Toward a Calculus of Signatures

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University

October 2, 2001

Abstract

The designers of languages such as Standard ML and Objective Caml have paid much attention to mechanisms for building modules from other modules. Both the underlying semantics and the expressive power have been well crafted. It is surprising then, that the the mechanisms used to build interfaces (signatures) from other interfaces are so impoverished. Signatures languages could be improved by borrowing ideas from the design of abstract data types, especially of sets. This note presents a few examples from real programs, which may convince readers that the shortcomings are real and that the proposed improvements really would help. The note closes with a brief syntactic proposal, which has not yet been fleshed out with semantics.

I have been using Standard ML and Objective Caml for some time now, and I love the sophisticated modules languages, which give me all sorts of wonderful tools I can use to create modules. But over time I have gotten grumpier and grumpier because these languages give me almost no tools to create interfaces to my modules. The lack of tools is especially grievous because the interfaces are the key to understanding any system, and these languages force me to write my interfaces in a way that is unnecessarily hard to understand. I hope to convince the ML community that it may be profitable to improve the situation by developing a more powerful calculus of signatures.

This note takes two approaches to signatures. Section 1, an argument from principle, treats the signatures language as a problem in the design of an abstract data type. It argues that signatures languages should borrow features used in the construction of sets. Section 2, an argument from practice, presents some awkard examples from real programs. These examples illuminate some shortcomings of existing signatures languages and provide some evidence that the ideas presented in Section 1 could help.

The note closes with an embryonic proposal for a new signatures language (Section 3). The proposal is embryonic because it contains only syntax, which has not yet been graced with semantics.

1 An argument from principle

What is a signature, anyway? We'll take a slightly simplified view of the presentation in Harper and Stone (2000); a signature is a collection of bindings. Names may be bound to type constructors, values, or nested structures (modules).

- If a name is bound to a type constructor, that binding assocates a *kind* with the type constructor. In addition, the binding may reveal or conceal a *representation* of that type constructor and the *identity* of that type constructor with some other type constructor. Revealing some representations may implicitly add other names to other spaces (e.g., datatype constructors), but I don't think that's important for our purposes.

  The details of type constructors are slightly different in Standard ML and Objective Caml; our view is a bit closer to the Caml view.

- If a name is bound to a value, the binding includes the type of the value.

- If a name is bound to a nested structure, the binding may reveal or conceal a *signature* for that nested structure.

Experience readers will recognize that a signature is not merely a collection of bindings; the internal dependencies between bindings are also important. We can, however, largely ignore this distinction until Section 3.

The design of abstract data types What can we learn about a calculus of signatures by thinking about the design of abstract data types? I have borrowed from work done in algebraic specification, as exemplified by Liskov and Guttag (1986). This work classifies the operations on an abstract data types t into four categories:

- A *primitive constructor* takes zero or more arguments that are not of type t and produces a new value of type t. The value `nil` is a primitive constructor for lists, for example.

- A *non-primitive constructor* takes arguments, at least one of which is of type t, and produces a new value of type t. The value `::` ("cons") is a non-primitive constructor for lists, for example.

- *Observers* take an argument of type t and produce a new value, which is not of type t. An observer "looks inside" its argument and produces some fact about the argument, or perhaps some constituent value. Observers are sometimes also called *selectors* or *accessors*. Functions `hd`, and `null` are observers for lists, for example. Perhaps surprisingly, `tl` is considered to be a constructor, not an observer. We call operations like `tl` "observing constructors;" such operations play a significant role in the design of a calculus of signatures.

- A *mutator* is an operation that mutates a value of type t; obviously, only mutable types have mutators. The value `:=` is a mutator for ref cells, for example.

These classifications lead to considerations for the design of abstract data types. Not all considerations are appropriate for every data type, but here are some examples:

- For every constructor C, the designer should consider providing observers (or perhaps observing constructors) that make it possible to recover from a value $C(x, y, z)$ the arguments x, y, z passed to C. For example, the observer `hd` and observing constructor `tl` make it possible to recover the arguments passed to `::`.

- The designer should consider providing a way to observe with which constructor a value was built. For example, the observer `null` makes it possible to distinguish lists built with `nil` from lists built with `::`.

- Constructors and observers form a term language, and for many abstract data types it is clear that only some of the terms in this language should be considered "values." For example, for lists, terms built with only `nil` and `::` should be considered values, but terms containing `tl` should not be considered values. An abstract data type should have a clear specification, perhaps a formal algebraic specification, that makes it possible to know what value any term in the language stands for, or at least shows how to rewrite terms into other terms. For example, the equation `hd (x :: xs) = x` might form part of such a specification for lists. This ability to rewrite terms motivates our use of the word "calculus."

- For mutable types, mutators and observers should correspond. For example, if the X protocol provides a way to mutate the background image displayed on the screen, it should also provide a way to observe the background image displayed on the screen.

