the elaborator systematically rewrites modules in a manner that changes their signatures, it also must take responsibility for converting those modules back to their expected signature wherever required. This means that the elaborator must track which pairs are "real" and which have been invented by the elaborator to circumvent the avoidance problem.

The elaborator does so using the types. When the elaborator invents a pair to circumvent the avoidance problem, it gives its signature using $\exists$ rather than Σ. In the internal language, $\exists s{:}\sigma_1.\sigma_2$ means the same thing as $\Sigma s{:}\sigma_1.\sigma_2$, but the elaborator treats the two signatures differently: When the elaborator expects (say) a functor and encounters a $\Sigma s{:}\sigma_1.\sigma_2$, it generates a type error. However, when it encounters an $\exists s{:}\sigma_1.\sigma_2$, it extracts the σ_2 component (the elaborator's invariants ensure that it always can do so), looking for the expected functor. Roughly speaking, the elaborator treats $\exists s{:}\sigma_1.\sigma_2(s)$ as a subsignature of σ whenever $\sigma_2(s)$ is a subsignature of σ.

Formalization The elaborator is defined in terms of the five judgements given in Figure 7. The metavariables $\hat{M}$, $\hat{\sigma}$, etc., range over expressions in the external language (these are the same as the internal language's expressions, except that unannotated let is supported), and the metavariables ς and Δ range over the elaborator's signatures and contexts (the same as the internal language's, except that $\exists$ is supported, as given in Figure 7).

The main judgement is module elaboration, written $\Delta \vdash_\kappa \hat{M} \rightsquigarrow M : \varsigma$, which means that the external module $\hat{M}$ elaborates to the internal module M, which has the signature ς and purity κ. The signature, type, and term elaboration judgements are similar (except that signatures

module elaboration	$\Delta \vdash_\kappa \hat{M} \rightsquigarrow M : \varsigma$
signature coercion	$\Delta \vdash M : \varsigma \leq \sigma \rightsquigarrow M'$
existential unpacking	$M : \varsigma \overset{unpack}{\Longrightarrow} M' : \varsigma'$
signature elaboration	$\Delta \vdash \hat{\sigma} \rightsquigarrow \sigma$
type elaboration	$\Delta \vdash \hat{\tau} \rightsquigarrow \tau$
term elaboration	$\Delta \vdash \hat{e} \rightsquigarrow e : \tau$
elaborator signatures	$\varsigma ::= 1 \mid [\![T]\!] \mid [\![\tau]\!] \mid \mathfrak{S}(M) \mid \Pi s{:}\sigma.\varsigma \mid \Pi^{\text{gen}} s{:}\sigma.\varsigma \mid \Sigma s{:}\varsigma_1.\varsigma_2 \mid \exists s{:}\varsigma_1.\varsigma_2$
elaborator contexts	$\Delta ::= \epsilon \mid \Delta, s{:}\varsigma$

Figure 7: Elaborator Judgements

and types have no classifiers to generate). Two judgements are included for eliminating existentials: signature coercion is used when the desired result signature is known, unpacking is used to unpack outermost existentials when the result is not known. The signature coercion judgement is written $\Delta \vdash M : \varsigma \leq \sigma \rightsquigarrow M'$, meaning that a (determinate) module M with signature ς when coerced to signature σ becomes the (determinate) module M'. The unpacking judgement is written $M : \varsigma \overset{unpack}{\Longrightarrow} M' : \varsigma'$, meaning that $M : \varsigma$ unpacks to $M' : \varsigma'$.

Some illustrative rules of the elaborator appear in Figure 8; the complete definition is given in Appendix C. In these rules, the auxiliary operation $\overline{\cdot}$ takes elaborator signatures and contexts to internal ones by replacing all occurrences of $\exists$ with Σ.

Theorem 4.1 (Elaborator Invariants)
Suppose $\overline{\Delta} \vdash \text{ok}$. *Then:*

1. *If* $\Delta \vdash_\kappa \hat{M} \rightsquigarrow M : \varsigma$ *then* $\overline{\Delta} \vdash_\kappa M \Rightarrow \overline{\varsigma}$ *(and hence* $\overline{\Delta} \vdash_\kappa M : \overline{\varsigma}$*).*
2. *If* $\Delta \vdash M : \varsigma \leq \sigma \rightsquigarrow M'$ *and* $\overline{\Delta} \vdash_{\mathrm{P}} M : \overline{\varsigma}$ *and* $\overline{\Delta} \vdash \sigma$ sig *then* $\overline{\Delta} \vdash_{\mathrm{P}} M' : \sigma$.
3. *If* $M : \varsigma \overset{unpack}{\Longrightarrow} M' : \varsigma'$ *and* $\Gamma \vdash_{\mathrm{P}} M \Rightarrow \overline{\varsigma}$ *then* $\Gamma \vdash_{\mathrm{P}} M' \Rightarrow \overline{\varsigma'}$.
4. *If* $\Delta \vdash \hat{\sigma} \rightsquigarrow \sigma$ *then* $\overline{\Delta} \vdash \sigma$ sig.
5. *If* $\Delta \vdash \hat{\tau} \rightsquigarrow \tau$ *then* $\overline{\Delta} \vdash \tau$ type.
6. *If* $\Delta \vdash \hat{e} \rightsquigarrow e : \tau$ *then* $\overline{\Delta} \vdash e \Rightarrow \tau$ *(and hence* $\overline{\Delta} \vdash e : \tau$*).*

Rules 13 and 14 illustrate how the elaborator constructs existential signatures to account for hidden, indeterminate modules: In each of these rules, indeterminate modules are let-bound, providing variables that may be used to satisfy the determinacy requirements on existential unpacking and signature coercion (required by the invariants in Theorem 4.1) and on functor application (required by the type system). These variables must leave scope, requiring the construction of a pair that the elaborator tags with an existential signature. (Each of these rules carries a side condition that certain modules involved are indeterminate; when those conditions do not hold, less interesting rules are used to produce more precise signatures.) Rule 15 illustrates the coercion of functors, and rules 16, 17, and 18 handle elimination of existentials.

Although our elaborator serves only to deal with the avoidance problem, a realistic elaborator would also address other issues such as coercive signature matching (where a field is either dropped or made less polymorphic), `open`, type inference, datatypes, and so forth [12]. We believe our elaborator extends to cover all these issues without difficulty.

4.2 Primitive Existential Signatures

In a sense, the elaborator solves the avoidance problem by introducing existential signatures to serve in place of the non-existent minimal supersignatures not mentioning a variable. In light of this, a natural question is whether the need for an elaborator could be eliminated by making existential signatures primitive to the type system.

