A Type System for Higher-Order Modules (Expanded Version)

Derek Dreyer Karl Crary Robert Harper

July 2002
CMU-CS-02-122

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

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

Support for this research was provided by the National Science Foundation through grants CCR-9984812, "Type-Driven Technology for Software and Information Infrastructure," and CCR-0121633, "Language Technology for Trustless Software Dissemination," and the Advanced Research Projects Agency CSTO through contract F19628-95-C-0050, "The Fox Project: Advanced Languages for System Software." The views and conclusions in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of these agencies or the U.S. Government.

Keywords: Type systems, module systems, abstract data types, singleton types, functional programming.

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 [14], 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 [16] and the Modula family of languages [32, 2] has been particularly influential. Much effort has gone into the design of modular programming mechanisms for the ML family of languages, notably Standard ML [21] and Objective Caml [25]. Numerous extensions and variations of these designs have been considered in the literature [18, 15, 26, 29, 5].

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

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

1.1 Design Issues

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

Controlled Abstraction Modularity is achieved by using signatures (interfaces) to mediate access between program components. The role of a signature is to allow the programmer to "hide" type information selectively. The mechanism for controlling type propagation is *translucency* [8, 11], with transparency and opacity as limiting cases.

Phase Separation ML-like module systems enjoy a *phase separation* property [9] 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 [11]. 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 [8, 22].

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 [6, 15].

1.2 A Type System for Modules

The type system proposed here takes into account all of these design issues. It consolidates and harmonizes design elements that were previously seen as disparate into a single framework. For example, rather than regard generativity of abstract types as an alternative to non-generative types, we make both mechanisms available in the language. We support both generative and applicative functors, admit translucent signatures, support separate compilation, and are able to accommodate modules as first-class values [22, 27].

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

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

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

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

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

Static and Dynamic Effects The "sealing" [8] of a module with a signature (*aka* "opaque signature ascription"), which is used to create abstract and/or generative types, is deemed by our type system to induce a *pro forma* computational effect. This is backed up by the intuition that generativity involves the generation of new types at run time. For our type system, we set the class of determinate modules to

[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].

be precisely the pure, effect-free modules. Thus, a module containing any type abstraction via sealing is rendered indeterminate, which is essential to achieving a correct semantics of type abstraction.

The framework of effects allows us to distinguish between, and admit both, generative and non-generative forms of type abstraction. In particular, we isolate two distinct sorts of computational effects, which we call *static* and *dynamic*. Static effects model the hiding of type information at compile time (*i.e.,* non-generative type abstraction), while dynamic effects model the generation of new types at run time (*i.e.,* type generativity). Dynamic effects allow us to encode generativity without resorting to the use of "generative stamps" [21, 18]. Since sealing is the principal means of creating abstract types, we also provide two forms of sealing corresponding to the two kinds of effects: *weak sealing*, which induces a static effect, and *strong sealing*, which induces both a static and a dynamic effect.

Applicative and Generative Functors An *applicative* functor [12] is one that respects static equivalence of its arguments. This models the behavior of functors in Objective Caml. A *generative* functor does not respect equivalence—the abstract types in the result differ on each application. The applicative/generative dichotomy falls out very cleanly from our treatment of abstraction via effects. Specifically, a generative functor is one whose body may contain dynamic effects, corresponding to generative types, while the body of an applicative functor may contain only static effects.

Existential Signatures In a manner similar to Shao [29], our type system is carefully crafted to circumvent the avoidance problem, so that every module enjoys a principal signature. However, this requires imposing restrictions on the programmer. To lift these restrictions, we 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 general, so, we restrict their use to a well-behaved setting. In the style of Harper and Stone [10], 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.* [22] and Russo [27], 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 the act of 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 [8] or incompleteness of type checking [25]. 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.

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

Figure 1: Syntax

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

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

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

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

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

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

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

Signatures and Modules There are seven basic signatures in our language. The atomic signature $[\![T]\!]$ is the type of an atomic module containing a single type, and the atomic signature $[\![\tau]\!]$ is the type of an atomic module containing a single term. The atomic modules are written $[\tau]$ and $[e : \tau]$, respectively. (We omit the type label "$: \tau$" 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 $\Pi s{:}\sigma_1.\sigma_2$ and $\Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2$ express the type of functors that accept an argument of signature σ_1 and return a result of signature σ_2 (possibly containing s). We discuss the difference between Π and Π^{gen} in detail below.

The structure signature $\Sigma s{:}\sigma_1.\sigma_2$ is the type of a pair of modules where the left-hand component has signature σ_1 and the right-hand component has signature σ_2, in which s refers to the left-hand component. 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

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

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

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

... is compiled as ...

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

Figure 2: ML Signature Example

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

The module syntax contains module variables (s), the atomic modules, and the usual introduction and elimination constructs for Π and Σ signatures, except that Σ modules are introduced by $\langle s = M_1, M_2\rangle$, in which s stands for M_1 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.

2.1 Module Equivalence and Implications

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

Often the literature on module systems has focused on which modules can appear in type projections, rather than on which modules can be compared for equivalence.[2] Here we stress equivalence as the primary

[2]For example, Harper *et al.* [9] included an explicit notion of module equivalence, but Harper and Lillibridge [8] and Leroy [11, 12] did not.

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

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

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

... is compiled as ...

$$\begin{array}{l}\langle s = [\texttt{bool}], \\ \quad \langle t = [\texttt{bool} \times \texttt{int}], \\ \qquad \langle S = \langle u = [\texttt{string}], \langle f = [\lambda y{:}\,\mathsf{Typ}\,u.\texttt{true}], \langle\rangle\rangle\rangle, \\ \qquad\quad \langle g = [\lambda y{:}\,\mathsf{Typ}\,t.\texttt{"hello world"}], \langle\rangle\rangle\rangle\rangle\end{array}$$

Figure 3: ML Structure Example

notion, since the choice of which modules may be compared for equality determines those from which types may be projected.

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

2.1.1 Determinacy

In the literature different accounts of higher-order modules provide different classes of determinate modules. For example, in Harper and Lillibridge's first-class module system [8], only values are considered determinate. This is necessary in their setting for type soundness because, in the presence of side-effects, non-values could compute different modules containing different types each time they are evaluated. Thus, the type components of a non-value are not well determined (hence the phrase "indeterminate") and cannot meaningfully be compared for equivalence.

In Leroy's second-class module calculi [11, 12], determinacy is limited to the syntactic category of paths. In Leroy's case this was not necessary for type soundness, since modules were never produced by run-time computations, but it served to provide a certain degree of abstraction. In Harper *et al.*'s "phase-distinction" calculus [9], an early higher-order, second-class module system, all modules were deemed to be determinate, but no means of abstraction—such as sealing—was provided.

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

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—*i.e.,* with signature $[\![T]\!]$—inside σ. When considered from the point of view of module equivalence, this opacity means that a sealed module should not be considered determinate.

To see this, suppose that $M = ([\texttt{int}] :> [\![T]\!])$ is determinate. Presumably, M will not be deemed equivalent to $[\texttt{bool}] :> [\![T]\!]$. However, since module equivalence is reflexive, M *will* be deemed equivalent to itself. For the type system to judge two opaque modules—M and M—as equivalent just because they have the same underlying implementation is a clear violation of the principle of type abstraction. Thus, it is imperative that a module containing any form of sealing be judged indeterminate.

Effects and Purity The irreflexivity of opaque module expressions is strongly reminiscent of the irreflexivity of expressions in languages with effects. In a first-class module system such as Harper and Lillibridge's [8], an effectful module expression must be ruled indeterminate, since it could yield a distinct type each time it is evaluated. In a second-class module system no such behavior is, in fact, possible, but it is useful to regard a sealed module as "hiding" an arbitrary computational effect.

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

Applicative vs. Generative Functors One of the key issues in module system design revolves around the interaction of abstract types and functors. Functors in Standard ML are *generative* in the sense that they generate distinct abstract types at each application. Functors in Objective Caml, however, are non-generative, or *applicative*. When applied to the same argument twice, an O'Caml functor yields results with equivalent abstract type components.

In order to model both kinds of functors in our system, we distinguish between two different forms of sealing—*strong*, written $M :> \sigma$ as before, and *weak*, written $M :: \sigma$. Both are opaque, as described above, but only strong sealing is *generative*. By "generative," we mean that each time the module expression $M :> \sigma$ is evaluated at run time, it "generates" abstract types distinct to that dynamic instance. In particular, a functor whose body is strongly sealed should generate distinct abstract types every time it is applied, as each application constitutes a separate dynamic instance of the body.

Weak sealing, on the other hand, allows one to write a functor whose body is sealed opaquely, but which does not generate fresh abstract types at each application. In other words, while $M :> \sigma$ is viewed as generating new type components at run time, $M :: \sigma$ is viewed as generating new type components at *compile time*. As a result, it is reasonable to distinguish between the sort of effects induced by weak and strong sealing. Specifically, opacity corresponds to a *static* effect, while generativity corresponds to a *dynamic* effect. Thus, weak sealing induces a static effect only, while strong sealing induces both a static and a dynamic effect. We discuss the importance of supporting *both* weak and strong sealing in Section 3.

Static vs. Dynamic Effects In our type system, as usual, dynamic effects are suspended when they appear within a lambda. Thus, a lambda is always dynamically pure (free of dynamic effects). When such a lambda is applied, whatever dynamic effects were suspended within it are released, and so a functor application may or may not have dynamic effects. In order that our type system can track dynamic effects, we use signatures to distinguish between functors that may or may not induce dynamic effects when applied. An applicative functor—*i.e.,* one with a Π signature—always has a dynamically pure body, and thus its application (to a dynamically pure argument) is dynamically pure. However, a generative functor—*i.e.,* one with a Π^{gen} signature—may have a dynamically impure body, and thus may generate dynamic effects when applied. One consequence is that the application of a generative functor must *not* be deemed determinate (even if the functor and its argument both are).

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

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

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

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

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

Figure 4: Key Typing Rules

In contrast, static effects, being generated at compile time, may be resolved under a lambda. More generally, no construct can suspend a static effect. Therefore, any module containing any strong or weak sealing is statically impure and therefore indeterminate. Nevertheless, a dynamically pure, statically impure module enjoys some privileges not enjoyed by modules in general. In particular, it can appear within the body of an applicative functor. Thus opaque sub-structures (such as ML-style datatypes [10]) can appear within applicative functors. Conversely, a statically pure, dynamically impure module may only appear in the body of a generative functor, but that functor then captures the dynamic effects, resulting in a determinate functor.

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

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

where `P` indicates that M is pure (and hence determinate), `D` indicates dynamic purity, `S` indicates static purity, and `W` indicates well-formedness only (no purity information). Hence, $\Gamma \vdash_{\mathtt{P}} M : \sigma$ is our determinacy 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 write functor signatures as $\Pi^{\delta} s{:}\sigma_1.\sigma_2$, where $\delta \in \{\epsilon, \mathtt{gen}\}$ and take $\epsilon \sqsubseteq \mathtt{gen}$.

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

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

2.1.2 Static Equivalence

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

The simplest answer to this question would be that $\mathsf{Typ}\, M = \mathsf{Typ}\, M'$ exactly when the modules M and M' are equal. This simple answer is naive because we cannot in general determine when two modules are equal. Suppose $F : [\![\mathtt{int}]\!] \to \sigma$ and $e, e' : \mathtt{int}$. Then $F\,[e] = F\,[e']$ if and only if $e = e'$, but the latter equality is undecidable in general. To preserve the phase distinction [9] between compile-time and run-time computation, we must ensure that type checking never requires the evaluation of program terms.

To resolve this problem, observe that our language provides no means by which a type component of a module can depend on a term component. (This is not happenstance, but the result of careful design. We will see in Section 5 that the matter is more subtle than it appears.) Consequently, we may ignore general equality and restrict our attention to the *static* equivalence of modules. Two modules are deemed to be equivalent if they agree on all type components.[4] In the sequel, we will sometimes refer to "static equivalence" to emphasize the static nature of our equivalence.

We write our module equivalence 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:

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

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

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

As these rules indicate, static equivalence of atomic term modules and generative functors is trivial, as is static equivalence of trivial modules (*i.e.*, modules of signature 1). In the meta-theory, it is therefore convenient to consider the class of signatures of the form 1, $[\![\tau]\!]$, and $\Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2$, which we call *unitary* to suggest that they behave like unit (*i.e.*, 1) with respect to static equivalence. (See Appendices D and E for further details on the utility of the "unitary" distinction.) The complete set of equivalence rules is given in Appendix A.

As an aside, the notion of static equivalence refutes the misconception that first-class modules are more general than second-class modules. In fact, first- and second-class modules are incomparable. First-class modules have the obvious advantage that they are first-class. 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 must use dynamic equivalence instead (in other words, one cannot phase-split modules as in Harper *et al.* [9]). Consequently, first-class modules cannot propagate as much type information as second-class modules can.

2.2 Singleton Signatures

Type sharing information is expressed in our language using singleton signatures [31], a derivative of translucent sums [8, 11, 15]. (An illustration of the use of singleton signatures to express type sharing appears in

[4] The phase distinction calculus of Harper, *et al.* [9] 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.

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:

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

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

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" [8]. For instance, if s is a variable bound with the signature $[\![T]\!]$, s can be given the fully transparent signature $\mathfrak{S}(s)$. This fact is essential to the existence of principal signatures in our type checking algorithm. Note that since singleton signatures express static equivalence information, the formation of singleton signatures is restricted to determinate modules. Thus, only determinate modules can be selfified (as in Harper and Lillibridge [8] and Leroy [11]).

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 $\lambda s{:}[\![T]\!].Fs$. In fact, it is not obvious that F belongs to $\mathfrak{S}_{[\![T]\!]\to[\![T]\!]}(F)$.

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

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

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

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

Lastly, it is worth noting that for all *unitary* signatures σ (as defined in Section 2.1.2), $\mathfrak{S}_\sigma(M) = \sigma$. This results from the fact that singletons express *static* equivalence, and all modules of a unitary signature are statically equivalent at that signature. The distinction between the higher-order singleton definitions for Π and Π^{gen} exhibits another instance of equivalence depending on context. In particular, while $\lambda s{:}[\![T]\!].[\mathtt{int}]$ and $\lambda s{:}[\![T]\!].s$ are inequivalent at $[\![T]\!] \to [\![T]\!]$, they *are* equivalent at $[\![T]\!] \stackrel{\mathrm{gen}}{\to} [\![T]\!]$ because they may both be assigned the latter signature, which is unitary, by subsumption.

2.3 Type Checking

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

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 C, and its correctness theorems are stated below. The main judgment of

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

Figure 5: Singletons at Higher Signatures

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 essentially Stone and Harper's algorithm [31] for type constructor equivalence in the presence of singleton kinds, extended to handle unitary signatures and the interplay between module equivalence and type equivalence. The algorithm, along with its proof of correctness and decidability, is detailed in Appendices D and E. The proof is mostly identical to the Stone-Harper proof. In extending it, we have taken the opportunity to revise a few minor errors and inelegancies; in particular, the lemmas leading up to the proof of soundness for the equivalence algorithm have been considerably simplified (Appendix D).

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 (see Appendix F for proof details):

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

3 Strong vs. 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.

```
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

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 [12] 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 determinate. Recall that a functor containing weak sealing is indeterminate 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 [29] 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

```
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

interpretation of ML datatypes as abstract types [10]. In both Standard ML and Caml datatypes are *opaque* in the sense that their representation as recursive sum types is not exposed. Standard ML and Caml differ, however, on whether datatypes are *generative*. In the presence of applicative functors (which are absent from Standard ML) there is excellent reason for datatypes *not* to be generative—namely, that a generative interpretation would prevent datatypes 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 [12] 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:

$$M :> \sigma \quad \stackrel{\text{def}}{=} \quad ((\lambda\_{:}1.M) :: (1 \stackrel{\text{gen}}{\to} \sigma))\,\langle\rangle$$

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

A similar restriction is imposed by Shao [29], but Harper and Lillibridge [8] propose an alternative that softens the restriction. Harper and Lillibridge's proposal (expressed in our terms) is to include a non-dependent typing rule without a determinacy restriction:

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

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

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

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

The *avoidance problem* [6, 15] 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 determinate (thereby eliminating any need to find non-dependent supersignatures), and provide a let construct so that such operations can still be applied to indeterminate modules. We have shown that, as a result, our type theory does enjoy principal signatures.

To achieve this, however, our let construct must be labelled 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 indeterminate 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 determinate, 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 indeterminate $M : \sigma_1$ can be rewritten as $\langle s = M, F(s) \rangle$ and given signature $\Sigma s{:}\sigma_1.\sigma_2$.

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

Figure 8: Elaborator Judgments

Following Harper and Stone [10], we propose the use of an elaboration algorithm to systematize these rewritings. This elaborator takes code written in an external language that supports unannotated lets, as well as indeterminate 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.

4.1.1 Formalization

The elaborator is defined in terms of the five judgments given in Figure 8. The metavariables $\hat{M}$, $\hat{\sigma}$, etc., range over expressions in the external language (these are the same as the internal language's expressions, except that unannotated let is supported), and the metavariables ς and Δ range over the elaborator's signatures and contexts (the same as the internal language's, except that $\exists$ is supported, as given in Figure 8).

The main judgment is module elaboration, written $\Delta \vdash_\kappa \hat{M} \rightsquigarrow M : \varsigma$, which means that the external module $\hat{M}$ elaborates to the internal module M, which has the signature ς and purity κ. The signature, type, and term elaboration judgments are similar (except that signatures and types have no classifiers to generate). Two judgments are included for eliminating existentials: signature coercion is used when the desired result signature is known, peeling is used to peel off the outermost existentials when the result is not known. The signature coercion judgment is written $\Delta \vdash M : \varsigma \leq \sigma \rightsquigarrow M'$, meaning that a (determinate) module M with signature ς when coerced to signature σ becomes the (determinate) module M'. The peeling judgment is written $M : \varsigma \stackrel{\text{peel}}{\Longrightarrow} M' : \varsigma'$, meaning that $M : \varsigma$ peels to $M' : \varsigma'$.

Some illustrative rules of the elaborator appear in Figure 9; the complete definition is given in Appendix G. In these rules, the auxiliary operation $\overline{\cdot}$ takes elaborator signatures and contexts to internal ones by replacing all occurrences of $\exists$ with Σ. They also use an elaborator signature analog of higher-order singletons, which are defined exactly as in Figure 5 with the additional case:

$$\mathfrak{S}_{\exists s:\varsigma_1.\varsigma_2}(M) \stackrel{\text{def}}{=} \exists \_{:}\mathfrak{S}_{\varsigma_1}(\pi_1 M).\mathfrak{S}_{\varsigma_2[\pi_1 M/s]}(\pi_2 M)$$

A disadvantage of employing an elaborator is that it is difficult to argue rigorously about whether it is correct. Unlike the internal language, which is defined by a declarative type system and proven decidable

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

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

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

$$\frac{\Delta \vdash \pi_2 M : \varsigma_2[\pi_1 M/s] \leq \sigma \rightsquigarrow N}{\Delta \vdash M : \exists s{:}\varsigma_1.\varsigma_2 \leq \sigma \rightsquigarrow N} \ (16) \qquad \frac{\pi_2 M : \varsigma_2[\pi_1 M/s] \stackrel{\text{peel}}{\Longrightarrow} M' : \varsigma}{M : \exists s{:}\varsigma_1.\varsigma_2 \stackrel{\text{peel}}{\Longrightarrow} M' : \varsigma} \ (17) \qquad \frac{\varsigma \text{ not an existential}}{M : \varsigma \stackrel{\text{peel}}{\Longrightarrow} M : \varsigma} \ (18)$$

Figure 9: Illustrative Elaboration Rules

by a sound and complete typechecking algorithm, the external language has no declarative definition but is defined directly via the elaboration algorithm itself, so there is no reference system against which to compare the elaborator. Nevertheless, we can still state and prove some important invariants about elaboration, as enumerated in the theorem below. In particular, the module and signature that are output by module elaboration are well-formed in the internal language and, moreover, the signature is principal.

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

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

Proof: By straightforward induction on the elaboration algorithm. ■

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

Rule 15 is interesting because it demonstrates the importance of static purity. The elaborator invariant requires that modules synthesized by signature coercion be determinate (because they are often fed back in as inputs), but in the case that $\delta = \mathtt{gen}$, the synthesized lambda is determinate only because the type system can recognize that its body is statically pure and its dynamic impurity is captured by the lambda.

Although our elaborator serves only to deal with the avoidance problem, a realistic elaborator would also address other issues such as coercive signature matching (where a field is either dropped or made less polymorphic), `open`, type inference, datatypes, and so forth [10]. We believe our elaborator extends to cover all these issues without difficulty.

4.1.2 Primitive Existential Signatures

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

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

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

and

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

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

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

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

and

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

holds if and only if there exists determinate $M : \sigma$ such that the types $\tau[M/s]$ and $\tau'[M/s]$ are equal. Thus, deciding subsignature or equivalence queries in the presence of existentials would be as hard as higher-order unification, which is known to be undecidable [7].

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

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 $\sigma_2 = \mathfrak{S}([\tau]) \rightarrow \mathfrak{S}([\tau'])$ 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). Unfortunately, fully general first-class modules present difficulties for static typing [15].

One practical approach to modules as first-class values was suggested by Mitchell, *et al.* [22], who propose that second-class modules automatically be wrapped as existential packages [23] to obtain first-class values. A similar approach to modules as first-class values is described by Russo and implemented in Moscow ML [27].

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:

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

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

The main limitation of existentially-packaged modules is the closed-scope elimination construct. It has been observed repeatedly in the literature[5] that this construct is too restrictive to be 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 $\langle\!|\sigma|\!\rangle\rightarrow\tau$.

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:

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

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 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 10. 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 10.

[5]originally by MacQueen [17] and later by Cardelli and Leroy [3] and Lillibridge [15], among others

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

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

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

Figure 10: Packaged Module Extension

Although our improved extension is not definable explicitly in terms of an existential encoding, one may still view the package type $\langle\!|\sigma|\!\rangle$ as a generalized form of existential type, where the existential type variable corresponds to the opaque type components of σ. In particular, every unpacking of a package results in the generation of new abstract types, just as every opening of an existential does (albeit in a closed scope). Since Mitchell and Plotkin [23], it has often been suggested that existential types may be used to model type generativity. This notion is relevant to our system in that we can *almost* eliminate the need for generative functor signatures by encoding them in terms of package types.