Connections to signatures Many of the ideas mentioned above are at most peripherally relevant to a calculus of signatures. Mutable signatures would be antithetical to the spirit of ML, so we dismiss mutation. More surprisingly, the modules language has no observers for signatures—the only way to observe a signature is to feed it to a compiler. Although it might be interesting to consider observers in the signatures language, such questions are beyond the scope of this paper. We can therefore focus on two questions:

- What are the constructors used to create signatures?

- How can we know when two terms written using different constructors actually denote the same signature?

My thesis is that *the existing term languages used to construct signatures are not rich enough.* In particular, Standard ML and Objective Caml don't have "observing constructors" that would make it possible to undo the effects of other constructors. As a result, too many signatures in ML programs are written in a "syntactic value" style. As a rough analogy, imagine what your ML programs would be like if every list expression had to use `nil` explicitly, or the `[..., ...]` notation.

Ideas from sets A signature is not a completely new kind of thing; it is a set of bindings.[1] If we want a rich collection of constructors for signatures, perhaps we can get some ideas from the design of abstractions for sets or tables. For an absolutely minimal implementation of sets, we might get away with the two constructors *empty* and *insert*, the observer *isEmpty*, and the "observing constructor" *choose*, which takes one element from a nonempty set, returning that element and a set containing the remaining elements.

$$\begin{array}{lcl} \mathit{empty} & : & \alpha\ \mathit{set} \\ \mathit{insert} & : & \alpha \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \\ \mathit{isEmpty} & : & \alpha\ \mathit{set} \to \mathit{bool} \\ \mathit{choose} & : & \alpha\ \mathit{set} \to \alpha \times \alpha\ \mathit{set} \end{array}$$

Programmers would be much happier if we provided the following additional constructors:

$$\begin{array}{lcl} \mathit{delete} & : & \alpha \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \\ \mathit{union} & : & \alpha\ \mathit{set} \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \\ \mathit{intersection} & : & \alpha\ \mathit{set} \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \\ \mathit{difference} & : & \alpha\ \mathit{set} \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \\ \mathit{map} & : & (\alpha \to \beta) \to \alpha\ \mathit{set} \to \beta\ \mathit{set} \\ \mathit{filter} & : & (\alpha \to \mathit{bool}) \to \alpha\ \mathit{set} \to \alpha\ \mathit{set} \end{array}$$

Of course, for sets, we would also like a wide variety of observers, including such higher-order functions as *fold*, but since we are considering only constructors for signatures, we omit them.

Of the operations above, what can we find in the existing languages for signatures?

- We can write the empty signature as `sig end`.
- We can add a *binding* to a signature S, although in somewhat stylized form, as follows:

  ```
  sig
    include S
    binding
  end
  ```

- We cannot delete a binding, and we cannot take unions, intersections, or differences of general signatures. At best, we can take *disjoint union* of two signatures S and S', which must not bind any names in common:

[1] For now, we ignore issues related to dependencies between bindings.

```
sig
  include S
  include S′
end
```

Existing signature languages are clearly impoverished when compared to typical abstractions for sets.

A closer look at signatures We've been considering signatures as sets of bindings. While such sets are a reasonable denotation for signatures, they are not a great model for a calculus of signatures, because of dependencies. Any binding, whether of a type constructor, value, or signature, can depend on other type-constructor bindings within the same signature; when we write signatures, order matters. We should therefore keep in mind not only sets, but lists, and in particular, lists to which we add bindings *at the end*.

How do existing signatures languages compare with languages for writing lists? Most signatures appear to be written using the "explicit-value" notation `sig`… `end`, which is a list of explicit bindings. This notation has the same kinds of structural properties as the bracket-comma[2] notation used for lists in the core language. But the explicit-value notation used for signatures is strictly more powerful, because it offers `include`, which can be used to "splice in" bindings from other signatures. We can think of `include` as analogous to the `,@` splicing notation used in Lisp-like languages. While `include` is sometimes useful, it is not expressive enough. One goal of this paper is to try to discover if there are other, richer ways to write signatures, which might possibly eliminate the need for `include`.

Existing signature languages also provide the ability to *change* existing bindings in some limited ways. Primarily, it is possible to *raise* the level of information associated with some types in signatures, by using the `where type` or `with type` notations.

Caml provides an actual calculus, which I should repeat here. I'm not sure I understand the SML story exactly. Note that `with type` *cannot be used with functor signatures!*

Characterizing a better language Here we argue about what a better calculus of signatures might look like. We defer a detailed proposal to Section 3.

- It makes sense to retain the explicit-value notation `sig` $\{$*binding*$\}$ `end`.[3] This notation directly expresses its denotation; we could say that such signature terms are in normal form. Moreover, the notation is easy to read and is sufficient for most signatures used in practice. Appendix A shows that in many Standard ML programs, at least 90% of the signatures are written using this form.

- It clearly makes sense to be able to add bindings to and remove bindings from an existing signature. There are semantic constraints. It is easy to identify the constraints on adding bindings; adding bindings is safe if

[2] Objective Caml uses a bracket-semicolon notation for lists.

