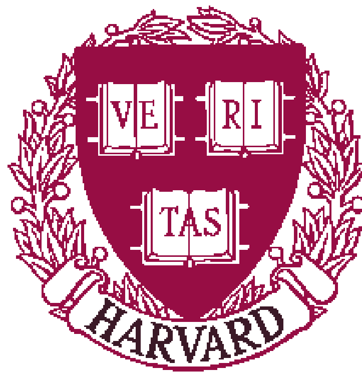
ML Module Mania: A Type-Safe, Separately Compiled, Extensible Interpreter

Norman Ramsey

TR-11-05



Computer Science Group
Harvard University
Cambridge, Massachusetts

ML Module Mania: A Type-Safe, Separately Compiled, Extensible Interpreter

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University

Abstract

The new embedded interpreter Lua-ML combines extensibility and separate compilation without compromising type safety. The interpreter's types are extended by applying a sum constructor to built-in types and to extensions, then tying a recursive knot using a two-level type; the sum constructor is written using an ML functor. The interpreter's initial basis is extended by composing initialization functions from individual extensions, also using ML functors. Lua-ML adds a detailed example to the debate over how much power is needed in a modules language.

1 Introduction

This paper solves a problem in the construction of interpreters: how to combine extensibility with separate compilation in a safe language. The solution makes essential use of ML modules. It uses functors (parameterized modules) not only in ways that are supported in all dialects of ML, but also in ways that require functors to be first-class, as they are in the modules language of Objective Caml (Leroy 2000). Powerful modules languages are objects of ongoing discussion in the functional-programming community. For example, Peyton Jones (2003b) recently likened the ML modules system to a Porsche, calling it "powerful, but breathtakingly expensive." To what degree such power is needed is a matter of debate. This paper uses the power to solve an open programming problem, adding an example to that debate.

We focus on a kind of interpreter for which extensibility and separate compilation are especially important: the *embedded* interpreter. Embedding is motivated by the problem of controlling a complex application, like a web server or an optimizing compiler, that is written in a statically typed, compiled language such as C, C++, or ML. Such an application will have lots of potential configurations and behaviors. How are you to control it? If you use command-line arguments, you may find yourself writing an interpreter for an increasingly complicated language of command-line arguments. If you use a configuration file, you will find yourself defining, parsing, and interpreting its syntax.

A better idea, which we owe to Ousterhout (1990), is to create a *reusable* language designed just for configuring and controlling application programs, i.e., for *scripting.* Making a scripting language reusable means making it easy to include its interpreter in an application, so the application can invoke the interpreter and the interpreter can call code in the application. Such an interpreter is called *embedded.*

An application that uses an embedded interpreter is written in two languages: Most code is written in the original, *host* language (e.g., C, C++, or ML), but key parts can be written in the embedded language. This organization has several benefits:

- Complex command-line arguments aren't needed; the embedded language can be used on the command line.
- A configuration file can be replaced by a program in the embedded language.
- It is easy to write an interactive loop that uses the embedded language to control the application.

The benefits above were demonstrated by Tcl (Ousterhout 1990), which was followed by embedded implementations of other languages, including Python, Perl, and several forms of Scheme (Benson 1994; Laumann and Bormann 1994; Jenness and Cozens 2002; van Rossum 2002), as well as by another language designed expressly for embedding: Lua (Ierusalimschy, de Figueiredo, and Celes 1996a).

Why have two languages? Can't the host language be used for scripting? Sometimes it can; for example, Leijen and Meijer (2000) show how to encode embedded-language terms as host-language terms, even to the extent of using the host-language type checker to enforce a static type discipline on the embedded language. But this technique requires that the host-language compiler translate phrases in the embedded language, and in many cases it is inconvenient or even impossible to make a host-language compiler available at run time. Aside from such considerations of implementation, some people would argue that two languages are needed because a language that is well suited for building large applications is inherently ill suited for scripting (Ousterhout 1998). We don't pursue that argument; we simply take for granted that there are two languages.

What are the requirements for a scripting language and its interpreter?

1. They must be extensible: The whole point is to add application-specific code and data to the scripting language, which we call *extension.*
2. The interpreter should be compiled separately from the application. In particular, it should be possible to compile an application-specific extension without using or changing the interpreter's source code. In other words, the interpreter should be isolated in a library.
3. The combination of application and scripting language should be type-safe, and this safety should be checked by the host-language compiler.

This paper presents Lua-ML, which to my knowledge is the first embedded interpreter to meet all three of these requirements. Lua-ML's API makes it possible to embed a Lua interpreter into an application written in Objective Caml. Lua-ML uses Objective Caml's modules language to compose the Lua-ML interpreter with its extensions.

At present, the primary application of Lua-ML is to script and control a nascent optimizing compiler for the portable assembly language C-- (Ramsey and Peyton Jones 2000). The compiler, which is roughly 25,000 lines of Objective Caml, uses about 1,000 lines of Lua to configure back ends and to call front ends, assemblers, linkers, and so on.

2 Related work

Prior work using C to implement embedded languages has produced interpreters that are extensible and separately compiled but not type-safe. Prior work using functional languages to implement modular interpreters has produced interpreters that are extensible and type-safe but not separately compiled. Lua-ML draws on both bodies of work, so we begin with a review.

In designs that are not type-safe, type safety is lost when values are converted between their host-language and embedded-language forms. We therefore pay special attention to the two forms of conversion: converting from host value to embedded value, which is called *embedding*, and converting from embedded value to host value, which is called *projection*.

2.1 Embedded languages and interpreters

Tcl The first language designed to be embedded in many applications was Tcl (Ousterhout 1990; Ousterhout 1994). Tcl is unusual in having no notion of type, not even a dynamic one: Every value is a string. A string may represent itself, a list of strings, the name of a hash table, the name of a private data object, some concrete syntax to be evaluated, or even a syntactic keyword. The meaning of any particular string depends on the context in which it appears.

Tcl is extended by adding a new *command* to an interpreter. A command is a C procedure that takes a list of strings, has side effects, and produces a string result. The command determines the meaning of each of its argument strings. For example, the `if` command treats one string as an expression to be evaluated, treats `then` and `else` as syntactic keywords, and treats two other strings as (lists of) commands to be evaluated. In addition to a list of strings, a command also receives a pointer to an interpreter and a value of type `ClientData`. This value may point to private data that is associated with the command; in this way, a Tcl command can simulate a closure or methods of an object.

Tcl's API enables an application to add new commands to an interpreter, evaluate Tcl code on demand, and so on. Adding a new command requires supplying not only an appropriate C procedure but also a value of type `ClientData`.

Tcl provides limited support for embedding and projection. The API provides functions to project a Tcl value (which is always a string) to an integer, a floating-point number, a Boolean, or a list of strings. The API provides no support for embedding; applications use standard C functions such as `sprintf`.

It is not possible to add an application-specific type to Tcl; extensions must make do with strings. Ousterhout (1994) recommends two ways of extending a Tcl interpreter with application-specific data:

- A pointer to the application-specific data can be associated with a Tcl command, which gets access through its `ClientData` parameter. Because the association is maintained by the Tcl implementation, it is relatively easy to get right, but the implementation of the command still requires an unsafe cast to project a value of type `ClientData` to a value of application-specific type.
- The application-specific data can be kept in a hash table or other data structure. One or more commands may keep a reference to the hash table and may interpret any string as a key in that table. This technique can be made type-safe, but it is more difficult to get right, because a Tcl value does not carry any sort of run-time type tag. It is up to the programmer of each command to make sure that strings are interpreted correctly.