Specifically, one might think of a generative functor as a regular (applicative) functor whose body is packaged as a term. Formally, $\Pi^{\text{gen}} s{:}\sigma_1.\sigma_2$ would be interpreted as $\Pi s{:}\sigma_1.[\![\langle\!|\sigma_2|\!\rangle]\!]$. Applying generative functor $F : \Pi^{\text{gen}} s{:}\sigma_1.\sigma_2$ to $M : \sigma_1$ would then be encoded as $\mathsf{unpack}\ \mathsf{Val}\,(FM)\ \mathsf{as}\ \sigma_2$. Thus, the dynamic effects encapsulated in generative functors and released in their application would be relayed through packing and unpacking.

We decided against employing this encoding for two reasons. First, discerning between applicative and generative functor application requires knowledge of the functor's signature—*i.e.*, it is not a purely syntactic encoding. Moreover, the encoding does not preserve the subtyping relation on generative functor signatures. If $\sigma_1 \leq \sigma_2$, then $\Pi^{\text{gen}} s{:}\sigma.\sigma_1 \leq \Pi^{\text{gen}} s{:}\sigma.\sigma_2$ in our type system, but not under the encoding. The problem is that the subsignature relationship $\sigma_1 \leq \sigma_2$ does not induce a *subtyping* relationship on the package types $\langle\!|\sigma_1|\!\rangle$ and $\langle\!|\sigma_2|\!\rangle$, since our language only provides subtyping on signatures. In addition, applicative functor signatures are no longer subtypes of generative functor signatures under the encoding, since σ and $[\![\langle\!|\sigma|\!\rangle]\!]$ are incomparable signatures. Although we believe it would be possible to work around these issues if the encoding were incorporated into an elaboration algorithm, we have opted instead for a purely type-theoretic treatment of generative functors.

6 Related Work

Harper, Mitchell and Moggi [9] 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 determinate).

MacQueen and Tofte [18] proposed a higher-order module extension to the original Definition of Standard ML [20], which was implemented in the Standard ML of New Jersey compiler. Their semantics involves a two-phase elaboration process, in which higher-order functors are re-elaborated at each application to take advantage of additional information about their arguments. This advantage is balanced by the disadvantage of inhibiting type propagation in the presence of separate compilation since functors that are compiled separately from their applications cannot be re-elaborated. A more thorough comparison is difficult because MacQueen and Tofte employ a stamp-based semantics, which is difficult to transfer to a type-theoretic setting.

Focusing on controlled abstraction, but largely neglecting higher-order modules, Harper and Lillibridge [8]

```
# module type S = sig type t end
  module F = functor (X : S) ->
    struct type u = X.t type v = X.t end
  module G = functor (X : S) ->
    struct type u = X.t type v = u end
  module AppF = F((struct type t = int end : S))
  module AppG = G((struct type t = int end : S));;

(* Output of the Objective Caml 3.04 compiler *)
module type S = sig type t end
module F : functor (X : S) ->
  sig type u = X.t and v = X.t end
module G : functor (X : S) ->
  sig type u = X.t and v = u end
module AppF : sig type u and v end
module AppG : sig type u and v = u end
```

Figure 11: Encoding of the Avoidance Problem in O'Caml

and Leroy [11, 13] introduced the closely related concepts of *translucent sums* and *manifest types.* These mechanisms served as the basis of the module system in the revised Definition of Standard ML 1997 [21], and Harper and Stone [10] 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* [12], which enables one to give fully transparent signatures for many higher-order functors. Leroy's formalism defined determinacy by a syntactic restriction that functor applications appearing in type paths must be in named form. On one hand, this restriction provides a weak form of structure sharing in the sense that the 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 the Objective Caml dialect of ML [25]. The manifest type formalism, like the translucent sum formalism, does not address the avoidance problem, and consequently it lacks principal signatures. This can lead Objective Caml to anomalous behavior such as that illustrated in Figure 11, which implements an instance of the avoidance problem. Two functors `F` and `G` are defined that have equivalent, fully transparent principal signatures. However, when applied to the same indeterminate module, the signatures of the results `AppF` and `AppG` differ rather arbitrarily (seemingly based on some purely syntactic discrepancy between `F`'s and `G`'s signatures). Aside from this fact (and disregarding Objective Caml's signature fields in structures, which makes type-checking undecidable [15]), the module language of Objective Caml can be viewed as essentially a subsystem of Russo's system (below).

More recently, Russo [26] has proposed a type system for applicative functors that generalizes Leroy's in much the same way we do, by abandoning the named form restriction. We adopt his use of existential signatures to address the avoidance problem, although Russo simultaneously uses existentials to model generativity, which we do not. Russo has implemented his system as an experimental extension to the Moscow ML compiler [24], which already supports SML-style generative functors. One can view Moscow ML's module system as a subsystem of ours in which weak sealing is the only available form of abstraction and in which all modules are considered dynamically pure. As we observed in Section 3, a system with only dynamically pure modules can provide a faithful account of generativity only in the absence of functors. Thus, the introduction of applicative functors into Moscow ML has the effect of making its generative functors weaker, since one can defeat generativity by eta-expanding a generative functor into an applicative one. Russo has also implemented an extension to Moscow ML for packaging modules as first-class values [27].

His extension is very similar to the existential-packaging approach discussed in the beginning of Section 5, and suffers likewise from the limitations of the closed-scope unpacking construct.

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

7 Conclusion

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

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

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

- *Structure Sharing.* The original version of Standard ML [20] included a notion of module equivalence that was sensitive to the dynamic, as well as static, parts of the module. Since this would violate our fundamental commitment to phase separation, it appears difficult to give an intrinsic type-theoretic account of this notion. It is in fact 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. But this would be merely an elaboration convention, not a true type-theoretic account of structure sharing.

- *Recursive Modules.* An important direction for future research is to integrate recursive modules [5] into the present framework. The chief difficulty is to achieve decidable type checking in the presence of general recursively dependent signatures, or to isolate a practical sub-language that avoids these problems. Russo [28] has formalized an extension to Moscow ML along these lines, but it suffers from weak support for separate compilation.

- *Extensibility.* A great deal of research has been devoted to incorporating forms of extensible programming, common in object-oriented settings, into functional languages. Many of the resulting languages include some kind of class mechanism [25], whose type-theoretic interpretation is often complex. Others make use of multimethods, for which properties such as redundancy and exhaustiveness are difficult to track in the presence of modules [19]. Understanding how these different forms of extensibility relate to one another, and how they interact with higher-order modules, is a substantial problem for future work.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

[19] Todd Millstein and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *9th Workshop on Foundations of Object-Oriented Languages*, pages 1–12, Portland, Oregon, January 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

A Type System

To reduce the number of freshness side-conditions, we adopt the convention that a context may not bind the same variable more than once. The second premises of Rules 42 and 44 appear within curly braces to denote that they are redundant once we have proven Validity in Appendix B. They are built into the declarative system as a technical device to shorten some of the proofs (in particular, for Theorems B.14 and E.7).

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

$$\frac{}{\epsilon \vdash \text{ok}}\ (1) \qquad \frac{\Gamma \vdash \sigma \ \text{sig}}{\Gamma, s{:}\sigma \vdash \text{ok}}\ (2)$$

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

$$\frac{\Gamma \vdash_{\text{P}} M : [\![T]\!]}{\Gamma \vdash \mathsf{Typ}\, M \ \text{type}}\ (3) \qquad \frac{\Gamma, s{:}\sigma \vdash \tau \ \text{type}}{\Gamma \vdash \Pi s{:}\sigma.\tau \ \text{type}}\ (4) \qquad \frac{\Gamma \vdash \tau' \ \text{type} \quad \Gamma \vdash \tau'' \ \text{type}}{\Gamma \vdash \tau' \times \tau'' \ \text{type}}\ (5) \qquad \frac{\Gamma \vdash \sigma \ \text{sig}}{\Gamma \vdash \langle\!|\sigma|\!\rangle \ \text{type}}\ (6)$$

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

$$\frac{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]}{\Gamma \vdash \tau_1 \equiv \tau_2}\ (7) \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2 \quad \Gamma, s{:}\sigma_1 \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \Pi s{:}\sigma_1.\tau_1 \equiv \Pi s{:}\sigma_2.\tau_2}\ (8)$$

$$\frac{\Gamma \vdash \tau_1' \equiv \tau_2' \quad \Gamma \vdash \tau_1'' \equiv \tau_2''}{\Gamma \vdash \tau_1' \times \tau_1'' \equiv \tau_2' \times \tau_2''}\ (9) \qquad \frac{\Gamma \vdash \sigma_1 \equiv \sigma_2}{\Gamma \vdash \langle\!|\sigma_1|\!\rangle \equiv \langle\!|\sigma_2|\!\rangle}\ (10)$$

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

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e : \tau}\ (11) \qquad \frac{\Gamma \vdash_{\kappa} M : [\![\tau]\!]}{\Gamma \vdash \mathsf{Val}\, M : \tau}\ (12) \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma \quad \Gamma, s{:}\sigma \vdash e : \tau \quad \Gamma \vdash \tau \ \text{type}}{\Gamma \vdash \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) : \tau}\ (13)$$

$$\frac{\Gamma, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\, f(s{:}\sigma){:}\tau.e : \Pi s{:}\sigma.\tau}\ (14) \qquad \frac{\Gamma \vdash e : \Pi s{:}\sigma.\tau \quad \Gamma \vdash_{\text{P}} M : \sigma}{\Gamma \vdash e\, M : \tau[M/s]}\ (15) \qquad \frac{\Gamma \vdash e' : \tau' \quad \Gamma \vdash e'' : \tau''}{\Gamma \vdash \langle e', e'' \rangle : \tau' \times \tau''}\ (16)$$

$$\frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_1 e : \tau'}\ (17) \qquad \frac{\Gamma \vdash e : \tau' \times \tau''}{\Gamma \vdash \pi_2 e : \tau''}\ (18) \qquad \frac{\Gamma \vdash_{\kappa} M : \sigma}{\Gamma \vdash \mathsf{pack}\ M\ \mathsf{as}\ \langle\!|\sigma|\!\rangle : \langle\!|\sigma|\!\rangle}\ (19)$$

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

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash 1 \ \text{sig}}\ (20) \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash [\![T]\!] \ \text{sig}}\ (21) \qquad \frac{\Gamma \vdash \tau \ \text{type}}{\Gamma \vdash [\![\tau]\!] \ \text{sig}}\ (22) \qquad \frac{\Gamma \vdash_{\text{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M) \ \text{sig}}\ (23)$$

$$\frac{\Gamma, s{:}\sigma' \vdash \sigma'' \ \text{sig}}{\Gamma \vdash \Pi^{\delta} s{:}\sigma'.\sigma'' \ \text{sig}}\ (24) \qquad \frac{\Gamma, s{:}\sigma' \vdash \sigma'' \ \text{sig}}{\Gamma \vdash \Sigma s{:}\sigma'.\sigma'' \ \text{sig}}\ (25)$$

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

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash 1 \equiv 1}\ (26) \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash [\![T]\!] \equiv [\![T]\!]}\ (27) \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \equiv [\![\tau_2]\!]}\ (28) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \equiv \mathfrak{S}(M_2)}\ (29)$$

$$\frac{\Gamma \vdash \sigma_2' \equiv \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Pi^{\delta} s{:}\sigma_1'.\sigma_1'' \equiv \Pi^{\delta} s{:}\sigma_2'.\sigma_2''}\ (30) \qquad \frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \equiv \Sigma s{:}\sigma_2'.\sigma_2''}\ (31)$$

$\boxed{\textbf{Signature subtyping: } \Gamma \vdash \sigma_1 \leq \sigma_2}$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash 1 \leq 1}\ (32) \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash [\![T]\!] \leq [\![T]\!]}\ (33) \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \leq [\![\tau_2]\!]}\ (34)$$

$$\frac{\Gamma \vdash \sigma_2' \leq \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \text{ sig} \quad \delta_1 \sqsubseteq \delta_2}{\Gamma \vdash \Pi^{\delta_1} s{:}\sigma_1'.\sigma_1'' \leq \Pi^{\delta_2} s{:}\sigma_2'.\sigma_2''}\ (35)$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_2'' \text{ sig}}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \leq \Sigma s{:}\sigma_2'.\sigma_2''}\ (36)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M) \leq [\![T]\!]}\ (37) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \leq \mathfrak{S}(M_2)}\ (38)$$

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

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s)}\ (39) \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle : 1}\ (40) \qquad \frac{\Gamma \vdash \tau \text{ type}}{\Gamma \vdash_{\mathtt{P}} [\tau] : [\![T]\!]}\ (41) \qquad \frac{\Gamma \vdash e : \tau \quad \{\Gamma \vdash \tau \text{ type}\}}{\Gamma \vdash_{\mathtt{P}} [e : \tau] : [\![\tau]\!]}\ (42)$$

$$\frac{\Gamma, s{:}\sigma' \vdash_\kappa M : \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma'.M : \Pi s{:}\sigma'.\sigma''}\ (43) \qquad \frac{\Gamma, s{:}\sigma' \vdash_\kappa M : \sigma'' \quad \{\Gamma, s{:}\sigma' \vdash \sigma'' \text{ sig}\}}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma'.M : \Pi^{\text{gen}} s{:}\sigma'.\sigma''}\ (44)$$

$$\frac{\Gamma \vdash_\kappa F : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_\kappa FM : \sigma''[M/s]}\ (45) \qquad \frac{\Gamma \vdash_\kappa F : \Pi^{\text{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} FM : \sigma''[M/s]}\ (46)$$

$$\frac{\Gamma \vdash_\kappa M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_\kappa M'' : \sigma''}{\Gamma \vdash_\kappa \langle s = M', M''\rangle : \Sigma s{:}\sigma'.\sigma''}\ (47) \qquad \frac{\Gamma \vdash_\kappa M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_\kappa \pi_1 M : \sigma'}\ (48) \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''[\pi_1 M/s]}\ (49)$$

$$\frac{\Gamma \vdash e : \langle\!|\sigma|\!\rangle}{\Gamma \vdash_{\mathsf{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma}\ (50) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} (M :: \sigma) : \sigma}\ (51) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma}\ (52)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash_{\mathtt{P}} M : \mathfrak{S}(M)}\ (53) \qquad \frac{\Gamma, s{:}\sigma' \vdash_{\mathtt{P}} Ms : \sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\rho}{\Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\sigma''}\ (54) \qquad \frac{\Gamma \vdash_{\mathtt{P}} \pi_1 M : \sigma' \quad \Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''}{\Gamma \vdash_{\mathtt{P}} M : \sigma' \times \sigma''}\ (55)$$

$$\frac{\Gamma \vdash_\kappa M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_\kappa M'' : \sigma \quad \Gamma \vdash \sigma \text{ sig}}{\Gamma \vdash_\kappa \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) : \sigma}\ (56) \qquad \frac{\Gamma \vdash_{\kappa'} M : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash_\kappa M : \sigma}\ (57)$$

$\boxed{\textbf{Module equivalence: } \Gamma \vdash M_1 \cong M_2 : \sigma}$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : \sigma}{\Gamma \vdash M \cong M : \sigma}\ (58) \qquad \frac{\Gamma \vdash M_2 \cong M_1 : \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma}\ (59) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma \quad \Gamma \vdash M_2 \cong M_3 : \sigma}{\Gamma \vdash M_1 \cong M_3 : \sigma}\ (60)$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]}\ (61) \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash [\mathsf{Typ}\ M] \cong M : [\![T]\!]}\ (62)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M_1 : \sigma \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma \quad \sigma \text{ is unitary}}{\Gamma \vdash M_1 \cong M_2 : \sigma}\ (63)$$

$$\frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash M_1 \cong M_2 : \sigma''}{\Gamma \vdash \lambda s{:}\sigma_1'.M_1 \cong \lambda s{:}\sigma_2'.M_2 : \Pi s{:}\sigma_1'.\sigma''} \; (64) \qquad \frac{\Gamma \vdash F_1 \cong F_2 : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash M_1 \cong M_2 : \sigma'}{\Gamma \vdash F_1 M_1 \cong F_2 M_2 : \sigma''[M_1/s]} \; (65)$$

$$\frac{\Gamma \vdash M_1' \cong M_2' : \sigma' \quad \Gamma, s{:}\sigma' \vdash M_1'' \cong M_2'' : \sigma''}{\Gamma \vdash \langle s = M_1', M_1'' \rangle \cong \langle s = M_2', M_2'' \rangle : \Sigma s{:}\sigma'.\sigma''} \; (66)$$

$$\frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma'} \; (67) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''[\pi_1 M_1/s]} \; (68)$$

$$\frac{\Gamma, s{:}\sigma' \vdash M_1 s \cong M_2 s : \sigma'' \quad \Gamma \vdash_{\mathrm{P}} M_1 : \Pi s{:}\sigma'.\rho_1 \quad \Gamma \vdash_{\mathrm{P}} M_2 : \Pi s{:}\sigma'.\rho_2}{\Gamma \vdash M_1 \cong M_2 : \Pi s{:}\sigma'.\sigma''} \; (69)$$

$$\frac{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma' \quad \Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''}{\Gamma \vdash M_1 \cong M_2 : \sigma' \times \sigma''} \; (70)$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\mathrm{P}} M'' : \sigma \quad \Gamma \vdash \sigma \text{ sig}}{\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \cong M''[M'/s] : \sigma} \; (71)$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M_1 : \mathfrak{S}(M_2)}{\Gamma \vdash M_1 \cong M_2 : \mathfrak{S}(M_2)} \; (72) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma} \; (73)$$

B Declarative Properties

This section is devoted to proving the essential properties of the declarative type system given in Appendix A. To minimize the amount of reproving, we have designed our type system so that the determinate (pure) fragment is as close as possible to the system of Stone and Harper [31], which we refer to hereafter as SH. Most of the theorems and proofs in this section are exactly analogous with the development in Appendix B of SH. Thus, the proofs here only give the (nontrivial) new cases, and we refer the reader to SH for the majority of them.

B.1 Preliminaries

Throughout we will write $\Gamma \vdash \mathcal{J}$ to denote any judgment with right hand side $\mathcal{J}$ (where $\mathcal{J}$ includes the purity level κ in module typing judgments). In addition, we use the "=" sign to indicate syntactic equality (modulo α-equivalence).

Proposition B.1 (Subderivations)

1. *Every proof of* $\Gamma \vdash \mathcal{J}$ *contains a subderivation of* $\Gamma \vdash \mathrm{ok}$.

2. *Every proof of* $\Gamma_1, s{:}\sigma, \Gamma_2 \vdash \mathcal{J}$ *contains a strict subderivation of* $\Gamma_1 \vdash \sigma$ sig.

Proof: By induction on derivations. ∎

Proposition B.2 (Free Variable Containment)
If $\Gamma \vdash \mathcal{J}$*, then* $FV(\mathcal{J}) \subseteq \mathrm{dom}(\Gamma)$.

Proof: By induction on derivations. ∎

- $\Delta \vdash \gamma : \Gamma$ iff
  1. $\Delta \vdash \text{ok}$
  2. And, $\forall s \in \text{dom}(\Gamma).\ \Delta \vdash_{\text{P}} \gamma s : \gamma(\Gamma(s))$
- $\Delta \vdash \gamma_1 \cong \gamma_2 : \Gamma$ iff
  1. $\Delta \vdash \text{ok}$
  2. And, $\forall s \in \text{dom}(\Gamma).\ \Delta \vdash \gamma_1 s \cong \gamma_2 s : \gamma_1(\Gamma(s))$

Figure 12: Typing and Equivalence Judgments for Substitutions

Proposition B.3 (Reflexivity)

1. *If* $\Gamma \vdash \tau$ type, *then* $\Gamma \vdash \tau \equiv \tau$.
2. *If* $\Gamma \vdash \sigma$ sig, *then* $\Gamma \vdash \sigma \equiv \sigma$ *and* $\Gamma \vdash \sigma \leq \sigma$.
3. *If* $\Gamma \vdash_{\text{P}} M : \sigma$, *then* $\Gamma \vdash M \cong M : \sigma$.

Proof: By induction on derivations. ■

Definition B.4 (Context/World Extension)
The context/world Γ_2 *is defined to* extend *the context* Γ_1 *(written* $\Gamma_2 \supseteq \Gamma_1$*) if the contexts viewed as partial functions give the same result for every module variable* s *in* $\text{dom}(\Gamma_1)$. *(Note that this is a purely syntactic condition and does not imply that either context is well-formed.)*

Proposition B.5 (Weakening)

1. *If* $\Gamma_1 \vdash \mathcal{J}$, $\Gamma_2 \supseteq \Gamma_1$, *and* $\Gamma_2 \vdash \text{ok}$, *then* $\Gamma_2 \vdash \mathcal{J}$.
2. *If* $\Gamma_1, s{:}\sigma_2, \Gamma_2 \vdash \mathcal{J}$, $\Gamma_1 \vdash \sigma_1 \leq \sigma_2$ *and* $\Gamma_1 \vdash \sigma_1$ sig, *then* $\Gamma_1, s{:}\sigma_1, \Gamma_2 \vdash \mathcal{J}$.

Proof: By induction on the derivation of the first premise. ■

Substitutions γ are defined as maps from variables to modules, which may be applied (in the usual capture-avoiding manner) to arbitrary syntactic expressions. We denote the identity substitution as **id** and substitution extension as $\gamma[s \mapsto M]$. Figure 12 defines typing and equivalence judgments for substitutions.

Proposition B.6 (Substitution)

1. *If* $\Gamma \vdash \mathcal{J}$ *and* $\Delta \vdash \gamma : \Gamma$, *then* $\Delta \vdash \gamma(\mathcal{J})$.
2. *If* $\Gamma_1, s{:}\sigma, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash_{\text{P}} M : \sigma$, *then* $\Gamma_1, \Gamma_2[M/s] \vdash \mathcal{J}[M/s]$.

Proof:

1. By induction on the derivation of the first premise.
2. By induction on the derivation of the first premise. Let $\Gamma = \Gamma_1, s{:}\sigma, \Gamma_2$. If $\mathcal{J} = \text{ok}$, then the proof is straightforward by cases on Γ_2. Otherwise, $\Gamma \vdash \text{ok}$ is a strict subderivation. Let $\Delta = \Gamma_1, \Gamma_2[M/s]$. By induction, $\Delta \vdash \text{ok}$. Let $\gamma = \mathbf{id}[s \mapsto M]$. It is easy to see that $\Delta \vdash \gamma : \Gamma$. The desired result then follows from Part 1.