One natural way to govern primitive existentials is with the introduction and elimination rules:

$$\frac{\Gamma \vdash_{\mathrm{P}} M : \sigma_1 \quad \Gamma \vdash \sigma \leq \sigma_2[M/s] \quad \Gamma, s{:}\sigma_1 \vdash \sigma_2 \text{ sig}}{\Gamma \vdash \sigma \leq \exists s{:}\sigma_1.\sigma_2}$$

and

$$\frac{\Gamma, s{:}\sigma_1 \vdash \sigma_2 \leq \sigma \quad \Gamma \vdash \sigma_1 \text{ sig} \quad \Gamma \vdash \sigma \text{ sig}}{\Gamma \vdash \exists s{:}\sigma_1.\sigma_2 \leq \sigma}$$

With these rules, the avoidance problem could be solved: The least supersignature of $\sigma_2(s)$ not mentioning $s{:}\sigma_1$ would be $\exists s{:}\sigma_1.\sigma_2(s)$.

Unfortunately, these rules (particularly the first) make type checking undecidable. For example, each of the queries

$$\Pi s{:}\sigma.[\![\tau]\!] \overset{?}{\leq} \exists s'{:}\sigma.\Pi s{:}\mathfrak{S}_\sigma(s').[\![\tau']\!]$$

and

$$(\lambda s{:}\sigma.[\tau]) \overset{?}{\cong} (\lambda s{:}\sigma.[\tau']) : \exists s'{:}\sigma.\Pi s{:}\mathfrak{S}_\sigma(s').[\![T]\!]$$

holds if and only if there exists determinate $M : \sigma$ such that the types $\tau[M/s]$ and $\tau'[M/s]$ are equal. Thus, deciding subsignature or equivalence queries in the presence of existentials would be as hard as higher-order unification, which is known to be undecidable [9].

We have explored a variety of alternative formalizations of primitive singletons as well, and none has led to a type system we have been able to prove decidable.

5 Other Issues

5.1 Modules as First-Class Values

It is desirable for modules to be usable as first-class values. This is useful to make it possible to choose at run time the most efficient implementation of a signature for a particular data set (for example, sparse or dense representations of arrays). Unfortunately, fully general first-class modules present difficulties for static typing [17].

One practical approach to modules as first-class values was suggested by Mitchell, *et al.* [22], who propose that second-class modules automatically be wrapped as existential packages [23] to obtain first-class values. A similar approach to modules as first-class values is implemented in Moscow ML [28].

This existential-packaging approach to modules as first-class values is built into our language. We write the wrapping of a module as a first-class value as $\langle\!\langle M \rangle\!\rangle$ and write its

$$\frac{\begin{array}{c}\Delta \vdash_{\kappa_F} \hat{F} \rightsquigarrow F : \varsigma_F \quad s_F : \mathfrak{S}_{\varsigma_F}(s_F) \overset{unpack}{\Longrightarrow} G : \Pi s{:}\sigma_1.\varsigma_2 \quad \kappa_M \neq \mathsf{P} \\ \Delta \vdash_{\kappa_M} \hat{M} \rightsquigarrow M : \varsigma_M \quad \Delta, s_F{:}\varsigma_F, s_M{:}\varsigma_M \vdash M : \varsigma_M \leq \sigma_1 \rightsquigarrow N\end{array}}{\begin{array}{l}\Delta \vdash_{\kappa_F \sqcup \kappa_M} \hat{F}\hat{M} \rightsquigarrow \langle s_F = F, \langle s_M = M, GN\rangle\rangle \\ \qquad : \exists s_F{:}\varsigma_F.\exists s_M{:}\varsigma_M.\varsigma_2[N/s]\end{array}} \quad (13)$$

$$\frac{\begin{array}{c}\Delta \vdash_{\kappa_1} \hat{M}_1 \rightsquigarrow M_1 : \varsigma_1 \quad \kappa_1 \sqcup \kappa_2 \neq \mathsf{P} \\ \Delta, s{:}\varsigma_1 \vdash_{\kappa_2} \hat{M}_2 \rightsquigarrow M_2 : \varsigma_2\end{array}}{\begin{array}{l}\Delta \vdash_{\kappa_1 \sqcup \kappa_2} \mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2 \\ \qquad \rightsquigarrow \langle s = M_1, M_2\rangle : \exists s{:}\varsigma_1.\varsigma_2\end{array}} \quad (14)$$

$$\frac{\begin{array}{c}\Delta, s{:}\sigma_1' \vdash s : \sigma_1' \leq \sigma_1 \rightsquigarrow M \qquad \delta \sqsubseteq \delta' \\ \Delta, s{:}\sigma_1', t{:}\varsigma_2[M/s] \vdash t : \varsigma_2[M/s] \leq \sigma_2' \rightsquigarrow N\end{array}}{\Delta \vdash F : \Pi^{\delta} s{:}\sigma_1.\varsigma_2 \leq \Pi^{\delta'} s{:}\sigma_1'.\sigma_2' \rightsquigarrow \lambda s{:}\sigma_1'.\,\mathsf{let}\ t = FM\ \mathsf{in}\ (N : \sigma_2')} \quad (15)$$

$$\frac{\Delta \vdash \pi_2 M : \varsigma_2[\pi_1 M/s] \leq \sigma \rightsquigarrow N}{\Delta \vdash M : \exists s{:}\varsigma_1.\varsigma_2 \leq \sigma \rightsquigarrow N} \quad (16)$$

$$\frac{\pi_2 M : \varsigma_2[\pi_1 M/s] \overset{unpack}{\Longrightarrow} M' : \varsigma}{M : \exists s{:}\varsigma_1.\varsigma_2 \overset{unpack}{\Longrightarrow} M' : \varsigma} \quad (17)$$

$$\frac{\varsigma \text{ not an existential}}{M : \varsigma \overset{unpack}{\Longrightarrow} M : \varsigma} \quad (18)$$

Figure 8: Illustrative Elaboration Rules

type as $\langle\!|\sigma|\!\rangle$. Elimination of wrapped modules (as for existentials) is performed using a closed-scope unpacking construct. These may be defined as follows:

$$\begin{array}{lcl}\langle\!|\sigma|\!\rangle & \overset{\text{def}}{=} & \forall\alpha.(\sigma \to \alpha) \to \alpha \\ \langle\!|M|\!\rangle & \overset{\text{def}}{=} & \Lambda\alpha.\lambda f{:}(\sigma \to \alpha).f\ M \\ \text{open}\ s{:}\sigma = e\ \text{in}\ (e' : \tau) & \overset{\text{def}}{=} & e\ \tau\ (\Lambda s{:}\sigma.e')\end{array}$$

(Compare the definition of $\langle\!|\sigma|\!\rangle$ with the standard encoding of the existential type $\exists\beta.\tau$ as $\forall\alpha.(\forall\beta.\,\tau \to \alpha) \to \alpha$.)

