A Type System for Higher-Order Modules*

Derek Dreyer Karl Crary Robert Harper

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

{dreyer,crary,rwh}@cs.cmu.edu

Abstract

We present a type theory for higher-order modules that accounts for many central issues in module system design, including translucency, applicativity, generativity, and modules as first-class values. Our type system harmonizes design elements from previous work, resulting in a simple, economical account of modular programming. The main unifying principle is the treatment of abstraction mechanisms as computational effects. Our language is the first to provide a complete and practical formalization of all of these critical issues in module system design.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types, Modules*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*

General Terms

Languages, Theory

Keywords

Type theory, modularity, computational effects, abstract data types, functors, generativity, singleton types

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 [17], 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 without them.

Much work has been devoted to the design of modular programming languages. Early work on CLU [19] and the Modula family of languages [34, 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 [23] and Objective Caml [27]. Numerous extensions and variations of these designs have been considered in the literature [21, 18, 28, 31, 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 to "hide" type information selectively. The mechanism for controlling type propagation is *translucency* [11, 14], with transparency and opacity as limiting cases.

*The ConCert Project is supported by the National Science Foundation under grant number 0121633: "ITR/SY+SI: Language Technology for Trustless Software Dissemination".

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

Phase Separation ML-like module systems enjoy a *phase separation* property [12] 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" symbol 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 [14]. 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.

Modules as First-Class Values Modules in ML are "second-class" in the sense that they cannot be computed as the results of ordinary run-time computation. It can be useful to treat a module as a first-class value that can be stored into a data structure, or passed as an ordinary function argument or result [11, 24].

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 [9, 18].

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 [24, 29].

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* [33, 32, 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 Signature subtyping is used to model "forgetting" type sharing, an essential part of signature matching. The coercive aspects of signature matching (dropping of fields and specialization of polymorphic values) are omitted here, since the required coercions are definable in the language.

Purity and Impurity Our type system classifies module expressions into *pure (effect-free)* and *impure (effectful)* forms. To ensure proper enforcement of abstraction, impure modules are *incomparable* (may not be compared for equality with any other module) and *non-projectible* (may not have type components projected from them). It follows that impure modules are also *non-substitutable* (may not be substituted for a module variable in a signature).

Abstraction and Sealing Modules that are *sealed* with a signature to impose type abstraction [11] are regarded as impure. In other words, sealing is regarded as a *pro forma* computational effect. This is consistent with the informal idea that generativity involves the generation of new types at run time. Moreover, this ensures that sealed modules are incomparable and non-projectible, which is sufficient to ensure the proper semantics of type abstraction.

Totality and Partiality Functors are λ-abstractions at the level of modules. A functor whose body is pure is said to be *total*; otherwise it is *partial*. It follows that the application of a pure, total functor to a pure argument is pure, whereas the application of a pure, partial functor to a pure argument is impure. Partial functors are naturally *generative*, meaning that the abstract types in its result are "new" for each instance; total functors are *applicative*, meaning that equal arguments yield equal types in the result. Generative functors are obtained without resort to "generative stamps" [23, 21].

Weak and Strong Sealing Since sealing induces a computational effect, only partial functors may contain sealed substructures; this significantly weakens the utility of total functors. To overcome this limitation we distinguish two forms of effect, *static* and *dynamic*, and two forms of sealing, *weak* and *strong*. Weak sealing induces a static effect, which we think of as occurring once during type checking; strong sealing induces a dynamic effect, which we think of as occurring during execution. Dynamic effects induce partiality, static effects preserve totality.

Existential Signatures In a manner similar to Shao [31], 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 propose the use of existential signatures to provide principal signatures where none would otherwise exist. We show that these existential signatures are type-theoretically ill-behaved in

[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 [4].

types	τ	$::=$	$\mathsf{Typ}\,M \mid \Pi s{:}\sigma.\tau \mid \tau_1 \times \tau_2$
terms	e	$::=$	$\mathsf{Val}\,M \mid \langle e_1, e_2\rangle \mid \pi_i e \mid eM \mid$ $\mathsf{fix}\, f(s{:}\sigma){:}\tau.e \mid \mathsf{let}\; s = M \;\mathsf{in}\; (e : \tau)$
signatures	σ	$::=$	$1 \mid [T] \mid [\tau] \mid \Pi^{\mathsf{tot}} s{:}\sigma_1.\sigma_2 \mid \Pi^{\mathsf{par}} s{:}\sigma_1.\sigma_2 \mid$ $\Sigma s{:}\sigma_1.\sigma_2 \mid \mathfrak{S}(M)$
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	$::=$	$\bullet \mid \Gamma, s{:}\sigma$

Figure 1. Syntax

general, so, we restrict their use to a well-behaved setting. In the style of Harper and Stone [13], we propose the use of an elaboration algorithm from an external language that may incur the avoidance problem, into our type system, which does not.