■

Proposition B.7 (Properties of Type Equivalence)

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$, *then* $\Gamma \vdash \tau_2 \equiv \tau_1$.

2. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$ *and* $\Gamma \vdash \tau_2 \equiv \tau_3$*, then* $\Gamma \vdash \tau_1 \equiv \tau_3$*.*

3. *If* $\Gamma \vdash M_1 \cong M_2 : [\![T]\!]$*, then* $\Gamma \vdash \mathsf{Typ}\, M_1 \equiv \mathsf{Typ}\, M_2$*.*

4. *If* $\Gamma \vdash \tau \text{ type}$*, then* $\Gamma \vdash \mathsf{Typ}[\tau] \equiv \tau$*.*

5. *If* $\Gamma \vdash_{\mathrm{P}} [\tau] : \sigma$*, where* σ *is either* $[\![T]\!]$ *or* $\mathfrak{S}(M)$*, then* $\Gamma \vdash \tau \text{ type}$*.*

Proof:

1-2. By Rules 7 and 61, $\Gamma \vdash \tau_1 \equiv \tau_2$ if and only if $\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]$, and module equivalence is symmetric and transitive.

3. By Rule 62, $\Gamma \vdash [\mathsf{Typ}\, M_1] \cong M_1 : [\![T]\!]$ and $\Gamma \vdash [\mathsf{Typ}\, M_2] \cong M_2 : [\![T]\!]$. By symmetry and transitivity, $\Gamma \vdash [\mathsf{Typ}\, M_1] \cong [\mathsf{Typ}\, M_2] : [\![T]\!]$. The desired result follows by Rule 7.

4. By Rule 62, $\Gamma \vdash [\mathsf{Typ}[\tau]] \cong [\tau] : [\![T]\!]$. The desired result follows by Rule 7.

5. By induction on the derivation of the premise.

■

The following definition of the sizes of signatures has the property that it is invariant under substitution, *i.e.*, $size(\sigma) = size(\sigma[M/s])$.

Definition B.8 (Sizes of Signatures)
Let the size *of a signature* σ *(written* $size(\sigma)$*) be defined inductively as follows:*

$$\begin{array}{lcl}
size(1) & \stackrel{\text{def}}{=} & 1 \\
size([\![T]\!]) & \stackrel{\text{def}}{=} & 1 \\
size([\![\tau]\!]) & \stackrel{\text{def}}{=} & 2 \\
size(\mathfrak{S}(M)) & \stackrel{\text{def}}{=} & 2 \\
size(\Pi^{\delta} s{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & 2 + size(\sigma_1) + size(\sigma_2) \\
size(\Sigma s{:}\sigma_1.\sigma_2) & \stackrel{\text{def}}{=} & 2 + size(\sigma_1) + size(\sigma_2)
\end{array}$$

B.2 Validity and Functionality

We are now prepared to prove the critical and nontrivial properties of Validity and Functionality. Validity states that every expression appearing within a provable judgment is well-formed. Functionality states that applying equivalent substitutions to equivalent expressions yields equivalent expressions. Given Validity, it is straightforward to prove Functionality directly. Unfortunately, the unavoidable asymmetry in rules such as Rule 68 forces Validity to depend on Functionality as well (see Section 3.2 of Stone's thesis for a more detailed description of the problem).

To prove the two properties simultaneously, we define a strengthened equivalence relation in Figure 13. For modules and types, the relation takes the form of a strengthened induction hypothesis for Validity, ensuring that the modules or types in question are not only equivalent, but well-formed. For signatures, we employ a Kripke-style logical relation that ensures the signatures are functional in their free variables. The proof involves a fairly standard form of logical relations argument.

Lemma B.9 (Monotonicity)
Suppose $\Delta' \vdash \text{ok}$ *and* $\Delta' \supseteq \Delta$*.*

1. *If* $\tau_1 \text{ is } \tau_2\ [\Delta]$*, then* $\tau_1 \text{ is } \tau_2\ [\Delta']$*.*

2. *If* $M_1 \text{ is } M_2 \text{ in } \sigma\ [\Delta]$*, then* $M_1 \text{ is } M_2 \text{ in } \sigma\ [\Delta']$*.*

3. *If* $\sigma_1 \text{ is } \sigma_2\ [\Delta]$*, then* $\sigma_1 \text{ is } \sigma_2\ [\Delta']$*.*

- σ_1 is σ_2 $[\Delta]$ iff
  1. $\Delta \vdash \mathrm{ok}$
  2. And,
     - $\sigma_1 = 1$ and $\sigma_2 = 1$
     - Or, $\sigma_1 = [\![T]\!]$ and $\sigma_2 = [\![T]\!]$
     - Or, $\sigma_1 = [\![\tau_1]\!]$ and $\sigma_2 = [\![\tau_2]\!]$ and τ_1 is τ_2 $[\Delta]$
     - Or, $\sigma_1 = \mathfrak{S}(M_1)$ and $\sigma_2 = \mathfrak{S}(M_2)$ and M_1 is M_2 in $[\![T]\!]$ $[\Delta]$
     - Or, $\sigma_1 = \Pi^\delta s{:}\sigma_1'.\sigma_1''$ and $\sigma_2 = \Pi^\delta s{:}\sigma_2'.\sigma_2''$ and σ_1' is σ_2' $[\Delta]$ and $\forall \Delta' \supseteq \Delta$ if M_1 is M_2 in σ_1' $[\Delta']$ then $\sigma_1''[M_1/s]$ is $\sigma_2''[M_2/s]$ $[\Delta']$
     - Or, $\sigma_1 = \Sigma s{:}\sigma_1'.\sigma_1''$ and $\sigma_2 = \Sigma s{:}\sigma_2'.\sigma_2''$ and σ_1' is σ_2' $[\Delta]$ and $\forall \Delta' \supseteq \Delta$ if M_1 is M_2 in σ_1' $[\Delta']$ then $\sigma_1''[M_1/s]$ is $\sigma_2''[M_2/s]$ $[\Delta']$
- τ_1 is τ_2 $[\Delta]$ iff
  1. $\Delta \vdash \tau_1 \equiv \tau_2$
  2. And, $\Delta \vdash \tau_1$ type
  3. And, $\Delta \vdash \tau_2$ type
- M_1 is M_2 in σ $[\Delta]$ iff
  1. $\Delta \vdash M_1 \cong M_2 : \sigma$
  2. And, $\Delta \vdash_{\mathtt{P}} M_1 : \sigma$
  3. And, $\Delta \vdash_{\mathtt{P}} M_2 : \sigma$
- γ_1 is γ_2 in Γ $[\Delta]$ iff
  1. $\Delta \vdash \mathrm{ok}$
  2. And, $\forall s \in \mathrm{dom}(\Gamma).\ \gamma_1(\Gamma(s))$ is $\gamma_2(\Gamma(s))$ $[\Delta]$
  3. And, $\forall s \in \mathrm{dom}(\Gamma).\ \gamma_1 s$ is $\gamma_2 s$ in $\gamma_1(\Gamma(s))$ $[\Delta]$

Figure 13: Logical Relations for Declarative Properties

4. *If* γ_1 is γ_2 in Γ $[\Delta]$, *then* γ_1 is γ_2 in Γ $[\Delta']$.

Proof:

1-2. By Weakening.

3. By induction on the sizes of σ_1 and σ_2, and by Parts 1 and 2.

4. By Parts 2 and 3.

■

Lemma B.10 (Properties of Related Signatures)
If σ_1 is σ_2 $[\Delta]$, *then* $\Delta \vdash \sigma_1$ sig, $\Delta \vdash \sigma_2$ sig, $\Delta \vdash \sigma_1 \equiv \sigma_2$, $\Delta \vdash \sigma_1 \leq \sigma_2$, *and* $\Delta \vdash \sigma_2 \leq \sigma_1$.

Proof: See proof of SH Lemma B.6. All new cases are trivial, except for Π^{gen}, for which the proof is the same as in the Π case. ■

Corollary B.11 (Logical Subsumption)
If M_1 is M_2 in σ_1 $[\Delta]$ *and* σ_1 is σ_2 $[\Delta]$, *then* M_1 is M_2 in σ_2 $[\Delta]$.

Corollary B.12 (Properties of Related Substitutions)
If γ_1 is γ_2 in Γ $[\Delta]$, *then* $\Delta \vdash \gamma_1 \cong \gamma_2 : \Gamma$, $\Delta \vdash \gamma_1 : \Gamma$ *and* $\Delta \vdash \gamma_2 : \Gamma$.

Lemma B.13 (Symmetry and Transitivity of Logical Relations)

1. *If* τ_1 is τ_2 $[\Delta]$, *then* τ_2 is τ_1 $[\Delta]$.
2. *If* τ_1 is τ_2 $[\Delta]$ *and* τ_2 is τ_3 $[\Delta]$, *then* τ_1 is τ_3 $[\Delta]$.
3. *If* M_1 is M_2 in σ $[\Delta]$, *then* M_2 is M_1 in σ $[\Delta]$.
4. *If* M_1 is M_2 in σ $[\Delta]$ *and* M_2 is M_3 in σ $[\Delta]$, *then* M_1 is M_3 in σ $[\Delta]$.
5. *If* σ_1 is σ_2 $[\Delta]$, *then* σ_2 is σ_1 $[\Delta]$.
6. *If* σ_1 is σ_2 $[\Delta]$ *and* σ_2 is σ_3 $[\Delta]$, *then* σ_1 is σ_3 $[\Delta]$.
7. *If* γ_1 is γ_2 in Γ $[\Delta]$, *then* γ_2 is γ_1 in Γ $[\Delta]$.
8. *If* γ_1 is γ_2 in Γ $[\Delta]$ *and* γ_2 is γ_3 in Γ $[\Delta]$, *then* γ_1 is γ_3 in Γ $[\Delta]$.

Proof:

1-2. By Parts 1 and 2 of Proposition B.7.

3-8. See proof of SH Lemma B.8. All new cases are trivial, except for Π^{gen}, for which the proof is the same as in the Π case.

■

It is worth noting that the existence of the redundant premises in Rules 42 and 44 allow the following statement of the Fundamental Theorem of Logical Relations to avoid mentioning the typing judgments for terms and indeterminate modules.

Theorem B.14 (Fundamental Theorem of Logical Relations)
Suppose γ_1 is γ_2 in Γ $[\Delta]$.

1. *If* $\Gamma \vdash \tau$ type, *then* $\gamma_1\tau$ is $\gamma_2\tau$ $[\Delta]$.

2. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$*, then* $\gamma_1\tau_1$ is $\gamma_2\tau_2$ $[\Delta]$.

3. *If* $\Gamma \vdash \sigma$ sig*, then* $\gamma_1\sigma$ is $\gamma_2\sigma$ $[\Delta]$.

4. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$*, then* $\gamma_1\sigma_1$ is $\gamma_2\sigma_2$ $[\Delta]$.

5. *If* $\Gamma \vdash \sigma_1 \leq \sigma_2$*, then* $\gamma_1\sigma_1$ is $\gamma_2\sigma_1$ $[\Delta]$, $\gamma_1\sigma_2$ is $\gamma_2\sigma_2$ $[\Delta]$*, and* $\Delta \vdash \gamma_1\sigma_1 \leq \gamma_2\sigma_2$.

6. *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$*, then* $\gamma_1 M$ is $\gamma_2 M$ in $\gamma_1\sigma$ $[\Delta]$ *and* $\gamma_1\sigma$ is $\gamma_2\sigma$ $[\Delta]$.

7. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$*, then* $\gamma_1 M_1$ is $\gamma_2 M_2$ in $\gamma_1\sigma$ $[\Delta]$ *and* $\gamma_1\sigma$ is $\gamma_2\sigma$ $[\Delta]$.

Proof: By induction on derivations. See proof of SH Theorem B.9. All new cases are straightforward, with the following exceptions:

- Case: Rule 3.
  1. By IH, $\gamma_1 M$ is $\gamma_2 M$ in $[\![T]\!]$ $[\Delta]$.
  2. So $\Delta \vdash_{\mathsf{P}} \gamma_1 M : [\![T]\!]$, $\Delta \vdash_{\mathsf{P}} \gamma_2 M : [\![T]\!]$, and $\Delta \vdash \gamma_1 M \cong \gamma_2 M : [\![T]\!]$.
  3. By Rule 3, $\Delta \vdash \mathsf{Typ}\,\gamma_1 M$ type and $\Delta \vdash \mathsf{Typ}\,\gamma_2 M$ type.
  4. Then, by Part 3 of Proposition B.7, $\Delta \vdash \mathsf{Typ}\,\gamma_1 M \equiv \mathsf{Typ}\,\gamma_2 M$.

- Case: Rule 42.
  1. By IH, $\gamma_1\tau$ is $\gamma_2\tau$ $[\Delta]$, so $[\![\gamma_1\tau]\!]$ is $[\![\gamma_2\tau]\!]$ $[\Delta]$.
  2. By Corollary B.12, $\Delta \vdash \gamma_1 : \Gamma$ and $\Delta \vdash \gamma_2 : \Gamma$.
  3. So, by Substitution, $\Delta \vdash_{\mathsf{P}} \gamma_1([e : \tau]) : [\![\gamma_1\tau]\!]$ and $\Delta \vdash_{\mathsf{P}} \gamma_2([e : \tau]) : [\![\gamma_2\tau]\!]$,
  4. By Lemma B.10 and subsumption, $\Delta \vdash_{\mathsf{P}} \gamma_2([e : \tau]) : [\![\gamma_1\tau]\!]$
  5. Thus, by Rule 63, $\Delta \vdash \gamma_1([e : \tau]) \cong \gamma_2([e : \tau]) : [\![\gamma_1\tau]\!]$.

- Case: Rule 44. Similar to the proof for Rule 42.

- Case: Rule 7.
  1. By IH, $[\gamma_1\tau_1]$ is $[\gamma_2\tau_2]$ in $[\![T]\!]$ $[\Delta]$.
  2. So $\Delta \vdash_{\mathsf{P}} [\gamma_1\tau_1] : [\![T]\!]$, $\Delta \vdash_{\mathsf{P}} [\gamma_2\tau_2] : [\![T]\!]$, and $\Delta \vdash [\gamma_1\tau_1] \cong [\gamma_2\tau_2] : [\![T]\!]$.
  3. By Rule 7, $\Delta \vdash \gamma_1\tau_1 \equiv \gamma_2\tau_2$.
  4. Then, by Part 5 of Proposition B.7, $\Delta \vdash \gamma_1\tau_1$ type and $\Delta \vdash \gamma_2\tau_2$ type.

■

Lemma B.15 (Identity Substitution is Related to Itself)
If $\Gamma \vdash$ ok*, then* **id** is **id** in Γ $[\Gamma]$.

Proof: By induction on the derivation of $\Gamma \vdash$ ok.

- Case: Rule 1. Trivial.

- Case: Rule 2.
  1. $\Gamma \vdash$ ok is a strict subderivation, so by IH, **id** is **id** in Γ $[\Gamma]$.
  2. By Monotonicity, **id** is **id** in Γ $[\Gamma, s{:}\sigma]$.
  3. By Theorem B.14, σ is σ $[\Gamma, s{:}\sigma]$.
  4. Since $\Gamma, s{:}\sigma \vdash_{\mathsf{P}} s : \sigma$ and $\Gamma, s{:}\sigma \vdash s \cong s : \sigma$,
  5. we have s is s in σ $[\Gamma, s{:}\sigma]$.

6. Thus, **id** is **id** in $\Gamma, s{:}\sigma$ $[\Gamma, s{:}\sigma]$.

■

Corollary B.16 (Validity)

1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$, *then* $\Gamma \vdash \tau_1$ type *and* $\Gamma \vdash \tau_2$ type.
2. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $\Gamma \vdash \sigma_1$ sig *and* $\Gamma \vdash \sigma_2$ sig.
3. *If* $\Gamma \vdash \sigma_1 \leq \sigma_2$, *then* $\Gamma \vdash \sigma_1$ sig *and* $\Gamma \vdash \sigma_2$ sig.
4. *If* $\Gamma \vdash_{\mathrm{P}} M : \sigma$, *then* $\Gamma \vdash \sigma$ sig.
5. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$, *then* $\Gamma \vdash_{\mathrm{P}} M_1 : \sigma$, $\Gamma \vdash_{\mathrm{P}} M_2 : \sigma$, *and* $\Gamma \vdash \sigma$ sig.

Proof: By Theorem B.14, Lemma B.15, and Lemma B.10. ■

Corollary B.17 (Symmetry and Transitivity of Signature Equivalence)

1. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $\Gamma \vdash \sigma_2 \equiv \sigma_1$.
2. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$ *and* $\Gamma \vdash \sigma_2 \equiv \sigma_3$, *then* $\Gamma \vdash \sigma_1 \equiv \sigma_3$.

Corollary B.18 (Equivalence Implies Subtyping)
If $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $\Gamma \vdash \sigma_1 \leq \sigma_2$ *and* $\Gamma \vdash \sigma_2 \leq \sigma_1$.

Proposition B.19 (Transitivity of Signature Subtyping)
If $\Gamma \vdash \sigma_1 \leq \sigma_2$ *and* $\Gamma \vdash \sigma_2 \leq \sigma_3$, *then* $\Gamma \vdash \sigma_1 \leq \sigma_3$.

Proof: By induction on the sizes of the signatures involved, and by Validity. ■

Here we define equivalence and subtyping for contexts. Context equivalence is used in proving soundness of the equivalence algorithm defined in Appendix D, which maintains two contexts. The concept of context subtyping is useful in proving completeness of principal signature synthesis (Appendix C).

Definition B.20 (Context Equivalence)
Define Γ_1 *to be* equivalent *to* Γ_2 *(written* $\Gamma_1 \equiv \Gamma_2$*), inductively as follows:*

$$\frac{}{\epsilon \equiv \epsilon} \qquad \frac{\Gamma_1 \equiv \Gamma_2 \quad \Gamma_1 \vdash \sigma_1 \equiv \sigma_2}{\Gamma_1, s{:}\sigma_1 \equiv \Gamma_2, s{:}\sigma_2}$$

Definition B.21 (Context Subtyping)
Define Γ_1 *to be a* subcontext *of* Γ_2 *(written* $\Gamma_1 \leq \Gamma_2$*)*
iff $\Gamma_2 \vdash$ ok *and* $\forall s \in \mathrm{dom}(\Gamma_1) \subseteq \mathrm{dom}(\Gamma_2).\ \Gamma_2 \vdash \Gamma_2(s) \leq \Gamma_1(s)$.

Proposition B.22 (Properties of Context Subtyping and Equivalence)

1. *If* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_1 \vdash \mathcal{J}$, *then* $\Gamma_2 \vdash \mathcal{J}$.
2. *If* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \vdash \sigma$ sig, *then* $\Gamma_1 \leq \Gamma_2, s{:}\sigma$.
3. *If* $\Gamma_1 \leq \Gamma_2$ *and* $\Gamma_2 \vdash \sigma_2 \leq \sigma_1$, *then* $\Gamma_1, s{:}\sigma_1 \leq \Gamma_2, s{:}\sigma_2$.
4. *Context subtyping is reflexive and transitive.*
5. *Context equivalence is reflexive, symmetric and transitive, and implies context subtyping.*

Proof: Straightforward. ■

Proposition B.23 (Validity for Other Judgments)

1. *If* $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash \tau$ type.

2. *If* $\Gamma \vdash_\kappa M : \sigma$, *then* $\Gamma \vdash \sigma$ sig.

Proof: By induction on derivations, and by Validity. ■

Lemma B.24 (Equivalent Substitutions are Related)
If $\Gamma \vdash$ ok *and* $\Delta \vdash \gamma_1 \cong \gamma_2 : \Gamma$, *then* γ_1 is γ_2 in Γ $[\Delta]$.

Proof: By induction on the derivation of $\Gamma \vdash$ ok.

- Case: Rule 1. Trivial.
- Case: Rule 2.
  1. $\Gamma \vdash$ ok is a strict subderivation, so by IH, γ_1 is γ_2 in Γ $[\Delta]$.
  2. By Theorem B.14, $\gamma_1\sigma$ is $\gamma_2\sigma$ $[\Delta]$.
  3. By assumption, $\Delta \vdash \gamma_1 s \cong \gamma_2 s : \gamma_1\sigma$.
  4. By Validity, $\gamma_1 s$ is $\gamma_2 s$ in $\gamma_1\sigma$ $[\Delta]$.
  5. Therefore, γ_1 is γ_2 in $\Gamma, s{:}\sigma$ $[\Delta]$.

■

Corollary B.25 (Functionality)
Suppose $\Delta \vdash \gamma_1 \cong \gamma_2 : \Gamma$.

1. *If* $\Gamma \vdash \tau$ type, *then* $\Delta \vdash \gamma_1\tau \equiv \gamma_2\tau$.

2. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$, *then* $\Delta \vdash \gamma_1\tau_1 \equiv \gamma_2\tau_2$.

3. *If* $\Gamma \vdash \sigma$ sig, *then* $\Delta \vdash \gamma_1\sigma \equiv \gamma_2\sigma$.

4. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $\Delta \vdash \gamma_1\sigma_1 \equiv \gamma_2\sigma_2$.

5. *If* $\Gamma \vdash \sigma_1 \leq \sigma_2$, *then* $\Delta \vdash \gamma_1\sigma_1 \leq \gamma_2\sigma_2$.

6. *If* $\Gamma \vdash_{\mathrm{P}} M : \sigma$, *then* $\Delta \vdash \gamma_1 M \cong \gamma_2 M : \gamma_1\sigma$.

7. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$, *then* $\Delta \vdash \gamma_1 M_1 \cong \gamma_2 M_2 : \gamma_1\sigma$.

Proof: By Theorem B.14, Lemma B.24, and Lemma B.10. ■

B.3 Admissible Rules

Here we enumerate some important admissible rules, which fall into two categories. First, Proposition B.26 states that the rules of our type system involving singleton signatures extend to higher-order singletons (as defined in Figure 5). Since the well-formedness of $\mathfrak{S}_\sigma(M)$ does not necessarily imply that M has signature σ, the latter must be added as a premise to the higher-order variants of some of the rules.

