Reprinted from *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* **(ICFP'05)**

An Expressive Language of Signatures

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University

Kathleen Fisher
AT&T Labs Research

Paul Govereau
Division of Engineering and Applied Sciences
Harvard University

Abstract

Current languages allow a programmer to describe an interface only by enumerating its parts, possibly including other interfaces wholesale. Such languages cannot express relationships between interfaces, yet when independently developed software components are combined into a larger system, significant relationships arise.

To address this shortcoming, we define, as a conservative extension of ML, a language for manipulating interfaces. Our language includes operations for adding, renaming, and removing components; for changing the type associated with a value; for making manifest types abstract and vice versa; and for combining interfaces. These operations can express useful relationships among interfaces. We have defined a formal semantics in which an interface denotes a group of four sets; we show how these sets determine a subtyping relation, and we sketch the elaboration of an interface into its denotation.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Design, Languages

Keywords Objective Caml, Standard ML, signatures, interfaces, signature manipulation, programming in the large

1. Introduction

It is convenient—if not quite correct—for a programmer to think of an interface as a *set* of named components, where a component is typically a value or a type. Surprisingly, however, programming languages offer few means of manipulating interfaces in the ways we usually manipulate sets. While some languages make it possible to add components to an existing interface, we know of no language that enables a programmer to remove or change an existing component, or to take the union of two sets of components. Most languages can define an interface only by enumerating all its components.

Despite their limitations, existing interface languages serve many programmers well. These languages can become awkward, however, when software is composed of multiple components over which no single group has control. Even then, provided the components fit together nicely, existing languages will do. The problem—and the need to manipulate interfaces as sets of components—arises when components do not fit together.

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

To take a simple example, there is an inconsistency between the `ORD_SET` interface—called a *signature*—in the Standard ML of New Jersey Library and the `LIST` signature in the Standard ML Basis Library. For years, the only way to add an element to an ordered set was to use an `add` function with the following type:

```
val add : set * item -> set
```

But `add`'s type fits badly with `foldr` in the `LIST` signature:

```
val foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

The difficulty is that the arguments to `add` come in the wrong order, so `add` cannot be passed to `foldr`. This difficulty is easy to work around; for example, we can define a signature `ORD_SET2`, which is just like `ORD_SET` except that `add` has type

```
val add : item * set -> set
```

Regrettably, the signature language of Standard ML cannot define `ORD_SET2` concisely: one must enumerate, by hand, all the components in `ORD_SET2`. And when `ORD_SET` changes, `ORD_SET2` must be updated by hand. This strategy of "clone and modify" leads to difficulties in maintaining software.

The example of the `add` function illustrates a general problem: when two interfaces are closely related, we would like to express that relationship formally in the interface language, rather than write out both interfaces and indicate the relationship informally. For example, we would like to write something like

```
signature ORD_SET2 = ORD_SET as S
  rebinding val S.add : S.item * S.set -> S.set
```

In this paper, we present a language that solves this problem. We make the following contributions:

- Our language expresses a useful algebra of operations on sets of components. The language is a conservative extension of Leroy's (1994) calculus, which forms the foundation of the module language of Objective Caml.
- We show by example how our language can describe groups of signatures in ways that are difficult or impossible to do in existing languages.
- We show that for ML-like languages, there is no greatest lower bound in the subtype ordering for signatures. That is, given two signatures S and S', even if there is a third signature that is a subtype of both S and S', there is no *least specified* signature I that is a subtype of both S and S'. This result implies that there is no signature I such that a module M implements both S and S' if and only if it implements I. Our result holds even in Leroy's calculus. If, however, we exclude code that implements a signature only by being more polymorphic than the signature requires, then we can compute greatest lower bounds for ML signatures.

Our language is implemented for a simple core language based on ML.

2. Motivation and background

Our goal is to help programmers express nontrivial relationships among signatures, the better to explain and understand the structure of complex software. To meet this goal, we introduce a new language design. To enable use by programmers, we have made the language fairly rich. We have also paid attention to concrete syntax.

Our work is a bit different from other work on language features for modules. In most other work, such features are typically studied in the context of a small core calculus; we use a surface language. This decision makes our formalism atypical: unlike the description of a typical calculus, our formal description covers the full translation from unannotated syntax to semantic objects. Our formal descriptions are therefore more detailed and complex than usual. For this reason, almost all of our formalism is relegated to a companion technical report (Ramsey, Fisher, and Govereau 2005): the point of this paper is not that we can analyze our language formally, but that a modest extension to ML solves a significant class of programming problems.

We had hoped to define our new language in terms of an existing semantic framework, such as that of Leroy (1994), of Harper and Stone (2000), of Dreyer, Crary, and Harper (2003), of Milner et al. (1997), or of Russo (2001). But these frameworks do not meet our needs: Although they are well suited to describing the meaning of a single signature, they are not appropriate for describing computations that manipulate signatures. The first three frameworks preserve too much of the original syntactic structure of the signature. In particular, they totally order the components of a signature. This total ordering ensures that important dependencies are preserved, but makes it awkward to manipulate the many independent components.[1] The last two frameworks support abstract types through the use of special names, which are introduced by binding constructs that do not appear in the source code. These hidden binding constructs make it difficult to manipulate signatures.

Because existing frameworks did not meet our needs, we developed a new framework, which we designed to satisfy these criteria:

- To help a programmer know when one signature can be used in place of another, we want two signatures to be deemed equivalent if and only if any module that implements one also implements the other. To meet this criterion, we define equivalence using Leroy's (1994) technique of principal signatures and signature subtyping.
- We want a form of referential transparency, such that if two signatures are equivalent, one can be used for the other in any context. This criterion means that operations such as `rebinding` cannot use certain syntactic tricks based on names. For example, the following two ML signatures are equivalent:

```
signature S1 = sig       signature S2 = sig
  type t = int             type t = int
  val  x : t               val  x : int
end                      end
```

  If `S1` appears in a context where its type `t` is rebound to `bool`, for example, the type of `x` should not change (*cf.* Section 5.4).
- Instead of giving semantics only for an internal language, we want to give semantics to surface syntax, so as to help programmers understand the language.

[1] The Dreyer/Crary/Harper calculus not only enforces order but also discards the names used in the source code. This calculus would make a good foundation for the internal language of a compiler, but it is not so good when we want to manipulate signatures using the names originally present in the source.

declaration ⇒ `module type` *signature-name* = *sig*

sig ⇒ `sig` {*binding*} `end`
  | `functor` (*module-name* : *sig*) `->` *sig*
  | *sig* `with type` $\tilde{p} = \tilde{\tau}$
  | *signature-name*

binding ⇒ `type` *name* :: $\tilde{\kappa}$ [= $\tilde{\tau}$]
  | `val` *name* : $\tilde{\tau}$
  | `module` *name* : *sig*
  | `include` *sig*

$\tilde{p}$ ⇒ Path (qualified name)

$\tilde{\tau}$ ⇒ Core-language type

$\tilde{\kappa}$ ⇒ Core-language kind

Figure 1. Syntax of current ML signature language

To leverage strengths of existing languages, we start with ML, which among languages in wide use has the most powerful constructs for programming with interfaces. In both major dialects (Standard ML and Objective Caml), the unit of implementation is the *module*; a module is either a *structure* or a *functor*. A structure is a collection of named *components*; a component may define a *value*, a *type*, or a nested submodule. A functor is a mapping from one module to another.

An ML module is described by its *signature*, which can be thought of as an interface or a type. The signature for a structure describes each component of the structure. A value is described by its type, and a nested submodule is described by its signature. A type may be described by its kind alone, in which case it is *abstract*, or by its definition, in which case it is *manifest*. The signature for a functor gives the signatures for the argument and result of the functor. To make it possible for components of the result signature to depend on types declared in the argument signature, the argument signature is given a name.

Figure 1 gives a grammar for ML signatures.[2] Figure 1 also shows a notational convention we use throughout this paper: when a symbol could stand either for abstract syntax or for a semantic object, we write abstract syntax with a tilde over it. For example, τ is a semantic type, but $\tilde{\tau}$ is the syntax for a type.

ML includes only two constructs for manipulating signatures: `with type` and `include`. The `with type` construct enables a programmer to make an abstract type manifest. The `include` construct is a syntactic way to combine two signatures; it is as if the components of the included signature appeared in place of the `include`. In Section 5.4, we present a more semantically grounded way of combining signatures.

Because of differences between dialects, Figure 1 is not a complete story about all module-level features of all ML dialects. In Objective Caml, the unit of compilation is the structure, whereas in Standard ML, true separate compilation is not possible. (Tools such as CM (Blume and Appel 1999) provide a reasonable facsimile.) In Standard ML, the argument or result of a functor must be a structure, whereas in Objective Caml, an argument or result may also be a functor. In addition to `with type`, Objective Caml provides an analogous `with structure`, which is syntactic sugar for a collection of `with type` constraints.

[2] For simplicity, Figure 1 ignores an important distinction; in ML, there are *two* sorts of core-language types: those that can appear in a type definition and those that can appear in a value declaration. For example, a type of kind $* \Rightarrow *$ can appear in a type definition but not in a value declaration, and a polymorphic type can appear in a value declaration but not in a type definition.

Of the two major dialects, Objective Caml is closer to the design we would like to reach: it supports first-class, nested functors, and it omits sharing constraints. (We discuss in Section 6.1 how our more general construct subsumes both `with type` and `sharing`.) For this reason, we use Objective Caml as our representative of existing languages.

3. Examples of the problem

Before we present our language of signatures (Section 4), we present examples of ways in which current languages are flawed. Because such examples require discussion of relationships among modules, we present a nontrivial collection of signatures, which takes substantial space. But the examples do double duty: we use them here to illustrate flaws, and in Section 4 we use them again to show how our language addresses the flaws.

Rather than contrive examples to illustrate our language, and rather than use standard abstract data types such as stacks, sets, and finite maps, we have chosen to use examples derived from real software. Such examples may require more effort to understand, and they run the risk of conflating software-design issues with language-design issues, but they help show that our proposal is grounded in real problems. Readers who are already familiar with the limitations of ML signatures may prefer to skip these examples and continue with Section 4.

To focus our presentation, we have chosen our examples from just one software system: an embedded interpreter intended for *scripting*. The idea is simple: a complex application is linked with a small, reusable interpreter that is used at run time to drive or control the application. Reuse requires extension; an embedded interpreter is extended by adding application-specific values and code. For example, the popular computer game *Baldur's Gate* is linked with an embedded interpreter for the programming language Lua (Ierusalimschy 2003). The game extends Lua's built-in values with application-specific values representing monsters, weapons, and so on. The game includes Lua code that is interpreted to help determine how monsters and other "non-player characters" behave. The combination of parts of the Lua interpreter with independently developed extensions leads to interesting problems in the description of interfaces.

Our examples are divided into three groups. Each group begins with one or more signatures that are required for background, then goes on to present signatures that illustrate problems with the ML signature language. At the end of each group, problems are summarized with bullet points. Solutions appear in Section 4.

Deriving one interface from another A Lua interpreter is presented to a game engine using a single interface, expressed as the ML signature `INTERP`. This signature defines type `value`, which includes all values that the interpreter can manipulate: built-in values such as strings, numbers, and hash tables, as well as application-specific values, which have type `usert`. The `INTERP` signature also includes many useful functions: for example, "create an instance of the interpreter" or "evaluate a script." The `INTERP` signature is derived from another signature, `EVALUATOR`, which is itself *almost* derived from a third signature, `CORE`. All three signatures depend on `VALUE`, which defines the types that represent Lua values and the state of a Lua interpreter.