In fact, our language is more powerful than one employing only a wrapping approach to modules as first-class values. The main limitation of existentially wrapped modules is the closed-scope elimination construct. That is, in "open $s{:}\sigma = e$ in $(e' : \tau)$", the result type τ may not mention s. Among other consequences, this means functions over wrapped modules may not be dependent; that is, the result type may not mention the argument. That limitation is not shared by our language, since functions over modules can use unwrapped arguments and thereby be given the type $\Pi s{:}\sigma.\tau(s)$ instead of $\langle\!|\sigma|\!\rangle \to \tau$.

Since the only form of subtyping our language provides is on signatures (not types), the subsignature relationship $\sigma_1 \leq \sigma_2$ does not induce a subtyping relationship on the wrapped signatures $\langle\!|\sigma_1|\!\rangle$ and $\langle\!|\sigma_2|\!\rangle$. One could obtain this by adding a subtyping judgment for types and the natural rules:

$$\frac{\Gamma \vdash \sigma_2 \leq \sigma_1 \quad \Gamma, s{:}\sigma_2 \vdash \tau_1 \leq \tau_2\ \text{type}}{\Gamma \vdash \Pi s{:}\sigma_1.\tau_1 \leq \Pi s{:}\sigma_2.\tau_2\ \text{type}} \qquad \frac{\Gamma \vdash \tau \leq \tau'\ \text{type}}{\Gamma \vdash [\![\tau]\!] \leq [\![\tau']\!]}$$

However, it is important *not* to add these rules, because this change would make the subsignature problem undecidable. This can be proven by reduction from Lillibridge's "simple type system" [17, section 10.6], the undecidability of which is shown by reduction from Pierce's rowing machines [26].

Intuitively, decidability fails with these rules because the subtyping and subsignature problems become mutually recursive. This forces the termination metric for subsignature checking to account for the size of atomic signatures (which can otherwise be neglected), which in turns means that substitution (or rebinding of variables to carry singleton signatures, which amounts to the same thing) can increase the metric.

5.2 Compilation

A dynamic implementation of our language (as opposed to a static type checker) requires few new ideas. We may compile our internal language simply by, first, deleting all sealing; second, phase-splitting modules into separate static and dynamic components [11]; and, third, eliminating resulting singleton kinds [3]. The end result of this process is a program in F_ω [8] (plus product kinds and recursion on terms), which is certainly implementable. Space considerations preclude further discussion of the details here.

5.3 Syntactic Principal Signatures

It has been argued for reasons related to separate compilation that principal signatures should be expressible in the syntax available to the programmer. This provides the strongest support for separate compilation, because a programmer can break a program at any point and write an interface that expresses all the information the compiler could have determined at that point. Such strong support does not appear to be vital in practice, since systems such as Objective Caml and Standard ML of New Jersey's higher-order modules have been used successfully for some time without principal signatures at all, but it is nevertheless a desirable property.

Our type system (*i.e.,* the internal language) does provide syntactic principal signatures, since principal signatures exist, and all the syntax is available to the programmer. However, the elaborator's *external* language does not provide syntax for the existential signatures that can appear in elaborator signatures, which should be thought of as the principal signatures of external modules. Thus, we can say that our basic type system provides syntactic principal signatures, but our external language does not.

In an external language where the programmer is permitted to write existential signatures, elaborating code such as:

$$(\lambda s'{:}(\exists s{:}\sigma_1.\sigma_2)\ldots)M$$

requires the elaborator to decide whether M can be coerced to belong to $\exists s{:}\sigma_1.\sigma_2$, which in turn requires the elaborator to produce a $M' : \sigma_1$ such that $M : \sigma_2[M'/s]$. Determining whether any such M' exists requires the elaborator to solve an undecidable higher-order unification problem: if $\sigma_2 =$

$\mathfrak{S}([\tau]) \to \mathfrak{S}([\tau'])$ and $M = \lambda t{:}[\![T]\!].t$, then $M : \sigma_2[M'/s]$ if and only if $\tau[M'/s]$ and $\tau'[M'/s]$ are equal.

Thus, to allow programmer-specified existential signatures in the greatest possible generality would make elaboration undecidable. Partial measures may be possible, but we will not discuss any here.

6 Related Work

Harper, Mitchell and Moggi [11] pioneered the theory of *phase separation*, which is fundamental to achieving maximal type propagation in higher-order module systems. Their non-standard equational rules, which identify higher-order modules with primitive "phase-split" ones, are similar in spirit to, though different in detail from, our notion of static module equivalence. One may view their system as a subsystem of ours in which there is no abstraction mechanism (and consequently all modules are determinate).

MacQueen and Tofte [19] proposed a higher-order module extension to the original Definition of Standard ML [20], which was implemented in the Standard ML of New Jersey compiler. Their semantics involves a two-phase elaboration process, in which higher-order functors are re-elaborated at each application to take advantage of additional information about their arguments. This advantage is balanced by the disadvantage of inhibiting type propagation in the presence of separate compilation since functors that are compiled separately from their applications cannot be re-elaborated. A more thorough comparison is difficult because MacQueen and Tofte employ a stamp-based semantics, which is difficult to transfer to a type-theoretic setting.

Focusing on controlled abstraction, but largely neglecting higher-order modules, Harper and Lillibridge [10] and Leroy [13, 15] introduced the closely related concepts of *translucent sums* and *manifest types.* These mechanisms served as the basis of the module system in the revised Definition of Standard ML 1997 [21], and Harper and Stone [12] have formalized the elaboration of Standard ML 1997 programs into a translucent sums calculus. To deal with the avoidance problem, Harper and Stone rely on elaborator mechanisms similar to ours. The Harper and Stone language can be viewed as a subsystem of ours in which all functors are generative and only strong sealing is supported.

Leroy introduced the notion of an *applicative functor* [14], which enables one to give fully transparent signatures for many higher-order functors. Leroy's formalism defined determinacy by a syntactic restriction that functor applications appearing in type paths must be in named form. On one hand, this restriction provides a weak form of structure sharing in the sense that the equivalence of `F(X).t` and `F(Y).t` implies that `X` and `Y` are the same structure. On the other hand, the restriction prevents the system from capturing the full equational theory of higher-order functors, since not all equations can be expressed in named form [3].