Second, Proposition B.27 states that β- and η-equivalence rules for functions and products are admissible in our system, as well as giving an alternative formulation of the typing, equivalence and extensionality rules for products. The presentation of these rules is taken directly from Section 2.2 of Stone's thesis [30] and the proofs are nearly identical to those given in Section 3.2 of Stone's thesis. (The singleton system presented in Stone's thesis is essentially the same as that of SH, with only a few minor differences in presentation.)

Proposition B.26 (Higher-Order Singleton Rules)

1. $\gamma(\mathfrak{S}_\sigma(M)) = \mathfrak{S}_{\gamma\sigma}(\gamma M)$.
2. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$, *then* $\Gamma \vdash M_1 \cong M_2 : \mathfrak{S}_\sigma(M_2)$.
3. *If* $\Gamma \vdash_{\mathtt{P}} M : \sigma$, *then* $\Gamma \vdash \mathfrak{S}_\sigma(M)$ sig *and* $\Gamma \vdash_{\mathtt{P}} M : \mathfrak{S}_\sigma(M)$.
4. *If* $\Gamma \vdash_{\mathtt{P}} M_1 : \mathfrak{S}_\sigma(M_2)$ *and* $\Gamma \vdash_{\mathtt{P}} M_2 : \sigma$, *then* $\Gamma \vdash M_1 \cong M_2 : \mathfrak{S}_\sigma(M_2)$.
5. *If* $\Gamma \vdash_{\mathtt{P}} M : \sigma$, *then* $\Gamma \vdash \mathfrak{S}_\sigma(M) \leq \sigma$.
6. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma_1$ *and* $\Gamma \vdash \sigma_1 \leq \sigma_2$, *then* $\Gamma \vdash \mathfrak{S}_{\sigma_1}(M_1) \leq \mathfrak{S}_{\sigma_2}(M_2)$.

Proof: By induction on the size of σ in Parts 1-5 and σ_1 in Part 6. The proof is almost identical to the proofs of Lemma 3.3.2 and Proposition 3.3.3 in Stone's thesis. All new cases involve unitary signatures, and most involve trivial applications of Validity and/or Reflexivity, since a unitary signature $\sigma = \mathfrak{S}_\sigma(M)$. Here are the two cases which do not follow directly from Validity and Reflexivity.

4. • Case: σ is unitary. Follows by Rule 63.

6. • Case: $\sigma_1 = \Pi s{:}\sigma_1'.\sigma_1''$ and $\sigma_2 = \Pi^{\mathrm{gen}} s{:}\sigma_2'.\sigma_2''$, so $\mathfrak{S}_{\sigma_2}(M_2) = \sigma_2$.
   (a) By Validity and Part 5, $\Gamma \vdash \mathfrak{S}_{\sigma_1}(M_1) \leq \sigma_1$.
   (b) Thus, by transitivity, $\Gamma \vdash \mathfrak{S}_{\sigma_1}(M_1) \leq \sigma_2$.

■

Proposition B.27 (Admissibility of Beta, Eta, and Alternative Product Rules)

1. *If* $\Gamma, s{:}\sigma' \vdash_{\mathtt{P}} M : \sigma''$ *and* $\Gamma \vdash_{\mathtt{P}} M' : \sigma'$, *then* $\Gamma \vdash (\lambda s{:}\sigma'.M)M' \cong M[M'/s] : \sigma''[M'/s]$.
2. *If* $\Gamma, s{:}\sigma' \vdash M_1 \cong M_2 : \sigma''$ *and* $\Gamma \vdash M_1' \cong M_2' : \sigma'$, *then* $\Gamma \vdash (\lambda s{:}\sigma'.M_1)M_1' \cong M_2[M_2'/s] : \sigma''[M_1'/s]$.
3. *If* $\Gamma \vdash_{\mathtt{P}} M_1 : \sigma_1$ *and* $\Gamma, s{:}\sigma_1 \vdash_{\mathtt{P}} M_2 : \sigma_2$, *then* $\Gamma \vdash \pi_1\langle s = M_1, M_2\rangle \cong M_1 : \sigma_1$ *and* $\Gamma \vdash \pi_2\langle s = M_1, M_2\rangle \cong M_2[M_1/s] : \sigma_2[M_1/s]$.
4. *If* $\Gamma \vdash M_1 \cong M_1' : \sigma_1$ *and* $\Gamma, s{:}\sigma_1 \vdash M_2 \cong M_2' : \sigma_2$, *then* $\Gamma \vdash \pi_1\langle s = M_1, M_2\rangle \cong M_1' : \sigma_1$ *and* $\Gamma \vdash \pi_2\langle s = M_1, M_2\rangle \cong M_2'[M_1'/s] : \sigma_2[M_1'/s]$.
5. *If* $\Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\sigma''$, *then* $\Gamma \vdash M \cong \lambda s{:}\sigma'.M s : \Pi s{:}\sigma'.\sigma''$.
6. *If* $\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma'.\sigma''$, *then* $\Gamma \vdash M \cong \langle \pi_1 M, \pi_2 M\rangle : \Sigma s{:}\sigma'.\sigma''$.
7. *If* $\Gamma \vdash \Sigma s{:}\sigma'.\sigma''$ sig, $\Gamma \vdash_{\mathtt{P}} M' : \sigma'$, *and* $\Gamma \vdash_{\mathtt{P}} M'' : \sigma''[M'/s]$, *then* $\Gamma \vdash_{\mathtt{P}} \langle M', M''\rangle : \Sigma s{:}\sigma'.\sigma''$.
8. *If* $\Gamma \vdash \Sigma s{:}\sigma'.\sigma''$ sig, $\Gamma \vdash M_1' \cong M_2' : \sigma'$, *and* $\Gamma \vdash M_1'' \cong M_2'' : \sigma''[M_1'/s]$, *then* $\Gamma \vdash \langle M_1', M_1''\rangle \cong \langle M_2', M_2''\rangle : \Sigma s{:}\sigma'.\sigma''$.
9. *If* $\Gamma \vdash \Sigma s{:}\sigma'.\sigma''$ sig, $\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma'$, *and* $\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''[\pi_1 M_1/s]$, *then* $\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''$.

Proof: See proof of Proposition 3.3.4 and Part 6 of Proposition 3.3.3 of Stone's thesis. ■

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

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} s \Rightarrow \mathfrak{S}_{\Gamma(s)}(s)} \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle \Rightarrow 1} \qquad \frac{\Gamma \vdash \tau \ \text{type}}{\Gamma \vdash_{\mathtt{P}} [\tau] \Rightarrow \mathfrak{S}([\tau])} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash_{\mathtt{P}} [e : \tau] \Rightarrow [\![\tau]\!]}$$

$$\frac{\Gamma, s{:}\sigma' \vdash_\kappa M \Rightarrow \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma'.M \Rightarrow \Pi s{:}\sigma'.\sigma''} \qquad \frac{\Gamma, s{:}\sigma' \vdash_\kappa M \Rightarrow \sigma'' \quad \mathtt{S} \sqsubseteq \kappa}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma'.M \Rightarrow \Pi^{\text{gen}} s{:}\sigma'.\sigma''}$$

$$\frac{\Gamma \vdash_\kappa F \Rightarrow \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M \Leftarrow \sigma'}{\Gamma \vdash_\kappa FM \Rightarrow \sigma''[M/s]} \qquad \frac{\Gamma \vdash_\kappa F \Rightarrow \Pi^{\text{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M \Leftarrow \sigma'}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} FM \Rightarrow \sigma''[M/s]}$$

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

$$\frac{\Gamma \vdash_\kappa M \Rightarrow \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_\kappa \pi_1 M \Rightarrow \sigma'} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M \Rightarrow \sigma' \times \sigma''}{\Gamma \vdash_{\mathtt{P}} \pi_2 M \Rightarrow \sigma''}$$

$$\frac{\Gamma \vdash_\kappa M \Leftarrow \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} M :: \sigma \Rightarrow \sigma} \qquad \frac{\Gamma \vdash_\kappa M \Leftarrow \sigma}{\Gamma \vdash_{\mathtt{W}} M :> \sigma \Rightarrow \sigma} \qquad \frac{\Gamma \vdash e \Leftarrow \langle\!|\sigma|\!\rangle}{\Gamma \vdash_{\mathtt{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma \Rightarrow \sigma}$$

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

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

Figure 14: Module Typechecking and Principal Signature Synthesis

C Typechecking and Synthesis

In this section we give an algorithm for typechecking modules and prove it sound and complete with respect to the declarative system. To decide whether a module has a given signature, the algorithm synthesizes the principal signature of a module and then checks whether the principal signature is a subtype of the given signature. The module typechecking judgment is written $\Gamma \vdash_\kappa M \Leftarrow \sigma$, where κ is the minimal purity of M but σ is any signature assignable to M in context Γ. The principal signature synthesis judgment is written $\Gamma \vdash_\kappa M \Rightarrow \sigma$, where κ is the minimal purity of M and σ is the principal signature of M in context Γ.

The synthesis algorithm itself is very straightforward. The astute reader will notice, however, that the synthesis rules for atomic term modules and unpacked modules rely on an undefined judgment for term typechecking of the form $\Gamma \vdash e \Leftarrow \tau$. Eventually we will give a term typechecking algorithm to implement this as well (Figure 18), but for technical reasons it turns out that we cannot do so yet. The problem is that the algorithm for term typechecking relies on the ability to reduce types to a normal form, which is a consequence of the proof of decidability for type and module equivalence. For the moment, then, we will take $\Gamma \vdash e \Leftarrow \tau$ to be synonymous with $\Gamma \vdash e : \tau$. Once we have proven decidability of type and module equivalence, we will be able to fully define module typechecking and prove it decidable (Appendix F).

Theorem C.1 (Soundness of Module Typechecking/Synthesis)
If $\Gamma \vdash_\kappa M \Leftarrow \sigma$ *or* $\Gamma \vdash_\kappa M \Rightarrow \sigma$, *then* $\Gamma \vdash_\kappa M : \sigma$.

Proof: By straightforward induction on the typechecking/synthesis algorithm. The only non-trivial case is that of determinate products, which follows easily from selfification (Rule 55) and Functionality. ■

It turns out to be important that the principal signatures of determinate modules are in what we call *determinate synthesis form*, which essentially means that they do not contain dependent product (Σ) signatures, except under Π^{gen}. In particular, if the synthesis rule for second projections ($\pi_2 M$) is not allowed to assume that M's principal signature is a non-dependent product, then the completeness proof (see Theorem C.5 below) appears to break for very subtle reasons in the case of Rule 55. This is the reason we have a separate synthesis rule for determinate pairs, although we do not have a counterexample to completeness in the system with only one synthesis rule for pairs.

Definition C.2 (Determinate Synthesis Form)
A signature σ is in determinate synthesis form *if:*

- *σ is unitary, $[\![T]\!]$, or $\mathfrak{S}(M)$,*
- *Or, $\sigma = \Pi s{:}\sigma_1.\sigma_2$, where σ_2 is in determinate synthesis form,*
- *Or, $\sigma = \sigma_1 \times \sigma_2$, where σ_1 and σ_2 are in determinate synthesis form.*

Proposition C.3 (Properties of Principal Signature Synthesis)

1. *Synthesis is deterministic, i.e. if $\Gamma \vdash_{\kappa_1} M \Rightarrow \sigma_1$ and $\Gamma \vdash_{\kappa_2} M \Rightarrow \sigma_2$, then $\sigma_1 = \sigma_2$ and $\kappa_1 = \kappa_2$.*
2. *If $\Gamma \vdash_{\mathtt{P}} M \Rightarrow \sigma$, then σ is in determinate synthesis form.*

Proof: By straightforward induction on the typechecking/synthesis algorithm. ■

Lemma C.4 (Weakening for Module Typechecking/Synthesis Algorithm)

1. *If $\Gamma_1 \vdash_\kappa M \Rightarrow \sigma$, $\Gamma_1 \subseteq \Gamma_2$, and $\Gamma_2 \vdash \text{ok}$, then $\Gamma_2 \vdash_\kappa M \Rightarrow \sigma$.*
2. *If $\Gamma_1 \vdash_\kappa M \Leftarrow \sigma$, $\Gamma_1 \subseteq \Gamma_2$, and $\Gamma_2 \vdash \text{ok}$, then $\Gamma_2 \vdash_\kappa M \Leftarrow \sigma$.*

Proof: By straightforward induction on the typechecking/synthesis derivation. ■

The statement of completeness involves a strengthened induction hypothesis that makes use of context subtyping (Definition B.21). The ability to strengthen the context (thus weakening the resulting judgment) is required to prove completeness for Rules 47 and 56.

Theorem C.5 (Completeness of Module Typechecking/Synthesis)
If $\Gamma \vdash_\kappa M : \sigma$ and $\Gamma \leq \Gamma'$, then $\Gamma' \vdash_{\kappa'} M \Leftarrow \sigma$, where $\kappa' \sqsubseteq \kappa$. Moreover, if $\kappa' = \mathtt{P}$, then $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \mathfrak{S}_\sigma(M)$.

Proof: By induction on derivations.

- Case: Rule 39.
  1. We have $\Gamma' \vdash_{\mathtt{P}} s \Rightarrow \mathfrak{S}_{\Gamma'(s)}(s)$.
  2. By definition of context subtyping, $\Gamma' \vdash \Gamma'(s) \leq \Gamma(s)$.
  3. By Proposition B.26, $\Gamma' \vdash \mathfrak{S}_{\Gamma'(s)}(s) \leq \mathfrak{S}_{\Gamma(s)}(s)$.
- Case: Rules 40, 41, and 42. Trivial.
- Case: Rule 43.
  1. By IH, $\Gamma', s{:}\sigma' \vdash_{\kappa'} M \Rightarrow \rho$, where $\Gamma', s{:}\sigma' \vdash \rho \leq \sigma''$ and $\kappa' \sqsubseteq \kappa \sqsubseteq \mathtt{D}$.
  2. Thus, $\Gamma' \vdash_{\kappa'} \lambda s{:}\sigma'.M \Rightarrow \Pi s{:}\sigma'.\rho$ and $\Gamma' \vdash \Pi s{:}\sigma'.\rho \leq \Pi s{:}\sigma'.\sigma''$.
  3. If $\kappa' = \mathtt{P}$, then by IH, $\Gamma', s{:}\sigma' \vdash \rho \leq \mathfrak{S}_{\sigma''}(M)$.
  4. By Proposition B.27, $\Gamma', s{:}\sigma' \vdash (\lambda s{:}\sigma'.M)s \cong M : \sigma''$.
  5. So, by Proposition B.26, $\Gamma', s{:}\sigma' \vdash \mathfrak{S}_{\sigma''}(M) \leq \mathfrak{S}_{\sigma''}((\lambda s{:}\sigma'.M)s)$,

6. and $\Gamma' \vdash \Pi s{:}\sigma'.\rho \leq \mathfrak{S}_{\Pi s:\sigma'.\sigma''}(\lambda s{:}\sigma'.M)$.

- Case: Rule 44.
  1. By IH, $\Gamma', s{:}\sigma' \vdash_{\kappa'} M \Rightarrow \rho$, where $\Gamma', s{:}\sigma' \vdash \rho \leq \sigma''$, and $\kappa' \sqsubseteq \kappa$.
  2. If $\kappa' \sqsubseteq \mathtt{D}$, then $\Gamma' \vdash_{\kappa'} \lambda s{:}\sigma'.M \Rightarrow \Pi s{:}\sigma'.\rho$ and $\Gamma' \vdash \Pi s{:}\sigma'.\rho \leq \Pi^{\text{gen}} s{:}\sigma'.\sigma''$.
  3. Otherwise, $\Gamma' \vdash_{\kappa' \sqcap \mathtt{D}} \lambda s{:}\sigma'.M \Rightarrow \Pi^{\text{gen}} s{:}\sigma'.\rho'$, $\Gamma' \vdash \Pi^{\text{gen}} s{:}\sigma'.\rho \leq \Pi^{\text{gen}} s{:}\sigma'.\sigma''$, and $\kappa' \sqcap \mathtt{D} \sqsubseteq \kappa \sqcap \mathtt{D}$.
  4. Since $\mathfrak{S}_{\Pi^{\text{gen}} s:\sigma'.\sigma''}(\lambda s{:}\sigma'.M) = \Pi^{\text{gen}} s{:}\sigma'.\sigma''$, we are done.

- Case: Rule 45.
  1. By IH and inversion on subtyping, $\Gamma' \vdash_{\kappa'} F \Rightarrow \Pi s{:}\rho'.\rho''$,
  2. where $\Gamma' \vdash \Pi s{:}\rho.\rho'' \leq \Pi s{:}\sigma'.\sigma''$ and $\kappa' \sqsubseteq \kappa$.
  3. So, $\Gamma' \vdash \sigma' \leq \rho'$ and $\Gamma', s{:}\sigma' \vdash \rho'' \leq \sigma''$.
  4. By IH, $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \sigma'$, and so $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \rho'$.
  5. Thus, $\Gamma' \vdash_{\kappa'} FM \Rightarrow \rho''[M/s]$, and by Substitution, $\Gamma' \vdash \rho''[M/s] \leq \sigma''[M/s]$.
  6. If $\kappa' = \mathtt{P}$, then by IH, $\Gamma', s{:}\sigma' \vdash \rho'' \leq \mathfrak{S}_{\sigma''}(Fs)$.
  7. By Substitution and Proposition B.26, $\Gamma' \vdash \rho''[M'/s] \leq \mathfrak{S}_{\sigma''[M/s]}(FM)$.

- Case: Rule 46.
  1. By IH and inversion on subtyping, $\Gamma' \vdash_{\kappa'} F \Rightarrow \Pi^{\delta} s{:}\rho'.\rho''$,
  2. where $\Gamma' \vdash \Pi^{\delta} s{:}\rho.\rho'' \leq \Pi^{\text{gen}} s{:}\sigma'.\sigma''$ and $\kappa' \sqsubseteq \kappa$.
  3. So, $\Gamma' \vdash \sigma' \leq \rho'$ and $\Gamma', s{:}\sigma' \vdash \rho'' \leq \sigma''$.
  4. By IH, $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \sigma'$, and so $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \rho'$.
  5. By Substitution, $\Gamma' \vdash \rho''[M/s] \leq \sigma''[M/s]$.
  6. If $\delta \neq \text{gen}$, then $\Gamma' \vdash_{\kappa'} FM \Rightarrow \rho''[M/s]$.
  7. In the case that $\kappa' = \mathtt{P}$, then by IH, $\Gamma', s{:}\sigma' \vdash \rho'' \leq \mathfrak{S}_{\sigma''}(Fs)$.
  8. By Substitution and Proposition B.26, $\Gamma' \vdash \rho''[M'/s] \leq \mathfrak{S}_{\sigma''[M/s]}(FM)$.
  9. If $\delta = \text{gen}$, then $\Gamma' \vdash_{\kappa' \sqcup \mathtt{S}} FM \Rightarrow \rho''[M/s]$ and $\kappa' \sqcup \mathtt{S} \sqsubseteq \kappa \sqcup \mathtt{S}$.

- Case: Rule 47.
  1. Let $M = \langle s = M', M''\rangle$.
  2. By IH, $\Gamma' \vdash_{\kappa'} M' \Rightarrow \rho'$, where $\Gamma' \vdash \rho' \leq \sigma'$ and $\kappa' \sqsubseteq \kappa$.
  3. Since $\Gamma, s{:}\sigma' \leq \Gamma', s{:}\rho'$,
  4. by IH, $\Gamma', s{:}\rho' \vdash_{\kappa''} M'' \Rightarrow \rho''$, where $\Gamma', s{:}\rho' \vdash \rho'' \leq \sigma''$ and $\kappa'' \sqsubseteq \kappa$.
  5. If $\kappa' \sqcup \kappa'' \neq \mathtt{P}$, then we are done,
  6. since $\Gamma' \vdash_{\kappa' \sqcup \kappa''} M \Rightarrow \Sigma s{:}\rho'.\rho''$, $\Gamma' \vdash \Sigma s{:}\rho'.\rho'' \leq \Sigma s{:}\sigma'.\sigma''$, and $\kappa' \sqcup \kappa'' \sqsubseteq \kappa$.
  7. Otherwise, $\Gamma' \vdash_{\mathtt{P}} M \Rightarrow \rho' \times \rho''[M'/s]$.
  8. In addition, by IH, $\Gamma' \vdash \rho' \leq \mathfrak{S}_{\sigma'}(M')$ and $\Gamma', s{:}\rho' \vdash \rho'' \leq \mathfrak{S}_{\sigma''}(M'')$.
  9. By Soundness, $\Gamma' \vdash_{\mathtt{P}} M' : \rho'$ and $\Gamma', s{:}\rho' \vdash_{\mathtt{P}} M'' : \rho''$.
  10. By Proposition B.27, $\Gamma' \vdash \pi_1 M \cong M' : \rho'$ and $\Gamma' \vdash \pi_2 M \cong M''[M'/s] : \rho''[M'/s]$.
  11. By Proposition B.26, $\Gamma' \vdash \rho' \leq \mathfrak{S}_{\sigma'}(\pi_1 M)$.
  12. By Substitution, $\Gamma' \vdash \rho''[M'/s] \leq \mathfrak{S}_{\sigma''[M'/s]}(M''[M'/s])$.
  13. Then, by Functionality and Proposition B.26, $\Gamma' \vdash \rho''[M'/s] \leq \mathfrak{S}_{\sigma''[\pi_1 M/s]}(\pi_2 M)$.
  14. Thus, $\Gamma' \vdash \rho' \times \rho''[M'/s] \leq \mathfrak{S}_{\Sigma s:\sigma'.\sigma''}(M)$.

15. Since $\Gamma' \vdash \Sigma s{:}\rho'.\rho'' \leq \Sigma s{:}\sigma'.\sigma''$ and, by Soundness, $\Gamma' \vdash_{\mathtt{P}} M : \Sigma s{:}\rho'.\rho''$,
16. we have $\Gamma' \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma'.\sigma''$, and so $\Gamma' \vdash \mathfrak{S}_{\Sigma s:\sigma'.\sigma''}(M) \leq \Sigma s{:}\sigma'.\sigma''$.