[3] The `include` syntax is not a *binding*.

and only if it would be safe to add them using the explicit-value notation. Deleting bindings is more interesting; in essence, it is safe to delete bindings if the resulting signature *subsumes* the original signature in the sense of Harper and Stone (2000). Therefore, it is always safe to delete a value binding. It is safe to delete a type binding if and only if no value bindings depend on the type binding. If is safe to delete a structure binding if and only if it is safe to delete the bindings of the types nested within that structure.

- We should be able not only to raise, but also to lower, the information associated with *any* type or signature binding. Such lowering is also related to subsumption.[4]

The operations above work with at most one existing signature, possibly by adding or removing bindings or other information. They are analagous to the *empty*, *insert*, and *delete* operations on sets. Because their arguments are a signature and a binding, we might call them the "signature–binding" operations. The "signature–signature" operations, which might be analogous to *union*, *intersection*, and *diff*, are more difficult to deal with. Of these, the most important appears to be union.

Existing languages provide *disjoint union*, which is more or less equivalent to taking all the bindings in one signature and adding them to another. Indeed, this is the effect of the `include` mechanism. By requiring the signatures involved to bind disjoint identifiers, we dodge semantic questions that must be addressed if we are to support arbitrary unions.

- Suppose there is a collision on a value binding (i.e., the two signatures bind values with the same name). If the types are different the result of including both bindings would be ambiguous, and it couldn't possibly denote a signature. If one binding is supposed to have priority, we could simply delete the unwanted binding before taking the union. If we want to preserve both bindings, we'll have to rename one. It's not clear whether an explicit "rename value" operation would be a good idea or whether we would be better off simply removing the old binding and inserting one with a new name.[5]

  If the two bindings have the same type, we could consider the union correct. To do so might give up some opportunities to detect errors in programs, however. I am agnostic on this point, but my gut feeling is to forbid.

- Suppose there is a collision on a type-constructor binding. This is a case where we want maximum flexibility. I know I want the ability to take the union of two signatures S and S' while arguing that type constructors $S.t$

[4] A third notion related to subsumption is polymorphic instantiation. It is not clear that an instantiating operation would be useful in a calculus of signatures, but completeness argues for having one.

[5] We argue below that orthogonality considerations militate for the rename operation.

and $S'.t$ should be considered the same. To exercise that ability, I expect to use some explicit notation, because I know I also want the ability to detect and recover from a situation in which the names of two types collide by accident. An appropriate notation might be something like `S union (S' where type t = S.t)`.

How do I detect and recover from accidents? Detection is easy; it should be a static error to take the union of two signatures if they each declare the same type constructor and if those type constructors are not known to be identical. To recover from such an accident, I want to be able to *rename* a type constructor within a signature. This renaming should change the name not only of the type constructor, but of *all the internal value, type-constructor, and signature bindings that depend on the renamed type constructor.*

- Suppose there is a collision on a structure binding. I want roughly the same power as for types, i.e., to detect and reject accidental collisions, to permit the union of signatures binding identical structures, and to rename structures to avoid collisions. I'm on shaky ground here because I don't know what is an appropriate definition of "identical structures." The Standard ML'97 idea appears to be to reduce a structure binding to a collection of type bindings, but I'm not entirely comfortable there.

To summarize, I argue that in addition to explicit bindings and operations for addition and removal, a calculus of signatures should provide support for unions.

- For unions to be useful, it must be possible to rename any value, type constructor, or structure within a signature.
- The language should permit non-disjoint unions if and only if the compiler can deduce that the common elements are identical. Such a deduction is straightforward for type constructors, possible for structures, and impossible for values. Therefore it will never be legal for two value bindings to collide in a union.

One question worth further exploration is whether there is a useful notion of least upper bound that could be applied to the semantics of a union operation.

Questions of type and structure identity also arise when we consider intersection and difference of signatures. Intersection and difference appear to present no additional problems, although we might ask if intersection takes a lower or upper bound of information associated with a type. This paper doesn't investigate the details of intersection and difference because in practice, they appear to be less important than union.

2 An argument from practice

This section presents a few motivating examples. My goal is to convince you both that we need *some* improvement to existing signatures languages and that the ideas discussed above may be useful in practice.

An interpreter for the embedded language Lua

Lua (Ierusalimschy, De Figueiredo, and Celes Filho 1996) is an embedded language and as such addresses the same problems as Tcl (Ousterhout 1990). I have recently written an implementation embedding Lua into Objective Caml. This implementation presents a couple of interesting problems, one of which is how to preserve extensibility, type safety, and separate compilation all in the same design. Other authors have wrestled with similar problems in the context of Haskell (Steele 1994; Liang, Hudak, and Jones 1995). My solutions to these problems are workable, but they expose several shortcomings of Objective Caml's signatures language. These shortcomings would be corrected by the proposals outlined in this note. The implementation problems are connected to the management of Lua types and libraries.