Together, manifest types and applicative functors form the basis of the module system of the Objective Caml dialect of ML [25]. The manifest type formalism, like the translucent sum formalism, does not address the avoidance problem, and consequently it lacks principal signatures. Aside from this fact (and disregarding Objective Caml's signature fields in structures, which makes typechecking undecidable [17]), the module language of Objective Caml can be viewed as essentially a subsystem of Russo's system (below).

More recently, Russo [27] has proposed a type system for applicative functors that generalizes Leroy's in much the same way we do, by abandoning the named form restriction. We adopt his use of existential signatures to address hidden components, although Russo simultaneously uses existentials to model generativity, which we do not. Russo has implemented his system as an experimental extension to the Moscow ML compiler [24], which already supports SML-style generative functors. One can view Moscow ML's module system as a subsystem of ours in which weak sealing is the only available form of abstraction and in which all modules are considered dynamically pure. As we observed in Section 3, a system with only dynamically pure modules can provide a faithful account of generativity only in the absence of functors. Thus, the introduction of applicative functors into Moscow ML has the effect of making its generative functors weaker, since one can defeat generativity by eta-expanding a generative functor into an applicative one.

Shao [29] proposes another type system for modules supporting both applicative and generative functors. Shao's system, in contrast to Russo's, may be viewed as a subsystem of ours based exclusively on strong sealing instead of weak sealing. As we observed in Section 3, a consequence of this is that the bodies of applicative functors may not contain any opaque substructures, such as (opaque) datatypes. Shao's system, like ours (recall Section 4), circumvents the avoidance problem by restricting functor application and projection to determinate arguments (which must be in named form in his system), and by eliminating implicit subsumption (in essence, this requires let expressions to be annotated, as in our system). We conjecture that our elaboration techniques could be applied to Shao's system to lift these restrictions in his system as well (at the expense of syntactic principal signatures).

7 Conclusion

Type systems for first-order module systems are reasonably well understood. In contrast, previous work on type-theoretic, higher-order modules has left that field in a fragmented state, with various competing designs and no clear statement of the trade-offs (if any) between those designs. This state of the field has made it difficult to choose one design over another, and has left the erroneous impression of trade-offs that do not actually exist. For example, no previous design supports both (undefeatable) generativity and applicative functors with opaque subcomponents.

Our language seeks to unify the field by providing a practical type system for higher-order modules that simultaneously supports the key functionality of preceding module systems. In the process we dispel some misconceptions, such as a trade-off between fully expressive generative and applicative functors, thereby eliminating some dilemmas facing language designers.

It is our hope that our type system will provide a jumping-off point for future work on type-theoretic module systems. For instance, it is important to note that some trade-offs still remain: between full propagation of type information and weak structure sharing via named form (Section 6); and between syntactic signatures, programming convenience, and decidable type checking (Section 5.3). It is too early to say for certain whether these trade-offs are essential, or whether they too can be avoided by additional research.

References

[1] David R. Aspinall. *Type Systems for Modular Programs and Specifications*. PhD thesis, Edinburgh University, Edinburgh, Scotland, December 1997.

[2] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalso, and Greg Nelson. The Modula-3 type system. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 202–212, Austin, TX, January 1989.

[3] Karl Crary. Sound and complete elimination of singleton kinds. In *Third Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, September 2000. Extended version published as CMU technical report CMU-CS-00-104.

[4] Karl Crary, Robert Harper, Perry Cheng, Leaf Petersen, and Chris Stone. Transparent and opaque interpretations of datatypes. Technical Report CMU–CS–98–177, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, November 1998. (First appeared as message to `types` list, November 13, 1998.).

[5] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 50–63, Atlanta, GA, 1999. ACM SIGPLAN.

[6] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. Technical Report CMU-CS-02-122, School of Computer Science, Carnegie Mellon University, March 2002.

[7] Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193:75–96, 1998.

[8] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[9] Warren D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[10] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[11] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[12] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. MIT Press, 2000.

[13] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994.

[14] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of POPL '95: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–153, San Francisco, CA, January 1995.

[15] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[16] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[17] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1996.

[18] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[19] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald T. Sannella, editor, *Programming Languages and Systems — ESOP '94*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

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

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

[22] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Eighteenth ACM Symposium on Principles of Programming Languages*, January 1991.

[23] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[24] Moscow ML. `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[25] Objective Caml. `http://www.ocaml.org`.

[26] Benjamin C. Pierce. Bounded quantification is undecidable. In *Theoretical Aspects of Object-Oriented Programming*, pages 427–459. The MIT Press, 1994.

[27] Claudio Russo. *Types for Modules*. PhD thesis, Edinburgh University, Edinburgh, Scotland, 1998. LFCS Thesis ECS–LFCS–98–389.

[28] Claudio V. Russo. First-class modules for Standard ML. *Nordic Journal of Computing*, 7(4):348–374, 2000.

[29] Zhong Shao. Transparent modules with fully syntactic signatures. In *International Conference on Functional Programming*, pages 220–232, Paris, France, September 1999.

[30] Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, August 2000.

[31] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 214–227, Boston, January 2000.

[32] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer-Verlag, 1983.

Notes on Appendices

Appendix A gives the type system for our module calculus.

Appendix B gives the module typechecking and principal signature synthesis judgements that form the core of our typechecking algorithm. We omit the term typechecking ($\Gamma \vdash e \Leftarrow \tau$) and unique type synthesis judgements ($\Gamma \vdash e \Rightarrow \tau$) for space reasons; they are fully detailed in the companion technical report [6].

Appendix C gives the signature coercion and module elaboration judgements that form the core of our elaboration algorithm. We omit the signature elaboration, type elaboration, and term elaboration judgements (see Figure 7) for space reasons; they also are fully detailed in the companion technical report.