- Case: Rule 48.
  1. By IH and inversion on subtyping, $\Gamma' \vdash_{\kappa'} M \Rightarrow \Sigma s{:}\rho.\rho''$,
  2. where $\Gamma' \vdash \Sigma s{:}\rho'.\rho'' \leq \Sigma s{:}\sigma'.\sigma''$ and $\kappa' \sqsubseteq \kappa$.
  3. Thus, $\Gamma' \vdash_{\kappa'} \pi_1 M \Rightarrow \rho$ and $\Gamma' \vdash \rho' \leq \sigma'$.
  4. If $\kappa' = \mathtt{P}$, then by IH, $\Gamma' \vdash \Sigma s{:}\rho'.\rho'' \leq \mathfrak{S}_{\Sigma s:\sigma'.\sigma''}(M)$,
  5. so $\Gamma' \vdash \rho' \leq \mathfrak{S}_{\sigma'}(\pi_1 M)$.
- Case: Rule 49.
  1. By IH, inversion on subtyping, and Proposition C.3, $\Gamma' \vdash_{\mathtt{P}} M \Rightarrow \rho' \times \rho''$,
  2. where $\Gamma' \vdash \rho' \times \rho'' \leq \mathfrak{S}_{\Sigma s:\sigma'.\sigma''}(M)$.
  3. Thus, $\Gamma' \vdash_{\mathtt{P}} \pi_2 M \Rightarrow \rho''$.
  4. By inversion, $\Gamma', s{:}\rho' \vdash \rho'' \leq \mathfrak{S}_{\sigma''[\pi_1 M/s]}(\pi_2 M)$.
  5. By Soundness, $\Gamma' \vdash_{\mathtt{P}} \pi_1 M : \rho'$.
  6. So by Substitution, $\Gamma' \vdash \rho'' \leq \mathfrak{S}_{\sigma''[\pi_1 M/s]}(\pi_2 M)$.
- Case: Rules 50, 51, 52, and 53. Trivial, by IH.
- Case: Rule 54.
  1. By IH and inversion on subtyping, $\Gamma' \vdash_{\mathtt{P}} M \Rightarrow \Pi s{:}\rho'.\rho''$, where $\Gamma' \vdash \sigma' \leq \rho'$.
  2. By Lemma C.4, $\Gamma', s{:}\sigma' \vdash_{\mathtt{P}} M \Rightarrow \Pi s{:}\rho'.\rho''$.
  3. Since $\Gamma', s{:}\sigma' \vdash_{\mathtt{P}} s \Leftarrow \rho'$, we have $\Gamma', s{:}\sigma' \vdash_{\mathtt{P}} M s \Rightarrow \rho''$.
  4. By IH and Proposition C.3, $\Gamma', s{:}\sigma' \vdash \rho'' \leq \mathfrak{S}_{\sigma''}(M s)$.
  5. Thus, $\Gamma' \vdash \Pi s{:}\rho'.\rho'' \leq \mathfrak{S}_{\Pi s:\sigma'.\sigma''}(M)$.
- Case: Rule 55.
  1. By IH, $\Gamma' \vdash_{\mathtt{P}} \pi_1 M \Rightarrow \rho'$ and $\Gamma' \vdash_{\mathtt{P}} \pi_2 M \Rightarrow \rho''$,
  2. where $\Gamma' \vdash \rho' \leq \mathfrak{S}_{\sigma'}(\pi_1 M)$ and $\Gamma' \vdash \rho'' \leq \mathfrak{S}_{\sigma''}(\pi_2 M)$.
  3. By inversion on synthesis and Proposition C.3, $\Gamma' \vdash_{\mathtt{P}} M \Rightarrow \rho' \times \rho''$.
  4. Since $\Gamma' \vdash \rho' \times \rho'' \leq \mathfrak{S}_{\sigma' \times \sigma''}(M)$, we are done.
- Case: Rule 56.
  1. Let $M = \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma)$.
  2. By IH, $\Gamma' \vdash_{\kappa'} M' \Rightarrow \rho'$, where $\Gamma' \vdash \rho' \leq \sigma'$ and $\kappa' \sqsubseteq \kappa$.
  3. Since $\Gamma, s{:}\sigma' \leq \Gamma', s{:}\rho'$,
  4. by IH, $\Gamma', s{:}\rho' \vdash_{\kappa''} M'' \Rightarrow \rho''$, where $\Gamma', s{:}\rho' \vdash \rho'' \leq \sigma$ and $\kappa'' \sqsubseteq \kappa$.
  5. If $\kappa' \sqcup \kappa'' \neq \mathtt{P}$, then $\Gamma' \vdash_{\kappa' \sqcup \kappa''} M \Rightarrow \sigma$ and $\kappa' \sqcup \kappa'' \sqsubseteq \kappa$.
  6. Otherwise, $\Gamma' \vdash_{\mathtt{P}} M \Rightarrow \mathfrak{S}_{\sigma}(M)$.
  7. By Soundness, $\Gamma' \vdash_{\mathtt{P}} M : \sigma$, so $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \sigma$ as well.
- Case: Rule 57.
  1. By IH, $\Gamma' \vdash_{\kappa''} M \Leftarrow \sigma'$, where $\kappa'' \sqsubseteq \kappa' \sqsubseteq \kappa$.
  2. Since $\Gamma' \vdash \sigma' \leq \sigma$, $\Gamma' \vdash_{\kappa''} M \Leftarrow \sigma$.
  3. If $\kappa'' = \mathtt{P}$, then by IH, $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \mathfrak{S}_{\sigma'}(M)$.
  4. By Soundness and Proposition B.26, $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \mathfrak{S}_{\sigma}(M)$.

$\blacksquare$

D An Algorithm for Deciding Equivalence

In this section we define an algorithm for deciding type, signature, and module equivalence, and prove it sound with respect to declarative equivalence. The algorithm, given in Figures 15 and 16, is nearly identical to the SH algorithm, with a few simple extensions. It makes use of two contexts and two signatures (which will always be equivalent when the algorithm is invoked) in order to ensure symmetry and transitivity of the algorithm. We refer to the reader to SH for discussion of how the algorithm works.

The interesting extensions consist of two new weak head reduction rules and two new kinds of paths. The reduction rule for $\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma)$ is the same as the one for $\pi_2\langle s = M', M''\rangle$, which corresponds to the intuition that the former is encodable as the latter when M' and M'' are determinate. There is also a weak head reduction rule for modules that merely contain a type projected from another module.

Module paths P are extended to include modules that consist solely of a product, function or package type. (The system in Stone's thesis achieves a similar effect by baking in $\times$ and $\rightarrow$ as type constructor constants of kind $T \rightarrow T \rightarrow T$, thus making $\tau_1 \times \tau_2$ and $\tau_1 \rightarrow \tau_2$ paths.) Formally, paths are defined as follows:

$$P ::= s \mid PM \mid \pi_1 P \mid \pi_2 P \mid [\Pi s{:}\sigma.\tau] \mid [\tau' \times \tau''] \mid [\langle\!|\sigma|\!\rangle]$$

Also note that any two modules of unitary signature are deemed equivalent by the algorithm automatically, as are any two modules of singleton signature.

The structure of our soundness proof is based closely on the SH soundness proof (SH Section 3). We simplify the proofs of the lemmas considerably, however, by performing induction on synthesis derivations instead of declarative derivations. We believe the same technique may be used to simplify the SH proof as well.

Lemma D.1 (Correspondence Between Natural and Principal Signatures)
If $\Gamma \vdash_{\mathsf{P}} P \Rightarrow \sigma$, then $\Gamma \vdash P \uparrow \sigma'$, where $\Gamma \vdash_{\mathsf{P}} P : \sigma'$ and $\sigma = \mathfrak{S}_{\sigma'}(P)$.

Proof: By induction on the principal signature synthesis algorithm.

- Case: $\Gamma \vdash_{\mathsf{P}} [\tau] \Rightarrow \mathfrak{S}([\tau])$, where $\Gamma \vdash \tau$ type. Trivial, since $\Gamma \vdash [\tau] \uparrow [\![T]\!]$ and $\Gamma \vdash_{\mathsf{P}} [\tau] : [\![T]\!]$.
- Case: $\Gamma \vdash_{\mathsf{P}} s \Rightarrow \mathfrak{S}_{\Gamma(s)}(s)$, where $\Gamma \vdash$ ok. Trivial, since $\Gamma \vdash s \uparrow \Gamma(s)$ and $\Gamma \vdash_{\mathsf{P}} s : \Gamma(s)$.
- Case: $\Gamma \vdash_{\mathsf{P}} PM \Rightarrow \sigma''[M/s]$, where $\Gamma \vdash_{\mathsf{P}} P \Rightarrow \Pi s{:}\sigma'.\sigma''$ and $\Gamma \vdash_{\mathsf{P}} M \Leftarrow \sigma'$.
  1. By IH, $\Gamma \vdash P \uparrow \Pi s{:}\sigma'.\sigma$, where $\Gamma \vdash_{\mathsf{P}} P : \Pi s{:}\sigma'.\sigma$ and $\sigma'' = \mathfrak{S}_{\sigma}(Ps)$.
  2. Thus, $\Gamma \vdash PM \uparrow \sigma[M/s]$ and $\Gamma \vdash_{\mathsf{P}} PM : \sigma[M/s]$.
  3. By Proposition B.26, $\sigma''[M/s] = \mathfrak{S}_{\sigma}(Ps)[M/s] = \mathfrak{S}_{\sigma[M/s]}(PM)$.
- Case: $\Gamma \vdash_{\mathsf{P}} \pi_1 P \Rightarrow \sigma'$, where $\Gamma \vdash_{\mathsf{P}} P \Rightarrow \Sigma s{:}\sigma'.\sigma''$.
  1. By IH, $\Gamma \vdash P \uparrow \Sigma s{:}\sigma_1.\sigma_2$, where $\Gamma \vdash_{\mathsf{P}} P : \Sigma s{:}\sigma_1.\sigma_2$ and $\sigma' = \mathfrak{S}_{\sigma_1}(\pi_1 P)$.
  2. Thus, $\Gamma \vdash \pi_1 P \uparrow \sigma_1$ and $\Gamma \vdash_{\mathsf{P}} \pi_1 P : \sigma_1$.
- Case: $\Gamma \vdash_{\mathsf{P}} \pi_2 P \Rightarrow \sigma''$, where $\Gamma \vdash_{\mathsf{P}} P \Rightarrow \sigma' \times \sigma''$.
  1. By IH, $\Gamma \vdash P \uparrow \Sigma s{:}\sigma_1.\sigma_2$, where $\Gamma \vdash_{\mathsf{P}} P : \Sigma s{:}\sigma_1.\sigma_2$, $\sigma' = \mathfrak{S}_{\sigma_1}(\pi_1 P)$, and $\sigma'' = \mathfrak{S}_{\sigma_2[\pi_1 P/s]}(\pi_2 P)$.
  2. Thus, $\Gamma \vdash \pi_2 P \uparrow \sigma_2[\pi_1 P/s]$ and $\Gamma \vdash_{\mathsf{P}} \pi_2 P : \sigma_2[\pi_1 P/s]$.

■

Lemma D.2 (Properties of Natural Signature Extraction)

1. *If $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$, then $\Gamma_1 \vdash P_1 \uparrow \sigma_1$ and $\Gamma_2 \vdash P_2 \uparrow \sigma_2$.*
2. *Natural signature extraction is deterministic, i.e. if $\Gamma \vdash P \uparrow \sigma_1$ and $\Gamma \vdash P \uparrow \sigma_2$, then $\sigma_1 = \sigma_2$.*

$\boxed{\textbf{Natural signature extraction: } \Gamma \vdash P \uparrow \sigma}$

$\Gamma \vdash [\tau] \uparrow [\![T]\!]$	
$\Gamma \vdash s \uparrow \Gamma(s)$	
$\Gamma \vdash PM \uparrow \sigma''[M/s]$	if $\Gamma \vdash P \uparrow \Pi s{:}\sigma'.\sigma''$
$\Gamma \vdash \pi_1 P \uparrow \sigma'$	if $\Gamma \vdash P \uparrow \Sigma s{:}\sigma'.\sigma''$
$\Gamma \vdash \pi_2 P \uparrow \sigma''[\pi_1 P/s]$	if $\Gamma \vdash P \uparrow \Sigma s{:}\sigma'.\sigma''$

$\boxed{\textbf{Weak head reduction: } \Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2}$

$\Gamma \vdash (\lambda s{:}\sigma'.M)M' \xrightarrow{\text{wh}} M[M'/s]$	
$\Gamma \vdash \pi_1\langle s = M', M''\rangle \xrightarrow{\text{wh}} M'$	
$\Gamma \vdash \pi_2\langle s = M', M''\rangle \xrightarrow{\text{wh}} M''[M'/s]$	
$\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \xrightarrow{\text{wh}} M''[M'/s]$	
$\Gamma \vdash [\mathsf{Typ}\, M] \xrightarrow{\text{wh}} M$	
$\Gamma \vdash P \xrightarrow{\text{wh}} M$	if $\Gamma \vdash P \uparrow \mathfrak{S}(M)$
$\Gamma \vdash F_1 M \xrightarrow{\text{wh}} F_2 M$	if $\Gamma \vdash F_1 \xrightarrow{\text{wh}} F_2$
$\Gamma \vdash \pi_1 M_1 \xrightarrow{\text{wh}} \pi_1 M_2$	if $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$
$\Gamma \vdash \pi_2 M_1 \xrightarrow{\text{wh}} \pi_2 M_2$	if $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$

$\boxed{\textbf{Weak head normalization: } \Gamma \vdash M \overset{\text{wh}}{\Longrightarrow} N}$

$\Gamma \vdash M \overset{\text{wh}}{\Longrightarrow} N$	if $\Gamma \vdash M \xrightarrow{\text{wh}} M'$ and $\Gamma \vdash M' \overset{\text{wh}}{\Longrightarrow} N$
$\Gamma \vdash M \overset{\text{wh}}{\Longrightarrow} M$	otherwise

Figure 15: Auxiliary Judgments for Equivalence Algorithm

Algorithmic type equivalence: $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$

$\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$	if $\Gamma_1 \vdash [\tau_1] : [\![T]\!] \Leftrightarrow \Gamma_2 \vdash [\tau_2] : [\![T]\!]$

Algorithmic signature equivalence: $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$

$\Gamma_1 \vdash 1 \Leftrightarrow \Gamma_2 \vdash 1$	
$\Gamma_1 \vdash [\![T]\!] \Leftrightarrow \Gamma_2 \vdash [\![T]\!]$	
$\Gamma_1 \vdash [\![\tau_1]\!] \Leftrightarrow \Gamma_2 \vdash [\![\tau_2]\!]$	if $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$
$\Gamma_1 \vdash \mathfrak{S}(M_1) \Leftrightarrow \Gamma_2 \vdash \mathfrak{S}(M_2)$	if $\Gamma_1 \vdash M_1 : [\![T]\!] \Leftrightarrow \Gamma_2 \vdash M_2 : [\![T]\!]$
$\Gamma_1 \vdash \Pi^\delta s{:}\sigma_1'.\sigma_1'' \Leftrightarrow \Gamma_2 \vdash \Pi^\delta s{:}\sigma_2'.\sigma_2''$	if $\Gamma_1 \vdash \sigma_1' \Leftrightarrow \Gamma_2 \vdash \sigma_2'$ and $\Gamma_1, s{:}\sigma_1' \vdash \sigma_1'' \Leftrightarrow \Gamma_2, s{:}\sigma_2' \vdash \sigma_2''$
$\Gamma_1 \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \Leftrightarrow \Gamma_2 \vdash \Sigma s{:}\sigma_2'.\sigma_2''$	if $\Gamma_1 \vdash \sigma_1' \Leftrightarrow \Gamma_2 \vdash \sigma_2'$ and $\Gamma_1, s{:}\sigma_1' \vdash \sigma_1'' \Leftrightarrow \Gamma_2, s{:}\sigma_2' \vdash \sigma_2''$

Algorithmic module equivalence: $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$

$\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$	if σ_1 and σ_2 are each unitary or singleton
$\Gamma_1 \vdash M_1 : [\![T]\!] \Leftrightarrow \Gamma_2 \vdash M_2 : [\![T]\!]$	if $\Gamma_1 \vdash M_1 \stackrel{\text{wh}}{\Longrightarrow} P_1$, $\Gamma_2 \vdash M_2 \stackrel{\text{wh}}{\Longrightarrow} P_2$, and $\Gamma_1 \vdash P_1 \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash P_2 \uparrow [\![T]\!]$
$\Gamma_1 \vdash M_1 : \Pi s{:}\sigma_1'.\sigma_1'' \Leftrightarrow \Gamma_2 \vdash M_2 : \Pi s{:}\sigma_2'.\sigma_2''$	if $\Gamma_1, s{:}\sigma_1' \vdash M_1 s : \sigma_1'' \Leftrightarrow \Gamma_2, s{:}\sigma_2' \vdash M_2 s : \sigma_2''$
$\Gamma_1 \vdash M_1 : \Sigma s{:}\sigma_1'.\sigma_1'' \Leftrightarrow \Gamma_2 \vdash M_2 : \Sigma s{:}\sigma_2'.\sigma_2''$	if $\Gamma_1 \vdash \pi_1 M_1 : \sigma_1' \Leftrightarrow \Gamma_2 \vdash \pi_1 M_2 : \sigma_2'$ and $\Gamma_1 \vdash \pi_2 M_1 : \sigma_1''[\pi_1 M_1/s] \Leftrightarrow \Gamma_2 \vdash \pi_2 M_2 : \sigma_2''[\pi_1 M_2/s]$

Algorithmic path equivalence: $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$

$\Gamma_1 \vdash s \uparrow \Gamma_1(s) \leftrightarrow \Gamma_2 \vdash s \uparrow \Gamma_2(s)$	
$\Gamma_1 \vdash P_1 M_1 \uparrow \sigma_1''[M_1/s] \leftrightarrow \Gamma_2 \vdash P_2 M_2 \uparrow \sigma_2''[M_2/s]$	if $\Gamma_1 \vdash P_1 \uparrow \Pi s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Pi s{:}\sigma_2'.\sigma_2''$ and $\Gamma_1 \vdash M_1 : \sigma_1' \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2'$
$\Gamma_1 \vdash \pi_1 P_1 \uparrow \sigma_1' \leftrightarrow \Gamma_2 \vdash \pi_1 P_2 \uparrow \sigma_2'$	if $\Gamma_1 \vdash P_1 \uparrow \Sigma s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Sigma s{:}\sigma_2'.\sigma_2''$
$\Gamma_1 \vdash \pi_2 P_1 \uparrow \sigma_1''[\pi_1 P_1/s] \leftrightarrow \Gamma_2 \vdash \pi_2 P_2 \uparrow \sigma_2''[\pi_1 P_2/s]$	if $\Gamma_1 \vdash P_1 \uparrow \Sigma s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Sigma s{:}\sigma_2'.\sigma_2''$
$\Gamma_1 \vdash [\Pi s{:}\sigma_1.\tau_1] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\Pi s{:}\sigma_2.\tau_2] \uparrow [\![T]\!]$	if $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$ and $\Gamma_1, s{:}\sigma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2, s{:}\sigma_2 \vdash \tau_2$
$\Gamma_1 \vdash [\tau_1' \times \tau_1''] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\tau_2' \times \tau_2''] \uparrow [\![T]\!]$	if $\Gamma_1 \vdash \tau_1' \Leftrightarrow \Gamma_2 \vdash \tau_2'$ and $\Gamma_1 \vdash \tau_1'' \Leftrightarrow \Gamma_2 \vdash \tau_2''$
$\Gamma_1 \vdash [\langle\!\vert \sigma_1 \vert\!\rangle] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\langle\!\vert \sigma_2 \vert\!\rangle] \uparrow [\![T]\!]$	if $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$

Figure 16: Equivalence Algorithm for Modules and Signatures

Proof: By straightforward induction on path equivalence and natural signature extraction. ■

Lemma D.3 (Determinacy of Weak Head Reduction and Normalization)

1. *Weak head reduction is deterministic, i.e. if* $\Gamma \vdash M \xrightarrow{\text{wh}} M_1$ *and* $\Gamma \vdash M \xrightarrow{\text{wh}} M_2$*, then* $M_1 = M_2$*.*

2. *Weak head normalization is deterministic, i.e. if* $\Gamma \vdash M \xRightarrow{\text{wh}} N_1$ *and* $\Gamma \vdash M \xRightarrow{\text{wh}} N_2$*, then* $N_1 = N_2$*.*

Proof:

1. By induction on weak head reduction.

2. By Part 1.

■

Lemma D.4 (Weak Head Reduction Implies Equivalence At Principal Signature)
If $\Gamma \vdash_{\mathsf{P}} M_1 \Rightarrow \sigma$ *and* $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$*, then* $\Gamma \vdash M_1 \cong M_2 : \sigma$*.*

Proof: By induction on the derivation of the second premise.

- Case: $\Gamma \vdash (\lambda s{:}\sigma'.M)M' \xrightarrow{\text{wh}} M[M'/s]$.
  1. By inversion on synthesis, $\Gamma \vdash_{\mathsf{P}} (\lambda s{:}\sigma'.M)M' \Rightarrow \sigma''[M'/s]$,
  2. where $\Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M \Rightarrow \sigma''$ and $\Gamma \vdash_{\mathsf{P}} M' \Leftarrow \sigma'$.
  3. By Theorem C.1 and Proposition B.27, $\Gamma \vdash (\lambda s{:}\sigma'.M)M' \cong M[M'/s] : \sigma''[M'/s]$.

- Case: $\Gamma \vdash \pi_1\langle s = M', M''\rangle \xrightarrow{\text{wh}} M'$.
  1. By inversion on synthesis, $\Gamma \vdash_{\mathsf{P}} \pi_1\langle s = M', M''\rangle \Rightarrow \sigma'$,
  2. where $\Gamma \vdash_{\mathsf{P}} M' \Rightarrow \sigma'$ and $\Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' \Rightarrow \sigma''$.
  3. By Theorem C.1 and Proposition B.27, $\Gamma \vdash \pi_1\langle s = M', M''\rangle \cong M' : \sigma'$.

- Case: $\Gamma \vdash \pi_2\langle s = M', M''\rangle \xrightarrow{\text{wh}} M''[M'/s]$.
  1. By inversion on synthesis, $\Gamma \vdash_{\mathsf{P}} \pi_2\langle s = M', M''\rangle \Rightarrow \sigma''[M'/s]$,
  2. where $\Gamma \vdash_{\mathsf{P}} M' \Rightarrow \sigma'$ and $\Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' \Rightarrow \sigma''$.
  3. By Theorem C.1 and Proposition B.27, $\Gamma \vdash \pi_2\langle s = M', M''\rangle \cong M''[M'/s] : \sigma''[M'/s]$.