The most salient difference between Lua and Tcl is that Lua supports multiple types, including a "userdata" type, which makes it possible for users embedding Lua to extend the interpreter with new types. We're using Lua embedded inside a compiler, so our extended types are compiler-related: abstract-syntax trees, environments, control-flow graphs, target machines, and more. In the C implementation of Lua, all "userdata" is represented as pointers of type `void *`, and the implementation must use unsafe casts to convert these pointers as appropriate. For an implementation embedded in Caml, I wanted to preserve type safety, so I developed a fairly elaborate mechanism for making "userdata" a sum type.

The role of a Lua library is to provide access to parts of the program inside which the Lua interpreter is embedded. Users of the program can then control it by writing Lua scripts. For example, our compiler provides libraries that provide access to the register allocator and various other optimization passes, and when we're experimenting with the compiler, we can reconfigure it by putting suitable Lua code on the command line. The mechanism used by Lua libraries is to install values, including functions, into the name space of the Lua interpreter. Lua libraries are connected to Lua types because installed functions must be able to manipulate certain types. For example, optimization passes must be able to manipulate control-flow graphs.

To argue that there are shortcomings in existing signatures languages, I show a substantial portion of the signatures that Lua uses to formalize types and libraries.

Lua values I begin by showing a small part of the signature that defines Lua values. Lua builds in a number of types, including nil, numbers, strings, and so on. The `userdata` type provides extensibility; the `userdata'` type constructor is left abstract by the signature. In an interpreter, the type parameter to `userdata'` is always instantiated as `value`; using such a type parameter makes it possible for user-defined types themselves to contain arbitrary Lua values. This technique, which is also used in Steele (1994) and is called "two-level types" in Sheard (2001), deserves to be more widely known.

8 ⟨*lua.mli* 8⟩≡ 9 ▷

```
module type VALUE = sig
  type 'a userdata'
  type info
  type value                          (* type of a Lua value *)
    = Nil
    | Number   of float
    | String   of string
    | Userdata of userdata
    ...
  and userdata  = value userdata'
  type state = ...                    (* state of a Lua interpreter *)

  val eq        : value -> value -> bool
  val to_string : value -> string
  ...
end
```

The real `VALUE` signature is a very broad signature containing many value bindings, but most of them are irrelevant to this discussion, so we omit them.

The problem we are trying to solve is to combine many types, each of which may be defined in a separately compiled module, into a single `userdata'` type. It therefore makes sense to define a signature indicating what constitutes a separately compiled type. Here is the `USERDATA` signature in full:

9 ⟨*lua.mli* 8⟩+≡ ◁8 16b ▷

```
module type USERDATA = sig
  type 'a t             (* type parameter will be Lua value *)
  val tname : string    (* name of this type, for error messages *)
  val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
  val to_string : ('a -> string) -> 'a t -> string
end
```

The `eq` and `to_string` operations support the same operations on values, and the `tname` value helps with error messages.

Lua libraries—an example A Lua interpreter provides only a small number of operations built in; most functionality is provided by libraries. Lua comes with a string library, a math library, and an I/O library, and we have defined several libraries that provide access to parts of our compiler, such as the register allocator.

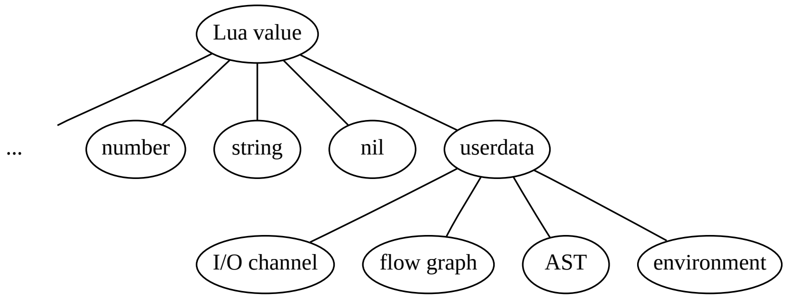


Figure 1: Types in one Lua interpreter

A Lua library has a very simple interface to the rest of the system; it provides a single function which, when a new interpreter is started, installs new functions and other values into the interpreter's state. To be able to define such functions, we must ensure that the Lua interpreter supports the types the functions need to manipulate. For example, the register-allocator can work only with an interpreter that supports control-flow graphs. It is a tricky business to get the types right, safely, while preserving separate compilation. It is easier to understand the problems if we show how to build an interpreter before moving on to the signatures that define the interfaces to the parts used to build interpreters.

Figure 1 shows one possible way of arranging types in a Lua interpreter. At the top of the figure is the type of Lua values, which is defined as type `value` in signature `VALUE` in chunk 8. Type `value` is a sum type, and it includes variants that represent numbers, strings, nil, userdata, and the other types defined in the `VALUE` signature. Figure 1 depicts these variants as children of the "Lua value" node. Because it is abstract in the `VALUE` signature, the userdata type can be different in different Lua interpreters. Figure 1 shows four variants of userdata: I/O channels, flow graphs, abstract-syntax trees (ASTs), and environments. These types are a subset of the types actually used in our compiler. As is standard with sum types, a value of any type in Figure 1 can be embedded into its parent type by applying an appropriate value constructor. For example, an I/O channel can be made into userdata, which in turn can be made into a Lua value. It is also possible to attempt a projection, which will fail if attempted with the wrong variant. For example, the register allocator can take an arbitrary Lua value and attempt to convert it to a flow graph. If this projection fails, the failure manifests as a run-time type error; Lua is a dynamically typed language.

