Thesis Proposal:
Effective Type Theory for Modularity

Derek Dreyer

November 21, 2002

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee
Robert Harper (co-chair)
Karl Crary (co-chair)
Peter Lee
David MacQueen (University of Chicago)

Abstract

The programming language Standard ML (SML) draws much of its expressive power from advanced concepts in type theory. In addition, unlike other HOT (Higher-Order Typed) languages, SML has the distinction of being formally defined. Unfortunately, the Definition of SML is not type-theoretic, making it difficult to analyze and extend. Much work in the last decade has thus been devoted to better understanding the type-theoretic underpinnings of SML, particularly with respect to its module system.

In my thesis work I plan to study several extensions to SML that would enhance its support for modular programming even further. My methodology for formalizing these extensions follows the approach advocated by Harper and Stone, who gave an interpretation of SML that involves elaborating SML programs into type theory. This approach allows me to design my extensions at the level of the underlying type theory, which is the ideal setting for language design, but provides the option to fall back on elaboration techniques when type inference or syntactic sugar is key to making a language feature palatable. I will incorporate my extensions into the TILT compiler for SML, whose front-end is a suitable testbed for implementation as it is based on the Harper-Stone framework.

This proposal describes my work thus far on extending SML with higher-order modules, modules as first-class values, and recursive modules. A running theme in my type theory for modules is the importance of effects. The design of my type system for the first two extensions is driven by the intuition that the creation of abstract data types should be thought of as an effect. My type system is more expressive than previous systems because it distinguishes type abstraction (a compile-time effect) from type generativity (a run-time effect). While my proposal for recursive modules relies more heavily on elaboration techniques, its success hinges critically on the introduction of a type of lazy memoized modules, used to encapsulate term-level computational effects.

Contents

1 Introduction

The most notable aspect of the programming language Standard ML (SML) is its powerful type system. By enforcing strong invariants on how program data may be reliably manipulated, types facilitate the coherent modular development of robust programs. The module system of SML extends the power of the core type system by allowing programmers to specify and enforce their own invariants in the form of abstract data types. Furthermore, advanced features of SML's type and module systems, like *parametric polymorphism* and *functors*, support generic programming and code reuse, thus carrying to a typed setting most of the purported flexibility of untyped languages.

Unlike other full-fledged strongly-typed languages, such as Objective Caml and Haskell, SML has the distinction of being formally defined.[1] The Definition of SML [28] provides a clear and rigorous point of reference for both users and implementers of the language. The static semantics of SML given by the Definition classifies well-typed program expressions with so-called *semantic objects*. Like the syntactic classifiers present in SML itself (*e.g.,* types and signatures), semantic objects describe how a program expression may be used in the rest of the program.

The reason that the Definition employs semantic objects, instead of defining SML via a direct type system, is that the language of syntactic classifiers is too limited. For example, consider the following polymorphic function:

```
fun f x =
  let fun g y = if true then x else y
  in  g x
  end
```

While typechecking `f`, assuming `x` has been assigned the type variable α, the semantic object assigned to the inner function `g` is $\alpha \to \alpha$. In contrast, the semantic object assigned to the entire function `f` generalizes the type variable α to become $\forall\alpha.\,\alpha \to \alpha$. The distinction is important: `f` is a polymorphic function, but `g` is *not* polymorphic inside the body of `f` and can only be applied to terms of the same type as `x`. The syntactic type language of SML lacks an explicit $\forall$ constructor and is therefore unable to encode this critical distinction between monotypes and polytypes.

The very existence of the Definition has been a key factor in the success of SML, encouraging the development of independent implementations of the language while providing stability of SML code bases across those implementations. The flip side of that stability is that the semantic object language is closely tailored to the needs of SML, often to the point of seeming *ad hoc* from a more general semantic perspective. For instance, SML supports two forms of type definitions. A `type` definition of the form `type t =` τ binds a type identifier `t` as shorthand for type τ. A `datatype` definition of the form `datatype t =` C_1 `of` τ_1 `|` `...` `|` C_n `of` τ_n creates an abstract type `t` whose elements can be constructed from values of types $\tau_1, \ldots, \tau_n$ using $C_1, \ldots, C_n$ respectively, where $\tau_1, \ldots, \tau_n$ may refer recursively to `t`. To accommodate the presence of both definition forms, the semantic objects of the Definition that correspond to type definitions are *type environments* that bind type identifiers to *type structures*. A type structure, in turn, is a pair (θ, VE) consisting of a type θ and a value environment VE. If the type structure corresponds to a `type` definition, then VE is empty. Otherwise, θ must be an abstract type name and VE is the environment listing θ's constructors along with their types. Thus, type structures do not provide a unified semantic account of SML type definitions so much as a merging of two utterly different definition forms into a single either/or form.

1.1 Type Theory and Elaboration

While the Definition's semantic approach is perfectly suitable for a formal definition, much work in the last decade has been devoted to better understanding the type-theoretic underpinnings of SML, particularly with respect to the module system [25, 15, 16, 14, 21, 22, 19]. There are several reasons for this. For one, the methodology of type theory offers an extensible framework for studying language features and clarifying the relationships between them. There are established techniques for proving properties like safety and decidability of type systems, making it easier to design sound and effective extensions to a type system.

[1] Faxén [10] recently gave a static semantics formalizing most of the Haskell 98 Report [1], in a manner rather similar to the approach of Harper and Stone [19] described below. The informal Report remains the definition of Haskell, however.

Another reason is the advent of type-directed compilation [17, 47, 43, 41]. In traditional compilers, even for a strongly-typed language like SML, type information is discarded after typechecking. In type-directed compilers, the intermediate languages of the compiler are typed so as to enable optimizations that rely on type information. For instance, the TILT compiler for SML developed at CMU [47, 45, 36] maintains type information in order to implement *intensional type analysis* [17] and *tag-free garbage collection* [32]. Thus, regardless of how SML programs are typechecked, a type-directed SML compiler will at some point need to translate them into an internal typed language. Such a translation may be construed as an interpretation of SML in terms of more basic type-theoretic mechanisms. TILT performs this translation as part of typechecking, based on the framework of Harper and Stone [19, 18].

Harper and Stone formalize the *elaboration* of SML programs into a comparatively simple, yet powerful, type theory, referred to in TILT as the HIL, or "High Intermediate Language". There is a natural tension in the design of such a type theory. On one hand, a simple design is better because it draws out the connections between related language features and avoids redundancy. On the other hand, it is desirable for the theory to be as high-level as possible, in order to avoid heavy reliance on elaboration tricks and come closer to modeling the actual semantics of SML. The HIL strikes a balance between these goals, especially in its module system, which is based on the Harper-Lillibridge formalism of translucent sums [14]. HIL modules are very close to their SML counterparts in many respects, and the presence of interfaces (in the form of HIL signatures) makes the HIL high-level enough to serve as an interchange format for separate compilation. At the same time, the elaborator uses HIL modules to encode polymorphism (using functors) and type generativity, including `datatype` generativity (using opaque signature ascription), thus simplifying the HIL itself.

The type system of the HIL consists of a set of judgments regarding well-formedness, type equivalence and subtyping. Well-formedness judgments typically have the form

$$\Gamma \vdash \textit{IL-phrase} : \textit{IL-class}$$

where *IL-phrase* is some HIL expression like a term, type constructor or module, and *IL-class* is its classifier, like a type, kind or signature, respectively. The Harper-Stone (HS) elaborator consists of a set of judgments transforming well-formed SML expressions into well-formed HIL expressions. These judgments generally have the form

$$\Gamma \vdash \textit{EL-phrase} \rightsquigarrow \textit{IL-phrase} : \textit{IL-class}$$

where *EL-phrase* is an external-language (SML) expression, *IL-phrase* is its translation, and *IL-class* the type of the translation. The form of these elaboration judgments is reminiscent of judgments in the Definition, with the HIL playing the role of the semantic object language. The difference is that the output of elaboration is a well-typed program in a self-contained type system.

An example of how elaboration is helpful is the problem of identifier lookup. In the HIL, the context Γ consists of a sequence of variable declarations (*decs*) of the form $var_1{:}class_1, \cdots, var_n{:}class_n$, where the $class_i$'s are HIL classifiers. A reference to a variable is only valid if it appears in Γ. In SML, identifier lookup is made more complicated by features like `open` and `local` declarations, which reveal and conceal namespaces, respectively. The HS elaborator handles these features by adding a *label—i.e.,* a non-alpha-variable identifier—to each declaration in Γ ($lab_i \triangleright var_i{:}class_i$), and the labels are used for a variety of elaboration tricks. The set of elaborator labels includes the set of SML identifiers but also special identifiers that the SML programmer cannot write. For instance, the appearance of the special `expose` label in a signature indicates to the elaborator that the underlying module implements a `datatype` definition. Bindings under a `local` declaration are kept hidden by placing them in a module bound to a dummy label that does not correspond to any SML identifier. Opening a module's namespace with `open` is implemented by binding the module to a "starred" label, which tells the elaborator to look inside the module's namespace during identifier lookup. Although not based in type theory, the use of these special labels is relatively simple and provides a uniform mechanism for dealing with namespace issues that do not seem to admit clean type-theoretic solutions.

1.2 Effective Type Theory for Modularity

The HS framework provides one with a flexible method of formalizing and extending a full-fledged programming language like SML. The programming language designer can first attempt to account for language

features at the level of the underlying HIL type theory, which is the ideal setting for language design. When type inference or syntactic sugar is key to making a language feature palatable, the designer can then fall back on elaboration techniques. Thus, HS allows one to develop an *effective type theory*, that is, a type theory that scales—by means of elaboration—to an external language that is more pleasant to program in.

As suggested above, much of the power of the HS approach to language definition comes from the consolidation of a variety of features under the concept of *modularity*. The module system of SML, which is reflected in the underlying type theory, provides support for namespace management, type abstraction, generic programming and separate compilation. In addition, HIL modules are used to interpret a variety of "core" SML constructs such as polymorphic functions and `datatype` declarations.

In my thesis work I propose to study several ways of enhancing SML's support for modular programming even further, and to use the Harper-Stone framework as a basis for formalizing and implementing my language extensions. In this proposal document I present my work thus far on the following two extensions to the SML module system:

Higher-Order Modules and Modules as First-Class Values SML functors are *first-order* in the sense that they can only be defined at top-level and cannot appear inside the arguments or bodies of other functors. They are also *generative*, in the sense that functors whose result signatures contain abstract type specifications generate fresh abstract types at each application. Lastly, SML modules are purely *second-class*, meaning that they exist at a separate syntactic level from core-language terms, and the language of module constructs is limited.

These characteristics of the SML module system define one point in the possible design space of modules. Another viable point is the module system of Objective Caml, which supports *higher-order* modules and a form of non-generative (or *applicative*) functors but, like SML, restricts modules to be second-class. Other module systems have been proposed that support modules as *first-class* values but lack applicative functors. In Section 2, I will show how all of these design points can potentially offer useful programming idioms, and how all can be encompassed by a new type-theoretic approach to the problem. Specifically, I will describe a type system for higher-order second-class modules that supports both generative and non-generative functors and allows modules to be packaged as first-class values. This approach is novel in treating type abstraction as an effect and distinguishing different kinds of such typing effects.

Recursive Modules Modules in SML are strictly hierarchical, meaning that there can be no cyclic dependencies between modules. If one wants to write mutually recursive function or data type definitions that belong conceptually in different program components, the hierarchical restriction inhibits natural modularization of the program. Separating those definitions into different modules, however, would require support for (mutually) *recursive modules*.

While several authors have proposed recursive module extensions to ML and Scheme [9, 11], few have attempted a type system or even a Definition-style formalization for their extensions. Crary *et al.* [3] have given a theoretical account of recursive modules, introducing the key ideas of *fixed-point modules* and *recursively dependent signatures*. More recently, Russo [40] has developed an extension to the Moscow ML compiler [33] that relaxes some of Crary *et al.*'s restrictions on the presence of computational effects in recursive modules, but offers inadequate support for separate compilation of recursive modules. In Section 3, I propose an approach to recursive modules that is capable of encoding Russo's semantics, but which overcomes its deficiencies by introducing an explicit type of *lazy memoized modules*. My proposal relies on a combination of extensions to both the type theory and the elaborator.

In the process of writing this proposal, I discovered a theme running through my type-theoretic accounts of hierarchical and recursive modules: namely, the interaction of modules and effects. Whereas the key to the success of my type theory of higher-order modules is the tracking of type abstraction and generativity, which I consider *typing effects*, the key to enabling separate compilation of recursive modules is the tracking of *term-level effects*, such as I/O and operations on mutable state. (Hence, the title of my proposal is a *double entendre*.) I am not sure at the moment what this connection signifies or how it might be exploited, but I am very interested in pursuing it further.

Along with scaling my theoretical work to the level of full-fledged extensions to the HS framework, there are several other issues less directly related to modular programming that I plan to examine as well. Discussed

speculatively in Section 4, these include extending the way that SML supports pattern matching (with *views*) and overloading (with *type classes*). Lastly, I plan to incorporate my extensions and revisions to SML into the TILT compiler, which is an ideal testbed for implementation since the current front-end is based on the HS framework. As one might expect, there are a number of aspects of the TILT elaborator that diverge or extrapolate from the HS formalization for practical reasons. Unfortunately, they are largely undocumented at present. I discuss some of these discrepancies briefly in Section 4.6, and I plan to re-examine them in light of my extensions to the language. At a minimum, as part of my thesis work, I will fully document the points of divergence and why I believe the elaborator to faithfully implement its (revised) formalization.

2 A Type System for Modules

Since the publication of the original Definition of Standard ML [27], there has been much research devoted to understanding SML's features better through the methodology of type theory. One result of this effort was the development (by Harper and Lillibridge [14, 24] and Leroy [21], independently) of the *translucent sums* (aka *manifest types*) formalism, which formed the basis of a major improvement to SML's module system in its Revised Definition of 1997 [28].

The motivation for translucent sums is rooted in the basic language design ideal that all the information about a term that is known to the rest of the program should be expressible in its type. Thus, unlike in SML '90, signatures in SML '97 are allowed to contain transparent type specifications (`type t =` τ), as well as abstract ones (`type t`), so that the most-specific "type" of a module can be expressed as a fully transparent signature that reveals the definitions of all its type components. Whereas the concept of type sharing constraints between modules was difficult to account for in previous type theories for modules, it falls naturally out of the translucent sums approach. In addition, the translucent sums type system allows the programmer to ascribe a signature to a module *opaquely*, *i.e.,* so that the type information available to clients of the module is precisely what appears in the signature. The presence of both abstract and transparent type specifications thus gives the programmer fine-grained component-wise control over the propagation of type information.

2.1 The Design Space

In addition to the key advance of the translucent sum formalism, studying SML from the perspective of type theory has shed light on limitations of its module system and suggested natural ways of extending and improving it. In this section I will describe several interesting directions that researchers have explored in the design space of module type systems.

2.1.1 Higher-Order Functors

It is common to view the SML module system as constituting its own λ-calculus and, viewing it as such, it is a natural question why functors are restricted to *first-order*, *i.e.,* why they may not take other functors as arguments or return them as results, as permitted in typical λ-calculi. The issue, it turns out, is not that allowing SML functors to be higher-order would create difficulties for typechecking, but that the *generative* nature of SML functors is inappropriate for certain key programming idioms in the higher-order case.

SML functors are generative in the sense that, every time they are applied, they generate fresh abstract types corresponding to the abstract type components in their result signatures. Consider the following higher-order functor `Apply`, written in pseudo-ML syntax, which takes as arguments a functor `F` of signature `SIG->SIG` and a structure `X` of signature `SIG`, and applies `F` to `X`:

```
signature SIG = sig type t val x : t end
functor Apply (F : SIG -> SIG, X : SIG) = F(X)
```

The result signature of `F` is abstract, so by functor generativity, the most-specific signature of `F(X)` is also abstract. Thus, any application of `Apply` will produce an abstract module of signature `SIG`.