```
module type VALUE = sig
  type value (* a Lua value *)
  type state (* the state of a Lua interpreter *)
  type usert (* a user-defined value *)
  ...
end
```

Here are the definitions of `CORE` and `EVALUATOR` and the derivation of `INTERP`.

```
module type CORE = sig
  module V : VALUE
  val apply : V.value -> V.state ->
    V.value list -> V.value
    (* apply function f in state s to list of args *)
  val setglobal :
    V.state -> string -> V.value -> unit
  ⟨five more functions common to core and evaluator⟩
end

module type EVALUATOR = sig
  module Value : VALUE
  module Ast   : AST with module Value = Value
  type state = Value.state
  type value = Value.value
  exception Error of String
  val compile : Ast.program -> ...
  val setglobal :
    state -> string -> value -> unit
  ⟨five more functions common to core and evaluator⟩
  ⟨additional definitions in evaluator but not core⟩
end

module type INTERP = sig
  include EVALUATOR
  module Parser : PARSER with type chunk = Ast.chunk
  val dostring  : state -> string -> value list
  val mk        : unit -> state
end
```

We find nothing wrong with the `VALUE` signature, but the other signatures exhibit several defects:

- Because `INTERP` is derived from `EVALUATOR`, a programmer who needs to understand `INTERP` must also consult the definition of `EVALUATOR`. Because `INTERP` is the only signature that a user of the library really needs to understand, we would prefer to derive `EVALUATOR` from `INTERP`—but this is not possible. (An alternative would be to write out both `INTERP` and `EVALUATOR` in full, but this alternative would not only create a maintenance headache but also obscure the close connection between the two signatures.)
- For two reasons, we are forced to duplicate the components that are common to `CORE` and `EVALUATOR`. First, `apply` appears in `CORE` but not in `EVALUATOR`. Second, the value module is named `Value` in `EVALUATOR` but `V` in `CORE`. We could work around the first reason by adding `apply` to `EVALUATOR`, which would do no harm. But the second reason should not be worked around: The name `Value` is used in `INTERP` (and therefore also in `EVALUATOR`) because `INTERP` is intended for a programmer who does not know all the details of the system, so full names are appropriate. The name `V` is used in `CORE` because `CORE` is intended for a different programmer—one who is defining an extension to the system and who will use the `V` module extensively.

A better structure, which we show in Section 4, would be to define `INTERP` by writing out all the components in a form suitable for understanding by a fairly naïve programmer, then to derive `EVALUATOR` from `INTERP` and `CORE` from `EVALUATOR`. (Programmers might disagree about whether to derive `EVALUATOR` from `INTERP` or `INTERP` from `EVALUATOR`; the main point is that we want a language that allows the programmer to decide.)

Describing properties of extensions A Lua interpreter is useful largely because it can manipulate values of the underlying game engine, such as monsters and weapons. Such manipulation is enabled by adding the monster and weapon types to the interpreter. Each such addition (or extension) must satisfy several properties. Ideally we would define each property using a signature, then combine the signatures, but doing so using ML is awkward.

To add a type, the game engine must provide an equality test and a string-conversion function. This requirement is described by the `USERTYPE` signature.

```
module type USERTYPE = sig
  type t
  val eq        : t -> t -> bool
  val to_string : t -> string
end
```

The `USERTYPE` signature is fine as it is; we need it only for background.

Extensions are compiled separately, then combined into a single type, which becomes the `usert` type mentioned in the `VALUE` signature. To get access to a single extension, we need an appropriate "view," which provides a bidirectional mapping between the combined type `combined` and the single extension `t`.

```
module type TYPEVIEW = sig
  type combined
  type t
  val map : (combined -> t) * (t -> combined)
end
```

In our running example, the `combined` type might be the `usert` in a `VALUE` module, and the type `t` might be "monster" or "weapon." Like `USERTYPE`, `TYPEVIEW` is a perfectly good signature that we need for background.

Because different parts of the game-engine code might need access to any or all application-specific extensions, we need a signature that provides access to views of *all* the extensions that go into making up a combined type. Here is a signature that combines just two extensions:

```
module type COMBINED_COMMON = sig
  type also_t
  module TV1 : TYPEVIEW with type combined = also_t
  module TV2 : TYPEVIEW with type combined = also_t
end
```

Here, the type `also_t` might be the `usert` type, type `TV1.t` might be "monster," and type `TV2.t` might be "weapon."

An application may need arbitrarily many extensions. For example, the game engine might need not only monsters and weapons but also armor. To support unlimited extension, our Lua interpreter makes it possible to treat "combined" extensions as a new extension, which can then itself be combined. We would like to express this possiblity by defining a signature `COMBINED_TYPE` that has all the properties of both `COMBINED_COMMON` and `USERTYPE`. Here is how we have to do it in ML:

```
module type COMBINED_TYPE = sig
  include COMBINED_COMMON
  include USERTYPE with type t = also_t
end
```

Here the new type `t` might represent "weapons or armor" while the two views `TV1.t` and `TV2.t` might represent weapons and armor, respectively.

ML's mechanism for combining signatures is ugly:

- In `COMBINED_COMMON`, we use the annoying name `also_t` instead of the idiomatic name `t`. We do this because we use `include` to combine signatures, and we have to avoid a collision on the name `t`. Then we need the additional `with type` to make `t` and `also_t` the same type.

  The problem with `also_t` goes beyond ugliness: the programmer who wrote the `COMBINED_COMMON` signature had to be prescient, knowing in advance that because `COMBINED_COMMON` would be combined with `USERTYPE`, the standard name `t` could not be used. In Section 4, we show how to combine two signatures without advance planning.

Refining description of a functor Application-specific code is linked into a Lua-ML interpreter when the interpreter is bootstrapped from a `CORE` interpreter. The application-specific code takes the form of an ML functor that matches signature `BARECODE`:

```
module type BARECODE =
  functor (C : CORE) -> sig
    val init : C.V.state -> unit
  end
```

Such a functor receives a `CORE` and provides an `init` function. When called, the `init` function uses the core's `setglobal` to add new Lua-ML code to the interpreter's state. For example, the *Baldur's Gate* game adds a "create weapon" function which can be invoked at run time to create a new instance of a weapon. This function is used by developers to debug and by players to cheat.

The `BARECODE` signature is perfectly good by itself, but if the create-weapon code needs access to the application-specific weapon type, `BARECODE` is not expressive enough. Here's the problem: the application-specific code needs a `TYPEVIEW` for its application-specific type, and the `combined` type in that `TYPEVIEW` must be the same as the `usert` type in the core's `VALUE` module. But given the signature `BARECODE` above, there is no way for anything outside of `BARECODE` to *name* the core's `VALUE` module. We therefore define a more elaborate signature for application-specific code.

```
module type USERCODE = sig
  type usert
  module M :
    functor (C : CORE with type V.usert = usert) ->
    sig
      val init : C.V.state -> unit
    end
end
```

Using the `USERCODE` signature, we can name the appropriate type. For example, here is the signature of a putative "weapons library."