Lua Like Tcl, Lua is a language that is designed expressly for embedding (Ierusalimschy, de Figueiredo, and Celes 1996a; Ierusalimschy, de Figueiredo, and Celes 2001). The Lua language, its library, and its API for embedding have all undergone considerable evolution. Lua-ML implements the Lua language version 2.5, which is described by Ierusalimschy, de Figueiredo, and Celes (1996b). Version 2.5 is relatively old, but it is mature and efficient, and it omits some complexities of later versions. The most recent version as of this writing is Lua 5.0, which was released in Spring of 2003; I mention differences where appropriate.

Lua is a dynamically typed language with six types: nil, string, number, function, table, and userdata. Nil is a singleton type containing only the value nil. A table is a mutable hash table in which any value except nil may be used as a key. Userdata is a catchall type, which enables an application program to add new types to the interpreter. Except for table, the built-in types are immutable; userdata is mutable at the application's discretion. Lua 5.0 adds two new types: Boolean and thread.

Lua is extended by adding a new value to the interpreter. A value may be a function; recent versions of Lua include first-class, nested functions in various forms. Lua's API also gives application code the ability to bundle a function with values to form a closure. Lua's API differs from Tcl's in that Lua values do not escape: a Lua value can be manipulated only when on the Lua stack, which is an abstraction provided by the API. This design simplifies garbage collection of Lua values.

Like Tcl's API, Lua's API enables an application to add new values to an interpreter, evaluate Lua code on demand, and so on.

Lua's API provides an embedding function and a projection function for each of its six types. Through its userdata type, Lua also provides support for embedding and projection of application-specific types: a userdata value is represented by a C pointer of type `void *` and a small-integer *tag*, which identifies the true C type of the pointer. The Lua API contains support for allocating a unique tag. Instead of a tag, Lua 5.0 associates each userdata value with a table; such a table is called a *metatable*.

The purpose of userdata is to enable a programmer to add an application-specific type to Lua. The type must be a pointer type. To add a type, an application allocates a unique tag (or metatable) for the type and represents a value of the type as userdata with this tag. This technique requires a small amount of unsafe code, but such code can be isolated in a couple of C procedures. An example appears in Appendix A.

2.2 Extensible interpreters

Lua-ML is inspired in part by Steele's (1994) beautiful paper on building interpreters by composing pseudomonads. Steele follows an agenda set by Wadler (1992), which is to use monads to express various language features that may be implemented in an interpreter. An "extension" may include not only a new type of value but also new syntax, new control flow, new rules for evaluation, or other new language features. Lua-ML is much less ambitious: as with Lua, an interpreter's syntax, control flow, and rules for evaluation cannot be extended; the only possible extensions are to add new types and values. We are interested in the mechanism used to add new types.

Steele's interpreter is built using a "tower" of types. In such a tower, an extension is defined using a type constructor of kind $* \times * \Rightarrow *$. Although Steele's interpreter is written in Haskell (Peyton Jones 2003a), we use Objective Caml for examples. For example, one might define an extension for arbitrary-precision rational arithmetic using the type constructor `arithx`:

```
type ('value, 'next) arithx =
  | Bignum of Big_int.big_int
  | Ratio  of 'value * 'value
  | Other  of 'next
```

The type constructor `arithx` represents one level of the tower. The type parameter `'next` represents the next level down, and the type parameter `'value` represents the (eventual) top of the tower. Thus, the extension above defines a value at the `arithx` level to be either an arbitrary-precision integer, a ratio of two values, or a value from the next level down.

In a tower of types, embedding and projection are implemented by composing functions that move up and down the tower. Each such function is simple; for example, a value from the level below `arithx` might be embedded by the function `fun v -> Other v`, and a value from the `arithx` level might be projected downward by the function `function Other v -> v | _ -> raise Projection`.

Building a full tower of types requires linking multiple levels through the `'next` parameter, then tying the knot with a recursive definition of `value`, in which `value` is used as the `'value` parameter. The use of a type parameter to tie a recursive knot is called *two-level types* by Sheard (2001). This technique deserves to be more widely known.

As an example, here is a very simple tower built with two levels: `void` (an empty type) and `arithx`. Tying the knot requires a recursive definition of `value`:

```
type void  = Void of void          (* no values *)
type value = (value, void) arithx (* illegal *)
```

Unfortunately, this definition of `value` is illegal in both ML and Haskell: In these languages, a recursive type definition is permitted only if the type in question is an algebraic data type, and this fact is not evident to the compiler. Steele solves this problem by using a program simplifier, which reduces the tower of types to a single recursive definition that is acceptable to a Haskell compiler. (The simplifier also eliminates the indirection inherent in the use of such value constructors as `Other` above.) Using a simplifier eliminates any possibility of separate compilation, because the simplifier performs what amounts to a whole-program analysis.

Liang, Hudak, and Jones (1995) also build interpreters by composing parts, but they use monad transformers, not pseudomonads. Again we focus on the definition of types. Liang, Hudak, and Jones use no type parameters.

- In place of Steele's `'value` parameter, they use mutually recursive type definitions—there are no two-level types.
- In place of Steele's `'next` parameter, they use a type constructor for *extensible unions*. This type constructor plays a role analagous to that of a cons cell in building lists in functional languages: It is applied to types in a union and is not part of either type. By contrast, Steele's `'next` parameter plays a role analagous to that of a linking pointer stored inside a heap-allocated structure in C: It is part of the definition of each type.

The extensible-union constructor is known as `Either` in Haskell 98 (Peyton Jones 2003a); in the earlier work it is called `OR`. In Objective Caml it could be written

```
type ('a, 'b) either = Left of 'a | Right of 'b
```

The extensible-union constructor simplifies the definition of types at each level, because a value constructor like `Other` is no longer necessary.

The example above could be written

```
type value  = (arithx, void) either
and  arithx =
  | Bignum of Big_int.big_int
  | Ratio  of value * value
and void = Void of void
```

(This style is even more convenient in Haskell, because mutually recursive definitions may be spread over a module with other definitions intercalated between them.) The `'value` parameter has been dropped; instead the `Ratio` constructor refers directly to the `value` type. Because mutually recursive types must be defined in a single module, this design sacrifices separate compilation for convenience.

Liang, Hudak, and Jones define embedding and projection functions using a multiparameter type class, which overloads the functions `embed` and `project`. (In their work, these functions are called `inj`, for "inject", and `prj`, for "project.") Suitable instance declarations automate the composition of these functions for types built with the extensible-union constructor.

Lua-ML borrows ideas from all of these sources.

- Like Tcl and Lua, Lua-ML provides an embedded interpreter as a separately compiled library.
- Like Lua, Lua-ML limits its extensibility to new types and values; the syntax and evaluation rules never change.
- Like Steele's interpreters, Lua-ML uses two-level types to create a recursive definition of `value`.
- Like Liang, Hudak, and Jones's interpreters, Lua-ML uses an external constructor to combine building blocks of different types. But instead of using a simple type constructor, Lua-ML uses a *functor* in the ML modules language.

The rest of this paper describes in detail what a Lua-ML extension looks like and how extensions are composed with Lua-ML's modules to produce a complete, extended interpreter. An ambitious example appears in Section 4.

3 Extension using libraries

Lua-ML respects the idioms of Lua, in which the unit of extension is the *library*. Lua comes with libraries for mathematics, string manipulation, and I/O. Application programmers use these libraries as models when designing their own extensions.