The big question is where to get the embedding and projection functions used to embed values into and project values out of the `value` and `userdata` types. The `VALUE` signature provides a large collection of embedding/projection pairs, but we have omitted them from this paper because they are not relevant to our discussion of signatures. The interesting examples arise from our treatment of the `userdata` type, because we use the modules language to build both a sum type and suitable embedding and projection functions.[6] For example, here is code that might be used to create the interpreter in Figure 1:

11a ⟨*example code to build a Lua interpreter* 11a⟩≡ 11b ▷

```
module T = Lua.Lib.Combine.T4                             (* types *)
             (Luaiolib.T)                (* T1 *)
             (Driver2.Export.FlowGraphT) (* T2 *)
             (Driver2.Export.AstT)       (* T3 *)
             (Driver2.Export.EnvT)       (* T4 *)
```

The `Lua.Lib.Combine.T`n functors each combine n modules matching `USERDATA` to a new module that also matches `USERDATA`. The type `'a T.t` is a sum type with four variants: one for each argument module. The new module `T` does more than just define a type, however; it also provides embedding and projection functions that map between the n argument types and the resulting "combined" type `'a T.t`. In the example above, modules `T.T1` through `T.T4` provide these embedding/projection pairs.

Once we have defined type constructor `T.t`, which will be the combined `userdata'` type constructor in our final interpreter, we can instantiate and combine libraries. Here we have

- The I/O library, which needs a mapping to the I/O types
- The string and math libraries, which need no types beyond the built-in types
- The register-allocation library, which needs the flow-graph type
- The driver library, which needs flow graphs, abstract-syntax trees, and environments, but not I/O channels

11b ⟨*example code to build a Lua interpreter* 11a⟩+≡ ◁ 11a 12a ▷

```
module L = Lua.Lib.Combine.L5                        (* libraries *)
              (Luaiolib.Make (T.T1))
              (Stringlib)
              (Mathlib)
              (Colorgraph.Make (T.T2))
              (Driver2.Export.MakeLib (T.T2) (T.T3) (T.T4))
```

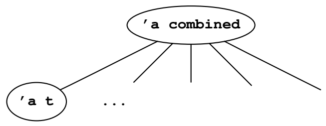
[6] An alternative would be to have users define a sum type explicitly and write their own embedding and projection functions. It might be acceptable to make authors of libraries write such code, but unfortunately such code would be needed to build any interpreter at all. We think this task would place an unacceptable burden on naïve users of the Lua system.

Module Lua.Lib.Combine.L5 combines these five libraries into a single library.

Finally, to build an interpreter, we supply three components: module T, which provides a userdata type; module L, which provides libraries; and module Lua.Parser.MakeStandard, which is the standard parser.

12a ⟨*example code to build a Lua interpreter* 11a⟩+≡ ◁11b
```
module I    = Lua.MakeInterp                    (* our interpreter *)
                (Lua.Parser.MakeStandard)
                (Lua.MakeCore (T) (L))
```
Having seen an example of how the parts of an interpreter are put together, we can now look at the signatures that define the interfaces between the parts.

Combining types in libraries, which shows a need for unions An essential operation in defining an interpreter is to combine the several types that make the userdata type. Functors such as Lua.Lib.Combine.T4 combine multiple types into single types, also providing mappings between the original and combined types. The mapping is provided by a "view" on each type. We might depict a view as follows:



View are defined by the signature TYPEVIEW, gives a combined ("parent") type, an individual ("child") type, and a mapping between them. The mapping, map, is an embedding/projection pair.[7]

12b ⟨*library support* 12b⟩≡ 14a ▷
```
module type TYPEVIEW = sig
  type 'a combined
  type 'a t  (* the individual type of which this is a view *)
  val map : ('a t -> 'a combined) * ('a combined -> 'a t)
end
```
If we combine four types into one, we get four type views, all sharing the same combined type. We describe these views using a signature, which we would like to write this way:

12c ⟨*code we wish we could write* 12c⟩≡ 13a ▷
```
module type COMBINED_VIEWS = sig
  type 'a t
  module type VIEW = TYPEVIEW with type 'a combined = 'a t
  module T1  : VIEW
  module T2  : VIEW
  module T3  : VIEW
  module T4  : VIEW
end
```
[7]The mappings in the real code are have more complicated Caml types, but we don't need the additional complexity to illustrate the problems with the signatures language.