```
module type WEAPON_LIB = sig
  type t = Weapon.t
  module T : USERTYPE with type t = t
  module Make :
    functor (TV : TYPEVIEW with type t = t) ->
    USERCODE with type usert = TV.combined
end
```

The surprising annotation "`with type t = t`" is a frequently used idiom. It says something nontrivial because the terms on the two sides of the equals sign are elaborated in different environments. For example, to read `USERTYPE with type t = t`, we must know that on the left of the equals sign, all the components of `USERTYPE` are implicitly included in the environment. But on the right of the equals sign, these components are not included. What the phrase says is that module `T` implements a refinement of `USERTYPE` in which the abstract type `t` is revealed to be identical to the type `Weapon.t` (which we assume is defined elsewhere).

The signatures above show two problems:

- One cannot name a type that appears in the argument or result of a functor.
- In the syntactic form *sig* `with type` $\tilde{p} = \tilde{\tau}$, the terms on either side of the = sign are elaborated in *different* environments. This semantics leads to confusing idioms such as `with type t = t`. It also limits the expressive power of `with type` (see Section 6.1).

Grammar

$$
\begin{array}{lcl}
S & \Rightarrow & \texttt{sig}\ Cs\ \texttt{end} \\
 & | & \texttt{functor}\ (X\,\texttt{:}\,S)\ \texttt{->}\ S \\
 & | & T \\
 & | & S\ ms \\
 & | & S\ \texttt{andalso}\ S \\
 & | & S\ \texttt{sealing with}\ S \\
Cs & \Rightarrow & \{C\} \\
ms & \Rightarrow & \{m\} \\
m & \Rightarrow & \texttt{revealing type}\ \tilde{p}\ \texttt{=}\ \tilde{\tau}\ \{\texttt{and type}\ \tilde{p}\ \texttt{=}\ \tilde{\tau}\} \\
 & | & \texttt{revealing module}\ \tilde{p}\ \texttt{=}\ \tilde{p}'\ \{\texttt{and type}\ \tilde{p}\ \texttt{=}\ \tilde{p}'\} \\
 & | & \texttt{adding}\ pC\ \{\texttt{and}\ pC\} \\
 & | & \texttt{removing}\ cpt\ \{\texttt{and}\ cpt\} \\
 & | & \texttt{rebinding}\ pC\ \{\texttt{and}\ pC\} \\
 & | & \texttt{moving}\ cpt\ \texttt{=>}\ \tilde{p}\ \{\texttt{and}\ cpt\ \texttt{=>}\ \tilde{p}\} \\
 & | & [\texttt{with}\ \tilde{p}]\ \texttt{as}\ pat \\
C & \Rightarrow & \texttt{type}\ t :: \tilde{\kappa}\ [\texttt{=}\ \tilde{\tau}] \\
 & | & \texttt{val}\ x : \tilde{\tau} \\
 & | & \texttt{module}\ X\ \texttt{:}\ S \\
pC & \Rightarrow & \texttt{type}\ \tilde{p} :: \tilde{\kappa}\ [\texttt{=}\ \tilde{\tau}] \\
 & | & \texttt{val}\ \tilde{p} : \tilde{\tau} \\
 & | & \texttt{module}\ \tilde{p}\ \texttt{:}\ S
\end{array}
$$

$$
\begin{array}{lcl}
pat & \Rightarrow & X \mid pat\ \texttt{->}\ pat \\
cpt & \Rightarrow & (\texttt{type} \mid \texttt{val} \mid \texttt{module})\ \tilde{p} \\
\tilde{p} & \Rightarrow & X \mid x \mid t \mid X.\tilde{p}
\end{array}
$$

Terminal and nonterminal symbols

C	A component (binding) used to define a signature
Cs	A sequence of components (between `sig`. . . `end`)
cpt	An existing component identified by path
$\tilde{\kappa}$	Syntax for a kind
m	A signature modifier (e.g., `adding`. . .)
ms	A sequence of modifiers
$\tilde{p}$	Syntax for a path
pC	A "path component," used to extend or change an existing signature
S	A signature
t	The name of a type
T	The name of a top-level signature
$\tilde{\tau}$	Syntax for a type
x	The name of a value
X	The name of a submodule

The metalanguage is Wirth's EBNF: square brackets for optional, braces for zero or more. The grammar is ambiguous; a sequence of modifiers $\{m\}$ binds as tightly as possible, and the *pat* arrow binds as in ML types. Parentheses can be used around S and *pat*.

Figure 2. Syntax of an expressive language of signatures

4. An expressive language of signatures

In each of the examples above, a problem arises because there is a relationship among signatures that cannot easily be expressed formally. To solve such problems, we have devised a more expressive language of signatures. Our language is an extension of Leroy's (1994, 2000) version of ML.

Our design extends signatures with operations analogous to set operations. We aim for two kinds of completeness: If we provide an operation that can make a small change in a signature, we also provide an inverse operation that can undo the change. And if we provide an operation that can change one component of a signature, we make sure it can change *any* component of the signature. We also try to make it easy for a programmer to understand when signatures or types are the same.

Our design is summarized in Figure 2. As in ML, signatures can be formed by `sig`. . . `end` and `functor` constructs, and they can be looked up by name T. In addition, a signature can be modified by "modifiers" ms, and two signatures can be combined using the `andalso` and `sealing with` operations.

Changing translucency The first modifiers are `revealing type` and `revealing module`, whose inverse is `sealing with`. In the terminology of Harper and Lillibridge (1994), these operations change the degree of translucency of a signature. In the terminology of Leroy (1994), `revealing` makes abstract types manifest (comparable to OCaml's `with type` and `with module`) while `sealing` makes manifest types abstract. Like OCaml's `with type`, `revealing type` is used not only to give a definition to an abstract type but also to force two abstract types to be the same. Unlike `with type`, our `revealing type` is also powerful enough to express equalities that would otherwise require Standard ML's `sharing` constraints. The `revealing module` modifier reveals all of a structure's abstract types at one go. Finally, `sealing` is the inverse of `revealing`. We discuss these modifiers in more detail in Section 5.4, where we discuss their elaboration.

Adding, removing, and changing components The `adding` and `removing` modifiers are provided by analogy with set operations. They are more restricted than actual set operations; for example, one cannot remove the declaration of an abstract type on which another declaration depends. The `rebinding` modifier is a bit like `removing` followed by `adding`, except that it can rebind "in place" an abstract type or submodule on which something else depends.

The `moving` modifier is more of an analogy with filesystems than with sets: it provides the ability not just to change the name of a value, type, or submodule, but even to move it to a different place in the signature, e.g., as in `S moving type TV1.combined => TV2.t'`. Unlike `removing` followed by `adding`, `moving` can move an abstract type to which other components refer.

Returning to our running example, these modifiers make it possible to define `INTERP` by explicit enumeration and to derive `EVALUATOR` from `INTERP`:

```
module type INTERP = sig
  module Value : VALUE
  ⟨ ... all components explicit ... ⟩
end

module type EVALUATOR =
  INTERP
  removing module INTERP.Parser
       and    val INTERP.dostring
       and    val INTERP.mk
```

To refer to components of a signature being modified, we use fully qualified names such as `INTERP.mk`. Some alternatives are discussed in Section 6.1.

Introducing new module names The last modifier, `with` $\tilde{p}$ `as` *pat*, does not actually modify a signature; instead, it introduces one or more new names for the signature's components. The initial `with` $\tilde{p}$ selects a component of the modified signature; if `with` $\tilde{p}$ is omitted, it selects the entire signature. The pattern *pat* matches the selected component and introduces one or more new module names. A newly introduced module name X can be used in later modifiers to refer to a component of that module, even if the component is the argument or result of a functor.

The `with` $\tilde{p}$ `as` *pat* modifier can be used for mere convenience, as shown in this derivation of `CORE` from `EVALUATOR`. We rename the `Value` component, add the declaration of the `apply` function, and remove unwanted components. So that we don't have to write `EVALUATOR` every time, we omit the `with` $\tilde{p}$ and use `as E` to define an abbreviation `E`.

```
module type CORE =
  EVALUATOR as E
  moving module E.Value => E.V
  adding   val E.apply : ...
  removing val E.compile
       and module E.Ast
       and ⟨additional components⟩
```

If we want to refine the description of a functor, `with` $\tilde{p}$ `as` *pat* is required. For example, the modifier `as (C -> I)` enables us to manipulate `BARECODE` directly; the ugly signature `USERCODE` becomes unnecessary.

```
module type BARECODE =
  functor (C : CORE) -> sig
    val init : C.V.state -> unit
  end

module type WEAPON_LIB = sig
  type t = Weapon.t
  module T : USERTYPE as U revealing type U.t = t
  module Make :
    functor (TV : TYPEVIEW
                    as TV revealing type TV.t = t)
      -> (BARECODE as (C -> I)
           revealing type C.V.usert = TV.combined)
end
```

Our `revealing type` is subtly different from Objective Caml's `with type` or Standard ML's `where type`: the terms on either side of the equals sign are elaborated in the same environment. This difference eliminates the unfortunate `with type t = t` from the original definition of `WEAPON_LIB`.

Combining signatures The last extension we provide is the `andalso` operation. The `andalso` operation is analogous to a union of components, but it is motivated from a more semantic model based on the implementation relation: informally, a module M implements the signature S `andalso` S' if and only if it implements both signature S and signature S'. The effect is somewhat like taking the union of components in S and S', except that if a name is declared in both S and S', the two declarations must be consistent. We discuss the precise definition in Section 5.4.

By using `andalso`, we can easily and concisely state that a `COMBINED_TYPE` has all the properties of both `USERTYPE` and `COMBINED_VIEWS`. And because `andalso` correctly handles situations in which type `t` is defined in both signatures, we don't have to plan ahead and use an ugly name like `also_t`:

```
module type COMBINED_VIEWS = sig
  type t
  module TV1 :
    TYPEVIEW revealing type TYPEVIEW.combined = t
  module TV2 :
    TYPEVIEW revealing type TYPEVIEW.combined = t
end
```