If the actual argument `F` to `Apply` is a functor with abstract result signature, then this is clearly the correct semantics. However, suppose that the actual argument `F` is a functor with transparent result signature, *e.g.,* the identity functor `Id` of signature `(X:SIG) -> SIG where type t = X.t`, or a constant functor that

```
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 1: Generative Functor Example

always returns a module with `type t = int`. In those cases, the signature of `F(X)` is transparent, while the signature of `Apply(F,X)` is not, signaling a loss of type propagation in the use of the higher-order functor. The problem is that in order for `Apply` to be applicable to a range of instantiations for `F` whose result signatures may differ, the signature of `F` needs to be kept abstract, which in SML is tantamount to being generative.

MacQueen and Tofte [26] proposed one solution to this problem, namely to "re-elaborate" higher-order functors at each point they are applied, in order to take advantage of type information regarding actual functor arguments like `F` in the above example. For instance, `Apply(Id,X)` would force the re-elaboration of `Apply` with `Id` substituted for `F`, resulting in a module whose type component `t` is transparently equal to `X.t`. There are two main disadvantages of this approach. One is that it relies heavily on having access to the code for the higher-order functor, which is not a valid assumption in the presence of separate compilation. The other is that the MacQueen-Tofte formalism is not based on type theory, but on the operational style of the original Definition of SML, which employed "stamps" to model generativity. It is thus not clear how their approach would transfer to a type-theoretic setting.

2.1.2 Applicative vs. Generative Functors

An alternative to the MacQueen-Tofte proposal is to consider what functors could be if they were not generative. The first to do this (in a language with abstract data types) was Leroy, in his work on *applicative* (or *non-generative*) functors [23], which serves as the basis of the module system of Objective Caml [34]. In Leroy's system, every time a functor is applied to the same argument, the abstract type components in the result are the same. This equivalence is made observable by allowing named functor applications to occur in types. For instance, the body of the `Apply` functor (above) could, in an applicative setting, be given the fully transparent signature `SIG where type t = F(X).t`. In the case that the actual argument `F` was 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 2: Applicative Functor Example

identity, the signature of `Apply` would then yield the desired equivalence `Apply(Id,X).t` = `Id(X).t` = `X.t`.

Although motivated by the desire to give more expressive types to higher-order functors, applicative functors pose an interesting alternative to generative functors even in the first-order case. The question of which one is more appropriate depends on whether or not one is writing code in a purely functional style.

Figures 1 and 2 illustrate the distinction.[2] In the first example in Figure 1, we define a functor that implements a symbol table, containing 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 (mutable) 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. This depends critically on the generativity of the `SymbolTableFun` functor. 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 distinct.

The symbol table example demonstrates the importance of generative functors for encoding abstract types in the presence of effects. In the absence of effects, however, Leroy [23] gives several examples to motivate the adoption of applicative functors. For instance, one may wish to implement persistent sets using ordered lists. Figure 2 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.

[2]The careful reader will notice that I have distinguished the functors in the two examples by using `:>` to seal the body of the generative functor and `::` to seal the body of the applicative one. This notation is non-standard for expressing this distinction, but will make sense in the context of the type theory I present in Sections 2.3 through 2.5.

Given the usefulness of both applicative and generative functors, several researchers—in particular, Russo [38] and Shao [42]—have proposed type systems that support both. As I will discuss in Section 2.2, both of their approaches are interesting, but have complementary deficiencies. One of the main contributions of my proposed type theory is that it gives a simultaneous account of applicative and generative functors that overcomes the problems of previous work.

2.1.3 First-Class vs. Second-Class Modules

Modules in SML and Objective Caml are purely *second-class*, meaning that the language of modules is separate from the core language of terms and is limited to the introduction and elimination forms for structures and functors, plus signature ascription. One benefit of these limitations, first observed and exploited by Harper, Mitchell and Moggi [16], is the principle of *phase separation*, stating that the type components of a module cannot depend on any run-time computations, *i.e.,* terms.

Phase separation is what makes the idea of applicative functors sound. The reason it is sensible to write functor applications in types such as `F(X).t` is that the type component `t` of `F(X)` can only depend on the (compile-time) type components of `F` and `X`, not their (run-time) term components, and thus it will be the same every time we apply `F` to `X`. In fact, Leroy's system does not utilize phase separation to its full extent. Specifically, he requires functor applications appearing in types to be in "named form", and only admits equivalence of types projected from syntactically identical modules. This is unnecessarily restrictive: `F(X).t` *should* equal `F(Y).t` so long as `X` and `Y` have equivalent type components, a notion that I call *static equivalence*. However, even if `Y` is defined to be precisely `X`, the equality of `F(X).t` and `F(Y).t` is not observable in Leroy's system as it is not syntactic identity. In contrast, the more recent treatments of applicative functors (by Russo and Shao), as well as the one I propose in Section 2.3, take full advantage of phase separation.

At the same time, it is also desirable for modules to be usable as first-class values. This would 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). In his thesis [24], Lillibridge explored the properties of the translucent sums formalism in a fully general first-class module system. One major problem in his system is that typechecking is undecidable. Yet there are other, less ambitious ways to incorporate some of the benefits of first-class modules into a second-class system, which avoid undecidability, as we discuss in Section 2.6. A problem that is central, however, to the idea of a module-as-first-class-value is that the type components of such a module may depend on run-time conditions, which breaks the principle of phase separation. Thus, a type system that encompasses both modules-as-first-class-values and applicative functors must have some way of distinguishing modules that obey phase separation from those that do not.

2.2 Previous Work on Type Systems for Applicative and Generative Functors

Before presenting my type system, it is important to describe the work of Russo [38] and Shao [42], each of whom has given a type system supporting both applicative and generative functors. Understanding the weaknesses in their systems will illuminate the design choices in mine.

Russo's thesis presents a more type-theoretic version of the Definition of SML that makes explicit the correspondence (observed by Mitchell and Plotkin [30]) between abstract types and existential types. In his framework, for example, a module sealed opaquely with an abstract signature (`MOD:>SIG`) would be assigned a semantic object that hides the abstract type components of `SIG` with existentially-bound type names.

The distinction between applicative and generative functors is then viewed as a question of where the abstraction (via existential quantification) happens in functors with abstract result signatures. For a generative functor, the result signature is an existential so that, every time the functor is applied, the types in the result are hidden under an existential and thus held distinct from any other types. In contrast, applicative functors hoist the existential quantification outside of the functor. In order for the abstract type components of the result to be existentially quantified outside of the functor, they must be parameterized over the abstract type components of the argument. For instance, consider the `SetFun` functor in Figure 2. Russo would treat this functor as defining an abstract type *constructor* `'a setfun`, resulting in the following signature:[3]

[3]My notation here is a morally accurate simplification of Russo's using pseudo-ML syntax.