Packaged Modules Modules in our system are "second-class" in the sense that the language of modules is separate from the language of terms. However, following Mitchell *et al.* [24] and Russo [29], we provide a way of packaging a module as a first-class value. In prior work, such packaged modules are typically given an existential type, whose closed-scope elimination construct can make for awkward programming. Instead, our account of type generativity allows us to employ a more natural, open-scope elimination construct, whereby unpackaging a packaged module engenders a dynamic effect.

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 [11] or incompleteness of type checking [27]. In contrast, our formalism leads to a practical, implementable programming language.

The rest of this paper is structured as follows: In Section 2 we present our core type system for higher-order modules, including the intuition behind its design and a brief description of the decidable typechecking algorithm. In Section 3 we discuss the programming importance of having both weak and strong forms of sealing. In Section 4 we explain the avoidance problem and how it can be circumvented using an elaboration algorithm. In Section 5 we present a very simple, orthogonal extension of our core system to provide support for packaging modules as first-class values. Finally, in Section 6 we compare our system with related work and in Section 7 we conclude.

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 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.

Terms 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 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 "$:\tau$" 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 $\Pi^{\mathsf{tot}} s{:}\sigma_1.\sigma_2$ and $\Pi^{\mathsf{par}} s{:}\sigma_1.\sigma_2$ express the type of functors that accept an argument of signature σ_1 and return a result of signature σ_2 (possibly containing s). The reason for two different Π signatures is to distinguish between *total* and *partial* functors, which we discuss in detail below. For convenience, we will take Π (without a superscript) to be synonymous with Π^{tot}. When s does not appear free in σ_2, we write $\Pi s{:}\sigma_1.\sigma_2$ as $\sigma_1 \to \sigma_2$.