A Type System

Well-formed contexts: $\Gamma \vdash \mathrm{ok}$

$$\frac{}{\epsilon \vdash \mathrm{ok}} \qquad \frac{\Gamma \vdash \sigma \ \mathrm{sig} \quad s \notin \mathrm{dom}(\Gamma)}{\Gamma, s{:}\sigma \vdash \mathrm{ok}}$$

Well-formed types: $\Gamma \vdash \tau \ \mathrm{type}$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash \mathsf{Typ}\, M \ \mathrm{type}} \qquad \frac{\Gamma, s{:}\sigma \vdash \tau \ \mathrm{type}}{\Gamma \vdash \Pi s{:}\sigma.\tau \ \mathrm{type}} \qquad \frac{\Gamma \vdash \tau' \ \mathrm{type} \quad \Gamma \vdash \tau'' \ \mathrm{type}}{\Gamma \vdash \tau' \times \tau'' \ \mathrm{type}}$$

Type equivalence: $\Gamma \vdash \tau_1 \equiv \tau_2$

$$\frac{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]}{\Gamma \vdash \tau_1 \equiv \tau_2} \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 \quad \Gamma, s{:}\sigma_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \Pi s{:}\sigma_1.\tau_1 \equiv \Pi s{:}\sigma_2.\tau_2} \qquad \frac{\Gamma \vdash \tau_1' \equiv \tau_2' \quad \Gamma \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash \tau_1' \times \tau_1'' \equiv \tau_2' \times \tau_2''}$$

Well-formed terms: $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash_{\kappa} M : [\![\tau]\!]}{\Gamma \vdash \mathsf{Val}\, M : \tau} \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma \quad \Gamma, s{:}\sigma \vdash e : \tau \quad \Gamma \vdash \tau \ \mathrm{type}}{\Gamma \vdash \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) : \tau}$$

$$\frac{\Gamma, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f(s{:}\sigma){:}\tau.e : \Pi s{:}\sigma.\tau} \qquad \frac{\Gamma \vdash e : \Pi s{:}\sigma.\tau \quad \Gamma \vdash_{\mathtt{P}} M : \sigma}{\Gamma \vdash e\, M : \tau[M/s]} \qquad \frac{\Gamma \vdash e' : \tau' \quad \Gamma \vdash e'' : \tau''}{\Gamma \vdash \langle e', e'' \rangle : \tau' \times \tau''} \qquad \frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_1 e : \tau'} \qquad \frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_2 e : \tau''}$$

Well-formed signatures: $\Gamma \vdash \sigma \ \mathrm{sig}$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \ \mathrm{sig}} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \ \mathrm{sig}} \qquad \frac{\Gamma \vdash \tau \ \mathrm{type}}{\Gamma \vdash [\![\tau]\!] \ \mathrm{sig}} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M) \ \mathrm{sig}} \qquad \frac{\Gamma, s{:}\sigma' \vdash \sigma'' \ \mathrm{sig}}{\Gamma \vdash \Pi^{\delta} s{:}\sigma'.\sigma'' \ \mathrm{sig}} \qquad \frac{\Gamma, s{:}\sigma' \vdash \sigma'' \ \mathrm{sig}}{\Gamma \vdash \Sigma s{:}\sigma'.\sigma'' \ \mathrm{sig}}$$

Signature equivalence: $\Gamma \vdash \sigma_1 \equiv \sigma_2$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \equiv 1} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \equiv [\![T]\!]} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \equiv [\![\tau_2]\!]} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \equiv \mathfrak{S}(M_2)}$$

$$\frac{\Gamma \vdash \sigma_2' \equiv \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Pi^{\delta} s{:}\sigma_1'.\sigma_1'' \equiv \Pi^{\delta} s{:}\sigma_2'.\sigma_2''} \qquad \frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \equiv \Sigma s{:}\sigma_2'.\sigma_2''}$$

Signature subtyping: $\Gamma \vdash \sigma_1 \leq \sigma_2$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \leq 1} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \leq [\![T]\!]} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \leq [\![\tau_2]\!]} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M) \leq [\![T]\!]} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \leq \mathfrak{S}(M_2)}$$

$$\frac{\Gamma \vdash \sigma_2' \leq \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \ \mathrm{sig} \quad \delta_1 \sqsubseteq \delta_2}{\Gamma \vdash \Pi^{\delta_1} s{:}\sigma_1'.\sigma_1'' \leq \Pi^{\delta_2} s{:}\sigma_2'.\sigma_2''} \qquad \frac{\Gamma \vdash \sigma_1' \leq \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_2'' \ \mathrm{sig}}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \leq \Sigma s{:}\sigma_2'.\sigma_2''}$$

Well-formed modules: $\Gamma \vdash_{\kappa} M : \sigma$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s)} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash \tau \ \mathrm{type}}{\Gamma \vdash_{\mathtt{P}} [\tau] : [\![\tau]\!]} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \ \mathrm{type}}{\Gamma \vdash_{\mathtt{P}} [e : \tau] : [\![\tau]\!]} \qquad \frac{\Gamma, s{:}\sigma' \vdash_{\kappa} M : \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_{\kappa} \lambda s{:}\sigma'.M : \Pi s{:}\sigma'.\sigma''}$$

$$\frac{\Gamma, s{:}\sigma' \vdash_{\kappa} M : \sigma'' \quad \Gamma, s{:}\sigma' \vdash \sigma'' \ \mathrm{sig}}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma'.M : \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash_{\kappa} F : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_{\kappa} F M : \sigma''[M/s]} \qquad \frac{\Gamma \vdash_{\kappa} F : \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} F M : \sigma''[M/s]}$$

$$\frac{\Gamma \vdash_{\kappa} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\kappa} M'' : \sigma''}{\Gamma \vdash_{\kappa} \langle s = M', M'' \rangle : \Sigma s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash_{\kappa} M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_{\kappa} \pi_1 M : \sigma'} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''[\pi_1 M/s]} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash_{\mathtt{P}} M : \mathfrak{S}(M)}$$

$$\frac{\Gamma \vdash_{\kappa} M : \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} (M :: \sigma) : \sigma} \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma}{\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma} \qquad \frac{\Gamma, s{:}\sigma' \vdash_{\mathtt{P}} M s : \sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\rho}{\Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash_{\mathtt{P}} \pi_1 M : \sigma' \quad \Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''}{\Gamma \vdash_{\mathtt{P}} M : \sigma' \times \sigma''}$$

$$\frac{\Gamma \vdash_{\kappa} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\kappa} M'' : \sigma \quad \Gamma \vdash \sigma \ \mathrm{sig}}{\Gamma \vdash_{\kappa} \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) : \sigma} \qquad \frac{\Gamma \vdash_{\kappa'} M : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash_{\kappa} M : \sigma}$$