A library may perform up to three tasks:

1. Every library defines additional *values* (usually functions) that are installed in an interpreter at startup time. These values may be stored in global variables, in tables that are global variables, and so on. They become part of the initial basis of Lua. For example, the Lua I/O library defines a function `write`, which performs output.
2. A library may define additional *types* of userdata. For example, the Lua I/O library defines a type representing an "open file handle."
3. A library may define additional mutable *state* for the interpreter. Such state may be exposed through Lua variables, or it may be hidden behind Lua functions. For example, the Lua I/O library defines a "current output file," which is an open file handle that `write` writes to.

In C, a Lua library is hidden behind a single function that installs Lua values in an interpreter, acquires tags for userdata, and initializes mutable state. For example, the Lua I/O library is hidden behind the function `lua_iolibopen`.

Lua-ML uses Lua's model of libraries, but the techniques used to express a library are different. In Lua-ML, we define libraries using ML modules. Relating the signatures of these modules to the tasks that libraries perform is one of the fine points of the design.

3.1 Signatures for libraries

Every library adds new values to an interpreter (task 1), but adding new types (task 2) and new state (task 3) are optional. Depending on which options are exercised, there are four kinds of library. There are designs in which each kind of library has its own signature, but these designs have two defects:

- Four signatures is too many, especially if we want libraries to be composable—the obvious composition scheme uses sixteen functors.
- It is not obvious how to manage libraries that *share* types or state.

In a complex application, sharing types is commonplace. For example, our optimizing compiler defines a type that represents a control-flow graph. This type is shared among the libraries for each back end, for the register allocator, and for optimization. State, by contrast, is seldom used and rarely shared. These issues are discussed in more detail in Section 5.

Instead of putting one library in one module and needing a distinct signature for each kind of library, Lua-ML *splits* a library into multiple modules. The definition of a new type (task 2) appears in a *type module*, which matches the `USERTYPE` signature. Definitions of new values, functions, or state (tasks 1 and 3) appear in a *code module*, which matches the `USERCODE` or `BARECODE` signature. These signatures are explained in Section 3.3.

If a code module depends on the presence of a particular type, Lua-ML represents the code module as an ML functor: Such a functor produces a result that matches `USERCODE`, and as arguments it takes one or more *views* of type modules. A view matches signature `TYPEVIEW` and is also explained in Section 3.3. If two or more code modules share a type, the sharing is expressed by applying them to the same view.

Because state is rarely shared, Lua-ML does not provide a similar mechanism to share state. Instead, if state is shared among two or more libraries, that state must be stored in a global Lua variable, which makes it accessible to *all* libraries and to Lua code in general. Such state can be protected from unwanted mutation by giving it an abstract type and by permitting only certain libraries to depend on the type. If state is private to a single library, which is the common case, it can be hidden behind one or more functions in that library. In other words, it can appear as one or more free variables of those library functions.

Type modules and code modules are examples of what Batory and O'Malley (1992) call *symmetric components*: Multiple type modules can be composed to form a new type module, and multiple code modules can be composed to form a new code module. This compositional technique was also used to good effect in the TCP/IP protocol stack developed for FoxNet (Biagioni et al. 1994). By exploiting composition, we can, if we like, define a library to be a pair consisting of one type module and one code module.

3.2 Elements of the design

Value and state Both a Lua value and the state of a Lua interpreter are represented as values in Objective Caml. A Lua interpreter must include a sub-module that matches the `VALUE` signature. An abbreviated version of this signature is

```
module type VALUE = sig
  type 'a userdata'
  type srcloc (* a source-code location *)
  type value =
    | Nil
    | Number   of float
    | String   of string
    | Function of srcloc * func
    | Userdata of userdata
    | Table    of table
  and func     = value list -> value list
  and table    = (value, value) Luahash.t
  and userdata = value userdata'
  and state    = { globals : table
                   (* other fields omitted *)
                 }
  val eq        : value -> value -> bool
  val to_string : value -> string
  ...
end
```

Type constructor `userdata'` is a two-level type that is formed by composing libraries. The type parameter represents a `value`, as you can see from the definition of `userdata`, where the recursive knot is tied. By this mechanism, the `value` type can be extended by libraries. The `state` type, by contrast, cannot be extended.

As an example of a type constructor that could be used as `userdata'`, here is the definition of `Luaiolib.t` (open file handle) from the Lua-ML I/O library:

```
type 'a t = In of in_channel | Out of out_channel
```

Because an open file handle does not contain a Lua value, the type parameter `'a` is not used.

Embedding and projection To convert from a Caml value to a Lua value (of Caml type `value`) requires an *embedding* function; to convert from Lua to Caml requires *projection*. Embedding and projection come in pairs, and to represent such a pair, Lua-ML defines type `('a, 'b) ep`: an `embed` function for converting a value of type `'a` into a value of type `'b` and a `project` function for the opposite

conversion. For the special case where we are embedding into a Lua `value`, we define type `'a map`.

```
type ('a, 'b) ep =
  { embed : 'a -> 'b; project : 'b -> 'a }
type 'a map = ('a, value) ep
```

Unlike APIs such as Tcl or Lua, Lua-ML uses higher-order functions to provide an *unlimited* supply of embedding/projection pairs: Embedding and projection are a *type-indexed family* of functions. The idea is inspired by Danvy (1996), who uses a similar family to implement partial evaluation. We build a type-indexed family of functions as follows.[1]

- For a base type, such as `float`, we provide a suitable embedding/projection pair. Lua-ML includes pairs for `float`, `int`, `bool`, `string`, `unit`, `userdata`, `table`, and `value`.
- For a type constructor that takes one argument, such as `list`, we provide a higher-order function that maps an embedding/projection pair to an embedding/projection pair. Lua-ML includes such functions for the `list` and `option` type constructors.
- For a type constructor of two or more arguments, we continue in a similar vein. Of greatest interest is the arrow constructor, which describes a function type.

In Lua-ML, the functions that build embedding/projection pairs are part of the `VALUE` signature; the details appear in a companion paper (Ramsey 2005). What is important for this paper is that we need an embedding/projection pair for each type module. These pairs are constructed by the functors used to build an interpreter.

A library may define its own embedding/projection pairs. For example the I/O library needs to convert from the type `Luaiolib.t` (open file handle) to the type `in_channel` (file open for input). The conversion is done by the embedding/projection pair `infile`, which has type `in_channel map`. It uses a pair `t`, which has type `Luaiolib.t map`. This pair is obtained from the view of the type module for type `Luaiolib.t`.

```
let infile =
  let fail v =
    raise (Projection (v, "input file")) in
  { embed   = (fun f -> t.embed (In f))
  ; project = (fun v -> match t.project v with
                        | In f -> f
                        | _ -> fail v)
  }
```

The exception `Projection` is raised whenever projection fails.

Registration The process of initializing an interpreter includes *registration*. A library registers a value by storing it in a global Lua variable, table, or other structure. Registration can be performed by directly manipulating the `globals` table in a Lua `state`, but Lua-ML provides two convenience functions. Function `register_globals` has type `(string * value) list -> state -> unit`; for each (s, v) pair on the list, it makes v the value of global variable s in the `state`. Function `register_module` has type `string -> (string * value) list -> state -> unit`; it embodies the common programming convention of putting a group of related functions in different named fields of a single, global table.

[1] Danvy (1998) credits Andrzej Filinski and Zhe Yang with developing this technique.

For example, the Lua-ML I/O library registers many functions at startup time. Registration takes place when `init` is called, receiving `interp`, which has type `state`.

```
let init interp =
  let io = {currentin=stdin; currentout=stdout} in
  ⟨definitions of the I/O library functions⟩
  register_globals
  [ "open_in",
       efunc (string **->> infile) open_in
  ; "close_in",
       efunc (infile **->> unit)   close_in
   ...
  ] interp
```

This example shows a library that extends the interpreter with new, private state: the `io` record. The mutable fields `currentin` and `currentout` maintain the current input and output file, which are accessible only to the functions in the I/O library.