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. 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 [33]. The definition of $\mathfrak{S}_\sigma(M)$ (the signature containing only modules equal to M at signature σ) is given in Figure 5.

```
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 ...

$$\Sigma s{:}[T].\ \Sigma t{:}\mathfrak{S}([\mathsf{Typ}\,s \times \mathtt{int}]).\ \Sigma S{:}(\Sigma u{:}[T].\Sigma f{:}[\mathsf{Typ}\,u \to \mathsf{Typ}\,s].1).\ \Sigma g{:}[\mathsf{Typ}\,t \to \mathsf{Typ}\,(\pi_1 S)].1$$

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 ...

$$\langle s = [\mathtt{bool}],\ \langle t = [\mathtt{bool} \times \mathtt{int}],\ \langle S = \langle u = [\mathtt{string}], \langle f = [\lambda y{:}\,\mathsf{Typ}\,u.\mathtt{true}], \langle\rangle\rangle\rangle,\ \langle g = [\lambda y{:}\,\mathsf{Typ}\,t.\mathtt{"hello\ world"}], \langle\rangle\rangle\rangle\rangle\rangle$$

Figure 3. ML Structure Example

Modules 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 and may appear free 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 is sealed either strongly or weakly, the result is *opaque*. 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. The distinction between strong and weak sealing is discussed in detail below.

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 $\lambda\alpha{:}(T \to T).(\mathtt{int} \times \alpha\,\mathtt{int})$ is represented by the module $\lambda s{:}([T] \to [T]).[\mathtt{int} \times \mathsf{Typ}\,(s\,[\mathtt{int}])]$.

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

Comparability and Projectibility Two closely related issues are crucial to the design of a module system supporting type abstraction:

1. When can a module be compared for equivalence with another module?
2. When can a type component be projected from a module and used as a type?

We say that a module is *comparable* iff it can be compared for equivalence with another module, and that a module is *projectible* iff its type components may be projected and used as type expressions. (In the literature most presentations emphasize projectibility [11, 14, 15].)

A simple analysis of the properties of comparability and projectibility suggests that they are closely related. Suppose that M is a projectible module with signature $[T]$, so that $\mathsf{Typ}\,M$ is a type. Since type equality is an equivalence relation, this type may be compared with any other, in particular, $\mathsf{Typ}\,M'$ for another projectible module M' of the same signature. But since $\mathsf{Typ}\,M$ and $\mathsf{Typ}\,M'$ fully determine M, we are, in effect, comparing M with M' for equivalence. This suggests that projectible modules be regarded as comparable for type checking purposes. Conversely, if M is a comparable module, then by extensionality M should be equivalent to $[\mathsf{Typ}\,M]$, which is only sensible if M is also projectible.

Purity and Impurity The design of our module system rests on the semantic notions of *purity* and *impurity* induced by computational effects. To motivate the design, first recall that in a first-class module system such as Harper and Lillibridge's [11] there can be "impure" module expressions that yield distinct type components each time they are evaluated. For example, a module expression M might consult the state of the world, yielding a different module for each outcome of the test. The type components of such a module are not statically well-determined, and hence should not be admitted as type expressions at all, much less compared for equivalence. On the other hand, even in such a general framework, pure (effect-free) modules may be safely regarded as both comparable and projectible.

In a second-class module system such examples are not, in fact, expressible, but we will nevertheless find it useful to classify modules according to their purity.[2] This classification is semantic, in the sense of being defined by judgments of the calculus, rather than syntactic, in the sense of being determined solely by the form of expression. Such a semantic approach is important for a correct account of type abstraction in a full-featured module language.

The axiomatization of purity and impurity in our system is based on a set of rules that takes account of the types of expressions, as well as their syntactic forms. The type system is conservative in that it "assumes the worst" of an impure module expression, ruling it

[2] Moreover, in Section 5 we will introduce the means to re-create these examples in our setting, making essential use of the same classification system.

incomparable and non-projectible, even when its type components are in fact statically well-determined. As we will see shortly, this is important for enforcing type abstraction, as well as ensuring soundness in the presence of first-class modules. In addition, since it is sound to do so, we deem all pure module expressions to be comparable and projectible. That is, to be as permissive as possible without violating soundness or abstraction, we identify comparability and projectibility with purity. Finally, note that a module is judged pure based on whether its type components are well-determined, which is independent of whether any term components have computational effects.

In the literature different accounts of higher-order modules provide different classes of pure modules. For example, in Harper and Lillibridge's first-class module system [11], only syntactic values are considered pure. In Leroy's second-class module calculi [14, 15], purity is limited to the syntactic category of paths. In Harper *et al.*'s early "phase-distinction" calculus [12] all modules are deemed to be pure, but no means of abstraction is provided.

Abstraction via Sealing The principal means for defining abstract types is *sealing*, written $M :> \sigma$. Sealing M with σ prevents any client of M from depending on the identities of any type components specified opaquely—with signature $[\![T]\!]$ rather than $\mathfrak{S}_{[T]}(M)$—inside σ. From the point of view of module equivalence, this means that a sealed module should be considered incomparable. To see this, suppose that $M = ([\mathtt{int}] :> [\![T]\!])$ is regarded as comparable. Presumably, M could not be deemed equivalent to $M' = ([\mathtt{bool}] :> [\![T]\!])$ since their underlying type components are different. However, since module equivalence is reflexive, if M is comparable, then M must be deemed equivalent to itself. This would mean that the type system would distinguish two opaque modules based on their underlying implementation, a violation of type abstraction.

A significant advantage of our judgmental approach to purity is that it affords a natural means of ensuring that a sealed module is incomparable, namely to judge it impure. This amounts to regarding sealing as a *pro forma* run-time effect, even though no actual effect occurs at execution time. Not only does this ensure that abstraction violations such as the one just illustrated are ruled out, but we will also show in Section 3 that doing so allows the type system to track the run-time "generation" of "new" types.

Applicative and Generative Functors Functors in Standard ML are *generative* in the sense that each abstract type in the result of the functor is "generated afresh" for each instance of the functor, regardless of whether or not the arguments in each instance are equivalent. Functors in Objective Caml, however, are *applicative* in the sense that they preserve equivalence: if applied to equivalent arguments, they yield equivalent results. In particular, the abstract types in the result of a functor are the same for any two applications to the same argument.