Module equivalence: $\Gamma \vdash M_1 \cong M_2 : \sigma$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : \sigma}{\Gamma \vdash M \cong M : \sigma} \qquad \frac{\Gamma \vdash M_2 \cong M_1 : \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma \quad \Gamma \vdash M_2 \cong M_3 : \sigma}{\Gamma \vdash M_1 \cong M_3 : \sigma} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M : [\![T]\!]}{\Gamma \vdash [\mathsf{Typ}\, M] \cong M : [\![T]\!]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M_1 : 1 \quad \Gamma \vdash_{\mathsf{P}} M_2 : 1}{\Gamma \vdash M_1 \cong M_2 : 1} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M_1 : [\![\tau]\!] \quad \Gamma \vdash_{\mathsf{P}} M_2 : [\![\tau]\!]}{\Gamma \vdash M_1 \cong M_2 : [\![\tau]\!]} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M_1 : \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathsf{P}} M_2 : \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma''}{\Gamma \vdash M_1 \cong M_2 : \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma''}$$

$$\frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash M_1 \cong M_2 : \sigma''}{\Gamma \vdash \lambda s{:}\sigma_1'.M_1 \cong \lambda s{:}\sigma_2'.M_2 : \Pi s{:}\sigma_1'.\sigma''} \qquad \frac{\Gamma \vdash F_1 \cong F_2 : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash M_1 \cong M_2 : \sigma'}{\Gamma \vdash F_1 M_1 \cong F_2 M_2 : \sigma''[M_1/s]}$$

$$\frac{\Gamma \vdash M_1' \cong M_2' : \sigma' \quad \Gamma, s{:}\sigma' \vdash M_1'' \cong M_2'' : \sigma''}{\Gamma \vdash \langle s = M_1', M_1'' \rangle \cong \langle s = M_2', M_2'' \rangle : \Sigma s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma'} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''[\pi_1 M_1/s]}$$

$$\frac{\Gamma, s{:}\sigma' \vdash M_1 s \cong M_2 s : \sigma'' \quad \Gamma \vdash_{\mathsf{P}} M_1 : \Pi s{:}\sigma'.\rho_1 \quad \Gamma \vdash_{\mathsf{P}} M_2 : \Pi s{:}\sigma'.\rho_2}{\Gamma \vdash M_1 \cong M_2 : \Pi s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma' \quad \Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''}{\Gamma \vdash M_1 \cong M_2 : \sigma' \times \sigma''}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' : \sigma \quad \Gamma \vdash \sigma \ \mathsf{sig}}{\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \cong M''[M'/s] : \sigma} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M_1 : \mathfrak{S}(M_2)}{\Gamma \vdash M_1 \cong M_2 : \mathfrak{S}(M_2)} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma}$$

B Typechecking Algorithm

Module typechecking: $\Gamma \vdash_\kappa M \Leftarrow \sigma$

$$\frac{\Gamma \vdash_\kappa M \Rightarrow \sigma' \quad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash_\kappa M \Leftarrow \sigma}$$

Principal signature synthesis: $\Gamma \vdash_\kappa M \Rightarrow \sigma$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathsf{P}} s \Rightarrow \mathfrak{S}_{\Gamma(s)}(s)} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathsf{P}} \langle\rangle \Rightarrow 1} \qquad \frac{\Gamma \vdash \tau \ \mathrm{type}}{\Gamma \vdash_{\mathsf{P}} [\tau] \Rightarrow \mathfrak{S}([\tau])} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash_{\mathsf{P}} [e : \tau] \Rightarrow [\![\tau]\!]} \qquad \frac{\Gamma, s{:}\sigma' \vdash_\kappa M \Rightarrow \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma'.M \Rightarrow \Pi s{:}\sigma'.\sigma''}$$

$$\frac{\Gamma, s{:}\sigma' \vdash_\kappa M \Rightarrow \sigma'' \quad \mathtt{S} \sqsubseteq \kappa}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma'.M \Rightarrow \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma''} \qquad \frac{\Gamma \vdash_\kappa F \Rightarrow \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathsf{P}} M \Leftarrow \sigma'}{\Gamma \vdash_\kappa F M \Rightarrow \sigma''[M/s]} \qquad \frac{\Gamma \vdash_\kappa F \Rightarrow \Pi^{\mathrm{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathsf{P}} M \Leftarrow \sigma'}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} F M \Rightarrow \sigma''[M/s]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M' \Rightarrow \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' \Rightarrow \sigma''}{\Gamma \vdash_{\mathsf{P}} \langle s = M', M'' \rangle \Rightarrow \sigma' \times \sigma''[M'/s]} \qquad \frac{\Gamma \vdash_{\kappa'} M' \Rightarrow \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\kappa''} M'' \Rightarrow \sigma'' \quad \kappa' \sqcup \kappa'' \neq \mathtt{P}}{\Gamma \vdash_{\kappa' \sqcup \kappa''} \langle s = M', M'' \rangle \Rightarrow \Sigma s{:}\sigma'.\sigma''}$$

$$\frac{\Gamma \vdash_\kappa M \Rightarrow \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_\kappa \pi_1 M \Rightarrow \sigma'} \qquad \frac{\Gamma \vdash_{\mathsf{P}} M \Rightarrow \sigma' \times \sigma''}{\Gamma \vdash_{\mathsf{P}} \pi_2 M \Rightarrow \sigma''} \qquad \frac{\Gamma \vdash_\kappa M \Leftarrow \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} M :: \sigma \Rightarrow \sigma} \qquad \frac{\Gamma \vdash_\kappa M \Leftarrow \sigma}{\Gamma \vdash_{\mathtt{W}} M :> \sigma \Rightarrow \sigma}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M' \Rightarrow \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' \Leftarrow \sigma \quad \Gamma \vdash \sigma \ \mathsf{sig}}{\Gamma \vdash_{\mathsf{P}} \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \Rightarrow \mathfrak{S}_\sigma(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma))} \qquad \frac{\Gamma \vdash_{\kappa'} M' \Rightarrow \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\kappa''} M'' \Leftarrow \sigma \quad \Gamma \vdash \sigma \ \mathsf{sig} \quad \kappa' \sqcup \kappa'' \neq \mathtt{P}}{\Gamma \vdash_{\kappa' \sqcup \kappa''} \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \Rightarrow \sigma}$$

C Elaboration Rules

Existential unpacking: $M : \varsigma \overset{unpack}{\Longrightarrow} M' : \varsigma'$

$$\frac{\pi_2 M : \varsigma_2[\pi_1 M/s] \overset{unpack}{\Longrightarrow} N : \varsigma}{M : \exists s{:}\varsigma_1.\varsigma_2 \overset{unpack}{\Longrightarrow} N : \varsigma} \qquad \frac{\varsigma \text{ not an existential}}{M : \varsigma \overset{unpack}{\Longrightarrow} M : \varsigma}$$