Functions `open_in` and `close_in` are pervasives in Caml. The values `efunc`, `string`, `**->>`, and `unit` all relate to embedding; the code embeds `open_in`, which has type `string -> in_channel`, and `close_in`, which has type `in_channel -> unit`. Details can be found in a companion paper (Ramsey 2005). The `init` function registers many other functions which are not shown, but which are defined in ⟨*definitions of the I/O library functions*⟩ so they have access to `currentin` and `currentout`.

3.3 Components of an interpreter

Figure 1 (on the next page) shows how modules are composed to build an interpreter. The figure is less frightening than it appears:

- The large box on the left holds type modules; the box on the right holds code modules.
- An oval with a double border represents a module that is written by hand. An oval with a single border represents a module that is supplied with Lua-ML or built by applying a functor. Almost every oval is labeled with the signature its module matches.
- A functor application is indicated by a tiny circle that is connected with arrows. The incoming arrows represent the functor's arguments; the outgoing arrow represents its result. Solid arrows represent a functor application that is hand-written by a user of Lua-ML; dotted arrows represent a functor application that is done "behind the scenes" by another functor. Such behind-the-scenes application indicates the presence of a higher-order functor.
- A dashed arrow represents inclusion, which may be direct or through submodules. As an example of direct inclusion, the `COMBINED_TYPE` signature includes `USERTYPE`. As an example of inclusion through submodules, any module matching `COMBINED_TYPE` contains submodules that match `TYPEVIEW`.

The final result of all the functor application in Figure 1 is an interpreter, which matches signature `INTERP` and is shown at the bottom of Figure 1. Since an interpreter is our ultimate goal, we begin our explanation there and work from the bottom up.

An interpreter An interpreter is built by applying the `MakeInterp` functor to an *evaluator* and a parser. By supplying a nonstandard parser, a user can extend the concrete syntax accepted by the interpreter. Such an extension must

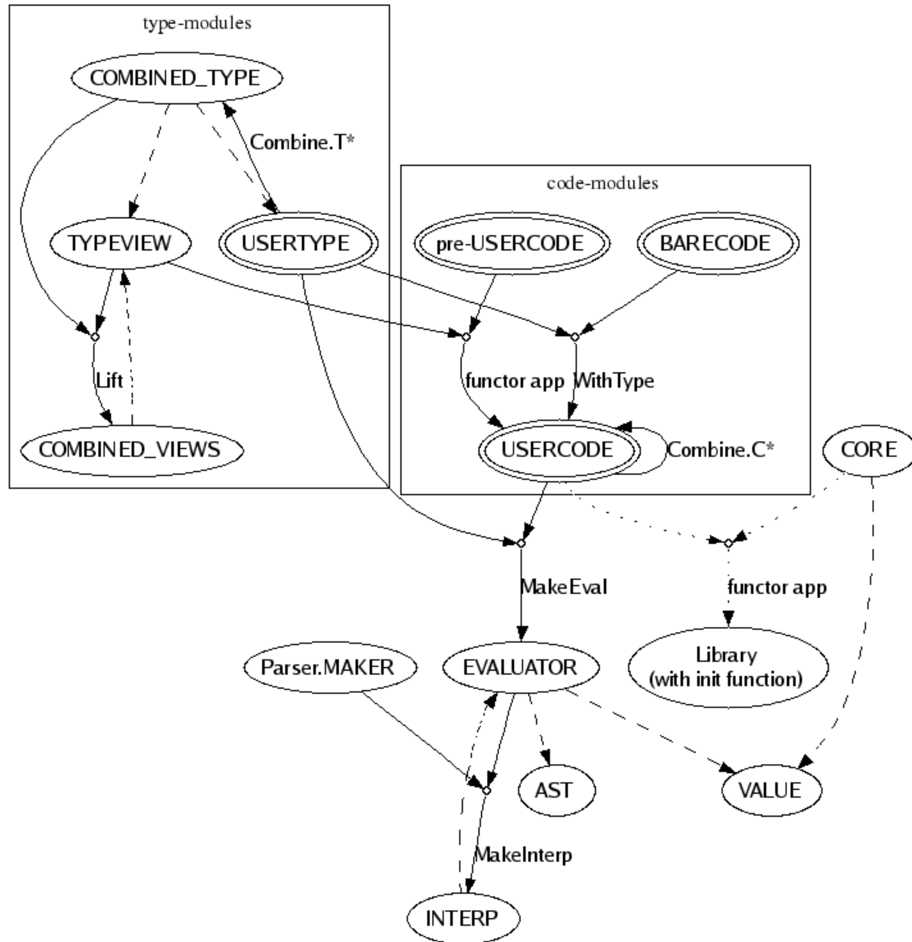


Figure 1: ML module mania: Components and construction of a Lua-ML interpreter

translate into existing abstract syntax, as the abstract syntax of Lua-ML is not extensible.

The signature `INTERP` and functor `MakeInterp` are declared as follows:

```
module type INTERP = sig
  include EVALUATOR
  module Parser :
    Luaparser.S with type chunk = Ast.chunk
  val mk        : unit -> state
  val dostring  : state -> string -> value list
   ...
end
module MakeInterp
  (MakeParser : Parser.MAKER) (E : EVALUATOR)
    : INTERP with module Value = E.Value
```

Within a module matching `INTERP`, function `mk` creates a fresh, fully initialized interpreter, and function `dostring` evaluates a string containing Lua source code. The details of the parser signatures `Luaparser.S` and `Parser.MAKER` are of little interest, so we ignore them.

An evaluator An evaluator is built using a type module and a code module. The signature of an evaluator is

```
module type EVALUATOR = sig
  module Value  : VALUE
  module Ast    : AST with module Value = Value
  type state    = Value.state
  type value    = Value.value
  val pre_mk    : unit -> state
  type compiled = unit -> value list
  val compile   :
    Ast.chunk list -> state -> compiled
  ...
end
```

The evaluator provides definitions of values and terms using the submodules `Value` and `Ast`. It provides `pre_mk`, which creates and initializes an interprter, and it provides `compile`, which translates abstract syntax into a form that can be evaluated efficiently. It also provides many convenience functions, which are not shown here.

To build an evaluator, one applies functor `MakeEval` to a type module `T` and a code module `C`, each of which is typ-

ically a composition of similar modules. The type module provides type constructor `T.t`, which is used as the definition of `Value.userdata'`. `MakeEval` ties the recursive knot as shown in Section 3.2, by defining `value` to include `userdata` and `userdata` to be `value T.t`. The code module provides an initialization and registration function, which is called by `pre_mk`.

```
module MakeEval
  (T : USERTYPE)
  (C : USERCODE with type 'a userdata' = 'a T.t)
  : EVALUATOR with type 'a Value.userdata' = 'a T.t
```

In this declaration, the `with type` constraint on the module `C` ensures that the type module and code module are consistent, which is required for type safety.

Defining and composing type modules The basic building block of a type module is a user-defined type, which is a module matching the `USERTYPE` signature.

```
module type USERTYPE = sig
 type 'a t (* type parameter 'a will be Lua value *)
 val eq : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool
 val to_string : ('a -> string) -> 'a t -> string
 val tname : string  (* type's name, for errors *)
end
```

The type constructor `'a t` is a two-level type; when the recursive knot is tied by the definition of `userdata`, the type parameter `'a` will be `value`. The semantics of Lua require that it be possible to compare any two values for equality and to convert any value to a string. These operations are defined as higher-order functions, since to compare two values of type `'a t` for equality may require comparing values of type `'a`, for example. Finally, Lua-ML names each type, so it can issue a reasonably informative error message if projection fails.