```
module type COMBINED_TYPE =
  USERTYPE andalso COMBINED_VIEWS
```

As another example, it is possible to use a signature to specify a single interesting property of a type, then combine properties using `andalso`:

```
signature EQUATABLE = sig
  type t  val eq : t -> t -> bool          end
signature COMPARABLE = sig
  type t  val compare : t -> t -> order    end
signature HASHABLE = sig
  type t  val hash : t -> word             end
signature SHOWABLE = sig
  type t  val show : t -> string           end
signature HEAVILY_FEATURED =
  COMPARABLE andalso HASHABLE andalso SHOWABLE
```

Similar signatures are found in commonly used ML libraries. These signatures offer possibilities for reuse; for example, if they were standard, we would probably change the definition of `USERTYPE` to be `EQUATABLE andalso SHOWABLE`.

5. Formal definitions

To make the meaning of our language clear, we have developed a formal semantics. Because our language manipulates and computes with signatures, our formal semantics is somewhat different from those used in related work (see Section 2). The main idea is that a signature denotes a set of components, divided into four subsets of different kinds. The details of elaborating abstract syntax into a denotation are less important and so are relegated to the companion technical report. Before getting to the main idea, we introduce some terminology and supporting concepts.

5.1 Preliminaries

Identifiers An identifier is simply a name. As metavariables, we use the identifier x to stand for the name of a value, t to stand for the name of a type, X to stand for the name of a module, and T to stand for the name of a signature. Unlike Leroy (1994, 2000), we do not require that an identifier be decorated or "stamped" with the location at which it is defined. We do assume, in order to simplify this paper, that module names X are disjoint from type names t.

Paths A path (sometimes called a "qualified name") selects a component from a module; every component can be identified uniquely by its path. Considered as a semantic object, a path p is defined inductively as follows:

$$p \Rightarrow \varepsilon \mid x \mid t \mid X.p \mid \textsc{arg}.p \mid \textsc{res}.p$$

Depending on whether it ends in ε, x, or t, a path can name a module, a value, or a type. To select a part of a module, a module name X selects a submodule of a structure; an ARG or RES selects the argument or result of a functor. When notating paths, we omit a trailing $.\varepsilon$. We also use the dot to concatenate paths; when used this way, the dot is associative and has ε as a left and right identity. We reassociate dots as needed.

By the obvious homomorphic embedding, a syntactic path $\tilde{p}$ is also a semantic path, and we use it as such. As shown in Figure 2, however, the set of syntactic paths does not include the empty path or any path that uses an ARG or RES selector. By not letting a programmer write ARG or RES, we preserve the existing syntax of paths, and we use the ML style of programming by pattern matching.

Core-language requirements Like Leroy (2000), we intend our work to apply to more than one core language, so we wish to minimize assumptions about the core language. Because our signature language is much richer than Leroy's, we have assumed a simpler core language: one in which there are no nontrivial relations among

Core-language judgments	
$\sigma, D \vdash \tilde{\tau} \hookrightarrow \tau$	In context σ, D, syntax $\tilde{\tau}$ elaborates to core-language type τ. Our context σ, D may look a bit strange, but it is straightforward: σ is a mapping from identifiers to paths, which gives the path at which each in-scope identifier is defined. D is the denotation of the parts of the signature elaborated so far. The substitution σ and denotation D are used only to look up paths of abstract types, so the core-language judgment $\sigma, D \vdash \tilde{\tau} \hookrightarrow \tau$ can be implemented by a function of type $(p \to \tau\ \texttt{option}) \to \tilde{\tau} \to \tau$.
$\vdash \tilde{\kappa} \hookrightarrow \kappa$	Syntax $\tilde{\kappa}$ elaborates to core-language kind κ.
$\tau \approx \tau'$	Core-language types τ and τ' are equivalent
$\tau <: \tau'$	Core-language subtyping
Core-language functions	
$\mathit{freePaths}(\tau)$	The set of abstract types mentioned in type τ.
$\mathit{kind}(\tau)$	The kind of type τ
$\mathit{substTy}(\tau', p, \tau)$	Substitute τ' for p in type τ.
$\mathit{replacePrefixTy}(p', p, \tau)$	For every abstract type in τ whose name begins with p, replace the prefix p with new prefix p'.
Selected signature-language judgments	
$\sigma \vdash \tilde{p} \hookrightarrow p$	In context σ, path $\tilde{p}$ is an abbreviation for p.
$\tau @ p$	Every abstract type in type τ may be mentioned in a declaration at path p.
D wf	Denotation D is well formed.

Figure 3. Judgments and functions

kinds. Our assumptions apply to ML and some ML-like languages, but they would need to be relaxed to accomodate object-oriented languages described with power kinds. We express our assumptions as formal judgments, proof rules for which or implementations of which must be provided by the designer of the core language. These judgments are discussed below and summarized at the top of Figure 3. Some judgments in Figure 3 involve a denotation D, which is introduced in Section 5.2 below.

We assume that the core language provides semantic objects representing types τ and kinds κ, and that for each there is corresponding syntax $\tilde{\tau}$ and $\tilde{\kappa}$. We assume that there is a core-language judgment for type equality $\tau \approx \tau'$. We also assume there is a core-language judgment for subtyping $\tau <: \tau'$. (In core ML, subtyping is the "at least as polymorphic as" relation.) Finally, we assume the function $\mathit{kind}(\tau)$ can compute the core-language kind of any core-language type.

In addition to these requirements, a core language must be extended with abstract types. As far as the core language is concerned, an abstract type is simply a path for which a real core-language type may be substituted later. The path is annotated with a kind, and any substitution must respect the kind.

Our "abstract type" is slightly different from what programming experience might lead one to expect: it is a "name that can be abstracted over," not a "type whose definition is hidden." In excruciating detail, our abstract type is a type in an *interface* whose definition is *unspecified*; it is represented by a path and kind. A type in an *implementation* whose definition is *hidden* is represented not as an abstract type but as a fresh *type constructor*, written μ. A fresh type constructor can never by introduced be a signature; a fresh type constructor can be introduced only by a structure.

The abstract types that appear in a core-language type τ are called the *free paths* of τ. We require that the semantic representation of types support two functions that help substitute for the free paths of a type. First, if Θ maps paths to core-language types, then function $\mathit{mapabs}\ \Theta$ performs the corresponding simultaneous substitution on a type. Second, the *prefix replacement* of p with p' in τ, written $\mathit{replacePrefixTy}(p, p', \tau)$, replaces every abstract type of the form $p.p'' :: \kappa$ with the new type $p'.p'' :: \kappa$. Abstract types not of the form $p.p'' :: \kappa$ are unchanged. This operation is useful in defining the `moving` modifier, among others.

A final requirement on the core language is that it be possible to *discover* substitutions for abstract types by unifying lists of core-language types, as explained in the discussion of `andalso` in Section 5.4.

Our core-language type-equality judgment $\tau \approx \tau'$ differs from the corresponding judgment in Leroy (1994, 2000). The difference is that Leroy's core language has to consult an environment to find definitions of manifest types. We handle manifest types in our elaboration step $\sigma, D \vdash \tilde{\tau} \hookrightarrow \tau$, so by the time we get to judging equality, we no longer need the environment that keeps definitions of manifest types.

Our handling of core-language types parallels that in the *Definition of Standard ML* (Milner et al. 1997). Our distinction between abstract type and type constructor is handled in the *Definition* using binders. Our abstract type corresponds to the *Definition*'s "flexible name," i.e., one that is bound in a signature. Our type constructor corresponds to a "rigid name," i.e., one that is free in a signature. Creating a fresh type constructor corresponds to creating a new "generative stamp." Our substitution for an abstract type corresponds to the *Definition*'s "type realization."

5.2 The denotation of a signature

Our semantic framework builds on the intuition that a signature is a set of components. Unfortunately, a real signature cannot be treated as a simple set—a semantics must account for *dependencies*.

- Because the type of a value can mention abstract types declared in the same signature, the declaration of a value can depend on one or more abstract types. (Syntactically, the type of a value can also mention manifest types, but elaboration replaces manifest types with their definitions, so semantically, a value cannot depend on a manifest type.)
- Because the definition of a manifest type can mention abstract types declared in the same signature, the definition of a manifest type can depend on one or more abstract types. As for a value declaration, elaboration replaces each *use* of a manifest type by its definition, so there is no dependence on other manifest types.

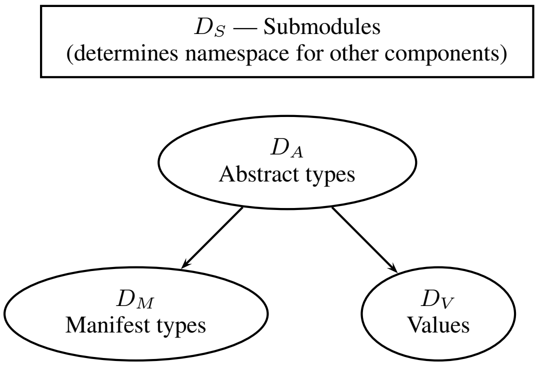

Figure 4. Structure of a SAMV denotation D

- Rules on dependencies are further complicated by the presence of functors: unlike a nested structure, a nested functor cannot introduce types on which declarations in other components can depend. There is one exception: the argument signature of a functor can introduce types on which the result signature of that functor depends.

We deal with these issues by defining the denotation of a signature to be not a single set of bindings, but four sets of bindings: Submodules, Abstract types, Manifest types, and Values. We call such a denotation a SAMV form, pronounced "sam-vee," and we write it as D. The four sets are D_S, D_A, D_M, and D_V.

- Each binding in the set D_S maps a path p to a tag that identifies a module at that path. The tag is either `structure` or `functor`. Given a binding (p, tag) in D_S, we may instead write $tag\ p$.

  A binding in D_S is independent of all other bindings.

- Each binding in the set D_A maps a path p to a core-language kind κ. The path uniquely identifies an abstract type. Given a binding (p, κ) in D_A, we may instead write $\mathtt{type}\ p :: \kappa$.

  A binding in D_A is independent of all other bindings.

- Each binding in the set D_M maps a path p to a manifest core-language type τ. Given a binding (p, τ) in D_M, we may instead write $\mathtt{type}\ p = \tau$.

  The bindings in D_M induce a substitution on core-language types. As noted above, we write this substitution $mapabs\ D_M$. Such substitutions play a crucial role in subtyping of signatures.

  Because a core-language type τ may mention abstract types in D_A, a binding in D_M may depend on bindings in D_A, but it is independent of all other bindings.

- Each binding in the set D_V maps a path p to a core-language type τ. An implementation of the signature promises to deliver, at that path, a value of type τ (or possibly a core-language subtype of τ). Given a binding (p, τ) in D_V, we may instead write $\mathtt{val}\ p : \tau$.

  Because a core-language type τ may mention abstract types in D_A, a binding in D_V may depend on bindings in D_A, but it is independent of all other bindings.

Figure 4 shows the structure of a denotation, with arrows to show possible dependencies.

As a very simple example of a denotation, the signature