```
∃ 'a setfun.
  (Elem : ORD) -> sig
                      type elem = Elem.elem
                      type set = Elem.elem setfun
                      ...
                  end
```

Applying `SetFun` to a module with `elem` defined as `elem1` and a module with `elem` defined as `elem2` will produce results with `set` defined as `elem1 setfun` and `elem2 setfun`, respectively. Thus, although the actual implementation of `setfun` is abstract, the functor respects static equivalence of its arguments.

The problem with Russo's system, which is implemented in the Moscow ML compiler [33], is that it allows any functor to be applicative or generative, the choice being one that the programmer makes at the definition of the functor. In particular, the body of an applicative functor may contain a generative functor application. First of all, this semantics severely diminishes the power of functor generativity. For example, the body of the functor `SymbolTableFun` in Figure 1 relies on the invariant that it only has to interpret indices into one symbol table. Suppose, however, that one defines an applicative functor `AppSymTabFun` that is merely the eta-expansion of `SymbolTableFun`. Two instantiations of `AppSymTabFun` would have compatible `symbol` types, thus breaking the implementation invariant. Since the main point of using a generative functor in the first place was to preserve that invariant, it is unclear what purpose generative functors serve under Russo's semantics. As I will show in Section 2.6, his extension for modules-as-first-class-values is also fundamentally limited by the weakness of his generative functors.

Moreover, I recently discovered that a variant of the eta-expansion example can be twisted into code that reveals Moscow ML's higher-order module extension to be *unsound* [4]. It is important to note that Russo's thesis is not unsound, because in his thesis he develops the languages of applicative and generative functors only in isolation. That I only uncovered the unsoundness of Moscow ML during the writing of this proposal is a testament to the subtlety of module type theory.

Shao defines a type system for modules based on the idea that applicative functors are fully transparent functors that have had the definitions of some of their type components hidden *after the fact*. This semantics is achieved in his type theory by assigning all functors a generative functor signature by default. A generative functor signature may then be coerced to an applicative functor signature only if the signature is fully transparent (or what Shao calls *instantiated*). Finally, type components in the result of an applicative functor signature may be made abstract while preserving its applicativity. Otherwise, Shao's applicative functors have the same semantics as Russo's, in particular they respect static equivalence of their arguments.

The problem with Shao's system is precisely the reverse of the problem with Russo's—as opposed to admitting too many applicative functors, he admits too few. Specifically, one cannot write an applicative functor in his system whose body has any opaque substructures. The best examples of why opaque substructures are important are provided by the interpretation of ML datatypes as abstract types [19]. In both SML and O'Caml, datatypes are *opaque* in the sense that their representation as recursive sum types is not exposed. Thus, in Shao's system, datatypes may not appear in the bodies of applicative functors, which considerably limits the utility of his applicative functors and seems like an arbitrary prohibition. For example, we would not be able to re-implement the `SetFun` functor with persistent splay trees, using a `datatype` declaration to define the tree type, without making the functor generative. In short, Shao's system is overly restrictive because it does not distinguish *type abstraction* from *type generativity*.

2.3 A Type System for Fully Transparent Modules

The previous section has hopefully made it clear that the issues concerning the interplay of abstraction and generativity are rather complex. The type theory I will now set forth is based on my work with Karl Crary and Bob Harper described in Dreyer *et al.* [5]. To simplify matters, I will begin the presentation of my type system for modules by omitting the constructs for abstraction and generativity, namely any form of sealing (*e.g.,* M `:>` σ). Thus, modules in this simplified system will be fully transparent and functors applicative (by default, since generativity is not supported yet). It is a virtue of my type-theoretic approach to the problem that the constructs for abstraction can be studied orthogonally from the rest of the language, and that in fact they form conservative extensions of the simplified type system.

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

Figure 3: Syntax of Fully Transparent Modules

2.3.1 Syntax

The syntax of my type system for fully transparent modules appears in Figure 3. Throughout the technical development I will consider alpha-equivalent expressions to be identical, and write the capture-avoiding substitution of module M for module variable s in a syntactic expression E as $E[M/s]$.

The language is divided into a *core* language of types and terms, and a *module* language of signatures and modules. First let us consider the atomic constructs. Modules consist of type and term components, so the most basic modules are the module $[\tau]$, containing a single type component τ, and the module $[e : \tau]$, containing a single term component e of type τ. (We will omit the type annotation on term modules when it is clear from context.) The corresponding signatures are $[\![T]\!]$, the signature of an atomic module containing a type, and $[\![\tau]\!]$, the signature of an atomic module containing a term of type τ. The elimination forms for these signatures are the core-language constructs $\mathsf{Typ}\, M$, which extracts the type from a module of signature $[\![T]\!]$, and $\mathsf{Val}\, M$, which extracts the term from a module of signature $[\![\tau]\!]$.

As for the remainder of the core level, I only include function and product types, but this can and will be extended in the actual HIL type system to include sums, recursive types, etc. The product type and term constructs are entirely standard. Note, however, that the (recursive) function construct $\mathsf{fun}\, f(s{:}\sigma){:}\tau.e$ takes a *module* as its argument. (In the case that a function is not recursive—*i.e.,* that f is not free in e—I will write $\Lambda s{:}\sigma.e$, and in the case that a function type is non-dependent, I will write $\sigma \to \tau$.) As modules may contain both type and term components, this function construct may be used to encode an ordinary function as one whose argument is an atomic term module, and a polymorphic abstraction as a function whose argument is an atomic type module:

$$
\begin{array}{lcllcl}
\tau_1 \to \tau_2 & \stackrel{\text{def}}{=} & [\![\tau_1]\!] \to \tau_2 & \forall\alpha.\tau & \stackrel{\text{def}}{=} & \Pi s{:}[\![T]\!].\tau[\mathsf{Typ}\, s/\alpha] \\
\lambda x{:}\tau.e & \stackrel{\text{def}}{=} & \Lambda s{:}[\![\tau]\!].e[\mathsf{Val}\, s/x] & \Lambda\alpha.e & \stackrel{\text{def}}{=} & \Lambda s{:}[\![T]\!].e[\mathsf{Typ}\, s/\alpha] \\
e_1 e_2 & \stackrel{\text{def}}{=} & e_1[e_2] & e\,\tau & \stackrel{\text{def}}{=} & e[\tau]
\end{array}
$$

More generally, we may encode *higher-order polymorphism* via a function whose argument is a functor representing a type *constructor*, *e.g.,* of signature $[\![T]\!] \to [\![T]\!]$, and *polymorphic recursion* via a *recursive* function whose argument has a type component.

The module level contains its own functions and products, although the function construct $\lambda s{:}\sigma.M$ is often called a *functor* and the product construct $\langle s = M_1, M_2\rangle$ is often called a *structure*. The variable s in $\langle s = M_1, M_2\rangle$ stands for the result of evaluating M_1 and is bound in M_2. Functors and structures have the corresponding signatures $\Pi s{:}\sigma_1.\sigma_2$ and $\Sigma s{:}\sigma_1.\sigma_2$, respectively. (When they are not dependent—*i.e.,* when s does not occur free in σ_2—I will write $\sigma_1 \to \sigma_2$ and $\sigma_1 \times \sigma_2$.) Note that unlike SML structures, products in this type system are restricted for simplicity to unlabeled pairs; the encoding of SML-style structures in terms of unlabeled pairs is left to elaboration, which I describe in Section 2.7.2.

Included at both module and term levels is an annotated let construct for hiding a module binding. Also included at the module level is a unit module $\langle\rangle$ of unit signature 1, as well as a Top signature, which is a supertype of all signatures. More important is the *singleton* signature $\mathfrak{S}(M)$, a subtype of the atomic type signature $[\![T]\!]$ that may be ascribed only to type modules whose type component is equivalent to $\mathsf{Typ}\, M$. In essence, the signatures $[\![T]\!]$ and $\mathfrak{S}(M)$ are the encodings of abstract (`type t`) and transparent (`type t = Typ` M) type specifications, respectively. That $\mathfrak{S}(M)$ is a subtype of $[\![T]\!]$ signifies that one may forget the definition of a type component during signature matching.

2.3.2 Static Module Equivalence

A key question in any type system, but particularly in one for modules, is: "When are two types τ_1 and τ_2 equivalent?" In the case that $\tau_1 = \mathsf{Typ}\, M_1$ and $\tau_2 = \mathsf{Typ}\, M_2$, the question becomes: "When are two modules M_1 and M_2 equivalent?" As suggested above in Section 2.1.3, in a second-class module system that obeys the principle of phase separation, the most liberal notion of equivalence for modules is *static equivalence*, *i.e.,* equivalence of their type components. Furthermore, static equivalence is the right notion because the only reason we care about equivalence of modules is to determine the equivalence of the *types* projected from those modules.

Given the idea of static equivalence, the formal equivalence rules are much as one would expect. The module equivalence judgment is written $\Gamma \vdash M_1 \cong M_2 : \sigma$ and type equivalence judgment $\Gamma \vdash \tau_1 \equiv \tau_2$. Atomic type modules are equivalent if their type components are equivalent, and atomic term modules of the same signature are always (trivially) equivalent since they have no type components:

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]} \qquad \frac{\Gamma \vdash M_1 : [\![\tau]\!] \quad \Gamma \vdash M_2 : [\![\tau]\!]}{\Gamma \vdash M_1 \cong M_2 : [\![\tau]\!]}$$

There are also rules for transferring module equivalence judgments to type equivalence judgments and reducing redundant conversions between the type and module levels:

$$\frac{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]}{\Gamma \vdash \tau_1 \equiv \tau_2} \qquad \frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash [\mathsf{Typ}\, M] \cong M : [\![T]\!]}$$

The remainder of the equivalence rules follow the rules of Stone and Harper's singleton kind system very closely [45]. In particular, there is a rule that observes that if a module M_1 has a singleton signature $\mathfrak{S}(M_2)$, then M_1 is equivalent to M_2. Finally, there are a standard set of congruence rules, which ensure that pairs of equivalent modules are equivalent, that applications of equivalent functors to equivalent arguments are equivalent, etc., in addition to rules that ensure β- and η-equivalence for modules.[4] Space considerations preclude further discussion of the complexities of equivalence in the presence of singleton signatures, but details are given in Dreyer *et al.* [6], and the full set of equivalence rules can be found in Appendix A.

2.3.3 Typing Rules

The typing rules for modules, written $\Gamma \vdash M : \sigma$, are mostly very straightforward. When eliminating functors and structures with dependent signatures, care must be taken to perform the proper substitution in the result signature. Specifically, the rules for functor application and second projection are as follows:

$$\frac{\Gamma \vdash F : \Pi s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash M : \sigma_1}{\Gamma \vdash FM : \sigma_2[M/s]} \qquad \frac{\Gamma \vdash M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash \pi_2 M : \sigma_2[\pi_1 M/s]}$$

The only rules of interest are the so-called "selfification" rules that allow a module to be given a more precise signature with singletons. The need for selfification dates back to Harper and Lillibridge's "VALUE" rules [14] and Leroy's "signature strengthening" rules [21], which allow modules, or at least module variables, to be assigned fully transparent signatures by specifying abstract type components to be transparently equal to themselves. For example, a module `X` with signature `sig type t end` can be given the more precise signature `sig type t = X.t end`. That way, if one defines another module `Y` to be precisely `X`, then `Y` will be given `X`'s principal signature and the equivalence of `Y.t` and `X.t` will be apparent from `Y`'s signature. Formally, for a module of atomic signature $[\![T]\!]$, selfification takes the form of singleton introduction:

$$\frac{\Gamma \vdash M : [\![T]\!]}{\Gamma \vdash M : \mathfrak{S}(M)}$$

The other selfification rules allow singletons to propagate through the signatures of functors and structures:

$$\frac{\Gamma \vdash M : \Pi s{:}\sigma_1.\sigma_2' \quad \Gamma, s{:}\sigma_1 \vdash Ms : \sigma_2}{\Gamma \vdash M : \Pi s{:}\sigma_1.\sigma_2} \qquad \frac{\Gamma \vdash \pi_1 M : \sigma_1 \quad \Gamma \vdash \pi_2 M : \sigma_2}{\Gamma \vdash M : \sigma_1 \times \sigma_2}$$

[4] In fact, following the system in Stone's thesis [44], we use extensionality rules in lieu of η-equivalence rules, and we omit β-equivalence rules because they can be shown admissible using singletons and extensionality.

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

Figure 4: Singletons at Higher Signatures

For instance, the rule on the left combined with singleton introduction allows a functor F of signature $[\![T]\!] \to [\![T]\!]$ to be given the fully transparent signature $\Pi s{:}[\![T]\!].\mathfrak{S}(Fs)$.

Although my type system, following Stone and Harper, only provides primitive singletons at signature $[\![T]\!]$, singletons at higher signatures are definable in the language, and the selfification rules ensure that the rules governing primitive singletons are admissible for higher-order singletons. Figure 4 gives the definition of the higher-order singleton $\mathfrak{S}_\sigma(M)$, which is assignable to all modules that are equivalent to M at signature σ.

2.4 Abstraction as an Effect

The type system presented thus far allows any two modules to be compared for equivalence, and likewise allows the type component to be extracted from any module with signature $[\![T]\!]$. Consider, however, what happens if we add a construct $M :\!: \sigma$ for opaquely sealing a module M with a signature σ. The intended semantics of sealing is that all the information observable about $M :\!: \sigma$ is what appears in σ.

Suppose that modules M_1 and M_2 are both defined to be $[\texttt{int}] :\!: [\![T]\!]$, but M_3 is defined to be $[\texttt{bool}] :\!: [\![T]\!]$. By reflexivity of module equivalence, M_1 must be equivalent to M_2 since they are syntactically identical, but clearly M_1 can never be equivalent to M_3 since they do not have equivalent type components under the sealing. This violates the semantics of sealing because the underlying implementation of M_1 as $[\texttt{int}]$, which is not visible in its signature, has an effect on its (in-)equality with other sealed modules.

The solution I propose is to treat sealing as an effectful operation that renders a module's type components *indeterminate*. Formally, let's say that a module that is free of sealing is *pure*, while a module that contains sealing is *impure*. Only pure modules are eligible for comparison with other modules, and $\mathfrak{S}(M)$ and $\mathsf{Typ}\, M$ are only valid if M is pure. (Since all modules in the type system of the previous section were pure, this constitutes a conservative extension.) Thus, if a module contains sealing, the only way to refer to its type components is to bind it to a variable, which may then appear in types because variables are pure.

Tracking purity in the type system is very straightforward. We simply need to annotate the module typing judgment with an indication of whether the module in question is known to be pure or not. The new judgment is written $\Gamma \vdash_\kappa M : \sigma$, where the purity level κ is either $\texttt{P}$ (for "Pure") or $\texttt{I}$ (for "Well-formed but possibly impure"). The purity levels form a simple lattice of two elements with $\texttt{P}$ at the bottom and $\texttt{I}$ at the top. Sealing introduces impurity, and purity can be forgotten by subsumption in the purity lattice:

$$
\frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\texttt{I}} (M :\!: \sigma) : \sigma} \qquad \frac{\Gamma \vdash_{\kappa'} M : \sigma \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash_\kappa M : \sigma}
$$

Otherwise, modules are deemed to be just as pure as their submodules. The only interesting rules are for function application and second projection:

$$
\frac{\Gamma \vdash_\kappa F : \Pi s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\texttt{P}} M : \sigma_1}{\Gamma \vdash_\kappa FM : \sigma_2[M/s]} \qquad \frac{\Gamma \vdash_{\texttt{P}} M : \Sigma s{:}\sigma_1.\sigma_2}{\Gamma \vdash_{\texttt{P}} \pi_2 M : \sigma_2[\pi_1 M/s]}
$$

In FM and $\pi_2 M$, the module M must be pure in order for the elimination construct to be even well-formed. The reason is that, in both cases, a module containing M is substituted for a variable in the resulting signature σ_2. Since variables are pure, they may occur in types and singletons, so M must be pure in order for the substitution to preserve the well-formedness of σ_2.

It is worth noting that functors in this system still behave applicatively, even if their bodies contain sealing. To see this, consider that while a functor containing sealing is impure, once we bind it to a variable, uses of that variable are pure. For instance, recalling the applicative functor example from Figure 2, the functor application `SetFun(IntOrd)` is pure since `SetFun` and `IntOrd` are both variables. Therefore, it can be selfified to the fully transparent signature whose `set` component is specified transparently to be `SetFun(IntOrd).set`, and that selfified signature is what is assigned to both `IntSet1` and `IntSet2`.

2.5 Generativity as a Dynamic Effect

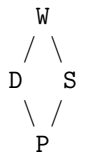
My type system in its current form provides essentially the same semantics for modules as the higher-order module system from Chapter 5 of Russo's thesis. The next step, then, is to incorporate an account of type generativity, which is often described as the generation of types at run time. This informal description gives the proper intuition for how to extend the type system I have developed so far.

In a purely second-class module system, the principle of phase separation ensures that type components of modules are not actually generated at run time. Therefore, modules containing generative type components are *pretending* to violate phase separation. Nevertheless, to achieve the desired semantics, we will treat them as if they really do violate phase separation.

In order to handle such generative modules soundly, we distinguish between two different kinds of effects: *static* effects, which correspond to type abstraction via sealing, and *dynamic* effects, which correspond to run-time type generation. The sealing construct I introduced in the previous section induces the purely static effect of hiding type information. I will hereafter refer to $M :\!: \sigma$ as *weak sealing*. To provide generative type abstraction, I now add a construct $M :\!> \sigma$ for *strong sealing*, which induces both a static and a dynamic effect. Not only does strong sealing hide the definitions of type components specified opaquely in σ, but it gives the impression (to the type system) that those definitions depend on run-time conditions. Given that "impression", it would be unsound for a strongly sealed module to appear inside an applicative functor, since each instantiation of the functor could potentially produce different types based on information available at run time.

Static (im-)purity is a syntactic condition determined by inspecting a module for sealing. In contrast, dynamic (im-)purity is a property of computations. Dynamic effects are encapsulated by suspended computations, *i.e.,* functors, and released by their applications. Thus, we track these effects in the manner of traditional effects systems—by introducing a new arrow type, or rather, a new functor signature representing functors whose bodies are dynamically impure. Such functors are better known, of course, as *generative* functors, and so the new functor signature is written $\Pi^{\text{gen}} s{:}\sigma_1.\sigma_2$ (or $\sigma_1 \xrightarrow{\text{gen}} \sigma_2$ if non-dependent). For example, the `SymbolTableFun` functor from Figure 1 would have to be assigned the signature $1 \xrightarrow{\text{gen}}$ `SYMBOL_TABLE` since its body is strongly sealed.

Formally, we extend the lattice of purity levels to form a diamond:



`P` is for "Pure", `D` is for "Dynamically pure, but possibly statically impure", `S` is for "Statically pure, but possibly dynamically impure", and `W` is for "Well-formed, but possibly statically or dynamically impure". Modules assigned the purity level `I` in the previous section would now be assigned `D` since the language of the previous section only dealt with static effects and thus all modules were dynamically pure. Modulo this renaming of purity levels, the extension to handle dynamic effects and strong sealing is conservative.

Some key typing rules governing the propagation of static and dynamic effects are given in Figure 5. Strongly sealed modules engender both a static and dynamic effect (rule 1). Weakly sealed modules contain at least a static effect, rendering them at best dynamically pure (2). Applicative functors may be coerced by subsumption to generative functors, thus forgetting that their bodies are dynamically pure (3). Applicative functors must have dynamically pure bodies (4), while generative functors suspend any dynamic effects in their bodies (5). Applications of applicative functors do not induce any effect (6), whereas applications of generative functors unleash a dynamic effect (7).

$$\frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma} \ (1) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} (M :: \sigma) : \sigma} \ (2) \qquad \frac{\Gamma, s{:}\sigma_1 \vdash \sigma_2 \ \mathrm{sig}}{\Gamma \vdash \Pi s{:}\sigma_1.\sigma_2 \leq \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2} \ (3)$$

$$\frac{\Gamma, s{:}\sigma_1 \vdash_\kappa M : \sigma_2 \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma_1.M : \Pi s{:}\sigma_1.\sigma_2} \ (4) \qquad \frac{\Gamma, s{:}\sigma_1 \vdash_\kappa M : \sigma_2}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma_1.M : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2} \ (5)$$

$$\frac{\Gamma \vdash_\kappa M_1 : \Pi s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma_1}{\Gamma \vdash_\kappa M_1 M_2 : \sigma_2[M_2/s]} \ (6) \qquad \frac{\Gamma \vdash_\kappa M_1 : \Pi^{\mathrm{gen}} s{:}\sigma_1.\sigma_2 \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma_1}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} M_1 M_2 : \sigma_2[M_2/s]} \ (7)$$

Figure 5: Key Typing Rules for Tracking Dynamic and Static Effects

The type system is now powerful enough for us to encode most of Shao's higher-order module calculus as a subsystem. In particular, Shao's system is like mine except that it only has strong sealing, not weak sealing. The restriction to strong sealing has the effect that functors containing opaque substructures must contain strong sealing and thus be generative, matching the semantics of Shao's system as described in Section 2.2. There is still one small corner of Shao's semantics that we cannot account for, but we will leave the discussion of that point until it becomes important in Section 3.3.5.

2.6 Packaging Modules as First-Class Values

As discussed in Section 2.1.3, the ability to treat modules as first-class values is a desirable feature. One practical approach to modules as first-class values that does not run afoul of decidability was suggested by Mitchell *et al.* [29], who propose that second-class modules automatically be wrapped as existential packages [30] to obtain first-class values. A similar approach to modules as first-class values is described by Russo and implemented in Moscow ML [39].

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:

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

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

The main limitation of existentially-packaged modules is the closed-scope elimination construct. It has been observed repeatedly in the literature[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 term-level functions over unpackaged, second-class modules, which can be given the dependent type $\Pi s{:}\sigma.\tau$ instead of $\langle\!|\sigma|\!\rangle \to \tau$.

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

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

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

[5] Originally by MacQueen [25] and later by Cardelli and Leroy [2] and Lillibridge [24], among others.

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

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

Figure 6: Packaged Module Extension

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 6. Note that the closed-scope unpacking construct is definable under this extension 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 in fact depend on run-time conditions. In the type system of the previous section, 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 6.

Nevertheless, the packaged module extension treads close enough to unsound waters to call for a formal proof of soundness with respect to a suitable dynamic semantics. I have worked out proofs of progress and preservation on paper and they are largely straightforward. The only real oddity is that in order to prove soundness, we must first erase all uses of *weak sealing* from the language. For instance, if the variable F is bound to an applicative functor with a weakly sealed body, then substituting the *impure* definition of F for occurrences of F in the rest of the program will be ill-typed. To avoid this problem, it is reasonable to erase uses of weak or strong sealing before evaluating a program, since sealing of any kind has no real run-time significance anyway. The dynamic semantics for the language without weak sealing is given in Appendix B.

2.7 Typechecking and Elaboration

A critical feature of my type system for modules is that typechecking is decidable, due to the existence of principal signatures. Formally, we can define a decidable algorithm $\Gamma \vdash_\kappa M \Rightarrow \sigma$ that computes M's principal signature σ and minimal purity level κ in context Γ. To determine whether $\Gamma \vdash_{\kappa'} M : \sigma'$ for arbitrary σ' and κ', we invoke the principal signature synthesis algorithm and then check whether $\Gamma \vdash \sigma \leq \sigma'$ and $\kappa \sqsubseteq \kappa'$. The proof of decidability is described in full in Dreyer *et al.* [6] and, though rather involved, adheres closely to the decidability proof for Stone and Harper's singleton kind system [45]. The synthesis algorithm, however, is relatively straightforward and is given in Appendix C.

2.7.1 The Avoidance Problem

The existence of principal signatures relies on some subtle choices made in the design of the type system. The clearest example of such a choice is the module-level let construct, which requires that a signature annotation be provided for the let body. Consider the principal signature synthesis rule for an impure let:

$$\frac{\Gamma \vdash_{\kappa_1} M_1 \Rightarrow \sigma_1 \quad \Gamma, s{:}\sigma_1 \vdash_{\kappa_2} M_2 \Rightarrow \sigma_2 \quad \Gamma, s{:}\sigma_1 \vdash \sigma_2 \leq \sigma \quad \Gamma \vdash \sigma\ \mathsf{sig} \quad \kappa_1 \sqcup \kappa_2 \neq \mathtt{P}}{\Gamma \vdash_{\kappa_1 \sqcup \kappa_2} \mathsf{let}\ s = M_1\ \mathsf{in}\ (M_2 : \sigma) \Rightarrow \sigma}$$

Note that the signature annotation σ needs to be well-formed in the ambient context Γ, so that it can be assigned to the let after the variable s is no longer in scope. If no annotation were provided, the synthesis

algorithm would need to find such a signature on its own. In other words, it would have to come up with the least supersignature of σ_2 not mentioning s. If M_1 is pure, then the solution is $\sigma_2[M_1/s]$, but if M_1 is impure, this signature is not well-formed.

The *avoidance problem* [12, 24] is that, in general, a signature does not necessarily have a least supersignature avoiding a particular variable. For example, consider the signature

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

The most obvious supersignature of σ avoiding s is $(\llbracket T \rrbracket \to \llbracket T \rrbracket) \times \llbracket T \rrbracket$. However, for any type τ, the signature $\sigma_\tau = \Sigma F{:}(\llbracket T \rrbracket \to \llbracket T \rrbracket).\mathfrak{S}(F[\tau])$ is a more precise supersignature of σ, as it brings out the connection between the first and second projections of a module of signature σ. Yet, since F is abstract, σ_{τ_1} and σ_{τ_2} are incomparable iff τ_1 is not equivalent to τ_2, so none of the σ_τ's is minimal.

The need to avoid the avoidance problem manifests itself in other constructs as well. In particular, the typing rules for functor applications $F(M)$ and second projections $\pi_2 M$ require that M be pure. As discussed in Section 2.3, this restriction is necessary for M to be substituted in the conclusions of those rules. A similar approach is taken by Shao [42], but an alternative is to follow Harper and Lillibridge [14] and use the following rules, which involve only non-dependent signatures:

$$\frac{\Gamma \vdash_\kappa F : \sigma_1 \to \sigma_2 \quad \Gamma \vdash_\kappa M : \sigma_1}{\Gamma \vdash_\kappa FM : \sigma_2} \qquad \frac{\Gamma \vdash_\kappa M : \sigma_1 \times \sigma_2}{\Gamma \vdash_\kappa \pi_2 M : \sigma_2}$$

In the case that M is pure, the typing rules in my system can be shown admissible using these rules, *e.g.,* in the case of $F(M)$, by promoting F with signature $\Pi s{:}\sigma_1.\sigma_2$ to the supersignature $\mathfrak{S}_{\sigma_1}(M) \to \sigma_2[M/s]$. Otherwise, however, we run into another instance of the avoidance problem, where we must find a supersignature of σ_2 that does not mention s in order to apply the rule. By forcing M to be pure, my system circumvents this, possibly at the cost of some expressiveness.

2.7.2 Elaboration

As I do not know how to relax these restrictions without sacrificing decidability of typechecking, I take the approach outlined in Section 1 and handle the avoidance problem as part of an *elaboration algorithm* from a more flexible *external* language to the more limited *internal* type system developed so far. The structure of the elaborator is based on Harper and Stone's elaborator for full SML [19], but for simplicity my external language (EL) only extends the internal language (IL) in a few ways. A significant part of my thesis will involve scaling this simplified EL to an extension of full SML, but it is useful to test out ideas first in a less daunting setting. Hereafter, I will distinguish EL syntax by writing $\hat{M}$, $\hat{\sigma}$, etc.

One way in which the EL is more flexible is that it supports *un*-annotated let's, as well as unrestricted functor applications and projections. Unannotated module-level let's are already present in SML in the form of `local` structure declarations and transparent signature ascription, and I employ an idealization of Harper and Stone's method of dealing with them. The idea is to introduce a form of existential signature $\exists s{:}\sigma_1.\sigma_2$ in the language of the elaborator, which represents conceptually the least supersignature of σ_2 not mentioning $s : \sigma_1$. Of course this signature does not actually exist in the IL; it is a purely elaboration-time notion, but it allows us to give a principal signature to the unannotated $\mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2$.

This idea of an *elaborator signature* is fundamental to the elaboration approach. However, since the target of the elaborator is the IL, an elaborator signature σ must have a representation, or *erasure*[6], into the IL, written $\overline{\sigma}$. The erasure of $\exists s{:}\sigma_1.\sigma_2$ is merely $\Sigma s{:}\overline{\sigma_1}.\overline{\sigma_2}$, and correspondingly the elaboration of $\mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2$ is $\langle s = M_1, M_2 \rangle$. In other words, let's are translated as pairs. The distinction between let's and pairs lies in how the elaborator treats existentials versus products. In particular, whenever a variable s with signature $\exists s'{:}\sigma_1.\sigma_2$ is used, the elaborator immediately projects out the second component $\pi_2 s$ with signature $\sigma_2[\pi_1 s/s']$, which effectively renders the let-bound first component inaccessible. Functor applications $\hat{F}(\hat{M})$ are elaborated as if $\hat{F}(\hat{M})$ were a derived form expanding to $\mathsf{let}\ s_F = \hat{F}\ \mathsf{in}\ \mathsf{let}\ s_M = \hat{M}\ \mathsf{in}\ s_F(s_M)$.

[6]To stem proliferation of signature modifiers, I implicitly extend the syntax of σ, M, Γ, etc. occurring in elaboration rules to include these *elaborator signatures*. However, when elaboration rules refer to IL judgments, all signatures are implicitly erased to IL signatures.

module elaboration	$\Gamma \vdash_\kappa \hat{M} \rightsquigarrow M : \sigma$
signature elaboration	$\Gamma \vdash \hat{\sigma} \rightsquigarrow \sigma$
type elaboration	$\Gamma \vdash \hat{\tau} \rightsquigarrow \tau$
term elaboration	$\Gamma \vdash \hat{e} \rightsquigarrow e : \tau$
coercive signature matching	$\Gamma \vdash M : \sigma \preceq \sigma' \rightsquigarrow M'$
signature matching right rules	$\Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \sigma' \rightsquigarrow M'$
signature matching left rules (*aka* "peeling")	$M : \sigma \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'$
label lookup	$M : \sigma \vdash \ell \stackrel{\text{look}}{\Longrightarrow} M' : \sigma'$

Figure 7: Elaborator Judgments

Another way in which the EL is more flexible is that it supports SML-style structures with labeled components that can be dropped or reordered during signature matching. Formally, EL structures have the form $\langle \hat{B} \rangle$, where $\hat{B}$ is a list of submodule bindings with the syntax

$$\hat{B} ::= \epsilon \mid \ell \triangleright s = \hat{M}, \hat{B}$$

Here, ϵ represents the empty list, and $\ell \triangleright s = \hat{M}$ binds $\hat{M}$ to variable s, with label ℓ. The distinction between label and variable [14] allows $\hat{M}$ to be externally accessible by the unchangeable ℓ, while its internal name s (whose scope is $\hat{B}$) remains alpha-variable. Analogously, the EL signatures of these structures have the form $\langle \hat{D} \rangle$, where $\hat{D}$ is a list of submodule declarations with the syntax

$$\hat{D} ::= \epsilon \mid \ell \triangleright s : \hat{\sigma}, \hat{D}$$

When the elaborator is scaled to full SML, labels and variables can be coalesced at the EL level into "identifiers" as they are in the Harper-Stone EL. I maintain the distinction in this idealized EL so as not to require a complex mechanism for identifier lookup.

As one would expect, EL structures are elaborated as products; specifically, $\langle \ell \triangleright s = \hat{M}, \hat{B} \rangle$ is translated as $\langle s = M, B \rangle$, where M is the translation of $\hat{M}$ and B is the translation of $\hat{B}$. In addition, a new elaborator signature $\langle \ell \triangleright s : \sigma, D \rangle$ is needed to keep track of labels for signature matching. The IL erasure of this signature naturally just drops the labels to become $\Sigma s{:}\overline{\sigma}.\overline{D}$.

The judgments defining the elaboration algorithm are listed in Figure 7. The form of the main translation judgments (the first four) is completely straightforward. The next three implement coercive signature matching, which is needed during elaboration of constructs like functor application or sealing. (IL subtyping alone is not sufficient because it does not account for existentials or the dropping or reordering of labeled structure components.) The main matching judgment matches a module M of signature σ against target signature σ', resulting in a coercion module M' with the target signature. Note that these are all elaborator-level modules and signatures, not EL code. The actual work of coercion is divided between two auxiliary judgments via the following single rule defining coercion:

$$\frac{M : \sigma \stackrel{\text{peel}}{\Longrightarrow} M'' : \sigma'' \quad \Gamma \vdash M'' : \sigma'' \preceq_{\mathrm{R}} \sigma' \rightsquigarrow M'}{\Gamma \vdash M : \sigma \preceq \sigma' \rightsquigarrow M'}$$

The left premise peels off the outermost existentials of σ, in essence ignoring the outermost hidden module components of M. This peeling judgment is useful in its own right, *e.g.,* when we need to coerce a signature into the form of a Π signature, but we do not have a specific target in mind. The right premise, which does most of the work, may then assume that the output signature of the peeling judgment, σ'', is not an existential. Lastly, the label lookup judgment checks whether module M with signature σ provides a label ℓ and, if so, constructs the appropriate sequence of projections to extract the ℓ component from M.

Figure 8 displays several illustrative elaboration rules, all of which are fairly self-explanatory, if noticeably hairier than the IL typing rules. Rules 8 and 9 illustrate the elaboration of let's and module projections $\hat{M}.\ell$, rules 10 and 11 the coercion right rules for functors and structures, rules 12 and 13 the peeling rules, and rules 14 and 15 the label lookup rules. The complete set of elaboration rules appears in Appendix D.

$$\frac{\Gamma \vdash_{\kappa_1} \hat{M}_1 \rightsquigarrow M_1 : \sigma_1 \quad \Gamma, s{:}\sigma_1 \vdash_{\kappa_2} \hat{M}_2 \rightsquigarrow M_2 : \sigma_2}{\Gamma \vdash_{\kappa_1 \sqcup \kappa_2} \mathsf{let}\ s = \hat{M}_1\ \mathsf{in}\ \hat{M}_2 \rightsquigarrow \langle s = M_1, M_2\rangle : \exists s{:}\sigma_1.\sigma_2} \quad (8)$$

$$\frac{\Gamma \vdash_{\kappa} \hat{M} \rightsquigarrow M : \sigma_M \quad s : \mathfrak{S}_{\sigma_M}(s) \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma' \quad M' : \sigma' \vdash \ell \stackrel{\text{look}}{\Longrightarrow} N : \sigma}{\Gamma \vdash_{\kappa} \hat{M}.\ell \rightsquigarrow \langle s = M, N\rangle : \exists s{:}\sigma_M.\sigma} \quad (9)$$

$$\frac{\Gamma, s{:}\sigma_1' \vdash s : \sigma_1' \preceq \sigma_1 \rightsquigarrow M \quad \Gamma, s{:}\sigma_1', t{:}\sigma_2[M/s] \vdash t : \sigma_2[M/s] \preceq \sigma_2' \rightsquigarrow N \quad (\delta, \delta') \neq (\text{gen}, \epsilon)}{\Gamma \vdash F : \Pi^{\delta} s{:}\sigma_1.\sigma_2 \preceq_{\text{R}} \Pi^{\delta'} s{:}\sigma_1'.\sigma_2' \rightsquigarrow \lambda s{:}\sigma_1'.\,\mathsf{let}\ t = FM\ \mathsf{in}\ (N : \sigma_2')} \quad (10)$$

$$\frac{M : \sigma \vdash \ell \stackrel{\text{look}}{\Longrightarrow} M' : \sigma' \quad \Gamma \vdash M' : \sigma' \preceq \sigma_\ell \rightsquigarrow N \quad \Gamma, s{:}\mathfrak{S}_{\sigma_\ell}(N) \vdash M : \sigma \preceq_{\text{R}} \langle D\rangle \rightsquigarrow \langle B\rangle}{\Gamma \vdash M : \sigma \preceq_{\text{R}} \langle \ell \triangleright s : \sigma_\ell, D\rangle \rightsquigarrow \langle \ell \triangleright s = N, B\rangle} \quad (11)$$

$$\frac{\pi_2 M : \sigma_2[\pi_1 M/s] \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'}{M : \exists s{:}\sigma_1.\sigma_2 \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'} \quad (12) \qquad \frac{\text{No other peeling rule applies.}}{M : \sigma \stackrel{\text{peel}}{\Longrightarrow} M : \sigma} \quad (13)$$

$$\frac{}{M : \langle \ell \triangleright s : \sigma, D\rangle \vdash \ell \stackrel{\text{look}}{\Longrightarrow} \pi_1 M : \sigma} \quad (14) \qquad \frac{\ell \neq \ell' \quad \pi_2 M : \langle D[\pi_1 M/s]\rangle \vdash \ell' \stackrel{\text{look}}{\Longrightarrow} M' : \sigma'}{M : \langle \ell \triangleright s : \sigma, D\rangle \vdash \ell' \stackrel{\text{look}}{\Longrightarrow} M' : \sigma'} \quad (15)$$

Figure 8: Illustrative Elaboration Rules

A disadvantage of employing an elaborator is that it is difficult to argue rigorously about whether it is correct. Unlike the IL, which is defined by a declarative type system and proven decidable by a sound and complete typechecking algorithm, the EL has no declarative definition but is defined directly via the elaborator, so there is no reference system against which to compare it. 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 IL and, moreover, the signature is principal. In addition, as the rules in Figure 8 indicate, the coercion, peeling and lookup judgments only handle pure modules.

Theorem 2.1 (Elaborator Invariants)
Suppose $\Gamma \vdash \text{ok}$. *Then:*

1. *If* $\Gamma \vdash_{\kappa} \hat{M} \rightsquigarrow M : \sigma$, *then* $\Gamma \vdash_{\kappa} M \Rightarrow \sigma$ *(and hence* $\Gamma \vdash_{\kappa} M : \sigma$*).*
2. *If* $\Gamma \vdash \hat{\sigma} \rightsquigarrow \sigma$, *then* $\Gamma \vdash \sigma$ sig.
3. *If* $\Gamma \vdash \hat{\tau} \rightsquigarrow \tau$, *then* $\Gamma \vdash \tau$ type.
4. *If* $\Gamma \vdash \hat{e} \rightsquigarrow e : \tau$, *then* $\Gamma \vdash e \Rightarrow \tau$ *(and hence* $\Gamma \vdash e : \tau$*).*
5. *If* $\Gamma \vdash M : \sigma \preceq \sigma' \rightsquigarrow M'$ *and* $\Gamma \vdash_{\text{P}} M : \sigma$ *and* $\Gamma \vdash \sigma'$ sig, *then* $\Gamma \vdash_{\text{P}} M' : \sigma'$.
6. *If* $\Gamma \vdash M : \sigma \preceq_{\text{R}} \sigma' \rightsquigarrow M'$ *and* $\Gamma \vdash_{\text{P}} M : \sigma$ *and* $\Gamma \vdash \sigma'$ sig, *then* $\Gamma \vdash_{\text{P}} M' : \sigma'$.
7. *If* $M : \sigma \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'$ *and* $\Gamma \vdash_{\text{P}} M \Rightarrow \sigma$ *(or* $\Gamma \vdash_{\text{P}} M : \sigma$*),*
   then $\Gamma \vdash_{\text{P}} M' \Rightarrow \sigma'$ *(or* $\Gamma \vdash_{\text{P}} M' : \sigma'$*).*
8. *If* $M : \sigma \vdash \ell \stackrel{\text{look}}{\Longrightarrow} M' : \sigma'$ *and* $\Gamma \vdash_{\text{P}} M \Rightarrow \sigma$ *(or* $\Gamma \vdash_{\text{P}} M : \sigma$*),*
   then $\Gamma \vdash_{\text{P}} M' \Rightarrow \sigma'$ *(or* $\Gamma \vdash_{\text{P}} M' : \sigma'$*).*

Proof: By straightforward induction on the elaboration algorithm. ■

3 Recursive Modules

Recursive modules are one of the most frequently requested extensions to SML. Intuitively, just as structures combine the product constructs at the type and term levels, and functors combine the function constructs at the type and term levels, it seems there should be a way to combine the constructs for recursive definitions at the type level (*i.e.,* `datatype`'s) and at the term level (*i.e.,* recursive functions) to form a recursive construct at the module level. Indeed, it would undoubtedly be straightforward to design a recursive module extension if one were to restrict recursive modules to contain only `datatype` and `fun` declarations.

Problems arise, however, if one wants to be less *ad hoc* about it and allow recursive modules to contain arbitrary code. It is easy to foresee the problems by first considering what would happen if we were to

- allow arbitrary recursive definitions at the type level like `type rec t = int * t`, instead of restricting recursion to `datatype` declarations
- allow arbitrary recursive definitions at the term level like `val rec x = 1::y and y = 2::x`, instead of restricting recursion to `fun` declarations

First, the kind of recursive type needed to implement transparently recursive type definitions such as `t = int * t` is different from the kind used—by the Harper-Stone interpretation, for instance—to implement SML `datatype`'s. To encode the transparent kind, we need so-called *equi-recursive* types, which admit equivalence between themselves and their recursive expansions. Formally, a recursive type $\mu\alpha.\tau$ is equi-recursive if one may observe the equivalence $\mu\alpha.\tau \equiv \tau[\mu\alpha.\tau/\alpha]$. In contrast, *iso-recursive* types require the use of explicit `fold` and `unfold` coercions to mediate between the recursive type and its expansion. Iso-recursive types, whose equational theory is much simpler than that of equi-recursive types, are enough to represent SML `datatype`'s because `datatype`'s are abstract—the `fold`'s and `unfold`'s are performed in a canonical way at constructor applications and during pattern matching, respectively, and those are the only places they can occur.

Likewise, the kind of recursive term-level construct needed to implement fixed-points over arbitrary expressions is beyond what is required for SML. In particular, the only recursive construct in the Harper-Stone HIL is a fixed-point over a set of mutually recursive function bindings. On the other end of the spectrum, Haskell supports recursive bindings of arbitrary expressions, but all terms in Haskell are lazy, memoized computations. The laziness is important: When lazily evaluated, the example recursive binding above produces an infinite stream of alternating 1's and 2's; when eagerly evaluated, the definition enters an infinite loop. There are, however, intermediate possibilities between the SML and Haskell forms of recursive definition as well. In short, the point is not that recursive modules are prohibitively difficult to implement, but rather that they may imply extensions to the power of the core language, so we must consider the design space carefully.

3.1 The Design Space

In this section I will examine several key axes in the design space of recursive modules from the perspective of type theory. The concepts and terminology that I present here are based to a large extent on the foundational work of Crary *et al.* [3]. I will discuss their type theory for recursive modules in more detail in Section 3.2. For now, I will use a loose combination of their formalism and mine to sketch out the terrain of recursive modules.

3.1.1 Fixed-Point Modules

The traditional type-theoretic model of recursion is the *fixed-point*, so a natural way to encode recursive modules is using a module-level fixed-point. In the context of my higher-order module type system of Section 2, consider adding the module construct $\mathsf{fix}(s : \sigma.\, M)$, where s is the recursive variable by which the module M may refer to itself, and σ is the signature of s. In a sense, $s : \sigma$ can be thought of as a *forward declaration* of M, somewhat like a header file in C. With this construct, mutually recursive modules can be encoded as substructures of M that refer to one another via projections from s. The topic of how to compile mutually recursive modules *separately* will be dealt with in Section 3.1.4.

The most obvious typing rule for fixed-point modules would require the body M to have the same signature as its forward declaration:

$$\frac{\Gamma, s{:}\sigma \vdash M : \sigma}{\Gamma \vdash \mathsf{fix}(s : \sigma.\, M) : \sigma}$$

Given this rule, the question arises how a fixed-point module should be evaluated. As SML is a strict functional language, the natural way to evaluate $\mathsf{fix}(s : \sigma.\, M)$ as an extension of SML is to eagerly evaluate M; once M is evaluated to a value V, $\mathsf{fix}(s : \sigma.\, V)$ steps to $V[\mathsf{fix}(s : \sigma.\, V)/s]$. What happens, however, if the evaluation of M involves the evaluation of s (*e.g.,* trivially, in the case that M is simply s)? The problem is that evaluating M means evaluating a module with free variables.[7]

There are at least two solutions to this problem. One is to leave the above typing rule as it is, but instrument the dynamic semantics so that every occurrence of s inside M incurs a well-definedness check. More formally, the idea is to treat a recursive module as a memoized computation. During the evaluation of M, s refers to the undefined computation $\bot$; attempts to evaluate s raise an exception. Then, once M evaluates to V, s can be memoized to V, and $\mathsf{fix}(s : \sigma.\, M)$ can step to V as well.

Another solution is to leave the dynamic semantics alone but modify the static semantics in order to prohibit the evaluation of M from ever encountering s. One easy way of accomplishing this is to restrict M to be a value. A less restrictive approach is to only require M to be *valuable*, where by "valuable" I mean that M is pure and terminating. While in general valuability is undecidable, Harper and Stone [18] employ a conservative approximation of valuability in their HIL, written $\Gamma \vdash M \downarrow \sigma$, in order to correctly interpret SML's value restriction. In the case of fixed-point modules, it is necessary for the valuability judgment to distinguish between ordinary module variables, which correspond to values at run time, and the recursive module variable, which is undefined at run time. For instance, if m is a module variable bound with σ in the context Γ, then $\mathsf{fix}(s : \sigma.\, m)$ should be well-formed in Γ, but $\mathsf{fix}(s : \sigma.\, s)$ should not. Formally, we add a new kind of context entry $s \uparrow \sigma$, indicating that s has σ and is *non-valuable*. The typing rule for fixed-point modules then becomes:

$$\frac{\Gamma, s \uparrow \sigma \vdash M \downarrow \sigma}{\Gamma \vdash \mathsf{fix}(s : \sigma.\, M) : \sigma}$$

Although I have phrased these two approaches as solutions to a problem regarding fixed-point modules, the same problem arises and the same solutions apply when considering term-level fixed-points $\mathsf{fix}(x : \tau.\, e)$, where e is an arbitrary term. This makes perfect sense, since the dynamic part of a recursive module is just a recursive term. In essence, as alluded to in the introduction above, the problem is that recursive modules implicitly extend the power of the term language and there is more than one way to extend it. While the latter solution—requiring the body of a fixed-point module to be valuable—would appear to banish side effects from recursive modules, I will explain in Section 3.3.3 how in fact it can *subsume* the flexibility of the memoizing approach, if we allow memoization to be encoded explicitly via a separate construct.

3.1.2 Recursively Dependent Signatures

Having added a recursive construct to the module level, it is natural to consider whether there is a need for a recursive construct at the signature level. Indeed, the signatures for strictly hierarchical modules that I gave in Section 2.3 are insufficient for recursive module programming, as the following example illustrates.

Suppose we wish to write a fixed-point module with two mutually recursive substructures `Exp` and `Dec`, which implement interpreters for "expressions" and "declarations", respectively, in the abstract syntax of some language. Each substructure exports an abstract data type representing its respective syntactic class, `exp` or `dec`, along with some functions for constructing and deconstructing elements of the abstract data type and a function `exec` that executes a program expression or declaration for side effects and, for simplicity, returns unit. If we attempt to write out the forward declaration for this fixed-point module (Figure 9), we immediately run into difficulties. The problem is that the type of the `Exp.makeLetExp` function refers to the

[7] In the dynamic semantics of pure λ-calculi with fix, it is common for a term-level fixed-point $\mathsf{fix}(x.\, e)$ to step to $e[\mathsf{fix}(x.\, e)/x]$, in which case the evaluation of $\mathsf{fix}(x.\, x)$ would not be ill-defined, but would enter an infinite loop. In an impure language, however, this semantics is inappropriate, as the evaluation of e may have side effects that ought not be repeated at every occurrence of x in e.