- Case: $\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \xrightarrow{\text{wh}} M''[M'/s]$.
  1. By inversion on synthesis, $\Gamma \vdash_{\mathsf{P}} \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \Rightarrow \mathfrak{S}_\sigma(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma))$,
  2. where $\Gamma \vdash_{\mathsf{P}} M' \Rightarrow \sigma'$, $\Gamma, s{:}\sigma' \vdash_{\mathsf{P}} M'' \Leftarrow \sigma$, and $\Gamma \vdash \sigma$ sig.
  3. By Theorem C.1 and Rule 71, $\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \cong M''[M'/s] : \sigma$.
  4. By Proposition B.26, $\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \cong M''[M'/s] : \mathfrak{S}_\sigma(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma))$.

- Case: $\Gamma \vdash [\mathsf{Typ}\, M] \xrightarrow{\text{wh}} M$.
  1. By inversion on synthesis, $\Gamma \vdash_{\mathsf{P}} [\mathsf{Typ}\, M] \Rightarrow \mathfrak{S}([\mathsf{Typ}\, M])$, where $\Gamma \vdash_{\mathsf{P}} M : [\![T]\!]$.
  2. By Rule 62, $\Gamma \vdash [\mathsf{Typ}\, M] \cong M : [\![T]\!]$.
  3. By Proposition B.26, $\Gamma \vdash [\mathsf{Typ}\, M] \cong M : \mathfrak{S}([\mathsf{Typ}\, M])$.

- Case: $\Gamma \vdash P \xrightarrow{\text{wh}} M$, where $\Gamma \vdash P \uparrow \mathfrak{S}(M)$.

1. By Lemma D.1, $\Gamma \vdash_{\mathrm{P}} P : \mathfrak{S}(M)$ and $\Gamma \vdash_{\mathrm{P}} P \Rightarrow \mathfrak{S}(P)$.
2. Thus, $\Gamma \vdash P \cong M : \mathfrak{S}(P)$.

- Case: $\Gamma \vdash F_1 M \xrightarrow{\text{wh}} F_2 M$, where $\Gamma \vdash F_1 \xrightarrow{\text{wh}} F_2$.

  1. By inversion on synthesis, $\Gamma \vdash_{\mathrm{P}} F_1 M \Rightarrow \sigma''[M/s]$,
  2. where $\Gamma \vdash_{\mathrm{P}} F_1 \Rightarrow \Pi s{:}\sigma'.\sigma''$ and $\Gamma \vdash_{\mathrm{P}} M \Leftarrow \sigma'$.
  3. By IH, $\Gamma \vdash F_1 \cong F_2 : \Pi s{:}\sigma'.\sigma''$.
  4. By Theorem C.1 and reflexivity, $\Gamma \vdash M \cong M : \sigma'$.
  5. By Rule 65, $\Gamma \vdash F_1 M \cong F_2 M : \sigma''[M/s]$.

- Case: $\Gamma \vdash \pi_1 M_1 \xrightarrow{\text{wh}} \pi_1 M_2$, where $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$.

  1. By inversion on synthesis, $\Gamma \vdash_{\mathrm{P}} \pi_1 M_1 \Rightarrow \sigma'$, where $\Gamma \vdash_{\mathrm{P}} M_1 \Rightarrow \Sigma s{:}\sigma'.\sigma''$.
  2. By IH, $\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''$.
  3. By Rule 67, $\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma'$.

- Case: $\Gamma \vdash \pi_2 M_1 \xrightarrow{\text{wh}} \pi_2 M_2$, where $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$.

  1. By inversion on synthesis, $\Gamma \vdash_{\mathrm{P}} \pi_2 M_1 \Rightarrow \sigma''$, where $\Gamma \vdash_{\mathrm{P}} M_1 \Rightarrow \sigma' \times \sigma''$.
  2. By IH, $\Gamma \vdash M_1 \cong M_2 : \sigma' \times \sigma''$.
  3. By Rule 68, $\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''$.

■

Corollary D.5 (A Module Is Equivalent To Its Weak Head Normal Form)

1. *If* $\Gamma \vdash_{\mathrm{P}} M_1 : \sigma$ *and* $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$, *then* $\Gamma \vdash M_1 \cong M_2 : \sigma$.

2. *If* $\Gamma \vdash_{\mathrm{P}} M : \sigma$ *and* $\Gamma \vdash M \stackrel{\text{wh}}{\Longrightarrow} N$, *then* $\Gamma \vdash M \cong N : \sigma$.

Proof:

1. By Theorem C.5, $\Gamma \vdash_{\mathrm{P}} M_1 \Rightarrow \sigma'$, where $\Gamma \vdash \sigma' \leq \sigma$. By Lemma D.4, $\Gamma \vdash M_1 \cong M_2 : \sigma'$, so by Rule 73, $\Gamma \vdash M_1 \cong M_2 : \sigma$.

2. By Part 1, reflexivity and transitivity.

■

Theorem D.6 (Soundness of Equivalence Algorithm)

1. *If* $\Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash \tau_1$ type, $\Gamma_2 \vdash \tau_2$ type, *and* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$, *then* $\Gamma_1 \vdash \tau_1 \equiv \tau_2$.

2. *If* $\Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash \sigma_1$ sig, $\Gamma_2 \vdash \sigma_2$ sig, *and* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$, *then* $\Gamma_1 \vdash \sigma_1 \equiv \sigma_2$.

3. *If* $\Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash \sigma_1 \equiv \sigma_2$, $\Gamma_1 \vdash_{\mathrm{P}} M_1 : \sigma_1$, $\Gamma_2 \vdash_{\mathrm{P}} M_2 : \sigma_2$, *and* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$, *then* $\Gamma_1 \vdash M_1 \cong M_2 : \sigma_1$.

4. *If* $\Gamma_1 \equiv \Gamma_2$, $\Gamma_1 \vdash_{\mathrm{P}} P_1 : \rho_1$, $\Gamma_2 \vdash_{\mathrm{P}} P_2 : \rho_2$, *and* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$, *then* $\Gamma_1 \vdash \sigma_1 \equiv \sigma_2$ *and* $\Gamma_1 \vdash P_1 \cong P_2 : \sigma_1$.

Proof: By induction on the algorithmic judgments.

1. (a) By Rule 41, $\Gamma_1 \vdash_{\mathrm{P}} [\tau_1] : [\![T]\!]$ and $\Gamma_2 \vdash_{\mathrm{P}} [\tau_2] : [\![T]\!]$.
   (b) By IH, $\Gamma_1 \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]$.

(c) By Rule 7, $\Gamma_1 \vdash \tau_1 \equiv \tau_2$.

2. All cases are straightforward by inversion on the signature formation rules and induction.

3.
   - Case: $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$, where σ_1 and σ_2 are unitary.
     Since $\Gamma_1 \vdash_{\mathtt{P}} M_1 : \sigma_1$ and $\Gamma_1 \vdash_{\mathtt{P}} M_2 : \sigma_1$, by Rule 63, $\Gamma_1 \vdash M_1 \cong M_2 : \sigma_1$.
   - The remaining cases are proved exactly as in Part 1 of SH Theorem 3.7, with the exception that the last step of the Σ case involves an application of Part 9 of Proposition B.27.

4.
   - Case: $\Gamma_1 \vdash s \uparrow \Gamma_1(s) \leftrightarrow \Gamma_2 \vdash s \uparrow \Gamma_2(s)$. Trivial.
   - Case: $\Gamma_1 \vdash P_1 M_1 \uparrow \sigma_1''[M_1/s] \leftrightarrow \Gamma_2 \vdash P_2 M_2 \uparrow \sigma_2''[M_2/s]$,
     where $\Gamma_1 \vdash P_1 \uparrow \Pi s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Pi s{:}\sigma_2'.\sigma_2''$ and $\Gamma_1 \vdash M_1 : \sigma_1' \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2'$.
     (a) For $i \in \{1,2\}$, by Theorem C.5, $\Gamma_i \vdash_{\mathtt{P}} P_i M_i \Rightarrow \rho_i''[M_i/s]$,
     (b) where $\Gamma_i \vdash_{\mathtt{P}} P_i \Rightarrow \Pi s{:}\rho_i'.\rho_i''$ and $\Gamma_i \vdash_{\mathtt{P}} M_i \Leftarrow \rho_i'$.
     (c) By Lemma D.1 and Lemma D.2, $\Gamma_i \vdash P_i \uparrow \Pi s{:}\sigma_i'.\sigma_i''$, $\Gamma_i \vdash_{\mathtt{P}} P_i : \Pi s{:}\sigma_i'.\sigma_i''$, and $\rho_i' = \sigma_i'$.
     (d) By IH, $\Gamma_1 \vdash \Pi s{:}\sigma_1'.\sigma_1'' \equiv \Pi s{:}\sigma_2'.\sigma_2''$ and $\Gamma_1 \vdash P_1 \cong P_2 : \Pi s{:}\sigma_1'.\sigma_1''$.
     (e) By inversion, $\Gamma_1 \vdash \sigma_1' \equiv \sigma_2'$ and $\Gamma_1, s{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''$.
     (f) By Theorem C.1, $\Gamma_i \vdash_{\mathtt{P}} M_i : \sigma_i'$, so by IH, $\Gamma_1 \vdash M_1 \cong M_2 : \sigma_1'$.
     (g) By Functionality, $\Gamma_1 \vdash \sigma_1''[M_1/s] \equiv \sigma_2''[M_2/s]$.
     (h) By Rule 65, $\Gamma_1 \vdash P_1 M_1 \cong P_2 M_2 : \sigma_1''[M_1/s]$.
   - Case: $\Gamma_1 \vdash \pi_1 P_1 \uparrow \sigma_1' \leftrightarrow \Gamma_2 \vdash \pi_1 P_2 \uparrow \sigma_2'$, where $\Gamma_1 \vdash P_1 \uparrow \Sigma s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Sigma s{:}\sigma_2'.\sigma_2''$.
     (a) For $i \in \{1,2\}$, by Theorem C.5, $\Gamma_i \vdash_{\mathtt{P}} \pi_1 P_i \Rightarrow \rho_i'$, where $\Gamma_i \vdash_{\mathtt{P}} P_i \Rightarrow \Sigma s{:}\rho_i'.\rho_i''$.
     (b) By Theorem C.1, $\Gamma_i \vdash_{\mathtt{P}} P_i : \Sigma s{:}\rho_i'.\rho_i''$.
     (c) By IH, $\Gamma_1 \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \equiv \Sigma s{:}\sigma_2'.\sigma_2''$ and $\Gamma_1 \vdash P_1 \cong P_2 : \Sigma s{:}\sigma_1'.\sigma_1''$.
     (d) By inversion, $\Gamma_1 \vdash \sigma_1' \equiv \sigma_2'$, and by Rule 67, $\Gamma_1 \vdash \pi_1 P_1 \cong \pi_1 P_2 : \sigma_1'$.
   - Case: $\Gamma_1 \vdash \pi_2 P_1 \uparrow \sigma_1''[\pi_1 P_1/s] \leftrightarrow \Gamma_2 \vdash \pi_2 P_2 \uparrow \sigma_2''[\pi_1 P_2/s]$,
     where $\Gamma_1 \vdash P_1 \uparrow \Sigma s{:}\sigma_1'.\sigma_1'' \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \Sigma s{:}\sigma_2'.\sigma_2''$.
     (a) For $i \in \{1,2\}$, by Theorem C.5, $\Gamma_i \vdash_{\mathtt{P}} \pi_2 P_i \Rightarrow \rho_i''$, where $\Gamma_i \vdash_{\mathtt{P}} P_i \Rightarrow \rho_i' \times \rho_i''$.
     (b) By Theorem C.1, $\Gamma_i \vdash_{\mathtt{P}} P_i : \rho_i' \times \rho_i''$.
     (c) By IH, $\Gamma_1 \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \equiv \Sigma s{:}\sigma_2'.\sigma_2''$ and $\Gamma_1 \vdash P_1 \cong P_2 : \Sigma s{:}\sigma_1'.\sigma_1''$.
     (d) By inversion, $\Gamma_1, s{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''$, and by Rule 67, $\Gamma_1 \vdash \pi_1 P_1 \cong \pi_1 P_2 : \sigma_1'$.
     (e) By Functionality, $\Gamma_1 \vdash \sigma_1''[\pi_1 P_1/s] \equiv \sigma_2''[\pi_1 P_2/s]$.
     (f) By Rule 68, $\Gamma_1 \vdash \pi_2 P_1 \cong \pi_2 P_2 : \sigma_1''[\pi_1 P_1/s]$.
   - Case: $\Gamma_1 \vdash [\Pi s{:}\sigma_1.\tau_1] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\Pi s{:}\sigma_2.\tau_2] \uparrow [\![T]\!]$,
     where $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$ and $\Gamma_1, s{:}\sigma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2, s{:}\sigma_2 \vdash \tau_2$.
     (a) By Theorem C.5 and inversion on principal signature synthesis,
     $\Gamma_1 \vdash \Pi s{:}\sigma_1.\tau_1$ type and $\Gamma_2 \vdash \Pi s{:}\sigma_2.\tau_2$ type.
     (b) So, $\Gamma_1 \vdash \sigma_1$ sig and $\Gamma_2 \vdash \sigma_2$ sig, and by IH, $\Gamma_1 \vdash \sigma_1 \equiv \sigma_2$.
     (c) In addition, $\Gamma_1, s{:}\sigma_1 \vdash \tau_1$ type and $\Gamma_2, s{:}\sigma_2 \vdash \tau_2$ type.
     (d) So by IH, $\Gamma_1, s{:}\sigma_1 \vdash \tau_1 \equiv \tau_2$.
     (e) Thus, $\Gamma_1 \vdash \Pi s{:}\sigma_1.\tau_1 \equiv \Pi s{:}\sigma_2.\tau_2$, and $\Gamma_1 \vdash [\Pi s \sigma_1.\tau_1] \cong [\Pi s{:}\sigma_2.\tau_2] : [\![T]\!]$.
   - Case: $\Gamma_1 \vdash [\tau_1' \times \tau_1''] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\tau_2' \times \tau_2''] \uparrow [\![T]\!]$, where $\Gamma_1 \vdash \tau_1' \Leftrightarrow \Gamma_2 \vdash \tau_2'$ and $\Gamma_1 \vdash \tau_1'' \Leftrightarrow \Gamma_2 \vdash \tau_2''$.
     The proof is similar to and simpler than the previous case.
   - Case: $\Gamma_1 \vdash [\langle\!|\sigma_1|\!\rangle] \uparrow [\![T]\!] \leftrightarrow \Gamma_2 \vdash [\langle\!|\sigma_2|\!\rangle] \uparrow [\![T]\!]$, where $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$.
     The proof is similar to the previous two cases.

■

Lemma D.7 (Symmetry and Transitivity of Equivalence Algorithm)

1. *If* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$*, then* $\Gamma_2 \vdash \tau_2 \Leftrightarrow \Gamma_1 \vdash \tau_1$.
2. *If* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$*, and* $\Gamma_2 \vdash \tau_2 \Leftrightarrow \Gamma_3 \vdash \tau_3$*, then* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_3 \vdash \tau_3$.
3. *If* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$*, then* $\Gamma_2 \vdash \sigma_2 \Leftrightarrow \Gamma_1 \vdash \sigma_1$.
4. *If* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$*, and* $\Gamma_2 \vdash \sigma_2 \Leftrightarrow \Gamma_3 \vdash \sigma_3$*, then* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_3 \vdash \sigma_3$.
5. *If* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$*, then* $\Gamma_2 \vdash M_2 : \sigma_2 \Leftrightarrow \Gamma_1 \vdash M_1 : \sigma_1$.
6. *If* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$*, and* $\Gamma_2 \vdash M_2 : \sigma_2 \Leftrightarrow \Gamma_3 \vdash M_3 : \sigma_3$*,* *then* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_3 \vdash M_3 : \sigma_3$.
7. *If* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$*, then* $\Gamma_2 \vdash P_2 \uparrow \sigma_2 \leftrightarrow \Gamma_1 \vdash P_1 \uparrow \sigma_1$.
8. *If* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$*, and* $\Gamma_2 \vdash P_2 \uparrow \sigma_2 \leftrightarrow \Gamma_3 \vdash P_3 \uparrow \sigma_3$*,* *then* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_3 \vdash P_3 \uparrow \sigma_3$.

Proof: By straightforward induction on algorithmic judgments. The proof of Part 8 makes use of Lemma D.2. ■

Lemma D.8 (Weakening for Equivalence Algorithm)
Suppose $\Gamma \subseteq \Gamma'$, $\Gamma_1 \subseteq \Gamma'_1$*, and* $\Gamma_2 \subseteq \Gamma'_2$.

1. *If* $\Gamma \vdash P \uparrow \sigma$*, then* $\Gamma' \vdash P \uparrow \sigma$.
2. *If* $\Gamma \vdash M_1 \xrightarrow{\text{wh}} M_2$*, then* $\Gamma' \vdash M_1 \xrightarrow{\text{wh}} M_2$.
3. *If* $\Gamma \vdash M \overset{\text{wh}}{\Longrightarrow} N$*, then* $\Gamma' \vdash M \overset{\text{wh}}{\Longrightarrow} N$.
4. *If* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$*, then* $\Gamma'_1 \vdash \tau_1 \Leftrightarrow \Gamma'_2 \vdash \tau_2$.
5. *If* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$*, then* $\Gamma'_1 \vdash \sigma_1 \Leftrightarrow \Gamma'_2 \vdash \sigma_2$
6. *If* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$*, then* $\Gamma'_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma'_2 \vdash M_2 : \sigma_2$.
7. *If* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$*, then* $\Gamma'_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma'_2 \vdash P_2 \uparrow \sigma_2$.

Proof: By straightforward induction on algorithmic judgments. ■

E Completeness of the Equivalence Algorithm

The completeness proof for the equivalence algorithm, which is the main contribution of Stone and Harper, extends easily to our system. As in SH, we employ a six-place Kripke-style logical relation, given in Figure 17, that involves two worlds and two signatures. All the extensions to the logical relation are straightforward. For modules of unitary signature the logical relation is trivial and only requires that the two signatures are related. We do not bother to define a logical relation for types because the algorithm treats them as modules of base signature $[\![T]\!]$, and logical equivalence at signature $[\![T]\!]$ coincides with algorithmic equivalence.

In adapting the SH proof, we have taken the opportunity to restructure the definition of the logical relation so as to eliminate the need for the logical "validity" predicate that SH defines. As shown in Figure 17, logical validity is definable as logical equivalence of a module or signature with itself, and using symmetry and transitivity of the equivalence algorithm it is easy to show that our relation is equivalent to SH's. The advantage of our formulation is that it obviates SH Lemma 4.4 (Reflexivity), which states precisely that our definition of logical validity is admissible.

- $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$ iff

  1. $(\Delta_1;\sigma_1)\approx(\Delta_2;\sigma_2)$, $(\Delta_1;\sigma_1)\approx(\Delta_1;\sigma_1)$, and $(\Delta_2;\sigma_2)\approx(\Delta_2;\sigma_2)$

- $(\Delta_1;\sigma_1)\approx(\Delta_2;\sigma_2)$ iff

  1. $\sigma_1=\sigma_2=1$
  2. Or, $\sigma_1=\sigma_2=[\![T]\!]$
  3. Or, $\sigma_1=[\![\tau_1]\!]$ and $\sigma_2=[\![\tau_2]\!]$ and $\Delta_1\vdash\tau_1\Leftrightarrow\Delta_2\vdash\tau_2$
  4. Or, $\sigma_1=\mathfrak{S}(M_1)$ and $\sigma_2=\mathfrak{S}(M_2)$ and $\Delta_1\vdash M_1:[\![T]\!]\Leftrightarrow\Delta_2\vdash M_2:[\![T]\!]$
  5. Or, $\sigma_1=\Pi^\delta s{:}\sigma_1'.\sigma_1''$ and $\sigma_2=\Pi^\delta s{:}\sigma_2'.\sigma_2''$ and $(\Delta_1;\sigma_1')$ **is** $(\Delta_2;\sigma_2')$ and $\forall\Delta_1'\supseteq\Delta_1,\Delta_2'\supseteq\Delta_2$ if $(\Delta_1';M_1;\sigma_1')$ **is** $(\Delta_2';M_2;\sigma_2')$ then $(\Delta_1';\sigma_1''[M_1/s])$ **is** $(\Delta_2';\sigma_2''[M_2/s])$
  6. Or, $\sigma_1=\Sigma s{:}\sigma_1'.\sigma_1''$ and $\sigma_2=\Sigma s{:}\sigma_2'.\sigma_2''$ and $(\Delta_1;\sigma_1')$ **is** $(\Delta_2;\sigma_2')$ and $\forall\Delta_1'\supseteq\Delta_1,\Delta_2'\supseteq\Delta_2$ if $(\Delta_1';M_1;\sigma_1')$ **is** $(\Delta_2';M_2;\sigma_2')$ then $(\Delta_1';\sigma_1''[M_1/s])$ **is** $(\Delta_2';\sigma_2''[M_2/s])$

- $(\Delta_1;\sigma_1\leq\rho_1)$ **is** $(\Delta_2;\sigma_2\leq\rho_2)$ iff

  1. $\forall\Delta_1'\supseteq\Delta_1,\Delta_2'\supseteq\Delta_2$ if $(\Delta_1';M_1;\sigma_1)$ **is** $(\Delta_2';M_2;\sigma_2)$ then $(\Delta_1';M_1;\rho_1)$ **is** $(\Delta_2';M_2;\rho_2)$.