```
signature S1 = sig
  type t = int
  val  x : t
end
```

from Section 2 has the denotation

$$\langle D_S = \{\mathtt{structure}\ \varepsilon\},\quad D_M = \{\mathtt{type\ t} = \mathtt{int}\},$$
$$D_A = \{\},\qquad D_V = \{\mathtt{val\ x} : \mathtt{int}\}\rangle.$$

As an example of a denotation that includes an abstract type, here is the denotation of the `USERTYPE` signature:

$$\begin{aligned}\langle D_S &= \{\mathtt{structure}\ \varepsilon\},\\ D_A &= \{\mathtt{type\ t} :: *\},\\ D_M &= \{\},\\ D_V &= \{\mathtt{val\ eq} : \mathtt{t} \to \mathtt{t} \to \mathtt{bool}, \mathtt{val\ to\_string} : \mathtt{t} \to \mathtt{string}\}\rangle.\end{aligned}$$

As an example that includes nested structures and a mix of abstract and manifest types, here is the denotation of signature `COMBINED_VIEWS`:

$$\begin{aligned}\langle D_S &= \{\mathtt{structure}\ \varepsilon, \mathtt{structure\ TV1}, \mathtt{structure\ TV2}\},\\ D_A &= \{\mathtt{type\ t} :: *, \mathtt{type\ TV1.t} :: *, \mathtt{type\ TV2.t} :: *\},\\ D_M &= \{\mathtt{type\ TV1.combined} = \mathtt{t}, \mathtt{type\ TV2.combined} = \mathtt{t}\},\\ D_V &= \{\mathtt{val\ TV1.map} : (\mathtt{t} \to \mathtt{TV1.t}) \times (\mathtt{TV1.t} \to \mathtt{t}),\\ &\qquad \mathtt{val\ TV2.map} : (\mathtt{t} \to \mathtt{TV2.t}) \times (\mathtt{TV2.t} \to \mathtt{t})\}\rangle.\end{aligned}$$

Finally, to show a functor, here is the denotation of `BARECODE`:

$$\begin{aligned}\langle D_S &= \{\mathtt{functor}\ \varepsilon, \mathtt{structure}\ \mathrm{ARG}, \mathtt{structure}\ \mathrm{ARG}.\mathtt{V},\\ &\qquad \mathtt{structure}\ \mathrm{RES}\},\\ D_A &= \{\mathtt{type}\ \mathrm{ARG}.\mathtt{V.value} :: *, \mathtt{type}\ \mathrm{ARG}.\mathtt{V.state} :: *,\\ &\qquad \mathtt{type}\ \mathrm{ARG}.\mathtt{V.usert} :: *\},\\ D_M &= \{\},\\ D_V &= \{\mathtt{val}\ \mathrm{ARG}.\mathtt{apply} : \cdots,\\ &\qquad \mathtt{val}\ \mathrm{ARG}.\mathtt{setglobal} : \cdots,\\ &\qquad \mathtt{val}\ \mathrm{RES}.\mathtt{init} : \mathrm{ARG}.\mathtt{V.state} \to \mathtt{unit}\}\rangle.\end{aligned}$$

Each set in a denotation is *flat*, which is a significant departure from previous work. In previous work, a language is typically defined using a term calculus with a reduction semantics; the denotation of a signature is a term in normal form. But components in a term are inherently ordered, and in many signatures the order of most components is irrelevant. Using a flat denotation frees the semanticist (and the programmer!) from having to reason about order, e.g., by introducing a permutation. And splitting the denotation into four sets simplifies the treatment of dependency. As compared with forms containing explicit binders (Milner et al. 1997; Russo 2001), we also make it easier to modify and compute with denotations.

The price we pay for the simplicity of a flat representation is that not every flat representation is meaningful. We require that a denotation be *well formed*. Well-formedness requires that D_A and D_M be disjoint, so that no type is identified as both abstract and manifest, and that the free paths of D_M and D_V be in D_A, so that every abstract type mentioned in a signature is defined in the signature.

But it is not enough to say that every abstract type mentioned in the signature is defined in the signature. Although we are as liberal as possible in allowing a type or value declared in one submodule to depend on an abstract type declared in another submodule, we must impose restrictions for functors: An abstract type declared in a functor's argument may be referred to only within that argument or within the corresponding result. An abstract type declared in a functor's result may be referred to only within that result. These restrictions are checked by the judgment $\tau@p$, which is pronounced "type τ may be mentioned at path p."

A type τ may be mentioned at path p if and only if for every abstract type p' that τ depends on, $p'@p$. The rules for $p'@p$ limit access to types declared in functors' argument and result signatures.

$$\frac{p' \text{ does not contain ARG or RES}}{p.p'@p.p''}$$

$$\frac{p' \text{ does not contain ARG or RES}}{p.\mathrm{ARG}.p'@p.\mathrm{RES}.p''}$$

$$\frac{\begin{array}{ll}\forall p \in \mathrm{dom}(D'_S) & : p \in \mathrm{dom}\, D_S \qquad\qquad\qquad \wedge\, D_S(p) = D'_S(p) \\ \forall p \in \mathrm{dom}(D'_A) & : p \in (\mathrm{dom}\, D_A \cup \mathrm{dom}\, D_M) \wedge \mathit{kind}_p(D) = D'_A(p) \\ \forall p \in \mathrm{dom}(D'_M) & : p \in (\mathrm{dom}\, D_A \cup \mathrm{dom}\, D_M) \wedge \mathit{mapabs}\ D_M\ p \approx \mathit{mapabs}\ D_M\ (D'_M(p)) \\ \forall p \in \mathrm{dom}(D'_V) & : p \in \mathrm{dom}\, D_V \qquad\qquad\qquad \wedge\, D_V(p) <: \mathit{mapabs}\ D_M\ (D'_V(p))\end{array}}{D <: D'}$$

$$\frac{\begin{array}{c}D <: D' \\ \forall p \in \mathrm{dom}(D'_V) : p \in \mathrm{dom}\, D_V \wedge D_V(p) \approx \mathit{mapabs}\ D_M\ (D'_V(p))\end{array}}{D \lesssim: D'}$$

Figure 5. Rules for subtyping and tight subtyping

In other words, either the two paths are in the same structure, or the mentioning path is the result of a functor of which the mentioned path is the argument. Mentioning is transitive, so if $p_1@p_2$ and $p_2@p_3$, then $p_1@p_3$.

Putting these conditions together, we get a rule that says when a denotation is well formed. The judgment "D_S well-parented" ensures that D_S faithfully describes the nested structure of submodules; its definition appears in the companion technical report.

$$\frac{\begin{array}{c}\mathrm{dom}\, D_A \cap \mathrm{dom}\, D_M = \{\,\} \\ \mathit{freePaths}(\mathrm{rng}\, D_M \cup \mathrm{rng}\, D_V) \subseteq \mathrm{dom}\, D_A \\ \forall (p,\tau) \in D_M \cup D_V : \tau @ p \\ D_S \text{ well-parented}\end{array}}{D \text{ wf}}$$

A well-formed denotation need not have quite all its parts. In particular, because a functor's argument signature is elaborated before its result signature, a well-formed denotation may have a functor's argument without the corresponding result. When elaboration is finished, however, all parts are present. We express this property by the judgment "D_S complete."

Our definition of well-formedness is more liberal than the syntactic definition of well-formedness used by Leroy (1994, 2000). In particular, our definition permits mutual dependencies between submodules of a structure. For example, we can write