```
signature EXPDEC = sig
  structure Exp : sig
    type exp
    (* makeLetExp(d,e) constructs "let d in e" *)
    val makeLetExp : Dec.dec * exp -> exp
    (* matchLetExp(e) returns SOME(d,e1) if e is of the form "let d in e1" *)
    val matchLetExp : exp -> (Dec.dec * exp) option
    ...
    val exec : exp -> unit
  end
  structure Dec : sig
    type dec
    (* makeValDec(v,e) constructs "val v = e" *)
    val makeValDec : var * Exp.exp -> dec
    (* matchValDec(d) returns SOME(v,e) if d is of the form "val v = e" *)
    val matchValDec : dec -> (var * Exp.exp) option
    ...
    val exec : dec -> unit
  end
end
```

Figure 9: Problematic Forward Declaration of `Exp` and `Dec`

abstract type `Dec.dec` (shown boxed above), but `Dec` is not in scope yet at that point. Switching the order of `Exp` and `Dec` does not help, since the type of `Dec.makeValDec` refers to `Exp` as well.

What is needed, then, is a means to express recursive dependencies between substructure specifications in a forward declaration signature. A simple way of supporting this is to add a new *recursively dependent signature* $\rho s.\,\sigma$, in which specifications in σ may refer recursively to one another through the module variable s. For example, the definition of the `EXPDEC` signature can now be made sensible by prefacing it with "ρ `X.`" and replacing the references to `Dec` (in the specification of `Exp`) with references to `X.Dec`. As for the semantics of recursively dependent signatures, or *rds*'s, there are three main questions: 1) when can a module be given an rds, 2) when is an rds well-formed, and 3) how are rds's implemented?

First, observe that the variable s in $\rho s.\,\sigma$ is essentially a stand-in for whatever module the rds is classifying. Thus, for a module M to be given the rds $\rho s.\,\sigma$, it is necessary for M to have the signature σ under the substitution of M for s. Conversely, there is the question of how to use a module whose signature is an rds. The answer is simple: if M has $\rho s.\,\sigma$, we can eliminate the rds by replacing the references to s in σ with M. Formally, the introduction and elimination rules for rds's are as follows:

$$\frac{\Gamma \vdash_{\mathtt{P}} M : \sigma[M/s]}{\Gamma \vdash_{\mathtt{P}} M : \rho s.\,\sigma} \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : \rho s.\,\sigma}{\Gamma \vdash_{\mathtt{P}} M : \sigma[M/s]}$$

Note that M is required to be pure in order for the signature $\sigma[M/s]$ to be valid. Adding these rules directly to my type system for higher-order modules would be ill-advised, as it would severely complicate the proof of decidability. I offer them here more as *properties* that ought to hold for any type system involving rds's.

The answers to the second and third questions are intertwined, since the implementation of rds's hinges on what sort of recursive dependencies are permitted in an rds. Obviously, all recursive dependencies in an rds are references *to type components*. Additionally, in the `EXPDEC` example, the references to the recursive variable `X` are all *from value component* specifications. I will call such dependencies *dynamic-on-static*, as opposed to *static-on-static* dependencies, which correspond to recursive references *from transparent type specifications* to other type components.

If we restrict rds's to contain only dynamic-on-static dependencies, then there is a straightforward way to implement rds's in terms of strictly hierarchical signatures. The idea is to hoist all the type specifications to the top of the signature; after hoisting, references from value components to type components are no longer recursive. For instance, the `EXPDEC` rds could be encoded by the hierarchical signature in Figure 10.