Continuing the analogy with computational effects, we will deem any functor whose body is pure to be *total*, otherwise *partial*. The application of a pure, total functor to a pure argument is pure, and hence comparable. Total functors are applicative in the sense that the application of a pure total functor to two equivalent pure modules yields equivalent pure modules, because the applications are pure, and hence comparable. Partial functors, on the other hand, always yield impure modules when applied. Therefore they do not respect equivalence of arguments (because the results, being impure, are not even comparable), ensuring that each instance yields a distinct result.

We distinguish the signatures of total (applicative) and partial (generative) functors. Total functors have Π signatures, whereas partial functors have Π^{par} signatures. The subtyping relation is defined so that every total functor may be regarded (degenerately) as a partial functor.

Weak and Strong Sealing In our system we identify applicative functors with total ones, and generative functors with partial ones. To make this work, however, we must refine the notion of effect. For if sealing is regarded as inducing a run-time effect, then it is impossible to employ abstraction within the body of a total functor, for to do so renders the body impure. (We may seal the *entire* functor with a total functor signature to impose abstraction, but this only ensures that the exported types of the functor are held abstract in any clients of that functor. It does not permit a substructure in the body of the functor to be held abstract in both the clients of the functor and in the remainder of the functor body.)

The solution is to distinguish two forms of sealing—*strong*, written $M :> \sigma$ as before, and *weak*, written $M :: \sigma$. Both impose abstraction in the sense of limiting type propagation to what is explicitly specified in the ascribed signature by regarding both forms of sealing as inducing impurity. However, to support a useful class of applicative functors, we further distinguish between *static* and *dynamic* effects. Weak sealing induces a static effect, whereas strong sealing induces dynamic effect.

The significance of this distinction lies in the definition of total and partial functors. A functor whose body involves a dynamic effect (*i.e.*, is *dynamically impure*), is ruled partial, and hence generative. Thus strong sealing within a functor body induces generativity of that functor. A functor whose body is either pure, or involves only a static effect (*i.e.*, is *dynamically pure*), is ruled total, and hence applicative. This ensures that applicative functors may use abstraction within their bodies without incurring generative behavior. The methodological importance of this distinction is discussed in Section 3.

A dynamic effect may be thought of as one that occurs during execution, whereas a static effect is one that occurs during type checking. Dynamic effects are suspended inside of a λ-abstraction, so functor abstractions are dynamically pure. However, when applied, the dynamic effects inside the functor are released, so that the application is dynamically impure. On the other hand, static effects occur during type checking, and hence are not suspended by λ-abstraction, nor released by application.

Formalization The typing judgment for our system is written $\Gamma \vdash_\kappa M : \sigma$, where κ indicates M's purity. The classifier κ is drawn from the following four-point lattice:

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

The point P indicates that M is pure (and hence comparable and projectible), D indicates dynamic purity, S indicates static purity, and W indicates well-formedness only (no purity information). Hence, $\Gamma \vdash_{\mathtt{P}} M : \sigma$ is our purity judgment. 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 P is taken as the bottom and W is taken as the top. We also sometimes find it convenient to use the notation $\Pi^{\delta} s{:}\sigma_1.\sigma_2$ for a functor signature that is either total or partial depending on whether $\delta = \mathsf{tot}$ or $\delta = \mathsf{par}$, respectively.

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

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

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

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

Figure 4. Key Typing Rules

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

Static Equivalence In the foregoing discussion we have frequently made reference to a notion of module equivalence, without specifying what this means. A key design decision for a module calculus is to define when two comparable modules are to be deemed equivalent. Different module systems arise from different notions of equivalence.

If a pure module has signature $[T]$, it is possible to extract the type component 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 would be to regard $\mathsf{Typ}\, M = \mathsf{Typ}\, M'$ exactly when the modules M and M' are equal. But this is too 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.

A characteristic feature of second class module systems is that they respect the *phase distinction* [12] between compile-time and run-time computation. This property of a module system states that type equivalence must be decidable independently of term equivalence. This should be intuitively plausible, since a second-class module system 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 that the matter is more subtle than it appears.)

Based on this principle, we define module equivalence to be "equivalence for type checking purposes", or *static equivalence*. Roughly speaking, two modules are deemed to be equivalent whenever they agree on their corresponding type components.[3]

We write our module equivalence judgment 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, pure generative functors are always statically equivalent, just as atomic term modules are:

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

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

As an aside, this discussion of module equivalence refutes the misconception that first-class modules are more general than second-class modules. In fact, the expressiveness of first- and second-class modules is incomparable. First-class modules have the obvious advantage that they are first-class. 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 one

[3] The phase distinction calculus of Harper, *et al.* [12] 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.

must use dynamic equivalence instead (in other words, one cannot phase-split modules as in Harper *et al.* [12]). Consequently, first-class modules cannot propagate as much type information as second-class modules can.

Singleton Signatures Type sharing information is expressed in our language using singleton signatures [33], a derivative of translucent sums [11, 14, 18]. (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" [11]. 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 pure modules. Thus, only pure modules can be selfified (as in Harper and Lillibridge [11] and Leroy [14]).

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 $F : [T] \to [T]$. 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 [33]) 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 meta-theoretic implications of such a change.

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

Figure 5. Singletons at Higher Signatures

Since a module with a (higher-order) singleton signature is fully transparent, it is obviously projectible and comparable, and hence could be judged to be pure, even if it would otherwise be classified as impure. This is an instance of the general problem of recognizing that "benign effects" need not disturb purity. Since purity is a judgment in our framework, we could readily incorporate extensions to capture such situations, but we do not pursue the matter here.

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.

Module typechecking 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 judgment 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 [33] 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 [7].

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* $\Gamma \vdash_\kappa M \Rightarrow \sigma$.