```
module type MUTUAL = sig
  module M : sig
    type t
  end
  module N : sig
    type t
    val  v : M.t
  end
end as S
  adding val S.M.v : S.N.t
```

This strange signature contains two nested submodules, each of which defines an abstract type and a value. The value in each submodule is of the type defined in the other. Although such a signature seems strange at first (and is impossible to write in ML), it has a perfectly good denotation:

$$\begin{array}{rl} D = \langle D_S & = \{\texttt{structure}\ \varepsilon, \texttt{structure}\ M, \texttt{structure}\ N\}, \\ D_A & = \{\texttt{type}\ M.t :: *, \texttt{type}\ N.t :: *\}, \\ D_M & = \{\}, \\ D_V & = \{\texttt{val}\ N.v :: M.t, \texttt{val}\ M.v :: N.t\}\rangle \end{array}$$

This denotation is in no way monstrous; there are plenty of structures that implement it. Signatures with mutually recursive components are used by Crary, Harper, and Puri (1999) to introduce a special context which enables the elaboration of an *implementation* that contains mutually recursive substructures.

Our definition of well-formedness does exclude mutually recursive manifest-type definitions; for example, there is no way to create a denotation that contains a pair of substructures `M` and `N` in which `type M.t = N.u * N.u` and `type N.u = M.t * M.t`.

5.3 Subtyping of denotations

Subtyping plays a crucial role in determining whether an implementation matches the corresponding interface: following Leroy (1994, 2000), one computes a principal signature for the implementation, and if this signature is a subtype of the interface, the two match.

In module subtyping, a functor's argument is contravariant. We handle contravariance by identifying covariant and contravariant *paths* within each denotation. To keep this paper simple, we present only the covariant case, which defines subtyping without functors. Full details for functors are in the companion technical report.

The rules for signature subtyping appear in Figure 5. It may help to think of D as an implementation and D' as an interface. To determine the kind of a type defined at path p in denotation D, we define

$$\mathit{kind}_p(D) = \begin{cases} \kappa & \text{if } (p,\kappa) \in D_A \\ \mathit{kind}(\tau) & \text{if } (p,\tau) \in D_M \\ \text{undefined} & \text{otherwise} \end{cases}$$

Ignoring functors, the only tricky parts of subtyping involve manifest types. Essentially, when examining a manifest type in one signature, we must account for the manifest-type definitions in the other signature. We do this by treating a D_M component as a substitution using the mapabs function. Here is a small but tricky example; each of these signatures is a subtype of the other:

```
module type S  = sig type t type u = t end
module type S' = sig type u type t = u end
```

For this example, the interesting subtyping conditions are

$$\begin{array}{l}\texttt{u} \in \mathrm{dom}\, D_M \wedge \mathit{kind}(D_M(\texttt{u})) = D'_A(\texttt{u}) \\ \texttt{t} \in \mathrm{dom}\, D_A \wedge \texttt{t} :: * = \mathit{mapabs}\ D_M\ (D'_M(\texttt{t})).\end{array}$$

The first condition is trivially true ($* = *$), and the second holds because $\mathit{mapabs}\ D_M\ (\texttt{u} :: *) = \texttt{t} :: *$.

As shown in Figure 5, a `val` binding in a subtype need not provide exactly what is called for in the supertype. For a supertype containing `val` $x : \tau'$, it suffices for the subtype to provide `val` $x : \tau$ where τ satisfies the core-language subtyping judgment $\tau <: \tau'$. For example, the empty list, which has type $\forall\alpha.\alpha$ `list`, could be used to satisfy a specification calling for a value of type `int list`. This ability is essential for implementors, but as explained below, it creates an ugly wart on the lattice structure of signatures. To remove this wart, we introduce *tight subtyping*, written $D \lesssim: D'$, which is also defined in Figure 5. Informally, we can interpret $D \lesssim: D'$ to mean that an implementation matching D satisfies the specification D' without resorting to core-language subtyping.

5.4 Elaboration

Our language consists of a sequence of top-level declarations of the form `signature` $T = S$. Elaborating a declaration may require that we elaborate a signature S, a component C, a sequence of components Cs, a modifier m, or a sequence of modifiers ms. The elaboration rules themselves are too detailed for discussion here; we describe some techniques below, but full details are relegated to the companion technical report.

The most important property of the elaboration rules is that if a top-level signature is elaborated successfully, the resulting denotation is well formed and complete. Moreoever, our rules are consistent with those of Leroy (1994, 2000). To be precise, Leroy's rules judge a source-language signature to be well formed if and only if our rules elaborate it successfully. The two systems of rules also define the same subtyping relationship on signatures. Our language is therefore a conservative extension of Leroy's. The key idea behind the proof is to choose the "stamps" used in Leroy's system such that each identifier is stamped with the path p at which it is declared. Because the two systems use different core-language judgments, the proof also requires a fairly ugly relation between the core-language judgments of our system and the core-language judgments of Leroy's system. Our elaboration rules also support a universality property: given a well-formed, complete denotation, we can produce a source term that elaborates to it.

Elaborating paths, components, and simple modifiers When a component of a signature is introduced, it can be referred to by its identifier. For example, in the result signature of `BARECODE`, the argument module `C` is referred to by its identifier. Such identifiers are also introduced by `with` $\tilde{p}$ `as` *pat*. Like Leroy (2000), we track these identifiers using a substitution σ. The substitution is used mostly to elaborate paths: the judgment $\sigma \vdash \tilde{p} \hookrightarrow p$ says that the first name in $\tilde{p}$ appears in σ, and that by replacing that name using σ, we get path p.

Elaboration of a component or modifier takes place in a particular context, which is identified by a path p. When elaboration adds a new component, rules rules check that new types may be mentioned at p, that new components are at paths not already occupied, and so on. For a component added by a modifier, which may include an arbitrary path, rules check that the component is added to the signature being modified and not to some other signature.

Removing components is easy; the only real restriction is that one may not remove an abstract type without also removing everything on which that type depends.

Rebinding a value or manifest type is equivalent to removing the old component, then adding the new one. Rebinding an abstract type is possible, but if anything depends on the type, its kind must not change. Rebinding a submodule is more complicated but the spirit is the same.

Moving components is also easy; the interesting aspect is that moving or renaming an abstract type updates everything that depends on it. Moving rules also check that restrictions on mentioning remain satisfied.

The most interesting rules are for changing translucency and combining signatures.

Changing translucency To make an abstract type manifest, that is, to reveal `type` $p = \tau$, there are a variety of side conditions on τ: it must not depend on p, it must have the right kind, and it must be mentionable at path p. The `revealing module` modifier is essentially a shorthand for revealing a collection of abstract types. But when we elaborate `revealing module` $\tilde{p}$ = $\tilde{p}'$, we also check to see that the denotations at $\tilde{p}$ and $\tilde{p}'$ have the same components at the same types (once type revelations are accounted for). This check is stricter than checks found in existing dialects of ML; we discuss alternatives in detail in Section 6.1.

Making a manifest type abstract is a more complicated problem. Consider

```
signature S = sig
  type t = int
  val a : int -> int
  val b : t   -> t
end
```

If we make the manifest type `t` abstract, what are we to do with the value bindings? As pointed out by Mitchell and Plotkin (1988), each value binding could legitimately be given type `int -> int`, `int -> t`, `t -> int`, or `t -> t`. What we want *not* to do is depend on the way the bindings are written syntactically. Values `a` and `b` have exactly the same type and should get exactly the same treatment; it shouldn't matter if their types are written with `int` or `t`.

By default, we keep the original denotations of value bindings, so each binding above would get type `int -> int`. If a programmer wants a more abstract type, the abstract version of the binding must be put into the signature used for sealing. For example, we could write

```
signature S' =
  S sealing with sig
    type t
    val a : int -> t
  end
```

Signature `S'` is equivalent to

```
signature S'_Equiv = sig
  type t
  val a : int -> t
  val b : int -> int
end
```

This example shows the difference between sealing and ascription: if a binding does not appear in the sealing signature, it "shines through" into the result.

Combining signatures The `andalso` operation computes a greatest lower bound, that is, the most abstract signature that is an implementation of the two input signatures. This operation is written $D \sqcap D'$, and one might hope to define it with respect to the signature-subtype ordering, but unfortunately, if the core language is ML, there is no greatest lower bound in this ordering, even when two signatures have a common lower bound. The reason is that an ML programmer can create an implementation *either* by using module-language mechnisms *or* by using polymorphism in the core language. For example, consider the following signatures:

```
signature T = sig          signature V = sig
  type t                     type t
  type u                     type u
  val x : t list             val x : u list
end                        end
```

A programmer can refine these two signatures to a common lower bound in two incomparable ways: by making `t` and `u` equal, or by giving `x` the polymorphic type `'a list`.