Signature coercion: $\Delta \vdash M : \varsigma \leq \sigma \rightsquigarrow N$

$$\frac{}{\Delta \vdash M : 1 \leq 1 \rightsquigarrow M} \qquad \frac{}{\Delta \vdash M : [\![T]\!] \leq [\![T]\!] \rightsquigarrow M} \qquad \frac{\overline{\Delta} \vdash \tau_1 \equiv \tau_2}{\Delta \vdash M : [\![\tau_1]\!] \leq [\![\tau_2]\!] \rightsquigarrow M}$$

$$\frac{}{\Delta \vdash M : \mathfrak{S}(N) \leq [\![T]\!] \rightsquigarrow M} \qquad \frac{\overline{\Delta} \vdash N_1 \cong N_2 : [\![T]\!]}{\Delta \vdash M : \mathfrak{S}(N_1) \leq \mathfrak{S}(N_2) \rightsquigarrow M} \qquad \frac{\Delta \vdash \pi_2 M : \varsigma_2[\pi_1 M/s] \leq \sigma \rightsquigarrow N}{\Delta \vdash M : \exists s{:}\varsigma_1.\varsigma_2 \leq \sigma \rightsquigarrow N}$$

$$\frac{\Delta, s{:}\sigma_1' \vdash s : \sigma_1' \leq \sigma_1 \rightsquigarrow M \quad \Delta, s{:}\sigma_1', t{:}\varsigma_2[M/s] \vdash t : \varsigma_2[M/s] \leq \sigma_2' \rightsquigarrow N \quad \delta \sqsubseteq \delta'}{\Delta \vdash F : \Pi^{\delta} s{:}\sigma_1.\varsigma_2 \leq \Pi^{\delta'} s{:}\sigma_1'.\sigma_2' \rightsquigarrow \lambda s{:}\sigma_1'.\,\mathsf{let}\ t = FM\ \mathsf{in}\ (N : \sigma_2')}$$

$$\frac{\Delta \vdash \pi_1 M : \varsigma_1 \leq \sigma_1 \rightsquigarrow N_1 \quad \Delta \vdash \pi_2 M : \varsigma_2[\pi_1 M/s] \leq \sigma_2[N_1/s] \rightsquigarrow N_2}{\Delta \vdash M : \Sigma s{:}\varsigma_1.\varsigma_2 \leq \Sigma s{:}\sigma_1.\sigma_2 \rightsquigarrow \langle N_1, N_2\rangle}$$

Module elaboration: $\Delta \vdash_{\kappa} \hat{M} \rightsquigarrow M : \varsigma$

$$\frac{}{\Delta \vdash_{\mathtt{P}} s \rightsquigarrow \mathfrak{H}_{\Delta(s)}(s)} \qquad \frac{}{\Delta \vdash_{\mathtt{P}} \langle\rangle \rightsquigarrow \langle\rangle : 1} \qquad \frac{\Delta \vdash \hat{\tau} \rightsquigarrow \tau}{\Delta \vdash_{\mathtt{P}} [\hat{\tau}] \rightsquigarrow [\tau] : \mathfrak{H}([\tau])} \qquad \frac{\Delta \vdash \hat{e} \rightsquigarrow e : \tau}{\Delta \vdash_{\mathtt{P}} [\hat{e}] \rightsquigarrow [e : \tau] : [\![\tau]\!]}$$

$$\frac{\Delta \vdash \hat{\sigma}_1 \rightsquigarrow \sigma_1 \quad \Delta, s{:}\sigma_1 \vdash_{\kappa} M \rightsquigarrow N : \varsigma_2 \quad \kappa \sqsubseteq \mathtt{D}}{\Delta \vdash_{\kappa} \lambda s{:}\hat{\sigma}_1.\hat{M} \rightsquigarrow \lambda s{:}\sigma_1.N : \Pi s{:}\sigma_1.\varsigma_2} \qquad \frac{\Delta \vdash \hat{\sigma}_1 \rightsquigarrow \sigma_1 \quad \Delta, s{:}\sigma_1 \vdash_{\kappa} M \rightsquigarrow N : \varsigma_2 \quad \mathtt{S} \sqsubseteq \kappa}{\Delta \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\hat{\sigma}_1.\hat{M} \rightsquigarrow \lambda s{:}\sigma_1.N : \Pi^{\mathrm{gen}} s{:}\sigma_1.\varsigma_2}$$

$$\frac{\Delta \vdash_{\mathtt{P}} \hat{F} \rightsquigarrow F : \varsigma_F \quad F : \varsigma_F \overset{unpack}{\Longrightarrow} G : \Pi s{:}\sigma_1.\varsigma_2 \quad \Delta \vdash_{\mathtt{P}} \hat{M} \rightsquigarrow M : \varsigma \quad \Delta \vdash M : \varsigma \leq \sigma_1 \rightsquigarrow N}{\Delta \vdash_{\mathtt{P}} \hat{F}\hat{M} \rightsquigarrow GN : \varsigma_2[N/s]}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\kappa} \hat{F} \rightsquigarrow F : \varsigma_F \qquad s_F : \mathfrak{H}_{\varsigma_F}(s_F) \overset{unpack}{\Longrightarrow} G : \Pi s{:}\sigma_1.\varsigma_2 \\ \Delta \vdash_{\mathtt{P}} \hat{M} \rightsquigarrow M : \varsigma \qquad \Delta, s_F{:}\varsigma_F \vdash M : \varsigma \leq \sigma_1 \rightsquigarrow N \qquad \kappa \neq \mathtt{P}\end{array}}{\Delta \vdash_{\kappa} \hat{F}\hat{M} \rightsquigarrow \langle s_F = F, GN\rangle : \exists s_F{:}\varsigma_F.\varsigma_2[N/s]}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\kappa_F} \hat{F} \rightsquigarrow F : \varsigma_F \qquad s_F : \mathfrak{H}_{\varsigma_F}(s_F) \overset{unpack}{\Longrightarrow} G : \Pi s{:}\sigma_1.\varsigma_2 \\ \Delta \vdash_{\kappa_M} \hat{M} \rightsquigarrow M : \varsigma_M \qquad \Delta, s_F{:}\varsigma_F, s_M{:}\varsigma_M \vdash s_M : \varsigma_M \leq \sigma_1 \rightsquigarrow N \qquad \kappa_M \neq \mathtt{P}\end{array}}{\Delta \vdash_{\kappa_F \sqcup \kappa_M} \hat{F}\hat{M} \rightsquigarrow \langle s_F = F, \langle s_M = M, GN\rangle\rangle : \exists s_F{:}\varsigma_F.\exists s_M{:}\varsigma_M.\varsigma_2[N/s]}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\kappa_F} \hat{F} \rightsquigarrow F : \varsigma_F \qquad s_F : \mathfrak{H}_{\varsigma_F}(s_F) \overset{unpack}{\Longrightarrow} G : \Pi^{\mathrm{gen}} s{:}\sigma_1.\varsigma_2 \\ \Delta \vdash_{\kappa_M} \hat{M} \rightsquigarrow M : \varsigma_M \qquad \Delta, s_F{:}\varsigma_F, s_M{:}\varsigma_M \vdash s_M : \varsigma_M \leq \sigma_1 \rightsquigarrow N\end{array}}{\Delta \vdash_{\kappa_F \sqcup \kappa_M \sqcup \mathtt{S}} \hat{F}\hat{M} \rightsquigarrow \langle s_F = F, \langle s_M = M, GN\rangle\rangle : \exists s_F{:}\varsigma_F.\exists s_M{:}\varsigma_M.\varsigma_2[N/s]}$$