```
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. Strong Sealing Example

```
signature ORD =
  sig
    type elem
    val compare : elem * elem -> order
  end
signature SET = (* persistent sets *)
  sig
    type elem
    type set
    val empty : set
    val insert : elem * set -> set
    ...
  end

functor SetFun (Elem : ORD)
     :: SET where type elem = Elem.elem =
  struct
    type elem = Elem.elem
    type set = elem list
    ...
  end

structure IntOrd = struct
  type elem = int
  val compare = Int.compare
end
structure IntSet1 = SetFun(IntOrd)
structure IntSet2 = SetFun(IntOrd)
```

Figure 7. Weak Sealing Example

3 Strong and 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 an abstract 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.

The symbol table example demonstrates the importance of strong sealing for encoding generative abstract types in stateful modules. Generativity is not necessary, however, for purely functional modules. Leroy [15] gives several examples of such modules as motivation for the adoption of applicative functors. For instance, one may wish to implement persistent sets using ordered lists. Figure 7 exhibits a purely functional `SetFun` functor, which is parameterized over an ordered element type, and whose implementation of the abstract `set` type is sealed. When `SetFun` is instantiated multiple times—*e.g.,* in different client modules—with the same element type, it is useful for the resulting abstract `set` types to be seen as interchangeable.

In our system, `SetFun` is made applicative, but still opaque, by *weakly* sealing its body. Specifically, `IntSet1.set` and `IntSet2.set` are both equivalent to `SetFun(IntOrd).set`. This type is well-formed because `SetFun` has an applicative functor signature, and `SetFun` and `IntOrd`, being variables, are both pure. Recall that a functor containing weak sealing is impure and must be bound to a variable before it can be used applicatively.

The astute reader may notice that weak sealing is not truly necessary in the `SetFun` example. In fact, one can achieve the same effect as the code in Figure 7 by leaving the body of the functor unsealed and (strongly) sealing the functor itself with an applicative functor signature before binding it to `SetFun`. This is the technique employed by Shao [31] for encoding applicative functors, as his system lacks an analogue of weak sealing. A failing of this approach is that it only works if the functor body is fully transparent—in the absence of weak sealing, any opaque substructures would have to be strongly sealed, preventing the functor from being given an applicative signature.

The best examples of the need for opaque substructures in applicative functors are provided by the interpretation of ML `datatype`'s as abstract types [13]. In both Standard ML and Caml, `datatype`'s are *opaque* in the sense that their representation as recursive sum types is not exposed, and thus distinct instances of the same `datatype` declaration create distinct types. Standard ML and Caml differ, however, on whether `datatype`'s are *generative*. In the presence of applicative functors (which are absent from Standard ML) there is excellent reason for `datatype`'s *not* to be generative—namely, that a generative interpretation would prevent `datatype`'s from appearing in the bodies of applicative functors. This would severely diminish the utility of applicative functors, particularly since in ML recursive types are provided only through the `datatype` mechanism. For example, an implementation of `SetFun` with splay trees, using a `datatype` declaration to define the tree type, would require the use of weak sealing.

For these reasons, strong sealing is no substitute for weak sealing. Neither is weak sealing a substitute for strong. As Leroy [15] 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.) With functors, however, strong sealing is necessary to provide true generativity. Nevertheless, it is worth noting that strong sealing is definable in terms of other constructs in our language, while weak sealing is not. In particular, we can define strong sealing, using a combination of weak sealing and generative functor application, as follows:

$$M :> \sigma \quad \stackrel{\text{def}}{=} \quad ((\lambda\_{:}1.M) :: (\Pi^{\mathsf{par}}\_{:}1.\sigma))\,\langle\rangle$$

The existence of this encoding does not diminish the importance of strong sealing, which we have made primitive in our language regardless.

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 pure, at the expense of requiring that functor and second-projection arguments are pure. This is necessary because the result of substituting an impure module into a well-formed signature can be ill-formed. Thus, to apply a functor to an impure argument, one must let-bind the argument and apply the functor to the resulting (pure) variable.

A similar restriction is imposed by Shao [31], but Harper and Lillibridge [11] 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 purity 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 pure, 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 impure, 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* [9, 18] is that there is no "best" way to do so. For example, consider the signature:

$$\sigma = ([T] \to \mathfrak{S}(s)) \times \mathfrak{S}(s)$$

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:

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

This gives us an infinite array of choices. Any of these choices is superior to the obvious $([T] \to [T]) \times [T]$, 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 pure (thereby eliminating any need to find non-dependent supersignatures), and provide a let construct so that such operations can still be applied to impure modules. We have shown that, as a result, our type theory does enjoy principal signatures.

To achieve this, however, our let construct must be labeled with its result signature (not mentioning the variable being bound), for otherwise the avoidance problem re-arises. This essentially requires that every functor application or projection involving an impure argument be labelled with its result signature as well, leading to potentially unacceptable syntactic overhead in practice. Fortunately, programs can be systematically rewritten to avoid this problem, as we describe next.

4.1 Elaboration and Existential Signatures

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 M_1 is pure, then the let expression can be given the minimal signature $\sigma_2(M_1)$. Otherwise, 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 impure $M : \sigma_1$ can be rewritten as $\langle s = M, F(s)\rangle$ and given signature $\Sigma s{:}\sigma_1.\sigma_2$.

Following Harper and Stone [13], we propose the use of an elaboration algorithm to systematize these rewritings. This elaborator takes code written in an external language that supports unannotated let's, as well as impure functor application and second-projection, and produces code written in our type system. Since the elaborator 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 an existential $\exists$ 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. Space considerations preclude further details of the elaboration algorithm, which appear in the companion technical report [7].