It might not be obvious how to extend Lua-ML with a type constructor that is polymorphic. For example, what if you don't like mutable tables and prefer an immutable binary-search tree of type `('k, 'v) tree`? You can easily introduce the `tree` constructor into Lua-ML, but with a key limitation: Type variables `'k` and `'v` may be instantiated only with types that are known to Lua-ML. Because Lua is dynamically typed, the correct thing to do is to instantiate both with `value`, but because `value` cannot be known at the time the type module for trees is defined, the type module must use its type parameter instead:

```
module TreeType : USERTYPE
                  with type 'a t = ('a, 'a) tree =
struct
  type 'a t = ('a, 'a) tree
  fun eq eq' t1 t2 = ...
  fun to_string _ _ = "a binary-search tree"
  val tname = "search tree"
end
```

A similar limitation applies to the introduction of polymorphic functions into Lua-ML, as described by Ramsey (2005).

A type module adds just one type to Lua, but a sophisticated application might need to add many types. To add many types, a programmer combines multiple type modules into one type module, which is passed to `MakeEval`. Type modules are combined using a functor like `Combine.T3`, defined below, which takes three `USERTYPE` modules as arguments and returns a `COMBINED_TYPE` module. The signature `COMBINED_TYPE` includes not only `USERTYPE` but also an embedding/projection pair for each constituent type. The embedding/projection pair is hidden inside a submodule that

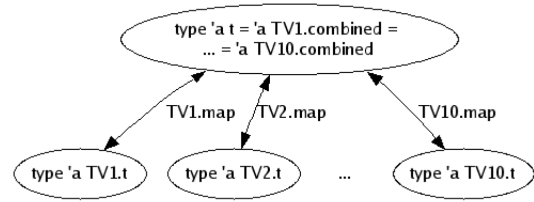


Figure 2: Views of combined types in `COMBINED_TYPE`

matches the `TYPEVIEW` signature, which is defined approximately as follows:[2]

```
module type TYPEVIEW = sig
  type 'a combined
  type 'a t (* the type of which this is a view *)
  val map : ('a t, 'a combined) ep
end
```

The type `'a combined` is the "combined type," which is a sum of individual types. The type `'a t` is one of these individual types. To see all of the individual types that make up a single combined type, one needs a "combined type module." It is the composition of up to ten type modules.

```
module type COMBINED_TYPE = sig
  include USERTYPE
  module type VIEW =
    TYPEVIEW with type 'a combined = 'a t
  module TV1  : VIEW
  module TV2  : VIEW
  module TV3  : VIEW
    ...
  module TV10 : VIEW
end
```

Each view's combined type is equal to the type `'a t` included from the `USERTYPE` signature. The combination may be better understood graphically; Figure 2 shows a single combined type and its relationships to its constituent types. Each constituent type can be embedded in the combined type above it; the combined type can be projected to any of the constituent types, but such projection might raise an exception.

The `Combine` module provides functors that combine up to ten types and return appropriate views. For example, it includes functor `Combine.T3`.

```
module Combine : sig
  module T3 (T1 : USERTYPE) (T2 : USERTYPE)
            (T3 : USERTYPE)
    : COMBINED_TYPE with type 'a TV1.t = 'a T1.t
                    with type 'a TV2.t = 'a T2.t
                    with type 'a TV3.t = 'a T3.t
  ⟨similar declarations for T1, T2, and T4 to T10⟩
  ...
end
```

The views of the constituent types in the `COMBINED_TYPE` signature are essential for building libraries that use the types. The views provide the projection functions that enable a library module to get from a value of combined type (which is probably `userdata`) to a value of the constituent type of its choice.

[2] The real definition of `TYPEVIEW` uses a different representation of the embedding/projection pair. This representation is more complicated, but it simplifies the implementation of code modules and of functors such as `Combine.T3`. The details are not worth the space it would take to explain them.

If there are more than ten types, then applications of `Combine.T*` functors must be nested. In this case, an inner application produces views that map between a child type and its parent type, but what are needed are views that map between a child type and its grandparent type. We can get these views by composing the combined parent type with the view mapping that type to the grandparent. The composition is implemented by a functor called `Lift`.

```
module Lift
  (T : COMBINED_TYPE)
  (View : TYPEVIEW with type 'a t = 'a T.t)
  : COMBINED_VIEWS
     with type 'a t     = 'a View.combined
     with type 'a TV1.t = 'a T.TV1.t
     with type 'a TV2.t = 'a T.TV2.t
      ...
     with type 'a TV10.t = 'a T.TV10.t
```

The collection of views matches `COMBINED_VIEWS`, which is just like `COMBINED_TYPE` except it does not include `USERTYPE`. An example appears in Section 4.

The idea of the `Combine.T*` functors is very similar to the idea of the `OR` type constructor of Liang, Hudak, and Jones (1995). Since Liang, Hudak, and Jones are using Haskell, they have the luxury of defining embedding and projection for `OR` types by using type classes. To achieve a similar effect in ML requires functors. These functors provide a branching factor of up to ten, which has two benefits: Less notation is needed to combine multiple types, and fewer pointers are dereferenced at run time in the implementations of embedding and projection.

Defining and composing code modules A code module is a library module that initializes an interpreter by registering interesting values and functions. A code module must know what sort of interpreter to initialize. Figure 1 shows that a final interpreter (`INTERP`) is produced from an `EVALUATOR`, which contains a translator (`compile`) and libraries. There is actually a stage before `EVALUATOR`: an *interpreter core*, which is shown at the middle right of Figure 1, as `CORE`.

```
module type CORE = sig
  module V : VALUE
    ...
  val register_globals :
    (string * V.value) list -> V.state -> unit
  val register_module  : string ->
    (string * V.value) list -> V.state -> unit
end
```

An interpreter core contains a submodule `V` that defines `value`. This definition includes a definition of `userdata` that is built using a type module. An interpreter core also contains convenience functions, of which we show only the most important: the registration functions mentioned above. These registration functions, along with the types `V.value` and `V.state`, are used by a code module to help initialize an interpreter.

The idea of a code module is simple: It is a functor that takes an interpreter core and produces an initialization function. The simplest kind of code module is from a library that adds no new types or new state and therefore does not depend on any application-specific types.

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

A code module of type `BARECODE` can be used with any module matching `CORE`. But if the code module depends on one or more application-specific types, there are two additional requirements:

- It must have suitable embedding and projection functions, which is to say *views*, with which it can map between `userdata` and values of the application-specific types.
- To ensure type safety, it can be used only with an interpreter core that provides a suitable definition of the `userdata'` type contructor. A definition is suitable if it is consistent with the embedding and projection functions.

We address the second requirement first.