But this signature alone is not good enough to build an interpreter. We cannot build an interpreter with just *any* combined type—because we need the `eq` and `to_string` operations, the combined type must match signature `USERDATA`. We therefore need to define a signature that subsumes both the `COMBINED_VIEWS` and the `USERDATA` signatures. This is a problem because both signatures define the type `'a t`, so with the existing signature languages we cannot combine them. We would like to write something like this:

13a ⟨*code we wish we could write* 12c⟩+≡ ◁12c 15b▷

```
module type COMBINED_TYPE =
  COMBINED_VIEWS union (USERDATA where type t = COMBINED_VIEWS.t)
     (* wishful thinking *)
```

To work around the lack of a union operation, we introduce an artificial `COMBINED_CORE` signature, which is part of both `COMBINED_VIEWS` and `COMBINED_TYPE`. In addition, to avoid a collision when we take the disjoint union of `COMBINED_CORE` and `USERDATA`, we have to introduce the awful `'a also_t` type.

13b ⟨*code we actually have to write* 13b⟩≡

```
module type COMBINED_CORE = sig
  type 'a also_t
  module type VIEW = TYPEVIEW with type 'a combined = 'a also_t
  module T1  : VIEW
  module T2  : VIEW
  module T3  : VIEW
  module T4  : VIEW
end
module type COMBINED_VIEWS = sig
  type 'a t
  include COMBINED_CORE with type 'a also_t = 'a t
end
module type COMBINED_TYPE = sig
  include USERDATA
  include COMBINED_CORE with type 'a also_t = 'a t
end
```

Because the signatures language has no union operation, our signatures are more complicated than we would prefer.

Defining libraries, which shows a need for expressive path names A Lua library works by installing new functions and values into an interpreter's state. The functions in a library require some interpreter support, e.g., the ability to signal errors, get access to global variables, etc. The full details of the interpreter core are unimportant; here are just a few bindings.

14a ⟨*library support* 12b⟩+≡ ◁12b 14b▷
```
module type CORE = sig
  module V : VALUE
  val error     : V.state -> string -> 'a  (* error fallback *)
  val getglobal : V.state -> V.value -> V.value
  val register_globals : (string * V.value) list -> V.state -> unit
    (* registers values as named global variables or functions *)
  ...
end
```

Because Lua is a dynamically typed, imperative language, Lua libraries work simply by mutating an interpreter's state, e.g., by installing values. A library therefore has a very simple signature.

14b ⟨*library support* 12b⟩+≡ ◁14a 15c▷
```
module type LIBRARY =
  functor (C : CORE) -> sig val init : C.V.state -> unit end
```

A library fulfills a promise of the following kind: you give me an interpreter core, and I'll give you an initialization function that takes interpreter state installs useful stuff in it.

Unfortunately, most libraries won't work with the `LIBRARY` signature, because the won't work with just any `CORE`—instead, they require that the `CORE` support certain types. For example, the I/O library requires the type of I/O channels, and the register allocator requires the type of flow graphs. Consider the I/O library. To map from a Lua value to an I/O channel, it needs to map first from the value to userdata, then from userdata to an I/O channel. The first mapping is standard in every Lua interpreter and is provided in the `VALUE` signature. The second mapping can be provided by a view in which the `combined` type is `userdata` and the individual type is I/O channel. In the example in chunk 11a, such a view could be found in module `T`, which matches signature `COMBINED_TYPE`. In fact, the view we want is in submodule `T.T1`, which is passed to `Luaiolib.Make` in chunk 11b. The functor returned by `Luaiolib.Make (T.T1)` will not work with just *any* `CORE`, but only with a `CORE` in which the type `'a userdata` is the same as `'a T.combined`. Let's consider how to express this constraint.

The Lua I/O library must define the type of I/O channel on which it depends, and it must provide a module matching signature `USERDATA` with which to build the combined type (`Luaiolib.T` in chunk 11a).

14c ⟨*interface for the Lua I/O library* 14c⟩≡
```
type 'a t = In  of in_channel
          | Out of out_channel
module T : Lua.USERDATA  with type 'a t = 'a t
```

The Lua I/O library must also export a functor which, given a view of type `t`, plus a core interpreter, produces an initialization function. As noted, the user-data type of the core interpreter has to be the combined type of the view. If we expand everything out, it is not difficult to write a suitable signature:

15a ⟨*fully expanded interface for the Lua I/O library* 15a⟩≡
```
module Make :
  functor (T : Lua.Lib.TYPEVIEW with type 'a t = 'a t) ->
  functor (C : CORE with type 'a V.userdata' = 'a T.combined) ->
  sig val init : C.V.state -> unit end
```

If, however, we want *not* to expand the signature by hand, but to use the existing `LIBRARY` signature, we run into difficulties. The problem is that the existing `LIBRARY` signature puts no constraints on `'a C.V.userdata'`, and the current signatures language provides no way to to constrain it. We might wish to write something like the following:

15b ⟨*code we wish we could write* 12c⟩+≡ ◁13a
```
module Make :
  functor (T : Lua.Lib.TYPEVIEW with type 'a t = 'a t) ->
  LIBRARY with type 'a arg.V.userdata' = 'a T.combined
```

But the `arg.` syntax is something we made up on the spot; there isn't really a way to modify a type binding that appears in the argument or result of a functor.