- $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$ iff

  1. $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$
  2. And, $(\Delta_1;M_1;\sigma_1)\approx(\Delta_2;M_2;\sigma_2)$, $(\Delta_1;M_1;\sigma_1)\approx(\Delta_1;M_1;\sigma_1)$, and $(\Delta_2;M_2;\sigma_2)\approx(\Delta_2;M_2;\sigma_2)$

- $(\Delta_1;M_1;\sigma_1)\approx(\Delta_2;M_2;\sigma_2)$ iff

  1. σ_1 and σ_2 are unitary
  2. Or, $\sigma_1=\sigma_2=T$ and $\Delta_1\vdash M_1:[\![T]\!]\Leftrightarrow\Delta_2\vdash M_2:[\![T]\!]$
  3. Or, $\sigma_1=\mathfrak{S}(N_1)$ and $\sigma_2=\mathfrak{S}(N_2)$
     and $\Delta_1\vdash M_1:[\![T]\!]\Leftrightarrow\Delta_1\vdash N_1:[\![T]\!]$ and $\Delta_2\vdash M_2:[\![T]\!]\Leftrightarrow\Delta_2\vdash N_2:[\![T]\!]$
  4. Or, $\sigma_1=\Pi s{:}\sigma_1'.\sigma_1''$ and $\sigma_2=\Pi s{:}\sigma_2'.\sigma_2''$ and $\forall\Delta_1'\supseteq\Delta_1,\Delta_2'\supseteq\Delta_2$ if $(\Delta_1';N_1;\sigma_1')$ **is** $(\Delta_2';N_2;\sigma_2')$ then $(\Delta_1';M_1N_1;\sigma_1''[N_1/s])$ **is** $(\Delta_2';M_2N_2;\sigma_2''[N_2/s])$
  5. Or, $\sigma_1=\Sigma s{:}\sigma_1'.\sigma_1''$ and $\sigma_2=\Sigma s{:}\sigma_2'.\sigma_2''$ and $(\Delta_1;\pi_1M_1;\sigma_1')$ **is** $(\Delta_2;\pi_1M_2;\sigma_2')$ and $(\Delta_1;\pi_2M_1;\sigma_1''[\pi_1M_1/s])$ **is** $(\Delta_2;\pi_2M_2;\sigma_2''[\pi_1M_2/s])$

- $(\Delta_1;\gamma_1;\Gamma_1)$ **is** $(\Delta_2;\gamma_2;\Gamma_2)$ iff

  1. $\forall s\in\mathrm{dom}(\Gamma_1)=\mathrm{dom}(\Gamma_2).\ (\Delta_1;\gamma_1s;\gamma_1(\Gamma_1(s)))$ **is** $(\Delta_2;\gamma_2s;\gamma_2(\Gamma_2(s)))$

- $(\Delta;\sigma)$ **valid** iff $(\Delta;\sigma)$ **is** $(\Delta;\sigma)$
- $(\Delta;M;\sigma)$ **valid** iff $(\Delta;M;\sigma)$ **is** $(\Delta;M;\sigma)$
- $(\Delta;\gamma;\Gamma)$ **valid** iff $(\Delta;\gamma;\Gamma)$ **is** $(\Delta;\gamma;\Gamma)$

Figure 17: Logical Relations for Proving Completeness of Equivalence Algorithm

Lemma E.1 (Monotonicity)
Suppose $\Delta_1' \supseteq \Delta_1$ *and* $\Delta_2' \supseteq \Delta_2$.

1. *If* $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$, *then* $(\Delta_1';\sigma_1)$ **is** $(\Delta_2';\sigma_2)$.
2. *If* $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$, *then* $(\Delta_1';M_1;\sigma_1)$ **is** $(\Delta_2';M_2;\sigma_2)$.
3. *If* $(\Delta_1;\gamma_1;\Gamma_1)$ **is** $(\Delta_2;\gamma_2;\Gamma_2)$, *then* $(\Delta_1';\gamma_1;\Gamma_1)$ **is** $(\Delta_2';\gamma_2;\Gamma_2)$.

Proof: By induction on the size of the signatures involved. ■

Lemma E.2 (Logical Equivalence Implies Logical Subsumption)
If $(\Delta_1;\sigma_1)$ **is** $(\Delta_1;\rho_1)$, $(\Delta_2;\sigma_2)$ **is** $(\Delta_2;\rho_2)$, *and* $(\Delta_1;\rho_1)$ **is** $(\Delta_2;\rho_2)$,
then $(\Delta_1;\sigma_1 \leq \rho_1)$ **is** $(\Delta_2;\sigma_2 \leq \rho_2)$.

Proof: See proof of SH Lemma 4.3. All the new cases involve unitary signatures and are trivial. ■

Lemma E.3 (Symmetry of Logical Relations)

1. *If* $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$, *then* $(\Delta_2;\sigma_2)$ **is** $(\Delta_1;\sigma_1)$.
2. *If* $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$, *then* $(\Delta_2;M_2;\sigma_2)$ **is** $(\Delta_1;M_1;\sigma_1)$.
3. *If* $(\Delta_1;\gamma_1;\Gamma_1)$ **is** $(\Delta_2;\gamma_2;\Gamma_2)$, *then* $(\Delta_2;\gamma_2;\Gamma_2)$ **is** $(\Delta_1;\gamma_1;\Gamma_1)$.

Proof: See proof of SH Lemma 4.5. All the new cases involve unitary signatures and are trivial, with the exception that the proof of Part 1 in the case that σ_1 and σ_2 are Π^{gen} is the same as for the Π and Σ cases. ■

Lemma E.4 (Transitivity of Logical Relations)

1. *If* $(\Delta_1;\sigma_1)$ **is** $(\Delta_1;\rho)$ *and* $(\Delta_1;\rho)$ **is** $(\Delta_2;\sigma_2)$, *then* $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$.
2. *If* $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_1;N;\rho)$ *and* $(\Delta_1;N;\rho)$ **is** $(\Delta_2;M_2;\sigma_2)$, *then* $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$.

Proof: See proof of SH Lemma 4.6. All the new cases involve unitary signatures and are trivial, with the exception that the proof of Part 1 in the case that σ_1 and σ_2 are Π^{gen} is the same as for the Π and Σ cases. ■

Define $\xrightarrow{\text{wh}}_*$ to be the reflexive, transitive closure of $\xrightarrow{\text{wh}}$.

Lemma E.5 (Closure of Logical Relations Under Weak Head Expansion)
If $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$, $\Delta_1 \vdash M_1' \xrightarrow{\text{wh}}_* M_1$, *and* $\Delta_2 \vdash M_2' \xrightarrow{\text{wh}}_* M_2$,
then $(\Delta_1;M_1';\sigma_1)$ **is** $(\Delta_2;M_2';\sigma_2)$.

Proof: See proof of SH Lemma 4.7. All the new cases involve unitary signatures and are trivial. ■

Lemma E.6 (Logical Relations Imply Algorithmic Equivalence)

1. *If* $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$, *then* $\Delta_1 \vdash \sigma_1 \Leftrightarrow \Delta_2 \vdash \sigma_2$.
2. *If* $(\Delta_1;M_1;\sigma_1)$ **is** $(\Delta_2;M_2;\sigma_2)$, *then* $\Delta_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Delta_2 \vdash M_2 : \sigma_2$.
3. *If* $(\Delta_1;\sigma_1)$ **is** $(\Delta_2;\sigma_2)$ *and* $\Delta_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Delta_2 \vdash P_2 \uparrow \sigma_2$, *then* $(\Delta_1;P_1;\sigma_1)$ **is** $(\Delta_2;P_2;\sigma_2)$.

Proof: See proof of SH Lemma 4.8. All the new cases involve unitary signatures and are trivial, with the exception that the proof of Part 1 in the case that σ_1 and σ_2 are Π^{gen} is the same as for the Π and Σ cases. ■

Theorem E.7 (Fundamental Theorem of Logical Relations)
Suppose $(\Delta_1;\gamma_1;\Gamma)$ **is** $(\Delta_2;\gamma_2;\Gamma)$.

1. *If* $\Gamma \vdash \tau$ type, *then* $\Delta_1 \vdash \gamma_1\tau \Leftrightarrow \Delta_2 \vdash \gamma_2\tau$.

2. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$, *then* $\Delta_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$, $\Delta_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_1$, *and* $\Delta_1 \vdash \gamma_1\tau_2 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$.

3. *If* $\Gamma \vdash \sigma$ sig, *then* $(\Delta_1; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2\sigma)$.

4. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_2)$, $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_1)$, *and* $(\Delta_1; \gamma_1\sigma_2)$ **is** $(\Delta_2; \gamma_2\sigma_2)$.

5. *If* $\Gamma \vdash \sigma_1 \leq \sigma_2$, *then* $(\Delta_1; \gamma_1\sigma_1 \leq \gamma_1\sigma_2)$ **is** $(\Delta_2; \gamma_2\sigma_1 \leq \gamma_2\sigma_2)$, $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_1)$, *and* $(\Delta_1; \gamma_1\sigma_2)$ **is** $(\Delta_2; \gamma_2\sigma_2)$.

6. *If* $\Gamma \vdash_{\mathsf{P}} M : \sigma$, *then* $(\Delta_1; \gamma_1 M; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M; \gamma_2\sigma)$.

7. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$, *then* $(\Delta_1; \gamma_1 M_1; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_2; \gamma_2\sigma)$, $(\Delta_1; \gamma_1 M_1; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_1; \gamma_2\sigma)$, *and* $(\Delta_1; \gamma_1 M_2; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_2; \gamma_2\sigma)$.

Proof: By induction on derivations. In all cases, $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_1; \gamma_1; \Gamma)$ and $(\Delta_2; \gamma_2; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$. For most of the cases, see proof of SH Theorem 4.9. Here we give the new cases:

Well-formed types: $\Gamma \vdash \tau$ type**.**

- Case: Rule 3.
  1. By IH, $(\Delta_1; \gamma_1 M; [\![T]\!])$ **is** $(\Delta_2; \gamma_2 M; [\![T]\!])$.
  2. By Lemma E.5, $(\Delta_1; [\mathsf{Typ}\,\gamma_1 M]; [\![T]\!])$ **is** $(\Delta_2; [\mathsf{Typ}\,\gamma_2 M]; [\![T]\!])$.
  3. So, $\Delta_1 \vdash [\mathsf{Typ}\,\gamma_1 M] : [\![T]\!] \Leftrightarrow \Delta_2 \vdash [\mathsf{Typ}\,\gamma_2 M] : [\![T]\!]$.
  4. Thus, $\Delta_1 \vdash \gamma_1(\mathsf{Typ}\,M) \Leftrightarrow \Delta_2 \vdash \gamma_2(\mathsf{Typ}\,M)$.
- Case: Rule 4. The proof is a reflexive instance of the proof for Rule 8.
- Case: Rule 5. The proof is a reflexive instance of the proof for Rule 9.
- Case: Rule 6. The proof is a reflexive instance of the proof for Rule 10.

Type equivalence: $\Gamma \vdash \tau_1 \equiv \tau_2$**.**
It suffices to prove that if $\Gamma \vdash \tau_1 \equiv \tau_2$ and $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ then $\Delta_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$, because we can apply this to get $\Delta_2 \vdash \gamma_2\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$, so $\Delta_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_1$ by Symmetry and Transitivity of the algorithm. A similar argument yields $\Delta_1 \vdash \gamma_1\tau_2 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$.

- Case: Rule 7.
  1. By IH, $(\Delta_1; [\gamma_1\tau_1]; [\![T]\!])$ **is** $(\Delta_2; [\gamma_2\tau_2]; [\![T]\!])$.
  2. So, $\Delta_1 \vdash [\gamma_1\tau_1] : [\![T]\!] \Leftrightarrow \Delta_2 \vdash [\gamma_2\tau_2] : [\![T]\!]$.
  3. Thus, $\Delta_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2$.
- Case: Rule 8.
  1. By IH, $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_2)$.
  2. By Lemma E.6, $\Delta_1 \vdash \gamma_1\sigma_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\sigma_2$.
  3. Now, $\Delta_1, s{:}\gamma_1\sigma_1 \vdash s \uparrow \gamma_1\sigma_1 \leftrightarrow \Delta_2, s{:}\gamma_2\sigma_2 \vdash s \uparrow \gamma_2\sigma_2$.
  4. So by Monotonicity and Lemma E.6, $(\Delta_1, s{:}\gamma_1\sigma_1; s; \gamma_1\sigma_1)$ **is** $(\Delta_2, s{:}\gamma_2\sigma_2; s; \gamma_2\sigma_2)$.
  5. By IH and Symmetry, $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_1)$ and $(\Delta_2; \gamma_2\sigma_2)$ **is** $(\Delta_2; \gamma_2\sigma_1)$.
  6. By Lemma E.2, $(\Delta_1; \gamma_1\sigma_1 \leq \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_2 \leq \gamma_2\sigma_1)$.
  7. So, $(\Delta_1, s{:}\gamma_1\sigma_1; s; \gamma_1\sigma_1)$ **is** $(\Delta_2, s{:}\gamma_2\sigma_2; s; \gamma_2\sigma_1)$.
  8. Thus, by Monotonicity, $(\Delta_1, s{:}\gamma_1\sigma_1; \gamma_1; \Gamma, s{:}\sigma_1)$ **is** $(\Delta_2, s{:}\gamma_2\sigma_2; \gamma_2; \Gamma, s{:}\sigma_1)$.
  9. Then, by IH, $\Delta_1, s{:}\gamma_1\sigma_1 \vdash \gamma_1\tau_1 \Leftrightarrow \Delta_2, s{:}\gamma_2\sigma_2 \vdash \gamma_2\tau_2$.

10. Therefore, $\Delta_1 \vdash [\Pi s{:}\gamma_1\sigma_1.\gamma_1\tau_1] \uparrow [\![T]\!] \leftrightarrow \Delta_2 \vdash [\Pi s{:}\gamma_2\sigma_2.\gamma_2\tau_2] \uparrow [\![T]\!]$,
11. and so $\Delta_1 \vdash [\Pi s{:}\gamma_1\sigma_1.\gamma_1\tau_1] : [\![T]\!] \Leftrightarrow \Delta_2 \vdash [\Pi s{:}\gamma_2\sigma_2.\gamma_2\tau_2] : [\![T]\!]$,
12. and finally $\Delta_1 \vdash \gamma_1(\Pi s{:}\sigma_1.\tau_1) \Leftrightarrow \Delta_2 \vdash \gamma_2(\Pi s{:}\sigma_2.\tau_2)$.

- Case: Rule 9.
  1. By IH, $\Delta_1 \vdash \gamma_1\tau_1' \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2'$ and $\Delta_1 \vdash \gamma_1\tau_1'' \Leftrightarrow \Delta_2 \vdash \gamma_2\tau_2''$.
  2. Therefore, $\Delta_1 \vdash [\gamma_1\tau_1' \times \gamma_1\tau_1''] \uparrow [\![T]\!] \leftrightarrow \Delta_2 \vdash [\gamma_2\tau_2' \times \gamma_2\tau_2''] \uparrow [\![T]\!]$,
  3. and so $\Delta_1 \vdash [\gamma_1\tau_1' \times \gamma_1\tau_1''] : [\![T]\!] \Leftrightarrow \Delta_2 \vdash [\gamma_2\tau_2' \times \gamma_2\tau_2''] : [\![T]\!]$,
  4. and finally $\Delta_1 \vdash \gamma_1(\tau_1' \times \tau_1'') \Leftrightarrow \Delta_2 \vdash \gamma_2(\tau_2' \times \tau_2'')$.

- Case: Rule 10.
  1. By IH, $\Delta_1 \vdash \gamma_1\sigma_1 \Leftrightarrow \Delta_2 \vdash \gamma_2\sigma_2$.
  2. Therefore, $\Delta_1 \vdash [\langle\!|\gamma_1\sigma_1|\!\rangle] \uparrow [\![T]\!] \leftrightarrow \Delta_2 \vdash [\langle\!|\gamma_2\sigma_2|\!\rangle] \uparrow [\![T]\!]$,
  3. and so $\Delta_1 \vdash [\langle\!|\gamma_1\sigma_1|\!\rangle] : [\![T]\!] \Leftrightarrow \Delta_2 \vdash [\langle\!|\gamma_2\sigma_2|\!\rangle] : [\![T]\!]$,
  4. and finally $\Delta_1 \vdash \gamma_1\langle\!|\sigma_1|\!\rangle \Leftrightarrow \Delta_2 \vdash \gamma_2\langle\!|\sigma_2|\!\rangle$.

Well-formed signatures: $\Gamma \vdash \sigma$ sig**.**
All the new cases involve unitary signatures and are trivial, with the exception of the Π^{gen} case, for which the proof is the same as for the Π and Σ cases.

Signature equivalence: $\Gamma \vdash \sigma_1 \equiv \sigma_2$**.**
All the new cases involve unitary signatures and are trivial, with the exception of the Π^{gen} case, for which the proof is the same as for the Π and Σ cases.

Signature subtyping: $\Gamma \vdash \sigma_1 \leq \sigma_2$**.**
In all the new cases, σ_2 is a unitary signature. By definition of the logical relation, this means that in order to show logical subsumption it suffices to show $(\Delta_1; \gamma_1\sigma_2)$ **is** $(\Delta_2; \gamma_2\sigma_2)$. Showing this is as well as $(\Delta_1; \gamma_1\sigma_1)$ **is** $(\Delta_2; \gamma_2\sigma_1)$ is trivial for the 1 and $[\tau]$ cases. For the (Π, Π^{gen}) and $(\Pi^{\text{gen}}, \Pi^{\text{gen}})$ cases, the proof is the same as the one for the (Π, Π) case.

Well-formed modules: $\Gamma \vdash_{\mathrm{P}} M : \sigma$**.**

- Case: Rule 40. Trivial.
- Case: Rule 42. By the same reasoning as for Rule 22, applied to the second (redundant) premise.
- Case: Rule 44. By the same reasoning as for Rule 24, applied to the second (redundant) premise.
- Case: Rule 47. The proof is a reflexive instance of the proof for Rule 66.
- Case: Rule 55. The proof is a reflexive instance of the proof for Rule 70.
- Case: Rule 56.
  1. By IH, $(\Delta_1; \gamma_1 M'; \gamma_1\sigma')$ **is** $(\Delta_2; \gamma_2 M'; \gamma_2\sigma')$.
  2. So, $(\Delta_1; \gamma_1[s \mapsto \gamma_1 M']; \Gamma, s{:}\sigma')$ **is** $(\Delta_2; \gamma_2[s \mapsto \gamma_2 M']; \Gamma, s{:}\sigma')$.
  3. By Proposition B.2, $s \notin FV(\sigma)$.
  4. Then by IH, $(\Delta_1; (\gamma_1 M'')[\gamma_1 M'/s]; \gamma_1\sigma)$ **is** $(\Delta_2; (\gamma_2 M'')[\gamma_2 M'/s]; \gamma_2\sigma)$.
  5. By Lemma E.5, $(\Delta_1; \gamma_1(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma)); \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma)); \gamma_2\sigma)$.

Module equivalence: $\Gamma \vdash M_1 \cong M_2 : \sigma$.
It suffices to prove that if $\Gamma \vdash M_1 \cong M_2 : \sigma$ and $(\Delta_1; \gamma_1; \Gamma)$ **is** $(\Delta_2; \gamma_2; \Gamma)$ then $(\Delta_1; \gamma_1 M_1; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_2; \gamma_2\sigma)$, because we can apply this to get $(\Delta_2; \gamma_2 M_1; \gamma_2\sigma)$ **is** $(\Delta_2; \gamma_2 M_2; \gamma_2\sigma)$, so $(\Delta_1; \gamma_1 M_1; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_1; \gamma_2\sigma)$ by Symmetry and Transitivity of the algorithm. A similar argument yields $(\Delta_1; \gamma_1 M_2; \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2 M_2; \gamma_2\sigma)$.

- Case: Rule 61. Trivial, by IH.
- Case: Rule 62.
  1. By IH, $(\Delta_1; \gamma_1 M; [\![T]\!])$ **is** $(\Delta_2; \gamma_2 M; [\![T]\!])$.
  2. By Lemma E.5, $(\Delta_1; \gamma_1[\mathsf{Typ}\, M]; [\![T]\!])$ **is** $(\Delta_2; \gamma_2 M; [\![T]\!])$.
- Case: Rule 63. Trivial, by IH.
- Case: Rule 66.
  1. By the same reasoning as in the proof for Rule 25, $(\Delta_1; \gamma_1(\Sigma s{:}\sigma'.\sigma''))$ **is** $(\Delta_2; \gamma_2(\Sigma s{:}\sigma'.\sigma''))$.
  2. By IH, $(\Delta_1; \gamma_1 M_1'; \gamma_1\sigma')$ **is** $(\Delta_2; \gamma_2 M_2'; \gamma_2\sigma')$.
  3. So, $(\Delta_1; \gamma_1[s \mapsto \gamma_1 M_1']; \Gamma, s{:}\sigma')$ **is** $(\Delta_2; \gamma_2[s \mapsto \gamma_2 M_2']; \Gamma, s{:}\sigma')$.
  4. Then by IH, $(\Delta_1; (\gamma_1 M_1'')[\gamma_1 M_1'/s]; \gamma_1\sigma''[\gamma_1 M_1'/s])$ **is** $(\Delta_2; (\gamma_2 M_2'')[\gamma_2 M_2'/s]; \gamma_2\sigma''[\gamma_2 M_2'/s])$.
  5. By Lemma E.5, $(\Delta_1; \pi_1\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle; \gamma_1\sigma')$ **is** $(\Delta_2; \pi_1\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle; \gamma_2\sigma')$
  6. and $(\Delta_1; \pi_2\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle; \gamma_1\sigma''[\gamma_1 M_1'/s])$ **is** $(\Delta_2; \pi_2\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle; \gamma_2\sigma''[\gamma_2 M_2'/s])$.
  7. Also by Lemma E.5, $(\Delta_1; \gamma_1 M_1'; \gamma_1\sigma')$ **is** $(\Delta_1; \pi_1\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle; \gamma_1\sigma')$
  8. and $(\Delta_2; \gamma_2 M_2'; \gamma_2\sigma')$ **is** $(\Delta_2; \pi_1\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle; \gamma_2\sigma')$.
  9. So, $(\Delta_1; \gamma_1\sigma''[\pi_1\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle/s])$ **is** $(\Delta_2; \gamma_2\sigma''[\pi_1\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle/s])$,
  10. $(\Delta_1; \gamma_1\sigma''[\gamma_1 M_1'/s])$ **is** $(\Delta_1; \gamma_1\sigma''[\pi_1\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle/s])$,
  11. and $(\Delta_2; \gamma_2\sigma''[\gamma_2 M_2'/s])$ **is** $(\Delta_2; \gamma_2\sigma''[\pi_1\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle/s])$.
  12. By Lemma E.2, $(\Delta_1; \pi_2\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle; \gamma_1\sigma''[\pi_1\langle s = \gamma_1 M_1', \gamma_1 M_1''\rangle/s])$ **is** $(\Delta_2; \pi_2\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle; \gamma_2\sigma''[\pi_1\langle s = \gamma_2 M_2', \gamma_2 M_2''\rangle/s])$.
  13. Therefore, $(\Delta_1; \gamma_1\langle s = M_1', M_1''\rangle; \gamma_1(\Sigma s{:}\sigma'.\sigma''))$ **is** $(\Delta_2; \gamma_2\langle s = M_2', M_2''\rangle; \gamma_2(\Sigma s{:}\sigma'.\sigma''))$.
- Case: Rule 70. Trivial, by IH.
- Case: Rule 71.
  1. By IH, $(\Delta_1; \gamma_1 M'; \gamma_1\sigma')$ **is** $(\Delta_2; \gamma_2 M'; \gamma_2\sigma')$.
  2. So, $(\Delta_1; \gamma_1[s \mapsto \gamma_1 M']; \Gamma, s{:}\sigma')$ **is** $(\Delta_2; \gamma_2[s \mapsto \gamma_2 M']; \Gamma, s{:}\sigma')$.
  3. By Proposition B.2, $s \notin FV(\sigma)$.
  4. Then by IH, $(\Delta_1; (\gamma_1 M'')[\gamma_1 M'/s]; \gamma_1\sigma)$ **is** $(\Delta_2; (\gamma_2 M'')[\gamma_2 M'/s]; \gamma_2\sigma)$.
  5. By Lemma E.5, $(\Delta_1; \gamma_1(\mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma)); \gamma_1\sigma)$ **is** $(\Delta_2; \gamma_2(M''[M'/s]); \gamma_2\sigma)$.