To ensure the suitability of `userdata'`, we would like to apply a `with type` constraint to the functor parameter `C` of module type `CORE`. Unfortunately, the signatures language of Objective Caml provides no way for a `with type` constraint to name a functor's parameter. To work around this limitation, we introduce another level of nesting and a new type constructor `userdata'`, the purpose of which is to be nameable in a `with type` constraint.

```
module type USERCODE = sig
  type 'a userdata' (* type on which lib depends *)
  module M : functor
    (C : CORE with type 'a V.userdata' = 'a userdata')
    -> sig
         val init : C.V.state -> unit
       end
end
```

Given this definition, we can write a signature such as `USERCODE with type 'a userdata' = ...` and be sure of properly constraining the functor parameter `C`. Such a constraint appears in the declaration of the `MakeEval` functor, which we repeat here:

```
module MakeEval
  (T : USERTYPE)
  (L : USERCODE with type 'a userdata' = 'a T.t)
  : EVALUATOR with type 'a Value.userdata' = 'a T.t
```

The signatures `BARECODE` and `USERCODE` describe two of the three kinds of hand-written code modules shown as double ovals in Figure 1. The third kind, "pre-`USERCODE`," is not a signature but a programming convention. A pre-`USERCODE` module is one that depends on particular *views* as well as a suitable interpreter core. Recall that a view provides an embedding/projection pair between an application-specific type and a `combined` type, where the `combined` type is to be used as userdata (Figure 2). A pre-`USERCODE` code module constrains both parts of a view:

- The view's application-specific type, `'a t`, must be one that is useful to the code module.
- The view's `combined` type constructor must be the same as the `userdata'` type constructor in the `USERCODE` signature.

As an example, here is a high-level view of the Lua-ML I/O library. It provides an application-specific type `'a t`, a type module `T`, and a pre-`USERCODE` code module `Make`.

```
type 'a t = In of in_channel | Out of out_channel
module T : USERTYPE  with type 'a t = 'a t
module Make (TV : TYPEVIEW with type 'a t = 'a t)
  : USERCODE with type 'a userdata' = 'a TV.combined
```

Code modules can be composed just as type modules can. The `Combine` module provides functors that combine up to ten code modules. For example, it includes functor `Combine.C3`.

```
module Combine : sig
  ...
  module C3
    (C1 : USERCODE)
    (C2 : USERCODE
          with type 'a userdata' = 'a C1.userdata')
    (C3 : USERCODE
          with type 'a userdata' = 'a C1.userdata')
    : USERCODE
      with type 'a userdata' = 'a C1.userdata'
  ⟨similar declarations for C1, C2, and C4 to C10⟩
end
```

Code modules that are combined must all agree on the `userdata'` type constructor.

The `Combine.C*` functors combine only code modules that match `USERCODE`. To combine a module that matches `BARECODE`, one must first use the `WithType` functor, which converts the bare code module to `USERCODE` by associating it with a type module.

```
module WithType (T : USERTYPE) (C : BARECODE)
  : USERCODE with type 'a userdata' = 'a T.t
```

4 Putting it all together

The details of Lua-ML's library support look daunting, but because library modules are combined in stylized ways, it is not difficult to write libraries and build interpreters. Each library defines its application-specific types in type modules matching signature `USERTYPE`. Each library defines its code in a code module, which is most typically either a structure matching `BARECODE` or a functor that accepts arguments matching `TYPEVIEW` and produces a result matching `USERCODE`. Both type modules and code modules can be compiled separately.

Once libraries are written, it is often easiest to write a single "linking module" that combines libraries and builds an interpreter. Such a module has a stylized structure:

1. Combine all relevant type modules using functors in the `Combine.T*` family. Call the result `T`.
2. From `T`, which matches `COMBINED_TYPE`, extract and rename each submodule matching `TYPEVIEW`. If `T` contains more than ten types, some types will be nested, and it will be necessary to use `Lift`.

   This step is not strictly necessary, but because the submodules have names like `T.TV4`, it is recommended to rename them so that subsequent code can use more readable names.
3. Combine all relevant code modules using functors in the `Combine.C*` family.
   - A code module matching `USERCODE` can be used directly.
   - A code module matching `BARECODE` must have the functor `WithType (T)` applied to it.
   - A code module that is pre-`USERCODE` must be applied to the relevant views from step 2.

   Call the resulting combination `C`.
4. Apply `MakeEval` to `T` and `C`, and apply `MakeInterp` to the result and to the standard parser maker. Call the result `I`.

With these linking steps completed, client code can create an interpreter by calling `I.mk`, and the `I` module contains all the procedures needed to evaluate Lua code.

As an example, here are some excerpts from our C-- compiler. The compiler defines many type modules. Here, for example, is one for the type `Ast2ir.proc`, which represents the intermediate form of a procedure and includes the procedure's control-flow graph.

```
module ProcType :
  USERTYPE with type 'a t = Ast2ir.proc =
struct
  type 'a t       = Ast2ir.proc
  let tname       = "proc"
  let eq _        = fun x y -> x = y
  let to_string _ =
    fun t -> "<proc " ^ t.Proc.name ^ ">"
end
```

The type modules `AsmType` and `TargetType` represent the types of an assembler and a target machine, respectively.

```
module AsmType    : USERTYPE with type ... = ...
module TargetType : USERTYPE with type ... = ...
```

There are many other type modules.

The compiler also defines code modules. Most parts of the compiler are exported to Lua in a single, pre-`USERCODE` code module called `MakeLib`.

```
module MakeLib
  (AsmV    : TYPEVIEW with type 'a t = 'a AsmType.t)
  (ProcV   : TYPEVIEW with type 'a t = 'a ProcType.t
             and type 'a combined = 'a AsmV.combined)
  ...
  (TargetV : TYPEVIEW with type 'a t = 'a TargetType.t
             and type 'a combined = 'a AsmV.combined)
  : USERCODE with type 'a userdata' = 'a AsmV.combined =
struct
  type 'a userdata' = 'a AsmV.combined
  module M
    (C : CORE with type 'a V.userdata' = 'a userdata') =
  struct
    module V = C.V
    let ( **-> ) = V.( **-> )
    let ( **->> ) t t' = t **-> V.result t'
    ⟨definitions of many embedding/projection pairs⟩
    let init interp =
      C.register_module "Asm"
      [ "x86" , V.efunc (outchan **->> asm)
                  (X86asm.make  Cfg.emit)
      ; "mips", V.efunc (outchan **->> asm)
                  (Mipsasm.make Cfg.emit)
      ...
      ] interp;
      C.register_module "Stack"
      [ "freeze",   V.efunc (proc **-> block **->> V.unit)
                  Stack.freeze
      ; "procname", V.efunc (proc **->> V.string)
                  (fun p -> p.Proc.name)
      ] interp;
      C.register_module "Targets"
      [ "x86",      target.V.embed X86.target
      ; "mips",     target.V.embed Mips.target
      ; "alpha",    target.V.embed Alpha.target
      ] interp;
      ...
    end (*M*)
end (*MakeLib*)
```

The `init` function defined by the code module registers many functions, each of which is embedded using `V.efunc`. Just a few examples are shown here. It also embeds a few non-function values, such as those in the `Targets` table.