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 pure $M : \sigma$ 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 [10].

4.2 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.

5 Packaging 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). However, fully general first-class modules present difficulties for static typing [18].

One practical approach to modules as first-class values was suggested by Mitchell, *et al.* [24], who propose that second-class modules automatically be wrapped as existential packages [25] to obtain first-class values. A similar approach to modules as first-class values is described by Russo and implemented in Moscow ML [29].

This existential-packaging approach to modules as first-class values is built into our language. We write the type of a packaged module as $\langle\!|\sigma|\!\rangle$ and the packaging construct as $\mathsf{pack}\,M\,\mathsf{as}\,\langle\!|\sigma|\!\rangle$. Elimination of packaged modules (as for existentials) is performed using a closed-scope unpacking construct. These may be defined as follows:

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

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

The main limitation of existentially-packaged modules is the closed-scope elimination construct. It has been observed repeatedly in the literature [20, 3, 18] that this construct is too restrictive to be very useful. For one, in "$\mathsf{unpack}\,e\,\mathsf{as}\,s{:}\sigma\,\mathsf{in}\,(e' : \tau)$", the result type τ may not mention s. As a consequence, functions over packaged modules may not be dependent; that is, the result type may not mention the argument. This deficiency is mitigated in our language by the ability to write functions over unpackaged, second-class modules, which can be given the dependent type $\Pi s{:}\sigma.\tau(s)$ instead of $\langle\!|\sigma|\!\rangle \to \tau$.

Another problem with the closed-scope elimination construct is that a term of package type cannot be unpacked into a *stand-alone* second-class module; it can only be unpacked inside an enclosing term. As each unpacking of a packaged module creates an abstract type in a separate scope, packages must be unpacked at a very early stage to ensure coherence among their clients, leading to "scope inversions" that are awkward to manage in practice.

What we desire, therefore, is a new module construct of the form "$\mathsf{unpack}\,e\,\mathsf{as}\,\sigma$", which coerces a first-class package e of type $\langle\!|\sigma|\!\rangle$ back into a second-class module of signature σ. The following example illustrates how adding such a construct carelessly can lead to unsoundness:

$$\begin{array}{l} \mathtt{module}\ F\ = \lambda s{:}[\langle\!|\sigma|\!\rangle].(\mathsf{unpack}\,(\mathsf{Val}\,s)\,\mathsf{as}\,\sigma) \\ \mathtt{module}\ X_1 = F\,[\mathsf{pack}\,M_1\,\mathsf{as}\,\langle\!|\sigma|\!\rangle] \\ \mathtt{module}\ X_2 = F\,[\mathsf{pack}\,M_2\,\mathsf{as}\,\langle\!|\sigma|\!\rangle] \end{array}$$

Note that the argument of the functor F is an atomic term module, so all arguments to F are statically equivalent. If F is given an applicative signature, then X_1 and X_2 will be deemed equivalent, even if the original modules M_1 and M_2 are not! Thus, F must be

$$
\begin{array}{lll}
\text{types} & \tau & ::= \ \cdots \mid \langle\!|\sigma|\!\rangle \\
\text{terms} & e & ::= \ \cdots \mid \mathsf{pack}\, M\, \mathsf{as}\, \langle\!|\sigma|\!\rangle \\
\text{modules} & M & ::= \ \cdots \mid \mathsf{unpack}\, e\, \mathsf{as}\, \sigma
\end{array}
$$

$$
\frac{\Gamma \vdash \sigma_1 \equiv \sigma_2}{\Gamma \vdash \langle\!|\sigma_1|\!\rangle \equiv \langle\!|\sigma_2|\!\rangle}
\qquad
\frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash \mathsf{pack}\, M\, \mathsf{as}\, \langle\!|\sigma|\!\rangle : \langle\!|\sigma|\!\rangle}
$$

$$
\frac{\Gamma \vdash e : \langle\!|\sigma|\!\rangle}{\Gamma \vdash_{\mathrm{S}} \mathsf{unpack}\, e\, \mathsf{as}\, \sigma : \sigma}
$$

Figure 8. Packaged Module Extension

deemed generative, which in turn requires that the unpack construct induce a *dynamic* effect.

Packaged modules that admit this improved unpacking construct are not definable in our core language, but they constitute a simple, orthogonal extension to the type system that does not complicate type checking. The syntax and typing rules for this extension are given in Figure 8. Note that the closed-scope unpacking construct is definable as

$$
\mathsf{let}\ s = (\mathsf{unpack}\ e\ \mathsf{as}\ \sigma)\ \mathsf{in}\ (e' : \tau)
$$

Intuitively, unpacking is generative because the module being unpacked can be an arbitrary term, whose type components may depend on run-time conditions. In the core system we presented in Section 2, the generativity induced by strong sealing was merely a *pro forma* effect—the language, supporting only second-class modules, provided no way for the type components of a module to be actually generated at run time. The type system, however, treats dynamic effects as if they are all truly dynamic, and thus it scales easily to handle the real run-time type generation enabled by the extension in Figure 8.