It is possible to work around this omission by pairing a functor with an abstract type, in such a way that the abstract type can be named and modified. For Lua libraries, we can write the following:

15c ⟨*library support* 12b⟩+≡ ◁14b 16a▷
```
module type TYPEFUL_LIBRARY = sig
  type 'a combined  (* the combined type of all the views on which lib depends *)
  module M : functor (C : CORE with type 'a V.userdata' = 'a combined) -> sig
    val init : C.V.state -> unit
  end
end
```

Now we can write the I/O library:

15d ⟨*code we actually write for the I/O library* 15d⟩≡
```
module Make (T : Lua.Lib.TYPEVIEW with type 'a t = 'a t)
    : Lua.TYPEFUL_LIBRARY with type 'a combined = 'a T.combined
```

This workaround is acceptable, but it is uglier than we would like, and it complicates the code for linking libraries together; both importers and exporters of libraries have to deal with the (otherwise superfluous) nested module.

Combining functors for types and libraries Although it is not part of our argument for a richer calculus of signatures, it helps clarify our example to present part of the interface of module `Combine`, which provides functors for combining up to 10 types and up to 10 libraries.

16a ⟨*library support* 12b⟩+≡ ◁15c

```
module Combine : sig
  module T4 (T1 : USERDATA) (T2 : USERDATA) (T3 : USERDATA) (T4 : USERDATA)
   : COMBINED_TYPE with type 'a T1.t = 'a T1.t and type 'a T2.t = 'a T2.t
                   and type 'a T3.t = 'a T3.t and type 'a T4.t = 'a T4.t
  module L5 (L1 : TYPEFUL_LIBRARY)
    (L2 : TYPEFUL_LIBRARY with type 'a combined = 'a L1.combined)
    (L3 : TYPEFUL_LIBRARY with type 'a combined = 'a L1.combined)
    (L4 : TYPEFUL_LIBRARY with type 'a combined = 'a L1.combined)
    (L5 : TYPEFUL_LIBRARY with type 'a combined = 'a L1.combined) :
    TYPEFUL_LIBRARY with type 'a combined = 'a L1.combined
  ...
end
```

Because the result of combining libraries is itself a library, users can combine arbitrarily many libraries by nesting the combining functors. Similarly, because `COMBINED_TYPE` includes `USERDATA`, we can also nest type-combining functors. Our Caml code also includes a functor to help compose the appropriate embedding/projection pairs.

Interpreters versus interpreter cores—an argument for removal A Lua interpreter embodies a particular combination of types and libraries. It provides a broad application program interface (API) to the Caml programmer, of which the ability to read and interpret Lua code is just a part. The API is expressed as a signature called `INTERP`. Unfortunately, because of the way the interpreter is bootstrapped, the `INTERP` signature has to be split into two pieces. The first piece, `INTERP_CORE`, provides basic functionality, plus a "compiler" that takes some abstract syntax and produces a function. Here's a small fraction of the interface:

16b ⟨*lua.mli* 8⟩+≡ ◁9 17a▷

```
module type INTERP_CORE = sig
  module Value : VALUE
  module Ast : AST with module Value = Value
  type state = Value.state
  type value = Value.value
  val state : unit -> state  (* produce a fresh, initialized state *)
  val compile : Ast.chunk list -> (state -> value list)
  ...
end
```

A full-blown interpreter, with signature INTERP, provides everything in an interpreter core, plus the ability to run Lua code in strings or files. To run this code, it needs a parser, which is also exposed as part of the API.

17a ⟨*lua.mli* 8⟩+≡ ◁16b 17b▷
```
module type INTERP = sig
  include INTERP_CORE
  module Parser : Luaparser.S with type chunk = Ast.chunk
  val dostring  : state -> string -> value list
  val dofile    : state -> string -> value list
end
```

Since INTERP is the API that is used by most clients, we would prefer to define INTERP using explicit values, then derive the INTERP_CORE signature by removing the bindings for Parser, dostring, and dofile.

For completeness, here are the signatures of the functors we use in chunk 12a to build an interpreter.

17b ⟨*lua.mli* 8⟩+≡ ◁17a
```
module MakeCore
    (T : Lib.COMBINED_TYPE)
    (L : Lib.TYPEFUL_LIBRARY with type 'a combined = 'a T.t)
    : INTERP_CORE with type 'a Value.userdata' = 'a T.t
module MakeInterp (MakeParser : Parser.MAKER) (I : INTERP_CORE)
    : INTERP with module Value = I.Value
```

Summary of findings based on Lua The examples above expose a number of shortcomings in Objective Caml's language for describing signatures.

- Because it is not possible to take the union of two signatures that define a type with the same name, our definition of the COMBINED_VIEWS and COMBINED_TYPE signatures (chunk 13b) is awkward, especially in its use of the type 'a also_t.
- Because it is not possible to give an equality or a representation for an abstract type that appears as an argument to a functor, we resort to an extra level of nesting and indirection in the definition of the TYPEFUL_LIBRARY signature (chunk 15c).
- Because it is not possible to define a signature by removing bindings from another signature, we have to expose our bootstrapping technique, as realized in signatures INTERP_CORE and INTERP, to programmers that will never write a library of their own, but will only write clients (chunk 17a).