Given a collection of type modules and code modules, we can write a linking module. We follow the four steps above. For step 1, we combine type modules. Because we once had more than ten type modules, we combine types in two stages.

```
module T1 =
  Combine.T5
    (DocType)             (* T1.TV1 *)
    (Luaiolib.T)          (* T1.TV2 *)
    (AsmType)             (* T1.TV3 *)
    (AstType)             (* T1.TV4 *)
    (Colorgraph.T)        (* T1.TV5 *)

module T =
  Combine.T6
    (T1)                  (* T.TV1 *)
    (Backplane.T)         (* T.TV2 *)
    (EnvType)             (* T.TV3 *)
    (ProcType)            (* T.TV4 *)
    (TargetType)          (* T.TV5 *)
    (BlockType)           (* T.TV6 *)
```

In step 2, we extract and rename the relevant views. The views in module `T1` provide mappings to type `T1.combined`, so we have to use `Lift` to get mappings to type `T.combined`.

```
module T1' = Lift (T1) (T.TV1)
```

The renaming is straightforward.

```
module DocTV        = T1'.TV1
module LuaioTV      = T1'.TV2
module AsmTV        = T1'.TV3
module AstTV        = T1'.TV4
module ColorgraphTV = T1'.TV5

module BackplaneTV  = T.TV2
module EnvTV        = T.TV3
module ProcTV       = T.TV4
module TargetTV     = T.TV5
module BlockTV      = T.TV6
```

In step 3, we combine code modules using `Combine.C7`. Three code modules are bare, so we define and use `WT` to use them with type module `T`.

```
module WT = WithType (T)
module C =
  Combine.C7
    (Luaiolib.Make (LuaioTV))
    (WT (Luastrlib.M))
    (WT (Luamathlib.M))
    (MakeLib
       (AsmTV) (AstTV) (EnvTV) (ProcTV) (TargetTV)
       (DocTV) (LuaioTV) (BlockTV))
    (Colorgraph.MakeLua
       (BackplaneTV) (ColorgraphTV) (ProcTV))
    (WT (Luautil.MakeLib))
    (Backplane.MakeLua (BackplaneTV) (ProcTV))
```

Finally, in step 4, we build an interpreter.

```
module I =
  MakeInterp
    (Parser.MakeStandard)
    (MakeEval (T) (C))
```

5 Discussion

Although Lua-ML's library support looks complex, it is not clear that anything significantly simpler will do, at least if we are using ML modules.

Composition of types The main source of complexity in Lua-ML is the need to compose separately compiled libraries. The composition of libraries determines the set of types included in an interpreter's `value` type. But if it is to be compiled separately, each library must be independent of `value` and of the set of types that make up `value`. Lua-ML solves this problem using Steele's (1994) technique of type parameterization, also called two-level types: a type constructor defined in a library takes a type parameter that is ultimately instantiated with `value`. By using a type parameter, one can define a data structure that can contain any `value` and can be compiled separately even when the full definition of `value` is unknown.

To define `value` once libraries have been chosen, Lua-ML uses an external sum constructor similar to that used by Liang, Hudak, and Jones (1995). The external sum is more convenient than Steele's tower of types, and it requires fewer pointer indirections at run time. Again, to be compiled separately, a library must be able to get values out of a sum without knowing the definition of the sum. Like the interpreters of Liang, Hudak, and Jones, Lua-ML solves this problem by using embedding and projection functions. Liang, Hudak, and Jones define the sum as a type constructor, and they use Haskell's type classes to define embedding and projection. Given an application of the type constructor, the Haskell compiler automatically composes the correct embedding and projection functions. The ML compiler is not so clever: Lacking type classes, we define the sum constructor as a functor (e.g., `Combine.T3`), not as a type constructor. Otherwise the designs are similar.

In summary, composing libraries requires that we compose types, and to compose types we must make two independent choices:

- To combine types, we may use an external sum constructor or we may build a tower using an additional type parameter. Both choices are consistent with separate compilation.
- To include a Lua value in a user-defined extension, we may use two-level types or we may provide a definition of `value` that is mutually recursive with the definitions of the constituent types, including extensions. Only two-level types are consistent with separate compilation.

These observations have guided the design of Lua-ML, but they do not determine it. We should ask if we could simplify Lua-ML significantly either by using another design or other language features to compose libraries.

Alternative designs Lua-ML splits each library into zero or more type modules plus a code module. A design that seems simpler is to write every Lua library as a single ML module. But there are four different kinds of library: one that adds a new type, new state, both, or neither. Because there are four kinds, the obvious "one library, one module" designs do not work out very well; the difficulty is what signature each kind of library should have.

- *Give each kind of library a different signature.* The design works well for describing individual libraries, but combining libraries is problematic: There are too many combinations of signatures.
- *Give each kind of library the most general signature.* In other words, pretend each library adds both a type and a state. This design seems reasonable at first, particularly if one provides functors analogous to `WithType`, so that a library can be coerced to a more general signature. But there is a problem: It is impossible to share types among

multiple libraries. This problem is quite significant if, for example, multiple libraries want to use I/O.

To share types among libraries is the primary reason that Lua-ML splits each library into multiple modules.

Another design that seems simpler is to treat both kinds of extensions, type and state, in the same way. But the mechanisms needed to share and compose types are complex, similar mechanisms for sharing and composing state would be unnecessary—types and state are used very differently in practice.

- Although most Lua libraries add neither a new type nor new state, it is still common for a library to add a new type. Moreover, added types are often shared; typical shared types include both general-purpose types like file descriptor and application-specific types like control-flow graph.
- A Lua library rarely adds state, and I have never observed such state to be shared with another library.
- ML library modules are similar to Lua libraries in their use of types and state. For example, a quick perusal of library modules distributed with Objective Caml shows that somewhat fewer than half define a distinct, new type. Only one appears to define new mutable state: the random-number generator `Random`. Some others provide access to existing mutable state: the thread library `Thread`, the bytecode loader `Dynlink`, and the windowing toolkit `Tk`. In all cases the mutable state is private to its module.

These practices justify Lua-ML's design, in which type extensions and state extensions are treated quite differently. Type extensions enjoy the full power of the modules system, and the presence of a needed type is checked at compile time. State extensions, by contrast, are second-class citizens. If you want some piece of shared state, your only option is to put it in a global variable, and you need to perform a dynamic check just to know it is there.[3] The benefit of this design is that the treatment of state is irrelevant to a library's signature, and the mechanisms for composing libraries are simplified thereby.

Alternative language mechanisms The complexity of composing libraries is apparent in the many different kinds of functors that must be composed to build an interpreter in Lua-ML. Perhaps it would be simpler to use a different language mechanism. There are several candidates:

- *Unsafe cast.* One could define `userdata` to be any pointer type, then use an unsafe cast to embed or project a particular extension. This solution, which is essentially the solution used in C for both Lua and Tcl, could also be used in ML. But it relies on the programmer to guarantee type safety, using code analogous to the C code in Appendix A. Such unsafe code tastes bad to an ML programmer.
- *Type dynamic.* One could define `userdata` to be the type "dynamic" and use the operations on that type to implement embedding and projection of each extension. Type dynamic is a frequently provided extension to a functional language, and in common languages it can be simulated: In ML, one can simulate dynamic by abusing the `exn` type, and in Haskell, one can simulate dynamic using universal and existential type qualifiers (Baars and Swierstra 2002).
- *Objects.* One could define `userdata` to be an object type and each extension to be a subtype. Embedding comes "for free" via subsumption, but projection requires that the language include a safe, downward cast, which involves a run-time check. No such cast is available in Objective Caml; a value of object type may be cast only to a supertype. Standard ML and Haskell, of course, lack objects entirely.
- *Extensible datatypes.* One might define `userdata` as an extensible datatype in the style of Millstein, Bleckner, and Chambers (2002). This mechanism looks attractive but is not available in widely deployed functional languages. It also has the limitation that only one definition of `userdata` may appear in any application that uses the embedded interpreter; in other words, it is not possible to embed two instances of the interpreter that use different `userdata` types.
- *Cross-module recursive types.* Given a language that allows the definition of a recursive type to extend across module boundaries, such that the extension defined by Russo (2001), one could define `userdata` directly using this extension instead of indirectly using functors and type parameters. Like the previous mechanism, this mechanism limits a program to a single instance of `userdata`.