```
signature L1 = sig         signature L2 = sig
  type t                     type t
  type u = t                 type u
  val x : t list             val x : 'a list
end                        end
```

It is easy to show that `L1` $<:$ `T`, `L1` $<:$ `V`, `L2` $<:$ `T`, `L2` $<:$ `V`, and there does not exist an M such that `L1` $<: M$ and `L2` $<: M$, $M <:$ `T`, and $M <:$ `V`. The insight is that in writing `L2`, the programmer has achieved signature subtyping by using core-language subtyping. If we rule out core-language subtyping, however, then we can compute greatest lower bounds. For this reason, we consider the partial order defined by the *tight subtyping* relation defined in

Figure 5. Using this order and the definition of $\sqcap$ below, we have shown that if $T \sqcap V$ is well formed, then $M \lesssim: T \sqcap V$ if and only if $M \lesssim: T$ and $M \lesssim: V$.

Here is how we compute $T \sqcap V$. As for subtyping, we ignore the contravariance introduced by functors. The computations of Å, E, and Θ are explained below.

$$\frac{\begin{array}{l} \text{Å} = (T_A \cup \mathit{mapr\ kind}\ T_M) \uplus (V_A \cup \mathit{mapr\ kind}\ V_M) \\ E = \{p :: \mathit{kind}(\tau) = \tau \mid (p, \tau) \in T_M\} \cup \\ \qquad \{p :: \mathit{kind}(\tau) = \tau \mid (p, \tau) \in V_M\} \cup \\ \qquad \{\tau_1 = \tau_2 \mid (p, \tau_1) \in T_V \wedge (p, \tau_2) \in V_V\} \\ \Theta = MGU(E) \qquad \mathrm{dom}\,\Theta \cap \mathit{freePaths}(\mathrm{rng}\,\Theta) = \{\,\} \\ \qquad \forall \tau : \mathit{kind}(\mathit{mapabs}\ \Theta\ \tau) = \mathit{kind}(\tau) \\ \qquad D_S \ = T_S \uplus V_S \\ \qquad D_A \ = \{(p, \kappa) \mid (p, \kappa) \in \text{Å} \wedge p \notin \mathrm{dom}\,\Theta\} \\ \qquad D_M = \Theta \\ \qquad D_V \ = \{(p, \mathit{mapabs}\ \Theta\ \tau) \mid (p, \tau) \in T_V\} \cup \\ \qquad\qquad\quad \{(p, \mathit{mapabs}\ \Theta\ \tau) \mid (p, \tau) \in V_V\} \end{array}}{D = T \sqcap V}$$

The function $\mathit{mapr\ kind}$ replaces each type in T_M and V_M with its corresponding kind. The $\uplus$ symbol is "consistent union:" if $\exists p : (p, v_1) \in A \wedge (p, v_2) \in B \wedge v_1 \neq v_2$, then $A \uplus B$ is undefined; otherwise $A \uplus B = A \cup B$.

The key to computing a *greatest* lower bound is to find as many manifest-type definitions as are necessary, but no more. We find these definitions through unification; the set E lists all of the equalities that have to hold for a common tight subtype of T and V, and we find a most general unifier Θ that will become the manifest-type definitions of the greatest lower bound. The unification is *not* the same as core-language unification such as is used to implement Hindley-Milner type inference. Our unification allows us to substitute a manifest type for an abstract type; there is no substitution for type variables.[3] Substitution for type variables makes a signature smaller, and we are computing the largest signature that is below both T and V. Once we have the substitution, computing the greatest lower bound is easy.

To prove that $\sqcap$ is a greatest lower bound requires one key lemma: if $M \lesssim: T$ and $M \lesssim: V$, and if $D = T \sqcap V$ is defined, then $\mathit{mapabs}\ M_M = \mathit{mapabs}\ M_M \circ \mathit{mapabs}\ D_M$. The lemma holds because M must satisfy all the equations in E.

The definition above assumes there are no functors. We have extended our definition to signatures that include functors, but only for the *positive* tight subtyping relation on signatures. That is, for $T \sqcap V$ to be defined, we require that the functor arguments be equivalent. Extension to fully contravariant functor arguments, like the dual greatest-upper-bound operation, awaits future work.

5.5 Scaling to realistic inputs

In this paper, we focus on elaboration of correct signatures. But we want to handle programs that have structures and functors, as well as programs that contain errors.

Structures and functors We have written and implemented rules that elaborate structures and functors to their principal signatures; syntax and semantics appear in the companion technical report. The rules are similar to the rules for signatures, with two kinds of additions: naming and generativity.

[3] There are restrictions; for example, the type that is substituted must be a *definable* type—in ML, not polymorphic. A significant difference from core-language unification is that in our unification problem, each equality constraint is associated with one or two named bindings. Therefore if the unification fails, it should be easy to explain what went wrong.

When we refer to a structure by name, we preserve identity of abstract types. For example, suppose we have

```
structure M = ...
structure N = M
```

The denotation at path `N` is like the denotation at path `M`, except that every abstract type must be made manifestly equal to the corresponding type at path `M` (unless the abstract type appears under a functor). This operation corresonds to Leroy's (1994) "strengthening" operation.

When we ascribe a signature to a structure, we replace each abstract type with a fresh type constructor: ascription is *generative*. If the ascription takes place in the body of a functor, generation of fresh type constructors is delayed until the functor is applied.

Error messages It is not enough to elaborate well-typed programs successfully; when a program fails to elaborate, a compiler must issue a reasonable error message. In our system, if a construct does not elaborate, the compiler has a context that includes a path and a denotation. Given this context, issuing reasonable error messages is easy. Here are a few examples:

- If a condition such as $p.x \notin \mathrm{dom}\, D_V$ fails, we can issue a message such as "multiple declarations of value $p.x$."
- If a condition such as $D <: D'$ fails, one or more of the universally quantified predicates in Figure 5 is unsatisfied. For each path not satisfying a predicate, we can issue a message such as "value $p.x$ expected but not provided" or "type τ of value $p.x$ does not match the expected type τ'."
- If a condition such as $\tau @ p$ fails, we can report that the type τ appearing outside a functor refers to an abstract type defined in that functor's argument or result signature.
- If we fail to form $T \sqcap V$, we can issue a message such as "in signatures joined with `andalso`, value $p.x$ has incompatible types τ and τ'" or "type $p.t$ has incompatible definitions τ and τ'."

In all cases, we can issue messages that refer not to whole denotations but to individual components of signatures; we hope a programmer could understand such messages without diving into details of SAMV form.

6. Discussion

To conserve space, we limit discussion to two topics: alternatives to the operations we propose and potential applications of our operations to structures and functors.

6.1 Design alternatives

The operations we propose may not be canonical or definitive—we intend them as a basis for discussion and refinement. In this section we discuss some design alternatives and motivations.

Environments and the expressive power of modifiers To make an abstract type manifest, Standard ML and Objective Caml offer a `where type` or `with type` operation. This operation differs from our proposal primarily in its use of environments. In our `revealing type` $\tilde{p}$ `=` $\tilde{\tau}$, the path on the left and the type on the right are evaluated in the *same* environment. In the existing languages, the path on the left is evaluated "inside" the signature being modified, where the type on the right is evaluated "outside" the signature. Thus ML's idiomatic `with type t = t` is not a tautology. Our operation evaluates both sides "outside" the signature, requiring a qualified path name to refer to any component of the signature. A seasoned ML programmer may find `revealing type S.t = int` ugly when compared with the more familiar `with type t = int`. And it would certainly be possible to change our proposal to be more consistent with ML. But we have two reasons to prefer our design. The first is a religious preference: we believe that the most appropriate way to use the equals sign is to elaborate left and right sides in the *same* environment, so a programmer can conclude that in the given environment, both sides mean the same thing.

The second reason is more substantial: by making both internal and external environments available on *each* side of the equals sign, we make `revealing type` expressive enough that it can subsume Standard ML's sharing constraints. As an example, Dave MacQueen provided a Standard ML version of the following signature:

```
signature WHERE_POWER =
  sig
    module A : sig
      type t
      type s
    end
    module B : sig
      val x : A.t
      type u
      type v = u list
    end
   (* SML: sharing type A.s = B.v *)
  end as S revealing type S.A.s = S.B.v
```

The two submodules `A` and `B` have a mutual dependency: value `B.x` depends on type `A.t`, and manifest type `A.s` depends on type `B.u`. This dependency can be seen in the denotation:

$$\begin{aligned}\langle D_S \ &= \{\texttt{structure}\ \varepsilon, \texttt{structure A}, \texttt{structure B}\},\\ D_A \ &= \{\texttt{type A.t} :: *, \texttt{type B.u} :: *\},\\ D_M \ &= \{\texttt{type B.v} = \texttt{B.u list}, \texttt{type A.s} = \texttt{B.u list}\},\\ D_V \ &= \{\texttt{val B.x} : \texttt{A.t}\}\rangle\end{aligned}$$

As the source-code comment shows, expressing this dependency in Standard ML requires a `sharing type` constraint. But a `sharing type` constraint simply equates two paths that are elaborated inside the signature, and because in our language the internals of the signature are available on both sides of the equals sign, we can express such constraints using only `revealing type`. The price is that we must use qualified names. For consistency, we use such names even for the `removing` modifier, which can never refer to anything outside the signature.

An alternative that might be more attractive to an ML traditionalist would be to define `revealing type` so that it would have the expressive power of our version without requiring qualified names everywhere. The critical requirement is that the signature's internals be accessible on the right side of the equals sign; our traditionalist could provide such access using special syntax while otherwise keeping the environments as they are now.

Families of signatures Why, if we wish to define an expressive language of signatures, do we not include a lambda-like parameterization mechanism, so a signature may be parameterized over a type, for example? In short, because we don't need to: the same ends can be achieved by putting an abstract type in a signature and later `revealing` that type. This design, called *fibration*, is preferable for other reasons (Harper and Pierce 2005, §§8.7–8.8). Using fibration, one need not identify certain abstract types as "parameters" in advance; as in the example where we refine `BARECODE`, one can simply modify an abstract type after the fact. Equally important, as explained by Harper and Pierce, fibration scales better when programs are built using deep hierarchies of functors.

Fibration of structures The `revealing module` modifier can be thought of as a sort of syntactic shorthand that applies `revealing type` to all the abstract types (D_A) of the signature being modified. In designing a semantics for `revealing module`, however, there are many choices what to do about other components. We begin by reviewing what happens in existing languages.

Objective Caml provides similar functionality in the form of `with module`. The only requirement stated in the manual is that given S `with module` p `=` p', each abstract type in the submodule at p must have a corresponding type component (abstract or manifest) in the module at p'. The implementation, however, appears to check consistency of other components of p, including values and manifest types. The module at p' may have additional value and type components that are not present at p.

Although it supports fibration via `where type`, Standard ML lacks a corresponding operation for structures. Such an operation has been added as an extension to Standard ML of New Jersey; documentation is thin, but experiments with the implementation show that it is more permissive than in Caml. In particular, manifest types need not be the same in modules at p and p'. Standard ML also has `sharing`, which as noted above, performs some of the same functions as `revealing`. The semantics of `sharing` has an interesting history.

When Standard ML was formalized in 1990, the semantics of `sharing` guaranteed identity of the two structures being shared. When Standard ML was revised in 1997, the notion of structure identity was dropped from the semantics, and a `sharing` constraint became syntactic sugar for a collection of `sharing type` constraints. The *Definition* (Milner et al. 1997, §G.3) notes that `sharing` was seldom used to enforce identity of values, and that dropping structure identity greatly simplified the semantics. Unfortunately, the expansion of `sharing` into `sharing type` has a property perhaps best regarded as a bug: it is defined only when the D_M components of both signatures are empty. Some compilers ignore this restriction, but because portable code must respect it, structure `sharing` is often inadequate to express requirements of complex Standard ML programs.

Influenced by the original definition of `sharing`, which enforced structure identity, we have chosen to impose strict requirements on `revealing module` $\tilde{p}$ `=` $\tilde{p}$'. When the denotations of the corresponding substructures are D and D', we consider all four SAMV parts:

- For abstract types in D_A, we have the same treatment as Objective Caml: each abstract type is made manifestly equal to the corresponding type in D', which may be abstract or manifest. We substitute accordingly in D_M and D_V.
- For D_S, D_M, and D_V, we enforce consistency with the corresponding components in D'. Here "consistency" means that both denotations have the same components, and after appropriate substitutions have been applied, corresponding types are equal.

These requirements can be expressed concisely using subtyping: $D' <: D$, and the domains of submodule, type, and value parts are equal. Our requirements are close to those of Objective Caml, but unlike Caml, we do not permit "extra" components in D'. Our requirements do not go so far as to enforce structure identity, but by being so restrictive, they increase the chances that `revealing module` on non-identical structures will cause a type error.

Sealing Unlike our other operations, `sealing` is grounded in theoretical considerations, not practical ones. Following Mitchell and Plotkin (1988), we believe that the appropriate way to make a manifest type abstract is to seal the type and its operations as a unit; only in this way can we decide what type should be given to each operation. This design makes `sealing` asymmetric with `revealing`; revealing the definition of an abstract type also reveals the types of all the operations, so the programmer has no decisions to make about the types of operations.

One alternative would separate a type from its operations and provide a more symmetric sealing operation that would simply make a manifest type abstract without changing the type of any operation. To make the types of some operations more abstract, a programmer could use `rebinding`. For example, if we call the symmetric sealing operation `hiding`, we might express the `sealing` example from Section 5.4 as