6 Related Work

Harper, Mitchell and Moggi [12] 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 sealing mechanism (and consequently all modules are pure).

MacQueen and Tofte [21] proposed a higher-order module extension to the original Definition of Standard ML [22], 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 [11] and Leroy [14, 16] 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 [23], and Harper and Stone [13] 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* [15], which enables one to give fully transparent signatures for many higher-order functors. Leroy's formalism may be seen as defining purity 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 abstract type `F(X).t` can only be the result of applying `F` to the module *named* `X`. 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 [4]. Together, manifest types and applicative functors form the basis of the module system of Objective Caml [27]. The manifest type formalism, like the translucent sum formalism, does not address the avoidance problem, and consequently it lacks principal signatures.

More recently, Russo, in his thesis [28], formalized two separate module languages: one being a close model of the SML module system, the other being a higher-order module system with applicative functors along the lines of O'Caml, but abandoning the named form restriction as we do. Russo's two languages can be viewed as subsystems of ours, the first supporting only strong sealing, the second supporting only weak sealing. We adopt his use of existential signatures to address the avoidance problem, although Russo also used existentials to model generativity, which we do not. Russo's thesis also describes an extension to SML for packaging modules as first-class values. This extension is very similar to the existential-packaging approach discussed in the beginning of Section 5, and therefore suffers from the limitations of the closed-scope unpacking construct.

While Russo defined these two languages separately, he implemented the higher-order module system as an experimental extension to the Moscow ML compiler [26]. Combining the two languages without distinguishing between static and dynamic effects has an unfortunate consequence. The Moscow ML higher-order module system places no restrictions on the body of an applicative functor; in particular, one can defeat the generativity of a generative functor by eta-expanding it into an applicative one. Exploiting this uncovers an unsoundness in the language [6], that, in retrospect, is clear from our analysis: one cannot convert a partial into a total functor.

Shao [31] has proposed a single type system for modules supporting both applicative and generative functors. Roughly speaking, Shao's system may be viewed as a subsystem of ours based exclusively on strong sealing and dynamic effects, but supporting both Π and Π^{par} signatures. As we observed in Section 3, this means that the bodies of applicative functors may not contain opaque substructures (such as `datatype`'s). Shao's system, like ours, circumvents the avoidance problem (Section 4) by restricting functor application and projection to pure arguments (which must be paths in his system), and by eliminating implicit subsumption, which amounts to requiring that let expressions be annotated, as in our system. It seems likely that our elaboration techniques could as well be applied to Shao's system to lift these restrictions, but at the expense of syntactic principal signatures. Shao also observes that fully transparent functors may be regarded as applicative; this is an instance of the general problem of recognizing benign effects, as described in Section 2.

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 (sound) generative 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.

Nevertheless, there are several important issues in modular programming that go beyond the scope of our type theory. Chief among these are:

- *Structure Sharing.* The original version of Standard ML [22] included a notion of module equivalence that was sensitive to the dynamic, as well as static, parts of the module. Although such a notion would violate the phase distinction, it might be possible to formulate a variation of our system that takes account of dynamic equivalence in some conservative fashion. It is possible to simulate structure sharing by having the elaborator add an abstract type to each structure to serve as the "compile-time name" of that structure. However, this would be merely an elaboration convention, not an intrinsic account of structure sharing within type theory.
- *Recursive Modules.* An important direction for future research is to integrate recursive modules [8, 5, 30] into the present framework. The chief difficulty is to achieve practical type checking in the presence of general recursively dependent signatures, or to isolate a practical sub-language that avoids these problems.

8 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] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In M. Broy and C. B. Jones, editors, *Proceedings IFIP TC2 working conference on programming concepts and methods*, pages 479–504. North-Holland, 1990. Also available as research report 56, DEC Systems Research Center.

[4] 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.

[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. Moscow ML's higher-order modules are unsound. Posted to the TYPES electronic forum, September 2002.

[7] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules (expanded version). Technical Report CMU-CS-02-122R, School of Computer Science, Carnegie Mellon University, December 2002.

[8] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, Montreal, Canada, June 1998.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

[20] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[21] 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.

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

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

[24] 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.

[25] 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.

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

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

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

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

[30] Claudio V. Russo. Recursive structures for Standard ML. In *International Conference on Functional Programming*, pages 50–61, Florence, Italy, September 2001.

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

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

[33] 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.

[34] 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 judgments for term typechecking ($\Gamma \vdash e \Leftarrow \tau$) and unique type synthesis ($\Gamma \vdash e \Rightarrow \tau$) for space reasons; they are fully detailed in the companion technical report [7].