```
sig
  structure X : sig
    structure Exp : sig type exp end
    structure Dec : sig type dec end
  end
  structure Exp : sig
    type exp = X.Exp.exp
    (* reference to X.Dec.dec no longer recursive *)
    val makeLetExp : X.Dec.dec * exp -> exp
    ...
  end
  structure Dec : sig
    type dec = X.Dec.dec
    ...
  end
end
```

Figure 10: Hierarchical Encoding of `EXPDEC`

This hoisting is only possible if the hoisted signature, consisting solely of the type specifications, is purely hierarchical, *i.e.,* if there are no static-on-static dependencies.

Static-on-static dependencies place stronger requirements on the underlying type system. Consider, for example, the rds $\sigma = \rho s.\mathfrak{S}([\texttt{int} \times \mathsf{Typ}\, s])$, which would correspond to the following pseudocode signature:

$$\rho\, \texttt{X. sig type t = int * X.t end}$$

By the rds elimination rule, any module M with this signature can be given the signature $\mathfrak{S}([\texttt{int} \times \mathsf{Typ}\, M])$. By straightforward singleton reasoning, this leads to the type equivalence $\mathsf{Typ}\, M \equiv \texttt{int} \times \mathsf{Typ}\, M$. The only way to account for this sort of equivalence is by using *equi-recursive types*, as I discussed at the beginning of Section 3. With equi-recursive types, σ can be viewed as equivalent to the signature $\mathfrak{S}([\mu\alpha.\texttt{int} \times \alpha])$. In fact, Crary *et al.* [3] show that in order to handle fully general rds's with static-on-static dependencies, the type theory must support equi-recursive *type constructors* of higher kind. The behavior of singletons in the presence of equi-recursive type constructors appears to be rather complex, and it is not known whether type equivalence in such a theory is decidable.

It is therefore desirable to avoid static-on-static dependencies if possible. At first glance, however, they appear necessary in order to write mutually recursive `datatype` specifications in separate substructures of an rds, which was one of the main motivations for recursive modules in the first place! Fortunately, they are not necessary, precisely because SML `datatype`'s are abstract. In particular, according to the HS interpretation, a `datatype` specification is translated as an abstract type spec together with its constructor and destructor functions, much like the signatures of `Exp` and `Dec` in the `EXPDEC` rds. Thus, what one might call "`datatype`-on-static" dependencies are revealed to be merely dynamic-on-static references emanating from the specifications of the `datatype`'s constructor and destructor functions, not from the type spec itself.

3.1.3 Opaque vs. Transparent Forward Declarations

The typing rules for fixed-point modules given in Section 3.1.1 do not make any restrictions on the form of the forward declaration signature. It turns out, surprisingly, that allowing an arbitrary signature to serve as a forward declaration raises several major problems.

The first, rather serious, problem is that simple recursive modules written in a completely straightforward manner fail to typecheck! Consider, for example, an attempted implementation of the `ExpDec` module, whose forward declaration signature `EXPDEC` motivated the development of recursively dependent signatures in the previous section. Figure 11 displays only a few lines of `ExpDec`, enough to illustrate the type error. Let us refer to the body of `ExpDec` as M. Then, by the rds introduction rule, in order for M to match `EXPDEC`, the function M`.Exp.makeLetExp` must have type M`.Dec.dec * `M`.Exp.exp -> `M`.Exp.exp`. The type

```
structure ExpDec = fix (X : EXPDEC. struct
  structure Exp = struct
    datatype exp = ... | LetExp of X.Dec.dec * exp | ...
    ...
    fun makeLetExp (d:X.Dec.dec, e:exp) = LetExp(d,e)
    ...
    fun exec (e:exp) =
      ... case X.Dec.matchValDec(d) of
            SOME(v:var, e1:X.Exp.exp) => ...
  end
  structure Dec = struct
    datatype dec = ... | ValDec of var * Exp.exp | ...
    ...
  end
end)
```

Figure 11: Problematic Implementation of `Exp` and `Dec`

of M.`Exp.makeLetExp`, however, is `X.Dec.dec *` M.`Exp.exp -> Exp.exp`, which does not match the specification because `X.Dec.dec` is not equivalent to M.`Dec.dec`. The error arises because, at the point where `makeLetExp` is defined, there is only one way to refer to the type `Dec.dec`, namely through the recursive module variable `X`. Thus, there is no way to make `ExpDec` typecheck under the fixed-point typing rule given so far. While Dreyer *et al.* [8] show how to overcome this first problem by generalizing the typing rule for fixed-point modules, the generalized rule is quite baroque and depends heavily on the use of an explicit phase-distinction formalism in the style of Harper *et al.* [16].

Another related, if less severe, problem is illustrated by the snippet of the `Exp.exec` function shown in Figure 11. Somewhere inside the function, a call is made to `X.Dec.matchValDec` which, if the match succeeds, returns a variable `v` and an expression $\mathtt{e}_1$ of type `X.Exp.exp`, *not* M.`Exp.exp`. What can the implementation of `Exp` do with an `X.Exp.exp`? The only way it can deconstruct a value of type `X.Exp.exp` is through a call to `X.Exp.matchLetExp` or some other forward-declared function. This is grossly inefficient compared to the cost of deconstructing a value of type M`.Exp.exp`, whose implementation is known in the body of `Exp`. The source of both this problem and the previous problem is the opacity of `exp` and `dec` in the forward declaration `EXPDEC`, which leaves no way to connect `X.exp` and `X.dec` to M`.exp` and M`.dec`.

A third problem arises when we consider how fixed-point modules are implemented. For the sake of brevity, consider the fixed-point module

$$\mathsf{fix}(s : \Sigma t{:}[\![T]\!].[\![\mathtt{unit} \to \mathsf{Typ}\, t]\!].\, \langle t = [\mathtt{int} \times \pi_1 s], [\lambda().(3, (\pi_2 s)())]\rangle)$$

corresponding to the pseudocode signature:

```
fix (S : sig type t val f : unit -> t end. struct
  type t = int * S.t
  val f = fn () => (3, S.f())
end)
```

Unlike `ExpDec`, this module is perfectly well-typed, if useless. When compiled, the module needs to be phase-split into a type part and a term part. For some definition of the type part `t`, the term part will clearly be the recursive function `fix(f : unit -> t. fn () => (3, f()))`. Observe, however, that the body of this fixed-point term has the type `unit -> int` $\times$ `t`. Thus, in order for the term to be well-typed, we must define `t` to be the *equi-recursive* type $\mu\alpha.\mathtt{int} \times \alpha$, so that `t` $\equiv$ `int` $\times$ `t`. In general, Crary *et al.* [3] have shown that, as for rds's with static-on-static dependencies, fixed-point modules with opaque forward declarations require the underlying type theory to support equi-recursive type constructors of higher kind.

In summary, opacity in forward declarations makes fixed-point modules hard to write, hard to compile and hard to typecheck! The simple panacea is to require all forward declarations to be *fully transparent*. First, this restriction eliminates the distinction between forward-declared type components and their definitions in the

module body, since those definitions must now be specified up front in the forward declaration. By the same token, it is easy to show that fully transparent fixed-point modules can be implemented straightforwardly without equi-recursive types (see Dreyer *et al.* [8] for details).

The downside of the transparency restriction is that it prevents `datatype` specifications from appearing in forward declarations since, according to SML semantics, `datatype`'s are abstract. In addition, it does not provide a way for the definitions of type components in mutually recursive substructures of a fixed-point module to be hidden from one another. The `datatype` problem is important and difficult to solve in a clean type-theoretic way; I will defer further discussion of it until Section 4.2. As for setting up abstraction boundaries between mutually recursive modules, one way to achieve this is by separately compiling each module, given only an abstract interface for the other modules. Separate compilation, in turn, introduces its own problems.

3.1.4 Separate Compilation of Mutually Recursive Modules

Suppose we have a fixed-point module with two substructures `A` and `B` that we would like to compile separately, defined as follows:

```
fix (X : σ. struct
  structure A = struct fun f(n) = X.B.g(n-1) end
  structure B = struct fun g(n) = X.A.f(n-1) end
end)
```

The standard way of encoding separate compilation in SML is through functors. In the case of this example, `A` and `B` could each be parameterized over `X`:

```
functor FunA (X : σ) = struct fun f(n) = X.B.g(n-1) end
functor FunB (X : σ) = struct fun g(n) = X.A.f(n-1) end
```

Linking the separately compiled modules is performed correspondingly by functor application:

```
fix (X : σ. struct
  structure A = FunA(X)
  structure B = FunB(X)
end)
```

Note that, while the forward declaration σ of the linking fixed-point is required to be fully transparent, the argument signatures of `FunA` and `FunB` are not. Specifically, the argument signature of `FunA` may hold the type specifications in the `B` substructure abstract, and vice versa, allowing for a form of *client-side abstraction*.[8]

The problem with this approach is that the semantics of fixed-point modules will cause the linking module to raise an error, either at compile time or run time, depending on which fixed-point typing rule is used. If the valuability-checking rule is used, the linking module will not typecheck—SML's call-by-value semantics evaluates the non-valuable variable `X` at each functor application, so the body of the linking module will not be deemed valuable. If the less restrictive typing rule is used in conjunction with a memoizing semantics, then the linking module will typecheck, but the references to `X` will raise an "undefined" exception when the module is evaluated at run time. One of the main contributions of the type-theoretic approach to recursive modules that I propose in Section 3.3 is that it offers a clean, simple solution to this dilemma.

3.2 Previous Work on Type Systems for Recursive Modules

Crary *et al.* [3] and Russo [40] have both given accounts of recursive SML-style modules. Crary *et al.* were the first to offer a foundational type-theoretic analysis of the problem and to identify many of the recursive module issues I described in the previous section. In contrast, Russo presents a formalization of the recursive module extension that he implemented in the Moscow ML compiler [33]. Although rather *ad hoc*, his extension provides an important practical perspective. In this section I will describe how both these accounts fit in to the design space I have laid out so far.

[8] In this particular example, abstraction/transparency is not an issue as σ has no type components.

Crary *et al.*'s type system for recursive modules employs an explicit phase-distinction formalism in the style of Harper *et al.* [16]. Such a formalism exposes, through the equivalence judgments of the type system, the implementation of module constructs in terms of more primitive term- and constructor-level constructs. In particular, every signature in their system is equivalent to a phase-split signature of the form $[\![\alpha{:}\kappa.\tau]\!]$, where κ is a kind, and every module is equivalent to a phase-split module of the form $[\alpha = c, e]$, where c is a type constructor. Thus, for instance, rds's with static-on-static dependencies are seen as equivalent to phase-split signatures that involve equi-recursive constructors. There is not a major difference between the explicitly phase-split approach and the approach of my higher-order module type system, which exploits phase separation implicitly in the static module equivalence judgment. However, signature equivalence/subtyping is somewhat more complex in the explicitly phase-split approach, as it is not syntax-directed.

Of the two typing rules for fixed-point modules that I presented in Section 3.1.1, Crary *et al.* use the rule that requires the body of a fixed-point module to be valuable. They point out the problem with opacity in forward declarations and suggest fully transparent fixed-point modules as the solution, although the order in which they motivate the ideas is somewhat different from my presentation above. Specifically, they use the problem with opacity to motivate rds's with *static-on-static* dependencies, which they in turn argue must be fully transparent in order to be phase-split properly. As I discussed in Section 3.1.2, rds's are a useful construct in their own right, independent of the opacity problem, and in the absence of static-on-static dependencies they do not need to be fully transparent. Lastly, Crary *et al.* propose the use of functors for separate compilation. In fact, their proposal does not work, for the example they give is very similar to the one I gave in Section 3.1.4 and thus does not typecheck. Their system therefore does not support separate compilation of recursive modules.

Russo formalizes a recursive module extension to SML in the style of his thesis [38], which in turn is a variation on the style of the Definition [28]. Russo's approach is not really a type system so much as a description of typechecking that employs a semantic object language to express types and signatures that cannot be written down in SML itself. His formalization depends heavily on the complex definition of signature matching given in his thesis, which corresponds to the notion of *coercive* signature matching that is handled in my framework by the elaborator. It is therefore difficult to translate his typing rules directly into my system, but I will attempt a close verbal approximation.

Russo extends SML with both a fixed-point and rds construct. The rds construct is fairly similar to rds's as I have described them, restricted to dynamic-on-static dependencies. In fact, Russo's are slightly more permissive, allowing static-on-static dependencies so long as they are ultimately acyclic and hence do not introduce equi-recursive type equivalences. His rds construct has the form $\rho s{:}\sigma_1.\sigma_2$, where σ_1 serves as the forward declaration of s during the typechecking of σ_2. The typing rule also checks that σ_2 (coercively) matches σ_1, as σ_2 must contain at least the components it expects itself to contain.

Russo employs the memoizing semantics for fixed-point modules that I sketched in Section 3.1.1. His typing rule for fixed-points, though, is different from either of the ones I described. While he does not require the body M of a fixed-point $\mathsf{fix}(s:\sigma.\, M)$ to be valuable, he does require that the signature of M match the signature $\mathfrak{S}_\sigma(s)$. Like requiring σ to be fully transparent, this restriction forces the type components in the forward declaration σ to coincide with their definitions in M, circumventing the problems I enumerated in Section 3.1.3. The only benefit of Russo's rule over the full transparency requirement is that it allows `datatype` specs to appear in a forward declaration, so long as they are *copied* into the body using SML's `datatype` copying primitive. (I will discuss the problem of `datatype`'s further in Section 4.2, along with an explanation of why Russo's solution is infeasible under the Harper-Stone interpretation of `datatype`'s.) Aside from `datatype`'s, however, abstract type specifications in σ are useless: if s`.t` is abstract, then M`.t` must be defined to be s`.t`, in which case `t` never gets to be defined!

A more significant benefit of Russo's fixed-point construct is that, while the body M of $\mathsf{fix}(s:\sigma.\, M)$ must match the fully transparent signature $\mathfrak{S}_\sigma(s)$, the signature of the fixed-point is not σ but rather the principal signature of M. In other words, σ serves as a forward declaration of components that M must provide *at a minimum*, but M may export other components as well, including abstract type components.

Finally, with regard to separate compilation, Russo recognizes precisely the problem that I have described. In the case that all the value components of the forward declaration are functions, he suggests that the linking module can first define a substructure that defines eta-expansions of all the forward-declared functions, and then use that structure in place of the recursive module variable. Given the limited applicability of this trick, as well as its inherent distastefulness, I do not consider it to be a solution.

3.3 A Type-Theoretic Account of Recursive Modules

In this section I will present my proposal for how to incorporate recursive modules into the type-theoretic framework for modules that I developed in Section 2. (Note that, unlike the formal development in that section, the extensions I present in this section have not been fully worked out yet, and I have only sketched out their implications in the meta-theory.) My proposal can be viewed as an attempt to account for some of the practical design decisions of Russo's recursive module extension in a more hygienic way. In the process, I expose the fact that separate compilation of recursive modules is not a problem.

3.3.1 Recursively Dependent Signatures

I propose to handle recursively dependent signatures purely through mechanisms of elaboration. In other words, I will extend the external language to contain an rds construct, and I will extend the elaborator in order to interpret external rds's correctly, but I will not add any rds construct to the HIL. My extensions to the elaborator are nevertheless based closely on the rds introduction and elimination rules. Russo provides some precedent for my approach—he defines his recursive module extension without any extensions to the language of semantic objects, which correspond roughly to HIL types and signatures in my framework.

First let us introduce a new external language signature $\rho s{:}\hat{\sigma}_1.\hat{\sigma}_2$, which elaborates to the corresponding elaborator-level signature $\rho s{:}\sigma_1.\sigma_2$. Note that I am using a Russo-style rds construct in which the signature of s is made explicit. The elaborator rds $\rho s{:}\sigma_1.\sigma_2$ is represented in the HIL simply as $\Sigma s{:}\overline{\sigma_1}.\overline{\sigma_2}$. I described in Section 3.1.2 how dynamic-on-static rds's could be encoded in terms of ordinary hierarchical signatures. For example, `EXPDEC`, which had the form $\rho\,\mathtt{X}.\,\sigma$, was encoded hierarchically in Figure 10 as $\Sigma\,\mathtt{X}{:}\sigma_1.\sigma_2$, where σ_1 specified the type components `Exp.exp` and `Dec.dec` that were recursively referenced in σ, and σ_2 was merely σ with its `Exp.exp` and `Dec.dec` components defined to equal the corresponding components in `X`. To understand my elaborator rds's, the essential observation to make is that by forcing the programmer to write the forward declaration σ_1 in an rds, we are already forcing them to write rds's hierarchically. For example, the programmer would write `EXPDEC` in my system directly as $\rho\,\mathtt{X}{:}\sigma_1.\sigma_2$.

The elaboration rule for rds's is fairly straightforward:

$$\frac{\Gamma \vdash \hat{\sigma}_1 \rightsquigarrow \sigma_1 \quad \Gamma, s{:}\sigma_1 \vdash \hat{\sigma}_2 \rightsquigarrow \sigma_2 \quad \Gamma, s{:}\sigma_1, t{:}\sigma_2 \vdash t : \sigma_2 \preceq \mathfrak{S}_{\sigma_1}(s) \rightsquigarrow M}{\Gamma \vdash \rho s{:}\hat{\sigma}_1.\hat{\sigma}_2 \rightsquigarrow \rho s{:}\sigma_1.\sigma_2}$$

The interesting part is the third premise, which forces any type components that are forward-declared in σ_1 to coincide with the corresponding components in σ_2. This rules out true static-on-static dependencies. For instance, suppose the `t` component in σ_2 were specified as `type t = int` $\times$ s`.t`. Since `t` is required to equal its forward declaration s`.t`, the latter would have to be specified in σ_1 in such a way that s`.t = int` $\times$ s`.t`, which is not possible in ML, *i.e.,* in the absence of equi-recursive types.

As rds's do not have explicit introduction and elimination constructs, the introduction and elimination rules for elaborator rds's take the form of signature coercion rules. The elimination rule for rds's peels off the rds's forward declaration, in a manner identical to the elimination rule for existentials. This is not surprising considering that rds's and existentials have the same underlying HIL representation.

$$\frac{\pi_2 M : \sigma_2[\pi_1 M/s] \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'}{M : \rho s{:}\sigma_1.\sigma_2 \stackrel{\text{peel}}{\Longrightarrow} M' : \sigma'}$$

Note that we unpeel $\rho s{:}\sigma_1.\sigma_2$ to $\sigma_2[\pi_1 M/s]$, not $\sigma_2[M/s]$ as the rds elimination rule of Section 3.1.2 dictates, because it is $\pi_1 M$ that contains the forward-declared type components. Adding this rule means that the signature coercion right rules can expect, when coercing σ_A to σ_B, that σ_A is not an rds.

The introduction rule for rds's takes the form of a signature coercion right rule:

$$\frac{\Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \sigma_1 \rightsquigarrow N_1 \quad \Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \sigma_2[N_1/s] \rightsquigarrow N_2}{\Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \rho s{:}\sigma_1.\sigma_2 \rightsquigarrow \langle N_1, N_2 \rangle}$$

To match M against $\rho s{:}\sigma_1.\sigma_2$, the rule requires that M match $\sigma_2[N_1/s]$, where N_1 is just M coerced into the right shape, *i.e.,* σ_1. This rule is morally the same as the rds introduction rule of Section 3.1.2, modulo the added complications of signature matching.

There are several reasons for handling rds's in the way I have. First, as a practical matter, it is much simpler to use a Russo-style rds construct of the form $\rho s{:}\sigma_1.\sigma_2$, where the forward declaration σ_1 is made explicit. The reader may have noticed that I avoided giving a well-formedness rule for rds's of the form $\rho s.\,\sigma$ when I first discussed them in Section 3.1.2. The reason was that the well-formedness rule given by Crary *et al.* is rather complex, depends heavily on the explicitly phase-split formalism, and essentially forces the typechecker to nondeterministically guess the right signature to assign s when typechecking σ. It may be possible for the elaborator to guess that signature as part of type inference, but I will not study that possibility further in this proposal.

Unfortunately, the Russo-style rds does not have the desired semantics when added to the HIL type theory. In particular, for the rds $\rho s{:}\sigma_1.\sigma_2$ to be well-formed, the signature σ_2 should *coercively* match σ_1, so that σ_1 only needs to specify the type components of s that σ_2 refers to. Coercive signature matching, however, is provided only by the elaborator, not by the HIL type system. One could argue that this is not a problem. For instance, suppose that the HIL rds rule required that σ_2 non-coercively match σ_1, *i.e.,* that $\sigma_2 \leq \sigma_1$. The elaborator might transform an external rds $\rho s{:}\sigma_1.\sigma_2$ into a valid HIL rds by "filling in" σ_1 structurally with dummy components (of signature Top), in order to make it a supersignature of σ_2. While this might work, it introduces an elaboration hack that is much less straightforward than the interpretation of rds's I have proposed.

Another pragmatic argument is that it is not clear how the introduction and elimination rules for rds's that I gave in Section 3.1.2 interact with the HIL typechecking algorithm, since they are non-syntax-directed retyping rules. The rds rules turn out to be admissible in the explicitly phase-split formalism but, as I have mentioned, that formalism introduces other complexities. In short, while the rds rules express intuitively the semantics of rds's, they would constitute an awkward and possibly ill-behaved extension to the type theory.

3.3.2 Fixed-Point Modules

I propose to extend my type theory with fixed-point modules, with the goal of encompassing as many points in the design space as possible. My fixed-point typing rule is as follows:

$$\frac{\Gamma \vdash \sigma \equiv \mathfrak{S}_{\sigma'}(M') \quad \Gamma, s \uparrow \sigma \vdash_\kappa M \downarrow \sigma_M \quad \Gamma \vdash \sigma_M \leq \sigma}{\Gamma \vdash_\kappa \mathsf{fix}(s:\sigma.\,M) \downarrow \sigma_M}$$

The first premise checks that the forward declaration signature σ is equivalent to some singleton signature, *i.e.,* that it is *fully transparent.* As I have pointed out already, this restriction seems to be unavoidable, although in Section 4.2 I will discuss possible ways to incorporate `datatype` specs into the forward declaration.

The second premise checks that the body M is valuable in a context where the recursive module variable s is not. Thus, I have chosen to employ the valuability semantics for fixed-points. Although I will not flesh out the definition of valuability in this proposal, I expect it to be similar to the definition of valuability that is already present in the Harper-Stone HIL to model the SML value restriction. As for the alternative memoizing semantics of fixed-point modules, I will show shortly how it can be encoded as a stylized use of the present rule in conjunction with a new type of *lazy modules.*

The third premise checks that the signature σ_M of M is a subtype of the forward declaration signature σ. The stronger signature σ_M is then returned as the signature of the fixed-point itself, with the intention that σ_M may export more components than are forward-declared in σ, including abstract type components. At first it may not be obvious how σ_M can contain abstract type components, since the type components of its supertype σ are all transparent. The trick is to use the Top signature.[9] For instance, suppose that $\sigma = \mathsf{Top} \times \sigma_2$ and $\sigma_M = \sigma_1' \times \sigma_2'$. While σ_2' must be a subtype of σ_2, σ_1' can be anything, and in particular may specify abstract types. Note that it is fine for Top to appear in σ because Top is trivially transparent, *i.e.,* $\mathsf{Top} = \mathfrak{S}_{\mathsf{Top}}(M)$ for any M.

One aspect of my fixed-point typing rule in particular may give cause for alarm, namely that the signature of the fixed-point body may not contain references to the recursive module variable. Thus, to synthesize the principal signature of a fixed-point, one must synthesize the principal signature σ_M of the body, and then find the least supersignature of σ_M avoiding the recursive module variable s. This would appear to incur precisely the avoidance problem! Fortunately, I believe this to be a solvable instance of the avoidance problem. Since

[9] In the interest of full disclosure, this is the reason I included the Top signature in my type system in the first place.

$$\frac{\Gamma \vdash_\kappa M : \sigma \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \mathsf{delay}(M) \downarrow \mathsf{comp}(\sigma)}\ (16) \qquad \frac{\Gamma \vdash_\kappa M : \mathsf{comp}(\sigma)}{\Gamma \vdash_\kappa \mathsf{force}(M) : \sigma}\ (17) \qquad \frac{\Gamma \vdash_\kappa M \downarrow \sigma}{\Gamma \vdash_\kappa M : \sigma}\ (18)$$

$$\frac{\Gamma \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash \mathsf{comp}(\sigma_1) \leq \mathsf{comp}(\sigma_2)}\ (19) \qquad \frac{\Gamma \vdash_{\mathtt{P}} \mathsf{force}(M) : \sigma}{\Gamma \vdash_{\mathtt{P}} M : \mathsf{comp}(\sigma)}\ (20) \qquad \frac{\Gamma \vdash \mathsf{force}(M_1) \cong \mathsf{force}(M_2) : \sigma}{\Gamma \vdash M_1 \cong M_2 : \mathsf{comp}(\sigma)}\ (21)$$

$$\frac{\Gamma \vdash M_1 \cong M_2 : \sigma}{\Gamma \vdash \mathsf{delay}(M_1) \cong \mathsf{delay}(M_2) : \mathsf{comp}(\sigma)}\ (22) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \mathsf{comp}(\sigma)}{\Gamma \vdash \mathsf{force}(M_1) \cong \mathsf{force}(M_2) : \sigma}\ (23)$$

Figure 12: Typing Rules for Lazy Memoized Modules

the avoided variable in question is assigned a fully transparent signature, it should be possible to substitute any references to the type components of s in σ_M with their definitions in the forward declaration signature. Moreover, the resulting signature should be equivalent to σ_M.

3.3.3 Lazy Memoized Modules

By restricting the body to be valuable, my fixed-point construct does not provide any way to write recursive modules with *term-level effects*. By term-level effects I mean effects such as I/O, references and exceptions, rather than the typing effects of abstraction and generativity.

Russo accounts for term-level effects by employing a memoizing semantics, as I described in Section 3.1.1. Inspired by Russo's approach, I propose to incorporate term-level effects into my type theory of recursive modules by adding a new kind of module: a *lazy memoized module*. The signature of a lazy module is $\mathsf{comp}(\sigma)$, which stands for the "computation of a module of signature σ". The introduction form for $\mathsf{comp}(\sigma)$ is $\mathsf{delay}(M)$, which suspends the computation of M (of signature σ). The elimination form for $\mathsf{comp}(\sigma)$ is $\mathsf{force}(M)$, which evaluates M to a value V and then checks whether V is a computation that has been evaluated and memoized yet. If so, it returns the memoized value of type σ; if not, it evaluates the suspended computation and memoizes it.

The point of introducing lazy modules is that we can now encode the Russo-style fixed-point construct (denoted $\mathsf{russofix}(s:\sigma.\, M)$) as a regular fixed-point whose body is a lazy memoized module. Formally, $\mathsf{russofix}(s:\sigma.\, M)$ can be defined as follows:

$$\mathsf{russofix}(s:\sigma.\, M) \stackrel{\text{def}}{=} \mathsf{force}(\mathsf{fix}(s':\mathsf{comp}(\sigma).\, \mathsf{delay}(M[\mathsf{force}(s')/s])))$$

Under this encoding, M may have arbitrary term-level effects, as any such effects are captured by the delay and only unleashed by the force *outside* of the fixed-point. In order to make this work, $\mathsf{delay}(M)$ must be considered valuable regardless of what M is.

The typing rules for lazy modules are given in Figure 12. Rule 16 requires the module M in $\mathsf{delay}(M)$ to be dynamically pure, *i.e.,* non-generative. The reason for this restriction is that if M has dynamic typing effects, then $\mathsf{delay}(M)$ suspends those effects, just as it suspends term-level effects. If the suspension is subsequently forced, the dynamic effects will be released. However, there is nothing about the signature $\mathsf{comp}(\sigma)$ that indicates whether forcing a module of that signature should induce a dynamic effect, and in general we do not want it to. In particular, the encoding of $\mathsf{russofix}$ relies on the ability to substitute $\mathsf{force}(s')$ for s, which is only sensible if $\mathsf{force}(s')$ is pure.

The restriction of Rule 16 has serious implications if we wish to be able to write recursive modules whose bodies contain dynamic effects, such as generative functor applications. I will defer the remedy to this restriction until Section 3.3.5, as it raises other interesting problems. The remaining rules in Figure 12 are completely straightforward, with the exception perhaps of Rules 20 and 21, which provide selfification and extensionality, respectively, at comp signatures. Correspondingly, the higher-order singleton $\mathfrak{S}_{\mathsf{comp}(\sigma)}(M)$ is defined to be $\mathsf{comp}(\mathfrak{S}_\sigma(\mathsf{force}(M)))$.

3.3.4 Handling Separate Compilation With Lazy Modules

An important benefit of separating fixed-point modules from lazy memoized modules is that it solves the problem of separate compilation. Recall the example from Section 3.1.4, in which we wanted to separately compile the two substructures of the module

```
fix (X : σ. struct
  structure A = struct fun f(n) = X.B.g(n-1) end
  structure B = struct fun g(n) = X.A.f(n-1) end
end)
```

First, let us rewrite the module as if it were a Russo-style fixed-point, encoded in terms of lazy modules:

```
force(fix (X : comp(σ). delay(struct
       structure A = struct fun f(n) = force(X).B.g(n-1) end
       structure B = struct fun g(n) = force(X).A.f(n-1) end
      end)))
```

Now, when we functorize `A` and `B` over `X`, we get:

```
functor FunA (X : comp(σ)) = struct fun f(n) = force(X).B.g(n-1) end
functor FunB (X : comp(σ)) = struct fun g(n) = force(X).A.f(n-1) end
```

The linking module is written essentially as it was before:

```
force(fix (X : comp(σ). delay(struct
       structure A = FunA(X)
       structure B = FunB(X)
      end)))
```

The difference here is that the body of the linking module is suspended, so the fixed-point evaluates to a module value V right away. Then, when V is promptly forced, the references to `X` in `FunA(X)` and `FunB(X)` do not raise an undefined exception because, after the fixed-point is unrolled, they are merely references to V, which is a value! Moreover, `FunA(`V`)` and `FunB(`V`)` evaluate without incident because they do not attempt to force V *during the forcing of* V.

Note that the comp signatures play a critical role here. If the functors `FunA` and `FunB` did not specify their arguments to be computations, the invocations of those functors would have had to force `X` to obtain a module of signature σ, resulting in the same divergent behavior as we saw in Section 3.1.4. In essence, the key idea that makes separate compilation work is that the separately compiled functors treat references to the recursive module variable the same way such references are treated *inside* the fixed-point: namely, as references to a memoized computation.

3.3.5 Recursive Modules With Dynamic Effects

Allowing dynamic effects in a lazy module is in fact not difficult. Suppose we forget about memoization and think of $\mathsf{comp}(\sigma)$ as essentially the thunk signature $1 \to \sigma$. Then, $\mathsf{delay}(M)$ is essentially the functor $\lambda\langle\rangle.M$. Consequently, if M has dynamic effects, then $\mathsf{delay}(M)$ is a *generative* functor of signature $1 \xrightarrow{\text{gen}} \sigma$.

Thus, an analogous way to deal with dynamic effects is to add a new signature $\mathsf{comp}^{\text{gen}}(\sigma)$ classifying lazy modules with dynamic effects, with the following introduction and elimination rules:

$$\frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \mathsf{delay}(M) \downarrow \mathsf{comp}^{\text{gen}}(\sigma)} \qquad \frac{\Gamma \vdash_\kappa M : \mathsf{comp}^{\text{gen}}(\sigma)}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} \mathsf{force}(M) : \sigma}$$

(I should point out that I only discovered the need for $\mathsf{comp}^{\text{gen}}(\sigma)$ when I attempted to prove that an earlier version of the typing rules for $\mathsf{comp}(\sigma)$ was sound, and discovered a subtle counterexample to soundness instead. While I believe the generative distinction eliminates any potential for unsoundness, this needs to be proven by extending the dynamic semantics and type safety proof to account for recursive modules.)

A subtle problem arises, though, if we try to use a $\mathsf{comp}^{\text{gen}}(\sigma)$ in conjunction with the russofix encoding. In particular, if the body M of $\mathsf{russofix}(s : \sigma.\, M)$ contains dynamic effects, then the encoding is not well-typed

because the body $\mathsf{delay}(M[\mathsf{force}(s')/s])$ has signature $\mathsf{comp}^{\mathrm{gen}}(\sigma)$, which is not a subtype of the forward declaration signature $\mathsf{comp}(\sigma)$. On the other hand, we cannot change the forward declaration signature to $\mathsf{comp}^{\mathrm{gen}}(\sigma)$, because then $\mathsf{force}(s')$ would become impure and $\mathsf{delay}(M[\mathsf{force}(s')/s])$ invalid. What is particularly frustrating about this problem is that, although M may have dynamic effects internally, when viewed at the fully transparent signature σ they are invisible. In essence, who cares whether a module generates abstract types at run time if it does not export any of them?

This dilemma points out a small but apparently important failing of my type theory of modules, namely that an impure module with a fully transparent signature is still irrevocably deemed impure. Indeed, this issue is the one point regarding which Shao's type system for higher-order modules [42], which I described in Section 2.2, has the upper hand. In his type system transparency *is* purity, and opacity *is* generativity. In the context of higher-order modules, it doesn't seem particularly important, for instance, whether a module sealed with a fully transparent signature is seen as pure or not. The module's signature reveals all there is to know about it regardless, so purity does not seem to achieve anything.

To allow dynamic effects in recursive modules, however, we appear to need a way of observing that a fully transparent module can only have *benign effects*, so it is for all intents and purposes pure. If the type system were to observe this fact automatically, the purity of a module would depend on the signature at which it is viewed. For example, every module would be considered pure at signature Top, which would falsify the claim that the signature synthesis algorithm outputs the minimal purity level of a module. Instead, I propose to introduce a new module construct $\mathsf{purify}_\sigma(M)$, which coerces M to purity at the fully transparent signature σ:

$$\frac{\Gamma \vdash \sigma \equiv \mathfrak{S}_{\sigma'}(M') \quad \Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{P}} \mathsf{purify}_\sigma(M) : \sigma}$$

The encoding of $\mathsf{russofix}(s:\sigma.\,M)$ can now be mended to handle a dynamically effectful M by purifying M at the forward declaration signature:

$$\mathsf{russofix2}(s:\sigma.\,M) \stackrel{\mathrm{def}}{=} \mathsf{force}(\mathsf{fix}(s':\mathsf{comp}(\sigma).\,\mathsf{delay}(\underline{\mathsf{purify}_\sigma}(M[\mathsf{force}(s')/s]))))$$

3.3.6 Elaboration of Recursive Modules

While clean and concise, the revised $\mathsf{russofix2}$ encoding of the previous section is a bit oversimplistic. In particular, by purifying M at the forward declaration signature, we lose the ability to export other, possibly abstract, components of M. Moreover, the $\mathsf{russofix2}$ encoding does not utilize the new $\mathsf{comp}^{\mathrm{gen}}$ signature. In this section I illustrate how elaboration techniques and the $\mathsf{comp}^{\mathrm{gen}}$ signature can be used to encode a more complete account of Russo's memoizing semantics in the form of an external language construct $\mathsf{rec}(s:\hat{\sigma}.\,\hat{M})$.

The following elaboration rule recovers the ability to export abstract types from a recursive module by translating $\mathsf{rec}(s:\hat{\sigma}.\,\hat{M})$ to a fixed-point over a pair of modules. The first module contains the translation M of $\hat{M}$, which we wish to export but not forward-declare as it may contain abstract types. The second module is a coercion from M to the translation σ of $\hat{\sigma}$, which we wish to forward-declare but not export. We avoid forward declaration of the first component by forward-declaring it with Top, as I suggested doing in Section 3.3.2. We avoid exporting the second component by only projecting out the first component of the fixed-point in the end.

$$\frac{\begin{array}{ccc}\Gamma \vdash \hat{\sigma} \rightsquigarrow \sigma & \Gamma \vdash \sigma \equiv \mathfrak{S}_{\sigma'}(M') & \Gamma, s{:}\sigma \vdash_\kappa \hat{M} \rightsquigarrow M : \sigma_M \\ \Gamma, s{:}\sigma \vdash \sigma_M \equiv \sigma'_M & \Gamma \vdash \sigma'_M \ \mathrm{sig} & \Gamma, u{:}\sigma'_M \vdash u : \sigma'_M \preceq \sigma \rightsquigarrow N\end{array}}{\begin{array}{l}\Gamma \vdash_\kappa \mathsf{rec}(s:\hat{\sigma}.\,\hat{M}) \rightsquigarrow \\ \quad \mathsf{let}\ s = \mathsf{fix}(r:\mathsf{Top}\times\mathsf{comp}(\sigma). \\ \qquad \langle t = \mathsf{delay}(M[\mathsf{force}(\pi_2 r)/s]), \\ \qquad\ \mathsf{delay}(\mathsf{purify}_\sigma(\mathsf{let}\ u = \mathsf{force}(t)\ \mathsf{in}\ (N:\sigma)))\rangle) \\ \quad \mathsf{in}\ (\mathsf{force}(\pi_1 s) : \sigma'_M) : \sigma'_M\end{array}}$$

Note that the $\mathsf{comp}^{\mathrm{gen}}$ signature is needed in order to classify the first component of the pair, in the case that M has dynamic effects. Also note that the fourth and fifth premises magically invent a signature σ'_M that is equivalent to σ_M but does not mention s. As I argued in Section 3.3.2, I believe this to be a solvable instance of the avoidance problem because s's signature σ is fully transparent.

In order to allow the programmer to separately compile recursive modules using the technique from Section 3.3.4, the external language must also be extended with lazy signatures $\mathsf{comp}(\hat{\sigma})$, whose elaboration rule is extremely simple:

$$\frac{\Gamma \vdash \hat{\sigma} \rightsquigarrow \sigma}{\Gamma \vdash \mathsf{comp}(\hat{\sigma}) \rightsquigarrow \mathsf{comp}(\sigma)}$$

The next question is obviously whether or not to include delay and force in the external language as well. For the purpose of separate compilation of recursive modules, it is not necessary to include them, since they are used in a highly idiomatic way that the elaborator can infer. In particular, as with rds's, the introduction and elimination of comp signatures can be performed as part of signature coercion. Whenever a lazy module is referred to, it is immediately forced; whenever a module is coerced to a lazy signature, it is suspended.

$$\frac{\Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \sigma' \rightsquigarrow M'}{\Gamma \vdash M : \sigma \preceq_{\mathrm{R}} \mathsf{comp}(\sigma') \rightsquigarrow \mathsf{delay}(M')} \qquad \frac{\mathsf{force}(M) : \sigma \overset{\mathrm{peel}}{\Longrightarrow} M' : \sigma'}{M : \mathsf{comp}(\sigma) \overset{\mathrm{peel}}{\Longrightarrow} M' : \sigma'}$$

The running separate compilation example can now be written in the external language as follows:

```
functor FunA (X : comp(σ)) = struct fun f(n) = X.B.g(n-1) end
functor FunB (X : comp(σ)) = struct fun g(n) = X.A.f(n-1) end
structure Linker = rec (X : σ. struct
                     structure A = FunA(X)
                     structure B = FunB(X)
                   end)
```

The peeling rule for $\mathsf{comp}(\sigma)$ elaborates the references to `X` in the bodies of `FunA` and `FunB` to $\mathsf{force}(\mathtt{X})$. The signature coercion right rule has the dual behavior, elaborating the references to `X` in the functor applications of the linking module to $\mathsf{delay}(\mathtt{X})$. Finally, the elaboration rule for recursive modules changes the forward declaration of the linking fixed-point to $\mathsf{comp}(\sigma)$ and substitutes $\mathsf{force}(\mathtt{X})$ for `X` in the body. So in the end, the occurrences of `X` in the linking module elaborate to $\mathsf{delay}(\mathsf{force}(\mathtt{X}))$, which is just the eta-expansion of `X`. Thus, elaboration is capable of producing extensionally the same code as the hand-coded version from Section 3.3.4, and in practice eta-redices like $\mathsf{delay}(\mathsf{force}(\mathtt{X}))$ can likely be eliminated as an optimization.

4 Directions for Thesis Work

The type-theoretic approach to modules that I have presented in Sections 2 and 3 will form the foundation of my thesis. The bulk of my thesis work will involve scaling my extensions to the level of the full SML language and implementing them. This final section is devoted to enumerating several related directions I intend to explore as well. Sections 4.1 through 4.3 discuss extensions and revisions to my type theory of modules. Sections 4.4 and 4.5 discuss the language features of *views* and *type classes*, both of which are relevant to modularity, and what may be involved in adding them to SML. Finally, Section 4.6 discusses some of the problems with the SML language and the TILT elaborator that I would like to resolve during the design and implementation of my language extensions.

4.1 A Monadic Approach to Generativity

In Section 2.5 I described generativity as a computational effect, encapsulated by functors and released upon their applications. My type system for modules tracks this computational effect by using different signatures (Π and Π^{gen}) to distinguish between functors with pure and impure bodies. This approach illustrates a beautiful connection between the applicative/generative distinction and the way effects are tracked in traditional *effect systems* [13, 46].

An alternative approach, originally proposed by Moggi [31], is to use *monadic types* to classify effectful computations. In the context of the module type theory, this would correspond to the addition of a new signature $\mathsf{gen}(\sigma)$, classifying dynamically effectful modules of signature σ. One could give this new generativity

monad the following introduction and elimination rules:

$$\frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{P}} \mathsf{gen}(M) : \mathsf{gen}(\sigma)} \qquad \frac{\Gamma \vdash_\kappa M : \mathsf{gen}(\sigma)}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} \mathsf{ungen}(M) : \sigma}$$

An impure module M may be packaged as a pure module $\mathsf{gen}(M)$ with a monadic signature. In order to use a module M of monadic signature, however, M must be ungen'd, unleashing a dynamic effect.

By providing a way for a dynamically effectful module to be packaged as a pure module, the monadic approach eliminates the need for generative functor signatures. In particular, the generative functor signature $\Pi^{\mathsf{gen}} s{:}\sigma_1.\sigma_2$ can be encoded as the (applicative) functor signature $\Pi s{:}\sigma_1.\mathsf{gen}(\sigma_2)$. Correspondingly, a generative functor $\lambda s{:}\sigma.M$, *i.e.,* where M is impure, can be encoded as the (applicative) functor $\lambda s{:}\sigma.\mathsf{gen}(M)$, and the application FM, where F is generative, can be encoded as $\mathsf{ungen}(FM)$. To preserve the functor subtyping of my original system, it is also necessary to add the following two subtyping rules for monadic signatures:

$$\frac{\Gamma \vdash \sigma_1 \leq \sigma_2}{\Gamma \vdash \mathsf{gen}(\sigma_1) \leq \mathsf{gen}(\sigma_2)} \qquad \frac{\Gamma \vdash \sigma \ \mathsf{sig}}{\Gamma \vdash \sigma \leq \mathsf{gen}(\sigma)}$$

There is a strong connection between the generative monadic *signature* $\mathsf{gen}(\sigma)$ and the package *type* $\langle\!|\sigma|\!\rangle$. Monads and packages, however, offer complementary benefits. While packages allow modules to be passed as first-class values, monadic signatures support subtyping, which is only available at the module level.

The monadic approach has several advantages over the effect-system approach in the context of modules. First, functors are not forced to be completely applicative or generative. Rather, some type components in the result signature of a functor may be specified as generative by wrapping them in monadic signatures, while others may remain applicative. This is potentially useful from a programming perspective, as it allows for fine-grained control over type generativity.

The monadic approach also turns out to simplify the module type theory by obviating the point S in the purity lattice. Recall that S corresponds to the classification of a module as statically pure but dynamically effectful. Since static effects correspond to type abstraction, and dynamic effects to generativity, it seems very strange to classify a module as generative but not abstract. In truth, the sole reason for including S is in order to allow generative functors with no sealing to be deemed pure, which in turn is important in order to preserve the invariants of elaboration. Specifically, in the case of the elaboration rule for coercion of functor signatures (Rule 10 in Figure 8), the invariants require that the output of the coercion rule, $\lambda s{:}\sigma_1'.\,\mathsf{let}\ t = FM\ \mathsf{in}\ (N : \sigma_2')$, is always pure, since the output of coercion rules may be fed back in as input. However, if F has a generative signature, then for the output to be pure, it is necessary to classify FM as S, not W.

With monads, this niggling excuse for the S classification falls by the wayside. In particular, any functor F with a generative signature, which under the monadic encoding means $\Pi s{:}\sigma_1.\mathsf{gen}(\sigma_2)$, can be eta-expanded into a pure functor $\lambda s{:}\sigma_1.\mathsf{gen}(\mathsf{ungen}(F\,s))$. Thus, the result of generative functor coercions can always be eta-expanded into a pure functor in the monadic system without needing S. The upshot, then, is that monads allow us to simplify the purity lattice to the more intuitive and catchy

where A stands for "non-generative, but possibly Abstract" and G stands for "possibly Generative". In such a lattice, type generativity clearly implies type abstraction.

Lastly, just as the monadic approach eliminates the need for the Π^{gen} signature, the distinction between $\mathsf{comp}(\sigma)$ and $\mathsf{comp}^{\mathsf{gen}}(\sigma)$ in the recursive module calculus should be rendered unnecessary as well, under the encoding of $\mathsf{comp}^{\mathsf{gen}}(\sigma)$ as $\mathsf{comp}(\mathsf{gen}(\sigma))$. Moreover, it is worth noting that the signature $\mathsf{comp}(\sigma)$ is a form of monadic signature itself as well, capturing the type of computations that may involve term-level effects. Monads and effects thus seem to be important concepts in the type theory of modules. I chose not to employ the monadic approach earlier in this proposal, as I have not yet worked out the full monadic type system

and its meta-theory, and I have not decided to what extent the generativity monad should be incorporated into the external language design.

4.2 Forward Declarations of Datatypes

A deficiency of my proposal for recursive modules is that it does not allow `datatype`'s to be specified in the forward declaration of a fixed-point. As I mentioned in Section 3.2, Russo's Moscow ML extension allows `datatype` specs in forward declarations, so long as they are copied into the recursive module body using the SML `datatype` copying primitive. From the perspective of the Harper-Stone framework, it is not clear how to implement Russo's semantics because those `datatype`'s specified in the forward declaration of a recursive module never actually get defined anywhere. The reason this is not a problem for Russo is that his Definition-based formalism only synthesizes semantic objects (*i.e.,* internal-language types) for external-language modules—it elaborates for the purpose of typechecking, but does not produce any HIL code as HS does. Nevertheless, the uselessness of forward-declaring any other kind of opaque type but a `datatype` indicates that the ability to forward-declare `datatype`'s is little more than a special-case loophole in otherwise fully transparent forward declarations. Still, it is an important loophole.

One way to translate Russo's loophole to our setting might be to view the `datatype`'s specified in a forward declaration as being defined *before* the fixed-point itself and subsequently copied into the forward declaration. For instance, to simulate `datatype t = <rhs>` in a forward declaration, first define `datatype prefix_t = <rhs>[prefix_t/t]`, and then replace the specification of `t` in the forward declaration with `datatype t = datatype prefix_t`. This has the effect of rendering the forward declaration transparent again, as required by my proposed fixed-point construct of Section 3.3.2.

The details of this encoding clearly need to be worked out, especially with regard to separate compilation. However, it should be noted that the principle behind the encoding is applicable not just to `datatype` specs, but to any *canonically implemented* specs. In other words, it is reasonable to specify types opaquely in a forward declaration if the elaborator knows a canonical way to define them outside of the recursive module and copy them into the forward declaration. It so happens that the only such canonically implemented specifications in SML are `datatype` specs.

4.3 Valuability vs. Evaluability

The purpose of the valuability restriction on fixed-point module bodies is to ensure that the recursive module variable is never accessed during the evaluation of the body. Valuability is sufficient, but it is not necessary. In particular, it prohibits side effects in the body that may have nothing to do with accessing the recursive module variable. For example, there should be nothing wrong with defining a mutable "flag" component of a recursive module to be `ref true`, but valuability disallows it. The flag is permitted under the Russo-style semantics of fixed-points, but at the expense of a memoization check at each reference to the recursive module variable.

A more appropriate restriction for fixed-point module bodies would be what I call *evaluability*, whose only purpose is to ensure that *undefined* (or *non-evaluable*) variables never get accessed during the evaluation of the module body. I expect the judgment for evaluability to be axiomatized similarly to valuability, save the allowance for side-effecting primitive operations like `ref`, but again the details must be worked out.

If we only wish to provide the Russo-style fixed-point in the external language, then it is somewhat moot whether we choose valuability or evaluability as the internal restriction on fixed-points. If, however, we want to allow the programmer to use the more efficient internal fixed-point construct directly, evaluability would provide considerably greater flexibility. In addition, it is worth noting that while my encoding of Russo-style recursive modules relies on memoized suspensions being considered valuable, they really are not. The evaluation of $\mathsf{delay}(M)$ allocates a new memoized cell, thus producing distinct values each time $\mathsf{delay}(M)$ is evaluated. Although I believe it is perfectly sound (for purposes of type safety) to consider $\mathsf{delay}(M)$ valuable, evaluability is the more appropriate notion.

4.4 Views

While an SML `datatype` declaration generates an abstract type, its underlying implementation is known to be isomorphic to a particular recursive sum type. The "concrete" nature of `datatype`'s has the advantage that constructing values of a `datatype` and pattern matching against them can be compiled efficiently.[10] On the other hand, the programming convenience benefits of pattern matching are tied to the `datatype` construct, with the consequence that one cannot pattern match against a value of an arbitrary abstract type.

To remedy this, Wadler [49] proposed the idea of *views*, which allow the programmer to write a non-standard implementation of a `datatype`. A view consists of two transformation functions: one from the `datatype` to the actual implementation type, which is called at applications of the datatype's constructors, and one in the other direction, which is called during pattern matching. By allowing one to take advantage of pattern matching while preserving data abstraction, views are closely tied to my thesis' theme of modularity.

Okasaki [35] later described, as an extension to SML, a variant of Wadler's views in which only the latter transformation is required. His simplification is motivated by the observation that the constructors that are appropriate for constructing values of an arbitrary type are not necessarily the same ones appropriate for destructing it. Okasaki also proposes ways to deal with the interaction of view transformations and effects. Essentially he suggests that, in order to avoid problems with redundant or nonexhaustive pattern matches, a view transformation should only be applied *once* during a pattern match and the result memoized.

Okasaki's extension is specified somewhat informally and treats views as derived forms. As part of my thesis work I propose to adapt his approach to the Harper-Stone framework, in the process exposing views to be special kinds of user-definable modules whose interfaces are recognized by the pattern compiler.

4.5 Type Classes

Polymorphism in SML is *parametric*, in that a polymorphic function has the same behavior for all instantiations of its type variables. In contrast, SML provides very little support for *ad hoc polymorphism* (or *overloading*), which refers to the ability to define a function at multiple types, with different implementations at different types. The only instances of overloading in SML are some of the arithmetic operators, which are applicable to both `int` and `real` arguments, and the equality function, `=`, which is defined only for so-called *equality types*.

As a way of generalizing SML's equality types to something less *ad hoc*, Wadler and Blott [50] proposed the idea of *type classes*. A type class defines a set of functions that must be implemented at any type belonging to the class. For example, the type class `Eq` could be defined to contain types `t` for which the function `=` of type `t * t -> bool` is implemented. Continuing the example, a function that uses the equality function on its input argument, but does not otherwise place any restrictions on the type of that argument, will be given a polymorphic type where the type variable classifying the input argument is required to belong to the `Eq` class. In addition to enhancing the flexibility of polymorphism, type classes provide a form of extensibility. One may declare a new type to be an *instance* of an old class by providing implementations of the overloaded functions associated with that class at the new type. Polymorphic code that makes use of those overloaded functions can then be instantiated at the new type and reused without modification.

Type classes are one of the most notable and successful features of the language Haskell [1]. In Haskell, whose module system is restricted to namespace management, type classes provide some of the reusability benefits of functors. The notion of *context reduction* in type class systems corresponds to automatic applications of functors with the default arguments supplied by instance declarations. While the default instantiation of functors (*aka dictionary passing*) implicit in the implementation of type classes allows for convenient programming idioms, it lacks the flexibility of SML functors. Recognizing this, Kahl and Scheffczyk [20] have recently proposed extending Haskell with *named instances* of type classes, so that one may explicitly indicate the instance to be used when satisfying the type class constraints of a polymorphic type variable.

While well-intentioned, Kahl and Scheffczyk are essentially backpatching a limited form of functors onto Haskell, which is I think approaching the problem from the wrong direction. Rather, I believe the more systematic approach would be to start with a full-featured module system and build type classes on top of it as an extension of type inference. From this perspective, a type class is a special form of signature, and

[10] Although doing so in a type-directed compiler without breaking the abstraction of the `datatype` requires some additional type-theoretic machinery [48].

an instance is a special form of module. The numerous variations and extensions of type classes, such as constructor classes and multi-parameter type classes [37], can then be viewed as other, less restricted forms of modules and signatures.

I have uncovered no prior work, however, on adding type classes to a language with a rich module system like SML. A major problem is that it is not clear how to define the scope of class and instance declarations. If we think of type classes as signatures, then class declarations should, like signature definitions, be permitted only at top level. As for instance declarations, however, Haskell skirts the issue by implicitly exporting *all* class instances declared in a module, and by prohibiting overlapping instances of the same type belonging to the same class. Such a solution would not carry over to SML modules, though. Module hierarchies in SML frequently contain several copies of a module as substructures of other modules, so it is not clear what to do if such a module exports instance declarations. One approach might be to allow overlapping instance declarations, accepting that type inference might resolve instance conflicts non-deterministically or according to an essentially *ad hoc* algorithm. As part of my thesis work, I propose to investigate these questions further: at a minimum to clearly formalize what the problems are, and ideally to incorporate some form of type class mechanism into SML as well.

4.6 Revising SML and the TILT Elaborator

My experience with the TILT elaborator so far suggests that implementing my language extensions will point out flaws and omissions in their design and/or formalization. In addition, it will raise issues that are not apparent when designing the semantics, such as time and space efficiency concerns. From working on the TILT elaborator, I am already aware of several problematic points in the design and implementation of SML itself that I plan to resolve as part of my thesis work:

Compilation Units Although most implementations of SML support some form of separate or incremental compilation, neither the Definition of SML nor Harper-Stone specifies what a *compilation unit* is. The TILT developers have found that the semantics of compilation units is rather subtle, and it is difficult to pin down the fine points without a more formal definition of compilation management. I expect formalization to be even more important in the presence of recursive modules and, potentially, recursive units.

Structure Sharing In the original Definition of SML [27], structure sharing constraints guaranteed the property that two substructures were copies of the same original module. Transitioning away from the stamp-based semantics of SML '90, the Revised Definition of SML '97 [28] redefined the *structure sharing* constraint `sharing A = B` to be syntactic sugar for a set of *type sharing* constraints between all the type components of the same name in `A` and `B`. Unfortunately, the way structure sharing is defined in the latter Definition renders the feature all but useless: `sharing A = B` will only be a valid constraint if *all* of the type components of the same name in `A` and `B` are *abstract*, which is rarely the case in practice.

In a note to the SML Implementers mailing list [7], I drafted a more sensible definition of structure sharing that we the TILT developers devised. However, the *implementation* of structure sharing in TILT has proven troublesome as well. In particular, structure sharing constraints cause the size of signatures to blow up. As a stopgap solution, the current implementation makes use of a signature construct `SIGNAT_OF(A)`, which refers to the principal signature of `A`. The `SIGNAT_OF` construct is poorly understood, though, and may not be type-theoretically valid. In addition to finding a sound solution to the practical problems with SML '97 structure sharing, I plan to re-incorporate an SML '90-style structure sharing construct, which would allow one to enforce stronger program coherence properties. It is folklore that the stamp-based semantics for structure sharing can be simulated type-theoretically by including a "hidden" abstract type in every structure definition, but the details of this idea must be worked out.

Signature Bindings Signature bindings in SML are only allowed at top level, and Harper-Stone assumes they are eliminated via a compiler pre-pass. However, from a programmer's point of view, the ability to bind signatures to variables, as well as specialize components of signatures using `where type`, is a critical feature of SML, and enables a form of *interface inheritance*. In addition, extending SML with the ability to write local signature bindings inside structures or signatures may offer a solution to the problems with signature blow-up in the elaboration of structure sharing constraints. Therefore, I intend to give serious thought to the semantics of signature variables and bindings in the process of revising and extending Harper-Stone.

References

[1] *The Haskell 98 Report*, February 1999. Available at `http://www.haskell.org/onlinereport`.

[2] 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.

[3] 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.

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

[5] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *2003 ACM Symposium on Principles of Programming Languages*.

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

[7] Derek Dreyer and Leaf Petersen. Sensible structure sharing for Standard ML. Posted to the `sml-implementers` mailing list, September 2001.

[8] Derek R. Dreyer, Robert Harper, and Karl Crary. Toward a practical type theory for recursive modules. Technical Report CMU–CS–01–112, School of Computer Science, Carnegie Mellon University, March 2001.

[9] Dominic Duggan and Constantinos Sourelis. Parameterized modules, recursive modules, and mixin modules. In *1998 ACM SIGPLAN Workshop on ML*, pages 87–96, Baltimore, Maryland, September 1998.

[10] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4&5):295–357, July 2002.

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

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

[13] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, Cambridge, MA, August 1986.

[14] 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.

[15] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[16] 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.

[17] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[18] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical Report CMU–CS–97–147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[19] 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.

[20] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In *Proceedings of the 2001 Haskell Workshop*.

[21] 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.

[22] Xavier Leroy. A syntactic theory of type generativity and sharing. In John H. Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, number 2265 in Rapport de Recherche, pages 1–12, Orlando, FL, June 1994. INRIA.

[23] 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.

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

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

[26] 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.

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

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

[29] 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.

[30] 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.

[31] Eugenio Moggi. Computational lambda calculus and monads. In *Fourth Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[32] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. Gordon and A. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Newton Institute, Cambridge University Press, 1997.

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

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

[35] Chris Okasaki. Views for Standard ML. In *1998 ACM SIGPLAN Workshop on Standard ML*, pages 14–23, Baltimore, MD, September 1998.

[36] Leaf Petersen, Perry Cheng, Robert Harper, and Chris Stone. Implementing the TILT internal language. Technical Report CMU-CS-00-180, Carnegie Mellon University, December 2000.

[37] Simon Peyton Jones and Mark Jones. Type classes: exploring the design space. In *Proceedings of the 1997 Haskell Workshop*.

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

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

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

[41] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

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

[43] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–129, La Jolla, CA, 1995.

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

[45] 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.

[46] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

[47] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[48] Joseph C. Vanderwaart, Derek Dreyer, Leaf Petersen, Karl Crary, Robert Harper, and Perry Cheng. Typed compilation of recursive datatypes. In *2003 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*.

[49] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[50] Philip Wadler and Stephen Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.

Notes on Appendices

The following appendices flesh out the type theory and elaborator for higher-order modules presented in Section 2. Appendices A and B define the static and dynamic semantics, respectively, of the underlying type theory. Appendix C gives the principal signature synthesis algorithm (modulo deciding type equivalence) for the static semantics of Appendix A. Appendix D defines the elaboration algorithm from the external language into the type theory.

A Static Semantics for Higher-Order Modules

To reduce the number of freshness side-conditions, we adopt the convention that a context may not bind the same variable more than once.

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

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

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

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

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: $\Gamma \vdash e : \tau$

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

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

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

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1\ \mathrm{sig}}\ (20) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!]\ \mathrm{sig}}\ (21) \qquad \frac{\Gamma \vdash \tau\ \mathrm{type}}{\Gamma \vdash [\![\tau]\!]\ \mathrm{sig}}\ (22) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathsf{Top}\ \mathrm{sig}}\ (23)$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M)\ \mathrm{sig}}\ (24) \qquad \frac{\Gamma, s{:}\sigma' \vdash \sigma''\ \mathrm{sig}}{\Gamma \vdash \Pi^{\delta} s{:}\sigma'.\sigma''\ \mathrm{sig}}\ (25) \qquad \frac{\Gamma, s{:}\sigma' \vdash \sigma''\ \mathrm{sig}}{\Gamma \vdash \Sigma s{:}\sigma'.\sigma''\ \mathrm{sig}}\ (26)$$

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \equiv 1}\ (27) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \equiv [\![T]\!]}\ (28) \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \equiv [\![\tau_2]\!]}\ (29) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathsf{Top} \equiv \mathsf{Top}}\ (30)$$

$$\frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \equiv \mathfrak{S}(M_2)}\ (31) \qquad \frac{\Gamma \vdash \sigma_2' \equiv \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Pi^{\delta} s{:}\sigma_1'.\sigma_1'' \equiv \Pi^{\delta} s{:}\sigma_2'.\sigma_2''}\ (32) \qquad \frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \equiv \sigma_2''}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \equiv \Sigma s{:}\sigma_2'.\sigma_2''}\ (33)$$

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

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash 1 \leq 1}\ (34) \qquad \frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash [\![T]\!] \leq [\![T]\!]}\ (35) \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\![\tau_1]\!] \leq [\![\tau_2]\!]}\ (36) \qquad \frac{\Gamma \vdash \sigma\ \mathrm{sig}}{\Gamma \vdash \sigma \leq \mathsf{Top}}\ (37)$$

$$\frac{\Gamma \vdash \sigma_2' \leq \sigma_1' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1''\ \text{sig} \quad \delta_1 \sqsubseteq \delta_2}{\Gamma \vdash \Pi^{\delta_1} s{:}\sigma_1'.\sigma_1'' \leq \Pi^{\delta_2} s{:}\sigma_2'.\sigma_2''} \ (38)$$

$$\frac{\Gamma \vdash \sigma_1' \leq \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash \sigma_1'' \leq \sigma_2'' \quad \Gamma, s{:}\sigma_2' \vdash \sigma_2''\ \text{sig}}{\Gamma \vdash \Sigma s{:}\sigma_1'.\sigma_1'' \leq \Sigma s{:}\sigma_2'.\sigma_2''} \ (39)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M) \leq [\![T]\!]} \ (40) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : [\![T]\!]}{\Gamma \vdash \mathfrak{S}(M_1) \leq \mathfrak{S}(M_2)} \ (41)$$

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

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} s : \Gamma(s)} \ (42) \qquad \frac{\Gamma \vdash \text{ok}}{\Gamma \vdash_{\mathtt{P}} \langle\rangle : 1} \ (43) \qquad \frac{\Gamma \vdash \tau\ \text{type}}{\Gamma \vdash_{\mathtt{P}} [\tau] : [\![T]\!]} \ (44) \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau\ \text{type}}{\Gamma \vdash_{\mathtt{P}} [e : \tau] : [\![\tau]\!]} \ (45)$$

$$\frac{\Gamma, s{:}\sigma' \vdash_\kappa M : \sigma'' \quad \kappa \sqsubseteq \mathtt{D}}{\Gamma \vdash_\kappa \lambda s{:}\sigma'.M : \Pi s{:}\sigma'.\sigma''} \ (46) \qquad \frac{\Gamma, s{:}\sigma' \vdash_\kappa M : \sigma'' \quad \Gamma, s{:}\sigma' \vdash \sigma''\ \text{sig}}{\Gamma \vdash_{\kappa \sqcap \mathtt{D}} \lambda s{:}\sigma'.M : \Pi^{\text{gen}} s{:}\sigma'.\sigma''} \ (47)$$

$$\frac{\Gamma \vdash_\kappa F : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_\kappa FM : \sigma''[M/s]} \ (48) \qquad \frac{\Gamma \vdash_\kappa F : \Pi^{\text{gen}} s{:}\sigma'.\sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \sigma'}{\Gamma \vdash_{\kappa \sqcup \mathtt{S}} FM : \sigma''[M/s]} \ (49)$$

$$\frac{\Gamma \vdash_\kappa M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_\kappa M'' : \sigma''}{\Gamma \vdash_\kappa \langle s = M', M''\rangle : \Sigma s{:}\sigma'.\sigma''} \ (50) \qquad \frac{\Gamma \vdash_\kappa M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_\kappa \pi_1 M : \sigma'} \ (51) \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''[\pi_1 M/s]} \ (52)$$

$$\frac{\Gamma \vdash e : \langle\!|\sigma|\!\rangle}{\Gamma \vdash_{\mathtt{S}} \mathsf{unpack}\ e\ \mathsf{as}\ \sigma : \sigma} \ (53) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\kappa \sqcup \mathtt{D}} (M :: \sigma) : \sigma} \ (54) \qquad \frac{\Gamma \vdash_\kappa M : \sigma}{\Gamma \vdash_{\mathtt{W}} (M :> \sigma) : \sigma} \ (55)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash_{\mathtt{P}} M : \mathfrak{S}(M)} \ (56) \qquad \frac{\Gamma, s{:}\sigma' \vdash_{\mathtt{P}} Ms : \sigma'' \quad \Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\rho}{\Gamma \vdash_{\mathtt{P}} M : \Pi s{:}\sigma'.\sigma''} \ (57) \qquad \frac{\Gamma \vdash_{\mathtt{P}} \pi_1 M : \sigma' \quad \Gamma \vdash_{\mathtt{P}} \pi_2 M : \sigma''}{\Gamma \vdash_{\mathtt{P}} M : \sigma' \times \sigma''} \ (58)$$

$$\frac{\Gamma \vdash_\kappa M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_\kappa M'' : \sigma \quad \Gamma \vdash \sigma\ \text{sig}}{\Gamma \vdash_\kappa \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) : \sigma} \ (59) \qquad \frac{\Gamma \vdash_{\kappa'} M : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma \quad \kappa' \sqsubseteq \kappa}{\Gamma \vdash_\kappa M : \sigma} \ (60)$$

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

$$\frac{\Gamma \vdash_{\mathtt{P}} M : \sigma}{\Gamma \vdash M \cong M : \sigma} \ (61) \qquad \frac{\Gamma \vdash M_2 \cong M_1 : \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma} \ (62) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma \quad \Gamma \vdash M_2 \cong M_3 : \sigma}{\Gamma \vdash M_1 \cong M_3 : \sigma} \ (63)$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash [\tau_1] \cong [\tau_2] : [\![T]\!]} \ (64) \qquad \frac{\Gamma \vdash_{\mathtt{P}} M : [\![T]\!]}{\Gamma \vdash [\mathsf{Typ}\, M] \cong M : [\![T]\!]} \ (65)$$

$$\frac{\Gamma \vdash_{\mathtt{P}} M_1 : \sigma \quad \Gamma \vdash_{\mathtt{P}} M_2 : \sigma \quad \sigma \text{ is of the form 1, } [\![\tau]\!], \mathsf{Top}, \text{ or } \Pi^{\text{gen}} s{:}\sigma_1.\sigma_2}{\Gamma \vdash M_1 \cong M_2 : \sigma} \ (66)$$

$$\frac{\Gamma \vdash \sigma_1' \equiv \sigma_2' \quad \Gamma, s{:}\sigma_1' \vdash M_1 \cong M_2 : \sigma''}{\Gamma \vdash \lambda s{:}\sigma_1'.M_1 \cong \lambda s{:}\sigma_2'.M_2 : \Pi s{:}\sigma_1'.\sigma''} \ (67) \qquad \frac{\Gamma \vdash F_1 \cong F_2 : \Pi s{:}\sigma'.\sigma'' \quad \Gamma \vdash M_1 \cong M_2 : \sigma'}{\Gamma \vdash F_1 M_1 \cong F_2 M_2 : \sigma''[M_1/s]} \ (68)$$

$$\frac{\Gamma \vdash M_1' \cong M_2' : \sigma' \quad \Gamma, s{:}\sigma' \vdash M_1'' \cong M_2'' : \sigma''}{\Gamma \vdash \langle s = M_1', M_1'' \rangle \cong \langle s = M_2', M_2'' \rangle : \Sigma s{:}\sigma'.\sigma''} \ (69)$$

$$\frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma'} \ (70) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \Sigma s{:}\sigma'.\sigma''}{\Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''[\pi_1 M_1/s]} \ (71)$$

$$\frac{\Gamma, s{:}\sigma' \vdash M_1 s \cong M_2 s : \sigma'' \quad \Gamma \vdash_{\mathrm{P}} M_1 : \Pi s{:}\sigma'.\rho_1 \quad \Gamma \vdash_{\mathrm{P}} M_2 : \Pi s{:}\sigma'.\rho_2}{\Gamma \vdash M_1 \cong M_2 : \Pi s{:}\sigma'.\sigma''} \ (72)$$

$$\frac{\Gamma \vdash \pi_1 M_1 \cong \pi_1 M_2 : \sigma' \quad \Gamma \vdash \pi_2 M_1 \cong \pi_2 M_2 : \sigma''}{\Gamma \vdash M_1 \cong M_2 : \sigma' \times \sigma''} \ (73)$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M' : \sigma' \quad \Gamma, s{:}\sigma' \vdash_{\mathrm{P}} M'' : \sigma \quad \Gamma \vdash \sigma \ \mathrm{sig}}{\Gamma \vdash \mathsf{let}\ s = M'\ \mathsf{in}\ (M'' : \sigma) \cong M''[M'/s] : \sigma} \ (74)$$

$$\frac{\Gamma \vdash_{\mathrm{P}} M_1 : \mathfrak{S}(M_2)}{\Gamma \vdash M_1 \cong M_2 : \mathfrak{S}(M_2)} \ (75) \qquad \frac{\Gamma \vdash M_1 \cong M_2 : \sigma' \quad \Gamma \vdash \sigma' \leq \sigma}{\Gamma \vdash M_1 \cong M_2 : \sigma} \ (76)$$