Each of these mechanisms enables a solution in which extensions can be independent and need not be composed explicitly by the programmer, which might be a worthwhile simplification. But it would be a mistake to think that libraries can be composed simply by composing their types: To implement Lua's semantics, it is also necessary to compose `eq` functions. Of the mechanisms enumerated above, only objects with a downward cast would provide a convenient way of attaching an `eq` operation to a type. Since no ML-like language provides such a mechanism, we would need to compose the `eq` functions in some other way. The `eq` functions would have to be defined and composed in a similar way to the `USERTYPE` structures in Lua-ML. We might hope to write the code differently, say by using type dynamic and moving the composition from the modules language into the core language, but it seems unlikely that the result would be any simpler than Lua-ML.

ML module mania Lua-ML's use of Objective Caml modules is aggressive—perhaps even maniacal. In particular, Lua-ML uses first-class functors, which may return a functor, take a functor as an argument, or be a component of a structure.

- A pre-`USERCODE` code module is a higher-order functor with a signature of the form $S_1 \to (S_2 \to S_3)$. Signature S_1 describes an application-dependent type, where signatures S_2 and S_3 belong to the Lua-ML interface; $S_2 \to S_3$ is approximately the signature of a code module (`USERCODE`), at least in spirit. If a functor could not return a functor, we would have to use a signature of the form $S_1 \times S_2 \to S_3$. In this form, there is no independent signature that describes a code module, and the argument signature $S_1 \times S_2$ does not describe an independently useful component. On these aesthetic grounds, I prefer the Curried form, but it is not essential.
- The `MakeEval` functor is a higher-order functor with a signature of the form $(S_2 \to S_3) \to S_4$. Here `USERCODE` is the argument functor, and the higher order enables

[3] Lua versions 4.0 and later provide a "registry," which is an *unnamed* table that is shared among all libraries. A similar registry could be implemented in Lua-ML as a library.

MakeEval to hide the details of building a suitable CORE module to which a USERCODE functor can then be applied. We could avoid an arrow on the left of an arrow by making the linking module do more work: It would have to build CORE explicitly (needing additional API functors to do so) and then apply each USERCODE functor to this CORE. The notational burden would be modest, but it is still worth avoiding. As in the previous case, the higher-order functor is not essential, but I prefer it.

- The USERCODE signature requires nesting a functor within a module. This nesting is only a device to enable us to constrain the functor's argument using `with type`, but such constraints are essential to get the separately compiled code to type-check.

This evidence shows that although first-class functors may please us, they are not needed to build a type-safe, separately compiled, extensible interpreter.

- If we allow functors, it is pleasant when they are first-class, but we can imagine building an extensible interpreter with only first-order, top-level functors, *provided* we have a signatures language that allows us to constrain a functor's argument using `with type`.
- It is even possible to imagine doing without functors—the problem that they solve is safely composing types and functions (such as eq) that are defined in separately compiled modules. Without functors, we could compose types using a mechanism such as type dynamic and compose functions using the core language. These mechanisms don't suffice to ensure that each type is associated with exactly one eq function, but a language designer could introduce other mechanisms for that purpose. An obvious candidate would be Haskell's type classes.

So what can we learn from Lua-ML, a modest-sized program that uses first-class functors aggressively? To me, the most surprising result is that the only first-class functor that would be difficult to get rid of—in the definition of USERCODE—is there purely as a workaround for a defect in the signatures language. For the rest, I am forced to conclude that Lua-ML doesn't really need first-class functors. While at first I found this conclusion discouraging, on reflection I am neither discouraged nor surprised; after all, although I normally use first-class *functions* heavily, I manage without them when I program in C. And some of my colleagues who are C programmers might well describe first-class functions as "powerful, but breathtakingly expensive." In the end, I hope you will believe as I do: It is a lot more fun to program with first-class functors than without them. And I hope that designers of future functional languages will include powerful modules systems, despite the expense.

Acknowledgements

João Dias, Simon Peyton Jones, and Sukyoung Ryu helped smooth some rough spots in the manuscript. Anonymous referees suggested a number of clarifications.

This work is part of the C-- project and was supported by NSF grant CCR-0096069, by a Sloan Research Fellowship, and by a gift from Microsoft. Code can be downloaded from www.cminusminus.org.

References

Baars, Arthur I. and S. Doaitse Swierstra. 2002. Typing dynamic typing. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 157–166.

Batory, Don and Sean O'Malley. 1992 (October). The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398.

Benson, Brent W. 1994 (October). Libscheme: Scheme as a C library. In *Proceedings of the USENIX Symposium on Very High Level Languages*, pages 7–19.

Biagioni, Edoardo, Robert Harper, Peter Lee, and Brian G. Milnes. 1994 (June). Signatures for a network protocol stack: A systems application of Standard ML. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 55–64. ACM Press.

Danvy, Olivier. 1996. Type-directed partial evaluation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257. ACM Press.

———. 1998. A simple solution to type specialization. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming (ICALP)*, number 1443 in Lecture Notes in Computer Science, pages 908–917. Springer-Verlag.

Ierusalimschy, Roberto, Luiz H. de Figueiredo, and Waldemar Celes. 1996a (June). Lua — an extensible extension language. *Software—Practice & Experience*, 26(6):635–652.

———. 1996b (November). *Reference Manual of the Programming Language Lua 2.5*. TeCGraf, PUC-Rio. Available from the author.

———. 2001 (May). The evolution of an extension language: A history of Lua. In *V Brazilian Symposium on Programming Languages*, pages B14–B28. (Invited paper).

Jenness, Tim and Simon Cozens. 2002 (July). *Extending and Embedding Perl*. Manning Publications Company.

Laumann, Oliver and Carsten Bormann. 1994 (Fall). Elk: The Extension Language Kit. *Computing Systems*, 7(4):419–449.

Leijen, Daan and Erik Meijer. 2000 (January). Domain-specific embedded compilers. *Proceedings of the 2nd Conference on Domain-Specific Languages,* in *SIGPLAN Notices*, 35(1):109–122.

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

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.

Millstein, Todd, Colin Bleckner, and Craig Chambers. 2002. Modular typechecking for hierarchically extensible datatypes and functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 110–122.

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

———. 1994. *Tcl and the Tk Toolkit*. Professional Computing Series. Reading, MA: Addison-Wesley.

———. 1998 (March). Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30.

Peyton Jones, Simon, editor. 2003a. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press. Also a special issue of the *Journal of Functional Programming*, 13(1):1-255, January 2003.

———. 2003b (January). Wearing the hair shirt: A retrospective on Haskell. Invited talk at the 30th Annual ACM Symposium on Principles of Programming Languages (POPL). Slides are available from `http://research.microsoft.com/Users/simonpj`.

Ramsey, Norman. 2005. Embedding an interpreted language using higher-order functions and types. *Journal of Functional Programming*. To appear. A preliminary version of this paper appeared in *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, June 2003.

Ramsey, Norman and Simon L. Peyton Jones. 2000 (May). A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices*, 35(5):285–298.

Russo, Claudio V. 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.

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.

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

van Rossum, Guido. 2002. *Extending and Embedding the Python Interpreter*. Release 2.2.2.

Wadler, Philip. 1992 (January). The essence of functional programming (invited talk). In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14.

A Application-specific conversion

Here is an example of C code used to convert a Lua value to and from an application-specific host value of type `Mine`. The function `getmine` contains a cast that cannot be checked by the C compiler. The code is safe only if other code avoids tagging userdata with `mine_tag`.

```
typedef struct mine *Mine;
int mine_tag;
int ismine(lua_State *L, int o) {
  return lua_isuserdata(L, o)
      && lua_tag(L, o) == mine_tag;
}

Mine getmine(lua_State *L, int o) {
  assert(ismine(L, o));
  return (Mine) lua_touserdata(L, o);
}

void pushmine(lua_State *L, Mine p) {
  lua_pushusertag(L, (void *) p, mine_tag);
}
```