$$\frac{\Delta \vdash_{\mathtt{P}} \hat{M}_1 \rightsquigarrow M_1 : \varsigma_1 \quad \Delta, s{:}\varsigma_1 \vdash_{\mathtt{P}} \hat{M}_2 \rightsquigarrow M_2 : \varsigma_2}{\Delta \vdash_{\mathtt{P}} \langle s = \hat{M}_1, \hat{M}_2\rangle \rightsquigarrow \langle s = M_1, M_2\rangle : \varsigma_1 \times \varsigma_2[M_1/s]} \qquad \frac{\Delta \vdash_{\kappa_1} \hat{M}_1 \rightsquigarrow M_1 : \varsigma_1 \quad \Delta, s{:}\varsigma_1 \vdash_{\kappa_2} \hat{M}_2 \rightsquigarrow M_2 : \varsigma_2 \quad \kappa_1 \sqcup \kappa_2 \neq \mathtt{P}}{\Delta \vdash_{\kappa_1 \sqcup \kappa_2} \langle s = \hat{M}_1, \hat{M}_2\rangle \rightsquigarrow \langle s = M_1, M_2\rangle : \Sigma s{:}\varsigma_1.\varsigma_2}$$

$$\frac{\Delta \vdash_{\mathtt{P}} \hat{M} \rightsquigarrow M : \varsigma \quad M : \varsigma \overset{unpack}{\Longrightarrow} N : \varsigma_1 \times \varsigma_2}{\Delta \vdash_{\mathtt{P}} \pi_1 \hat{M} \rightsquigarrow \pi_1 N : \varsigma_1} \qquad \frac{\Delta \vdash_{\kappa} \hat{M} \rightsquigarrow M : \varsigma \quad s : \mathfrak{H}_{\varsigma}(s) \overset{unpack}{\Longrightarrow} N : \varsigma_1 \times \varsigma_2 \quad \kappa \neq \mathtt{P}}{\Delta \vdash_{\kappa} \pi_1 \hat{M} \rightsquigarrow \langle s = M, \pi_1 N\rangle : \exists s{:}\varsigma.\varsigma_1}$$

$$\frac{\Delta \vdash_{\mathtt{P}} \hat{M} \rightsquigarrow M : \varsigma \quad M : \varsigma \overset{unpack}{\Longrightarrow} N : \varsigma_1 \times \varsigma_2}{\Delta \vdash_{\mathtt{P}} \pi_2 \hat{M} \rightsquigarrow \pi_2 N : \varsigma_2} \qquad \frac{\Delta \vdash_{\kappa} \hat{M} \rightsquigarrow M : \varsigma \quad s : \mathfrak{H}_{\varsigma}(s) \overset{unpack}{\Longrightarrow} N : \varsigma_1 \times \varsigma_2 \quad \kappa \neq \mathtt{P}}{\Delta \vdash_{\kappa} \pi_2 \hat{M} \rightsquigarrow \langle s = M, \pi_2 N\rangle : \exists s{:}\varsigma.\varsigma_2}$$

$$\frac{\Delta \vdash_{\kappa} \hat{M} \rightsquigarrow M : \varsigma_M \quad \Delta \vdash \hat{\sigma} \rightsquigarrow \sigma \quad \Delta, s{:}\varsigma_M \vdash s : \varsigma_M \leq \sigma \rightsquigarrow N}{\Delta \vdash_{\kappa \sqcup \mathtt{D}} \hat{M} :: \hat{\sigma} \rightsquigarrow (\mathsf{let}\ s = M\ \mathsf{in}\ (N : \sigma) :: \sigma) : \sigma} \qquad \frac{\Delta \vdash_{\kappa} \hat{M} \rightsquigarrow M : \varsigma_M \quad \Delta \vdash \hat{\sigma} \rightsquigarrow \sigma \quad \Delta, s{:}\varsigma_M \vdash s : \varsigma_M \leq \sigma \rightsquigarrow N}{\Delta \vdash_{\mathtt{W}} \hat{M} :> \hat{\sigma} \rightsquigarrow (\mathsf{let}\ s = M\ \mathsf{in}\ (N : \sigma) :> \sigma) : \sigma}$$

$$\frac{\Delta \vdash_{\mathtt{P}} \hat{M}_1 \rightsquigarrow M_1 : \varsigma_1 \quad \Delta, s{:}\varsigma_1 \vdash_{\mathtt{P}} \hat{M}_2 \rightsquigarrow M_2 : \varsigma_2}{\Delta \vdash_{\mathtt{P}} \mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2 \rightsquigarrow \pi_2\langle s = M_1, M_2\rangle : \varsigma_2[M_1/s]} \qquad \frac{\Delta \vdash_{\kappa_1} \hat{M}_1 \rightsquigarrow M_1 : \varsigma_1 \quad \Delta, s{:}\varsigma_1 \vdash_{\kappa_2} \hat{M}_2 \rightsquigarrow M_2 : \varsigma_2 \quad \kappa_1 \sqcup \kappa_2 \neq \mathtt{P}}{\Delta \vdash_{\kappa_1 \sqcup \kappa_2} \mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2 \rightsquigarrow \langle s = M_1, M_2\rangle : \exists s{:}\varsigma_1.\varsigma_2}$$