A Type System

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

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

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

$$\frac{\Gamma \vdash_{\mathsf{P}} M : [T]}{\Gamma \vdash \mathsf{Typ}\, M\ \mathrm{type}} \qquad \frac{\Gamma, s{:}\sigma \vdash \tau\ \mathrm{type}}{\Gamma \vdash \Pi s{:}\sigma.\tau\ \mathrm{type}}$$

$$\frac{\Gamma \vdash \tau'\ \mathrm{type} \quad \Gamma \vdash \tau''\ \mathrm{type}}{\Gamma \vdash \tau' \times \tau''\ \mathrm{type}} \qquad \frac{\Gamma \vdash \sigma\ \mathrm{sig}}{\Gamma \vdash \langle\!|\sigma|\!\rangle\ \mathrm{type}}$$

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

$$\frac{\Gamma \vdash [\tau_1] \cong [\tau_2] : [T]}{\Gamma \vdash \tau_1 \equiv \tau_2} \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 \quad \Gamma, s{:}\sigma_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \Pi s{:}\sigma_1.\tau_1 \equiv \Pi s{:}\sigma_2.\tau_2}$$

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \quad \Gamma \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash \tau_1' \times \tau_1'' \equiv \tau_2' \times \tau_2''} \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2}{\Gamma \vdash \langle\!|\sigma_1|\!\rangle \equiv \langle\!|\sigma_2|\!\rangle}$$

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

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash_{\kappa} M : [\tau]}{\Gamma \vdash \mathsf{Val}\, M : \tau}$$

$$\frac{\Gamma \vdash_{\kappa} M : \sigma \quad \Gamma, s{:}\sigma \vdash e : \tau \quad \Gamma \vdash \tau\ \mathrm{type}}{\Gamma \vdash \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) : \tau} \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma}{\Gamma \vdash \mathsf{pack}\, M\ \mathsf{as}\ \langle\!|\sigma|\!\rangle : \langle\!|\sigma|\!\rangle}$$

$$\frac{\Gamma, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f(s{:}\sigma){:}\tau.e : \Pi s{:}\sigma.\tau} \qquad \frac{\Gamma \vdash e : \Pi s{:}\sigma.\tau \quad \Gamma \vdash_{\mathsf{P}} M : \sigma}{\Gamma \vdash eM : \tau[M/s]}$$

$$\frac{\Gamma \vdash e' : \tau' \quad \Gamma \vdash e'' : \tau''}{\Gamma \vdash \langle e', e'' \rangle : \tau' \times \tau''} \qquad \frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_1 e : \tau'} \qquad \frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_2 e : \tau''}$$

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1\ \mathrm{sig}} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [T]\ \mathrm{sig}} \qquad \frac{\Gamma \vdash \tau\ \mathrm{type}}{\Gamma \vdash [\tau]\ \mathrm{sig}}$$

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

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \leq 1} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [T] \leq [T]} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \leq [\tau_2]}$$

$$\frac{\Gamma \vdash_{\mathsf{P}} M : [T]}{\Gamma \vdash \mathfrak{S}(M) \leq [T]} \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [T]}{\Gamma \vdash \mathfrak{S}(M_1) \leq \mathfrak{S}(M_2)}$$

$$\frac{\Gamma \vdash \sigma_2' \leq \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1''\ \mathrm{sig} \quad (\delta_1, \delta_2) \neq (\mathsf{par}, \mathsf{tot})}{\Gamma \vdash \Pi^{\delta_1} s{:}\sigma_1'.\sigma_1'' \leq \Pi^{\delta_2} s{:}\sigma_2'.\sigma_2''}$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_2''\ \mathrm{sig}}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \leq \Sigma s{:}\sigma_2'.\sigma_2''}$$

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s)} \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash \tau\ \mathrm{type}}{\Gamma \vdash_{\mathtt{P}} [\tau] : [T]} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau\ \mathrm{type}}{\Gamma \vdash_{\mathtt{P}} [e : \tau] : [\tau]} \qquad \frac{\Gamma, s{:}\sigma' \vdash_{\kappa} M : \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_{\kappa} \lambda s{:}\sigma'.M : \Pi s{:}\sigma'.\sigma''}$$

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

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

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

$$\frac{\Gamma \vdash_{\kappa} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\kappa} M'' : \sigma \quad \Gamma \vdash \sigma\ \mathrm{sig}}{\Gamma \vdash_{\kappa} \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) : \sigma} \qquad \frac{\Gamma \vdash e : \llparenthesis\sigma\rrparenthesis}{\Gamma \vdash_{\mathtt{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma} \qquad \frac{\Gamma \vdash_{\kappa'} M : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash_{\kappa} M : \sigma}$$

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

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

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

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

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

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

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

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: $\Gamma \vdash_{\kappa} M \Rightarrow \sigma$

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

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

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

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

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