```
signature S' =
  S hiding    type S.t
    rebinding val S.a : int -> t
```

For two reasons, this alternative makes us uneasy:

- The alternative assumes that when we wish to make a type and a set of operations abstract, we are willing to write out new types for all the operations explicitly. This assumption reminds us of the current situation in ML, in which the way to write a signature is to write out all its components. Our preferred design makes it possible to seal with a pre-existing signature, or to seal multiple signatures with some common third signature. But because we have not been able to show that these possibilities are useful in real programs, this objection may not have much force.
- The symmetry of the alternative is more apparent than real. In particular, if a programmer were to reveal the definition of an abstract type, then hide it again using the alternative operation, any operations that originally used the abstract type would still use the revealed representation, so the result would not necessarily be the same as the original signature.

Conclusions on the merits of `sealing` await practical experience.

Combining signatures The most useful way we have found to combine signatures is with `andalso`: the result signature has all the components of both argument signatures. But signatures could also be combined in other ways. For example, one could consider taking an intersection of components. This operation, which might be called `orelse`, would have the property that a module M would implement the signature S `orelse` S' iff it implemented either signature S or signature S'. The `orelse` operation presents some technical challenges, and because we have not found a use for it, we have postponed work on it.

We also considered defining an operation analogous to set difference, so that given signatures S and S', it might be possible to find an S'' such that $S = (S\ \texttt{andalso}\ S')\ \texttt{minus}\ S''$. But in the interesting cases, S'' is not a complete signature, so we have not looked further into set difference.

Concrete syntax Regarding concrete syntax, we have borrowed existing ML forms and keywords where we could. In particular, we have used postfix modifiers to change the meaning of a signature; we have used `as` for pattern matching; and we have kept close to the standard `type`, `val`, and `module` syntax for components. Where no existing form or keyword seemed to serve, we have aimed for regular, orthogonal syntax by making each new keyword the gerund form of a verb. What we have *not* tried to do is keep new keywords to a minimum; for this reason, our concrete syntax may be ill suited to adoption in an existing language.

6.2 Extension to implementations

Modification of signatures is only a beginning; programmers will want to combine implementations as well as interfaces. Although we have not formalized operations on implementations, we believe that most of our language should extend nicely to implementations.

Preliminary observations Implementations include both structures and functors, but we would like to take modification of structures as fundamental. We hope to treat modification of functors by "sinking" any modification inside the functor, where it would be applied to the functor's body. Loosely speaking, we hope to define a functor modification m in terms of a structure modification m' such that $(\texttt{F}\ m)(\texttt{Arg})$ would be equivalent to $(\texttt{F}(\texttt{Arg}))\ m'$.

When we speak of "modifying" a structure, we are really speaking of modying the *interface* to that structure—the internals remain unchanged. In core-language terms, a structure modification might hide old bindings and add new ones, but internal references between existing components should never be affected. In implementation terms, if a structure is represented as a record of values, structure modifications should build new records from old, but no values should be changed.

After all functors have been applied, the denotation of a structure has no abstract types: signature ascription and functor application replace abstract types with fresh type constructors. Keeping this property in mind helps us think about modifying structures.

As in our language of signatures, we hope to help programmers reason about equality. But reasoning about equality of structures is more difficult: to decide when two signatures are the same, we need only reason about equality of types, but to decide when two structures are the same, we must reason about equality of values.

Applying our operations to structures Many of our operations should extend to structures without much thought, but there are a few cases where care should be taken. We consider each group of operations in turn.

Changing translucency Because a structure has no abstract types, only fresh type constructors, the operations that change translucency don't apply. But the `sealing` operation has a close analog: signature ascription, which can replace existing types with fresh type constructors. To reduce the notational burden in cases where most bindings are unchanged, we could extend `sealing` to structures. But if a structure already has an explicit signature, it is just as expressive to modify that signature using `sealing` and then use ascription with the result.

Looking for an analog of `revealing` would be a bad move, since it would break the abstraction imposed by ascription. We want not to support such things.

Adding, removing, and changing components Adding, removing, and moving (renaming) a structure's components appear to present no problems. Rebinding values is also simple. Rebinding types or modules would have to be restricted to avoid changing a type on which other types or values might depend.

Combining structures Combining structures with `andalso` should be very useful: several simple components could be merged to implement a complex absraction. Deciding *which* structures should be combinable exposes some interesting design choices. These choices arise when the two structures have components in common. When such components are types, we can require them to be equal, or if we want to exploit polymorphism, we can take the least upper bound of two types.

Values offer more choices. The choice most consistent with our philosophy is to require common value components to be equal, but if the type of such components is not an equality type, this requirement cannot be checked. If having value components in common is mostly accidental, we could ask the programmer to remove conflicting components before combining the structures. But if we think it is likely that combined structures will often have value components in common, and if those components are likely to include functions or other values that can't be compared for equality, it makes more sense to define `andalso` to be asymmetric (as it is in the term language) and to prefer value components from its left-hand argument.

7. Conclusion

Large software systems are built from modules, many of which may be provided and maintained by separate groups. The relationships among the signatures of these modules are important enough to be expressed formally, but existing languages are not expressive enough. We have designed a significantly more expressive language, proven that it is a conservative extension of ML, and shown its use in an extended example. We have also implemented the language and have mechanically checked the indented examples from this paper. In future work, we hope to make our language part of the Moby programming language (Fisher and Reppy 1999).

Acknowledgments

Jürgen Pfitzenmaier extended the implementation of Objective Caml to support an earlier version of our language. Greg Morrisett provided several helpful comments on an earlier draft. Matthew Fluet suggested several signatures intended to be combined with `andalso`. Anonymous referees contributed materially to the improvement of this paper; we are especially grateful to the referee who called our attention to issues with Standard ML `sharing` constraints. Several referees were unusually generous with their time and effort; we regret not being able to keep up with all of their suggestions. This work was supported in part by NSF grants CCR-0311482 and ITR-0325460 and by an Alfred P. Sloan Research Fellowship,

References

Matthias Blume and Andrew W. Appel. 1999 (July). Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847.

Karl Crary, Robert Harper, and Sidd Puri. 1999 (May). What is a recursive module? *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 34(5):50–63.

Derek Dreyer, Karl Crary, and Robert Harper. 2003 (January). A type system for higher-order modules. *Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages,* in *SIGPLAN Notices*, 38(1):236–249.

Kathleen Fisher and John Reppy. 1999 (May). The design of a class mechanism for Moby. *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 34(5):37–49.

Robert Harper and Mark Lillibridge. 1994 (January). A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 123–137.

Robert Harper and Benjamin C. Pierce. 2005. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8. MIT Press.

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

Roberto Ierusalimschy. 2003 (December). *Programming in Lua*. Lua.Org. ISBN 85-903798-1-7.

Xavier Leroy. 1994 (January). Manifest types, modules, and separate compilation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 109–122.

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

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

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

Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005 (September). An expressive language of interfaces — extended version. Technical report, Division of Engineering and Applied Sciences, Harvard University. To appear.

Claudio V. Russo. 2001 (October). Recursive structures for Standard ML. *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01),* in *SIGPLAN Notices*, 36(10):50–61.