■

Lemma E.8 (Identity Substitution Is Related To Itself)
If $\Gamma \vdash \text{ok}$, *then* $(\Gamma; \mathbf{id}; \Gamma)$ **is** $(\Gamma; \mathbf{id}; \Gamma)$.

Proof: See proof of SH Lemma 4.10. ■

Corollary E.9 (Completeness of Equivalence Algorithm)
1. *If* $\Gamma \vdash \tau_1 \equiv \tau_2$, *then* $\Gamma \vdash \tau_1 \Leftrightarrow \Gamma \vdash \tau_2$.

2. *If* $\Gamma \vdash \sigma_1 \equiv \sigma_2$, *then* $\Gamma \vdash \sigma_1 \Leftrightarrow \Gamma \vdash \sigma_2$.

3. *If* $\Gamma \vdash M_1 \cong M_2 : \sigma$, *then* $\Gamma \vdash M_1 : \sigma \Leftrightarrow \Gamma \vdash M_2 : \sigma$.

Proof: By Lemma E.6, Theorem E.7, and Lemma E.8. ■

Lemma E.10

1. *If* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_1 \vdash \tau_1$ *and* $\Gamma_2 \vdash \tau_2 \Leftrightarrow \Gamma_2 \vdash \tau_2$, *then* $\Gamma_1 \vdash \tau_1 \Leftrightarrow \Gamma_2 \vdash \tau_2$ *is decidable.*

2. *If* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_1 \vdash \sigma_1$ *and* $\Gamma_2 \vdash \sigma_2 \Leftrightarrow \Gamma_2 \vdash \sigma_2$, *then* $\Gamma_1 \vdash \sigma_1 \Leftrightarrow \Gamma_2 \vdash \sigma_2$ *is decidable.*

3. *If* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_1 \vdash M_1 : \sigma_1$ *and* $\Gamma_2 \vdash M_2 : \sigma_2 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$, *then* $\Gamma_1 \vdash M_1 : \sigma_1 \Leftrightarrow \Gamma_2 \vdash M_2 : \sigma_2$ *is decidable.*

4. *If* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_1 \vdash P_1 \uparrow \sigma_1$ *and* $\Gamma_2 \vdash P_2 \uparrow \sigma_2 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$, *then* $\Gamma_1 \vdash P_1 \uparrow \sigma_1 \leftrightarrow \Gamma_2 \vdash P_2 \uparrow \sigma_2$ *is decidable.*

Proof: See proof of SH Lemma 4.12. ■

Corollary E.11 (Decidability of Equivalence Algorithm on Well-Formed Things)

1. *If* $\Gamma \vdash \tau_1$ type *and* $\Gamma \vdash \tau_2$ type, *then* $\Gamma \vdash \tau_1 \Leftrightarrow \Gamma \vdash \tau_2$ *is decidable.*

2. *If* $\Gamma \vdash \sigma_1$ sig *and* $\Gamma \vdash \sigma_2$ sig, *then* $\Gamma \vdash \sigma_1 \Leftrightarrow \Gamma \vdash \sigma_2$ *is decidable.*

3. *If* $\Gamma \vdash_{\mathrm{P}} M_1 : \sigma$ *and* $\Gamma \vdash_{\mathrm{P}} M_2 : \sigma$, *then* $\Gamma \vdash M_1 : \sigma \Leftrightarrow \Gamma \vdash M_2 : \sigma$ *is decidable.*

Proof: By Corollary E.9, comparison of each well-formed type, signature or module with itself is decidable. The desired result then follows from Lemma E.10. ■

F Decidability

With a decidable equivalence algorithm in hand, we may now define a term typechecking algorithm. Given a term and a type, the algorithm synthesizes the (unique) type of a term and checking that the synthesized type is equivalent to the given type. In the cases of function application $(e\,M)$ or projection from pairs $(\pi_i e)$, the synthesized type of e is not necessarily in the correct form and must be weak-head-reduced to a type of the form $\Pi s{:}\sigma.\tau$ or $\tau' \times \tau''$, respectively.

Now that we have filled in the definition of term typechecking, we proceed to prove soundness and completeness of term *and* module typechecking, reusing the proofs for module typechecking from Appendix C. Once we have shown this, decidability of the entire type system follows straightforwardly from the fact that the typechecking algorithm and subsignature checking are syntax-directed and module and type equivalence are decidable.

Theorem F.1 (Soundness of Full Typechecking/Synthesis)

1. *If* $\Gamma \vdash e \Leftarrow \tau$ *or* $\Gamma \vdash e \Rightarrow \tau$, *then* $\Gamma \vdash e : \tau$.

2. *If* $\Gamma \vdash_\kappa M \Leftarrow \sigma$ *or* $\Gamma \vdash_\kappa M \Rightarrow \sigma$, *then* $\Gamma \vdash_\kappa M : \sigma$.

Proof: By straightforward induction on the typechecking/synthesis algorithm, as before. In the term application and projection cases ($e\,M$ and $\pi_i e$), the proof requires a straightforward application of Corollary D.5 to show that $[\tau_e]$ is equivalent to its weak head normal form. ■

Proposition F.2 (Properties of Typechecking/Synthesis)

1. *Type synthesis is deterministic, i.e. if* $\Gamma \vdash e \Rightarrow \tau_1$ *and* $\Gamma \vdash e \Rightarrow \tau_2$, *then* $\tau_1 = \tau_2$.

Term typechecking: $\Gamma \vdash e \Leftarrow \tau$

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e \Leftarrow \tau}$$

Type synthesis for terms: $\Gamma \vdash e \Rightarrow \tau$

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

$$\frac{\Gamma, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash e \Leftarrow \tau}{\Gamma \vdash \mathsf{fix}\, f(s{:}\sigma){:}\tau.e \Rightarrow \Pi s{:}\sigma.\tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau_e \quad \Gamma \vdash [\tau_e] \overset{\text{wh}}{\Longrightarrow} [\Pi s{:}\sigma.\tau] \quad \Gamma \vdash_{\mathtt{P}} M \Leftarrow \sigma}{\Gamma \vdash e\, M \Rightarrow \tau[M/s]}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \quad \Gamma \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle \Rightarrow \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau_e \quad \Gamma \vdash [\tau_e] \overset{\text{wh}}{\Longrightarrow} [\tau_1 \times \tau_2]}{\Gamma \vdash \pi_i e \Rightarrow \tau_i}\ (i \in \{1,2\})$$

Figure 18: Term Typechecking and Unique Type Synthesis

2. *Signature synthesis is deterministic, i.e. if* $\Gamma \vdash_{\kappa_1} M \Rightarrow \sigma_1$ *and* $\Gamma \vdash_{\kappa_2} M \Rightarrow \sigma_2$, *then* $\sigma_1 = \sigma_2$ *and* $\kappa_1 = \kappa_2$.
3. *If* $\Gamma \vdash_{\mathtt{P}} M \Rightarrow \sigma$, *then* σ *is in determinate synthesis form.*

Proof: By straightforward induction on the typechecking/synthesis algorithm, with applications of Lemma D.3 in the term application and projection cases. ■

Lemma F.3 (Weakening for Full Typechecking/Synthesis)

1. *If* $\Gamma_1 \vdash e \Rightarrow \tau$, $\Gamma_1 \subseteq \Gamma_2$, *and* $\Gamma_2 \vdash \text{ok}$, *then* $\Gamma_2 \vdash e \Rightarrow \tau$.
2. *If* $\Gamma_1 \vdash e \Leftarrow \tau$, $\Gamma_1 \subseteq \Gamma_2$, *and* $\Gamma_2 \vdash \text{ok}$, *then* $\Gamma_2 \vdash e \Leftarrow \tau$.
3. *If* $\Gamma_1 \vdash_\kappa M \Rightarrow \sigma$, $\Gamma_1 \subseteq \Gamma_2$, *and* $\Gamma_2 \vdash \text{ok}$, *then* $\Gamma_2 \vdash_\kappa M \Rightarrow \sigma$.
4. *If* $\Gamma_1 \vdash_\kappa M \Leftarrow \sigma$, $\Gamma_1 \subseteq \Gamma_2$, *and* $\Gamma_2 \vdash \text{ok}$, *then* $\Gamma_2 \vdash_\kappa M \Leftarrow \sigma$.

Proof: By straightforward induction on the typechecking/synthesis algorithm, with applications of Lemma D.8 in the term application and projection cases. ■

Theorem F.4 (Completeness of Full Typechecking/Synthesis)

1. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \leq \Gamma'$, *then* $\Gamma' \vdash e \Leftarrow \tau$.
2. *If* $\Gamma \vdash_\kappa M : \sigma$ *and* $\Gamma \leq \Gamma'$, *then* $\Gamma' \vdash_{\kappa'} M \Leftarrow \sigma$, *where* $\kappa' \sqsubseteq \kappa$.
Moreover, if $\kappa' = \mathtt{P}$, *then* $\Gamma' \vdash_{\mathtt{P}} M \Leftarrow \mathfrak{S}_\sigma(M)$.

Proof: The proof of Part 2 is the same as for Theorem C.5. Here is the proof of Part 1:

- Case: Rules 11 and 12. Trivial, by IH.
- Case: Rule 13.
  1. By IH, $\Gamma' \vdash_{\kappa'} M \Rightarrow \sigma'$, where $\Gamma' \vdash \sigma' \leq \sigma$.
  2. Since $\Gamma, s{:}\sigma \leq \Gamma', s{:}\sigma'$, by IH, $\Gamma', s{:}\sigma' \vdash e \Leftarrow \tau$.
  3. Since $\Gamma' \vdash \tau$ type, we have $\Gamma' \vdash \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) \Rightarrow \tau$.
- Case: Rule 14. Trivial, by IH.

- Case: Rule 15.

   1. By IH, $\Gamma' \vdash e \Rightarrow \tau_e$, where $\Gamma' \vdash \tau_e \equiv \Pi s{:}\sigma.\tau$.
   2. By Corollary E.9, $\Gamma' \vdash \tau_e \Leftrightarrow \Gamma' \vdash \Pi s{:}\sigma.\tau$.
   3. By inspection of the equivalence algorithm, $\Gamma' \vdash [\tau_e] \stackrel{\text{wh}}{\Longrightarrow} [\Pi s{:}\sigma'.\tau']$,
   4. where $\Gamma' \vdash \sigma' \Leftrightarrow \Gamma' \vdash \sigma$ and $\Gamma', s{:}\sigma' \vdash \tau' \Leftrightarrow \Gamma', s{:}\sigma \vdash \tau$.
   5. By Corollary D.5, $\Gamma' \vdash_{\mathrm{P}} [\Pi s{:}\sigma'.\tau'] : [\![T]\!]$.
   6. So, $\Gamma' \vdash \sigma'$ sig, $\Gamma' \vdash \sigma$ sig, $\Gamma', s{:}\sigma' \vdash \tau'$ type and $\Gamma', s{:}\sigma \vdash \tau$ type.
   7. Thus by Theorem D.6, $\Gamma' \vdash \sigma' \equiv \sigma$.
   8. Then since $\Gamma', s{:}\sigma' \equiv \Gamma', s{:}\sigma$, by Theorem D.6, $\Gamma', s{:}\sigma' \vdash \tau' \equiv \tau$.
   9. By IH, $\Gamma' \vdash_{\mathrm{P}} M \Leftarrow \sigma$, so $\Gamma' \vdash_{\mathrm{P}} M \Leftarrow \sigma'$.
   10. Thus, $\Gamma' \vdash e\, M \Rightarrow \tau'[M/s]$, and by Substitution, $\Gamma' \vdash \tau'[M/s] \equiv \tau[M/s]$.

- Case: Rule 16. Trivial, by IH.
- Case: Rules 17 and 18. Similar to the proof for Rule 15.
- Case: Rule 19. Trivial, by IH.

■

Theorem F.5 (Decidability of Judgments with Well-Formed Contexts)
Suppose $\Gamma \vdash \mathrm{ok}$.

1. *If* $\Gamma \vdash \sigma_1$ sig *and* $\Gamma \vdash \sigma_2$ sig, *then* $\Gamma \vdash \sigma_1 \leq \sigma_2$ *is decidable.*
2. $\Gamma \vdash \tau$ type *is decidable.*
3. *It is decidable whether there exists* τ *such that* $\Gamma \vdash e \Rightarrow \tau$.
4. $\Gamma \vdash e : \tau$ *(or equivalently,* $\Gamma \vdash e \Leftarrow \tau$*) is decidable.*
5. $\Gamma \vdash \sigma$ sig *is decidable.*
6. *It is decidable whether there exist* σ *and* κ *such that* $\Gamma \vdash_\kappa M \Rightarrow \sigma$.
7. $\Gamma \vdash_\kappa M : \sigma$ *(or equivalently,* $\Gamma \vdash_{\kappa'} M \Leftarrow \sigma$ *for* $\kappa' \sqsubseteq \kappa$*) is decidable.*

Proof: Part 1 is by straightforward induction on the structure of σ_1 and σ_2 and by Corollary E.11. Parts 2, 3, 5 and 6 are by straightforward induction on the structure of τ, e, σ and M, respectively. Part 4 follows from Parts 2 and 3 and Corollary E.11. Part 7 follows from Parts 1, 5 and 6.

It is worth noting that as $\Gamma \vdash \mathrm{ok}$ is an assumption of the theorem, it must be preserved upon invocations of the IH, which means that all signatures must be checked for well-formedness before being added to the context. This well-formedness check is always either decidable by induction or else unnecessary because the signature is the result of synthesis (and thus well-formed by Theorem F.1).

Also worth noting is that in the application and projection cases of term synthesis, if $\Gamma \vdash e \Rightarrow \tau_e$, then $\Gamma \vdash \tau_e \equiv \tau_e$ and $\Gamma \vdash \tau_e \Leftrightarrow \Gamma \vdash \tau_e$ by Corollary E.9. Thus, by inspection of the algorithm, weak head normalization of $[\tau_e]$ terminates. ■

Lemma F.6 (Decidability of Context Well-formedness)
$\Gamma \vdash \mathrm{ok}$ *is decidable.*

Proof: By straightforward induction on Γ and by Theorem F.5. ■

Corollary F.7 (Decidability)
The type system is decidable.

G Elaboration Rules

Existential peeling: $M : \varsigma \overset{\text{peel}}{\Longrightarrow} M' : \varsigma'$

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

Type elaboration: $\Delta \vdash \hat\tau \rightsquigarrow \tau$

$$\frac{\Delta \vdash_{\mathrm{P}} \hat M \rightsquigarrow M : \varsigma \quad \Delta \vdash M : \varsigma \le [\![T]\!] \rightsquigarrow N}{\Delta \vdash \mathsf{Typ}\,\hat M \rightsquigarrow \mathsf{Typ}\,N}$$

$$\frac{\Delta \vdash \hat\sigma \rightsquigarrow \sigma \quad \Delta, s{:}\sigma \vdash \hat\tau \rightsquigarrow \tau}{\Delta \vdash \Pi s{:}\hat\sigma.\hat\tau \rightsquigarrow \Pi s{:}\sigma.\tau} \qquad \frac{\Delta \vdash \hat\tau_1 \rightsquigarrow \tau_1 \quad \Delta \vdash \hat\tau_2 \rightsquigarrow \tau_2}{\Delta \vdash \hat\tau_1 \times \hat\tau_2 \rightsquigarrow \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash \hat\sigma \rightsquigarrow \sigma}{\Delta \vdash \langle\!|\hat\sigma|\!\rangle \rightsquigarrow \langle\!|\sigma|\!\rangle}$$

Term elaboration: $\Delta \vdash \hat e \rightsquigarrow e : \tau$

$$\frac{\Delta \vdash_{\mathrm{P}} \hat M \rightsquigarrow M : \varsigma \quad M : \varsigma \overset{\text{peel}}{\Longrightarrow} N : [\![\tau]\!]}{\Delta \vdash \mathsf{Val}\,\hat M \rightsquigarrow \mathsf{Val}\,N : \tau}$$

$$\frac{\Delta \vdash_\kappa \hat M \rightsquigarrow M : \varsigma \quad \Delta \vdash \hat\tau \rightsquigarrow \tau \quad \Delta, s{:}\varsigma \vdash \hat e \rightsquigarrow e : \tau' \quad \overline{\Delta, s{:}\varsigma} \vdash \tau' \equiv \tau}{\Delta \vdash \mathsf{let}\ s = \hat M\ \mathsf{in}\ (\hat e : \hat\tau) \rightsquigarrow \mathsf{let}\ s = M\ \mathsf{in}\ (e : \tau) : \tau}$$

$$\frac{\Delta \vdash \Pi s{:}\hat\sigma.\hat\tau \rightsquigarrow \Pi s{:}\sigma.\tau \quad \Delta, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash \hat e \rightsquigarrow e : \tau' \quad \overline{\Delta}, f{:}[\Pi s{:}\sigma.\tau], s{:}\sigma \vdash \tau' \equiv \tau}{\Delta \vdash \mathsf{fix}\,f(s{:}\hat\sigma){:}\hat\tau.\hat e \rightsquigarrow \mathsf{fix}\,f(s{:}\sigma){:}\tau.e : \Pi s{:}\sigma.\tau}$$

$$\frac{\Delta \vdash \hat e \rightsquigarrow e : \tau_e \quad \overline{\Delta} \vdash [\tau_e] \overset{\text{wh}}{\Longrightarrow} [\Pi s{:}\sigma.\tau] \quad \Delta \vdash_{\mathrm{P}} \hat M \rightsquigarrow M : \varsigma \quad \Delta \vdash M : \varsigma \le \sigma \rightsquigarrow N}{\Delta \vdash \hat e\,\hat M \rightsquigarrow e\,N : \tau[N/s]}$$

$$\frac{\Delta \vdash \hat e_1 \rightsquigarrow e_1 : \tau_1 \quad \Delta \vdash \hat e_2 \rightsquigarrow e_2 : \tau_2}{\Delta \vdash \langle \hat e_1, \hat e_2 \rangle \rightsquigarrow \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Delta \vdash \hat e \rightsquigarrow e : \tau_e \quad \overline{\Delta} \vdash [\tau_e] \overset{\text{wh}}{\Longrightarrow} [\tau_1 \times \tau_2]}{\Delta \vdash \pi_i \hat e \rightsquigarrow \pi_i e : \tau_i}\ (i \in \{1,2\})$$

$$\frac{\Delta \vdash_\kappa \hat M \rightsquigarrow M : \varsigma_M \quad \Delta \vdash \hat\sigma \rightsquigarrow \sigma \quad \Delta, s{:}\varsigma_M \vdash s : \varsigma_M \le \sigma \rightsquigarrow N}{\Delta \vdash \mathsf{pack}\,\hat M\ \mathsf{as}\ \langle\!|\hat\sigma|\!\rangle \rightsquigarrow \mathsf{pack}\ (\mathsf{let}\ s = M\ \mathsf{in}\ (N : \sigma))\ \mathsf{as}\ \langle\!|\sigma|\!\rangle : \langle\!|\sigma|\!\rangle}$$

Signature elaboration: $\Delta \vdash \hat\sigma \rightsquigarrow \sigma$

$$\frac{}{\Delta \vdash 1 \rightsquigarrow 1} \qquad \frac{}{\Delta \vdash [\![T]\!] \rightsquigarrow [\![T]\!]} \qquad \frac{\Delta \vdash \hat\tau \rightsquigarrow \tau}{\Delta \vdash [\![\hat\tau]\!] \rightsquigarrow [\![\tau]\!]} \qquad \frac{\Delta \vdash_{\mathrm{P}} \hat M \rightsquigarrow M : \sigma \quad \Delta \vdash M : \sigma \le [\![T]\!] \rightsquigarrow N}{\Delta \vdash \mathfrak{S}(\hat M) \rightsquigarrow \mathfrak{S}(N)}$$

$$\frac{\Delta \vdash \hat\sigma_1 \rightsquigarrow \sigma_1 \quad \Delta, s{:}\sigma_1 \vdash \hat\sigma_2 \rightsquigarrow \sigma_2}{\Delta \vdash \Pi^\delta s{:}\hat\sigma_1.\hat\sigma_2 \rightsquigarrow \Pi^\delta s{:}\sigma_1.\sigma_2} \qquad \frac{\Delta \vdash \hat\sigma_1 \rightsquigarrow \sigma_1 \quad \Delta, s{:}\sigma_1 \vdash \hat\sigma_2 \rightsquigarrow \sigma_2}{\Delta \vdash \Sigma s{:}\hat\sigma_1.\hat\sigma_2 \rightsquigarrow \Sigma s{:}\sigma_1.\sigma_2}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Delta \vdash \hat{e} \rightsquigarrow e : \tau \quad \Delta \vdash \hat{\sigma} \rightsquigarrow \sigma \quad \overline{\Delta} \vdash \tau \equiv (\!|\sigma|\!)}{\Delta \vdash_{\mathtt{S}} \mathsf{unpack}\ \hat{e}\ \mathsf{as}\ \hat{\sigma} \rightsquigarrow \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma}$$