These examples provide an argument from practice that improvements in signatures languages are needed. They apply with equal force to the signatures languages of Objective Caml and Standard ML.

3 A modest proposal

This section proposes a syntax for a new calculus of signatures. I had hoped also to produce a semantics by giving rewriting rules, in classic calculational style, but I'm caught up on some snags having to do with the environments in which type expressions are elaborated. I'm circulating this note anyway because I'm trolling for potential coauthors who might be interested in this problem.

sig ⇒ `sig` {*binding*} `end`
  | `functor` (*id* `:` *sig*) `->` *sig*
  | *id*
  | *sig* `adding` {*longbinding*} `end`
  | *sig* `removing` {(`type`|`val`|`structure`|`module`) *pathname*} `end`
  | *sig* `with` (`type`|`val`|`structure`|`module`) *pathname* `as` *pathname* [`and` ...]
  | *sig* `with type` *tyexp* `=` (`abstract`|*tyexp*) [`and` ...]
  | *sig* `union` *sig*   # maybe also `inter` and `diff`?

N.B. *pathname* and *longbinding* must be able to name functor arguments and results, not just nested modules. This is an unsolved problem. In Caml, since `arg` and `result` can't name modules, we might use those names.

The proposal deliberately omits `include`, since `union` and `adding` ought to replace all uses of `include`. Although it is tempting to think about parameterization at the signature level, we omit that case as well—although mappings from signatures to signatures may well be interesting, they are not signatures, and one hopes they could be considered independently from this proposal.

There are a variety of ways to think about the problem of semantics. In general, one could define denotations of the terms above, define denotations of modules, define a module-level signature-matching algorithm, and show the signature matching to be sound with respect to the denotations. This would be a bad plan. It would repeat lots of work already done by others, and it would be unlikely to attract the attention of those who write ML compilers.

A better plan would be to find a way to translate the *sig* terms into a normal form. This normal form might itself be expressible as a *sig* term (thereby justifying the phrase "calculus of signatures"). It appears, however, that the normal form defined by Harper and Stone (2000), or a variation, might be more promising. In any case, we would like to pick a normal form for which someone else has already done the hard work of showing that signature matching is sound. If the normal form is well chosen, this plan should make it easy for compiler writers to adopt our proposal, simply by replacing that part of the compiler that translates signatures into internal form.

To finish this paper will therefore require

- Choosing a normal form
- Giving elaboration rules to map *sig* terms into normal form
- Perhaps showing some sort of consistency

Please let me know if these problems interest you.

References

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

Ierusalimschy, Roberto, Luis H. De Figueiredo, and Waldemar Celes Filho. 1996 (June).
Lua — an extensible extension language.
Software—Practice & Experience, 26(6):635–??

Liang, Sheng, Paul Hudak, and Mark Jones. 1995.
Monad transformers and modular interpreters.
In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 333–343. ACM SIGACT and SIGPLAN, ACM Press.

Liskov, Barbara and John Guttag. 1986.
Abstraction and Specification in Program Development.
MIT Press / McGraw-Hill.

Ousterhout, John K. 1990 (January).
Tcl: An embeddable command language.
In *Proceedings of the Winter USENIX Conference*, pages 133–146, Washington, DC.

Sheard, Tim. 2001 (September).
Generic unification via two-level types and parameterized modules.
In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 86–97, New York, NY.

Steele, Guy Lewis, Jr. 1994.
Building interpreters by composing monads.
In ACM, editor, *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 472–492, New York, NY, USA.

A Statistics on usage of include

The following table gives a *very* rough idea of how often `include` is used in Standard ML programs. In September, 2001, I appealed to the Usenet newsgroup `comp.programming.literate` for statistics on the use of `include` in signatures. The table presents the results. It lists the number of *named* signatures and the number of instances of `include` in the sources. The "*Min w/o inc*" column gives the ratio of the number of includes to the number of signatures, which (under)estimates the fraction of signatures that have no include.

Program (Source)	*Sigs*	*Includes*	*Min w/o inc*
CIL (Dimock)	124	4	0.97
Edinburgh Concurrency Workbench (Stevens)	64	13	0.80
Egregious (Scaife)	268	21	0.92
Foxnet (Swasey)	350	137	0.61
MLton (Weeks)	363	208	0.43
MLRISC (Reig/George)	359	30	0.92
SLED (Ramsey)	104	13	0.88
TILT (Swasey)	292	20	0.93
Twelf 1.2R5 (Swasey)	105	4	0.96
Twelf (Pfenning)	125	12	0.90

A note of explanation may be in order. Stephen Weeks, who provided data for MLton, uses a style in which a functor is always specified by two named signatures, one for the functor argument and one for the functor result. The functor result signature almost always includes the functor argument so that the signatures will be closed.