Reprinted from the 2005 ACM SIGPLAN Workshop on ML

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

Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University

1 Introduction

ML provides unusually powerful mechanisms for building programs from reusable modules. Such power is not available in other popular languages, and programmers accustomed to those languages have wondered if a powerful modules system is really necessary. This paper explores the power of ML modules—including higher-order functors—via an extended programming example. The example solves a problem in the construction of interpreters: how to combine extensibility with separate compilation in a safe language.

We focus on a kind of interpreter for which extensibility and separate compilation are especially important: the *embedded* interpreter. An embedded interpreter implements a *reusable* scripting language that can be used to control a complex application—like a web server or an optimizing compiler—which is written in a statically typed, compiled *host* language like ML. The interpreter becomes part of the application, so the application can invoke the interpreter and the interpreter can call code in the application. The idea was first demonstrated by Ousterhout (1990) and has been widely imitated (Benson 1994; Laumann and Bormann 1994; Ierusalimschy, de Figueiredo, and Celes 1996a; Jenness and Cozens 2002; van Rossum 2002). Sometimes a host language can also be used for scripting (Leijen and Meijer 2000), but often it is inconvenient or even impossible to make a host-language compiler available at run time.

A scripting language and its interpreter must meet several requirements:

1. They must be extensible: the whole point is to add application-specific data and code to the scripting language.
2. The interpreter should be compiled separately from the host 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 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 an 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 Background: Extensible interpreters

Prior work on extensible interpreters comes in two flavors. Work done using C has produced embedded interpreters that are extensible and separately compiled but not type-safe: safety is lost because each host value is given a "universal" type such as `void *` or `char *`, and application-specific code must use unsafe casts between this type and the actual host-language type. Work done using functional languages has produced interpreters that are extensible and type-safe but not separately compiled. Because this work has informed the design of Lua-ML, we begin by reviewing it.

Lua-ML is inspired partly 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 (Ierusalimschy 2003), 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 $* \times * \Rightarrow *$. 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 any embedded interpreter, a critical issue is how to convert between native host-language values, such as `Big_int.big_int`, and embedded-language values, for which the type variable `'value` stands. The conversion from host value to embedded value is called *embedding*, and the conversion from embedded value to host value is called *projection*.

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 Pasalic and Sheard (2004).

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, in both ML and Haskell this definition of `value` is illegal: 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 binary sum-type constructor to build what they call *extensible unions*. This type constructor plays a role analogous to that of a cons cell in ML: it is applied to types in a union and is not part of either type. By contrast, Steele's `'next` parameter plays a role analogous to that of a linked-list pointer stored inside a heap-allocated structure in C: it is part of the definition of each type.

In Haskell 98, the sum constructor is known as `Either` (Peyton Jones 2003);

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 sum constructor simplifies the definition of types at each level, because value constructors like `Other` are 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
```

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.

Liang, Hudak, and Jones define embedding and projection functions using a multiparameter type class, which overloads the functions `embed` and `project` (there called `inj` and `prj`). For types built with `OR`, suitable instance declarations automate the composition of these functions.

Lua-ML borrows ideas from all of these sources.

- Like embedded interpreters written in C, Lua-ML is a separately compiled library.
- Like one of these interpreters, Lua, Lua-ML limits its extensibility to new types and values; 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 type constructor with type classes, Lua-ML uses an ML functor.

The rest of this paper describes 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 Extending Lua using libraries

Lua-ML is based on Lua, a language that is designed expressly for embedding (Ierusalimschy, de Figueiredo, and Celes 1996a, 2001). 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 is Lua 5.0; 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, the purpose of which is to enable an application program to add new types to the interpreter. Such a type must be a pointer type. To add a new type, an application allocates a unique tag (or in Lua 5.0, a *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. Lua-ML uses the same overall model, but Lua-ML can extend userdata with *any* type, and it does so *without* unsafe code—a requirement for an interpreter written in ML.

In both Lua and Lua-ML, the idiomatic unit of extension is the *library*. Lua comes with libraries for mathematics, string manipulation, and I/O. Application programmers can 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 program constructs used to encapsulate a library are different: each library is defined 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. It is possible to give each kind of library its own signature, but such 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 libraries can *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 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 using 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. A type module also includes a few associated functions, e.g., a function used to print a value of the new type.
- Definitions of new values, functions, or state (tasks 1 and 3) appear in a *code module*, which matches the `USERCODE` or `BARECODE` signature (Section 3.4). The most interesting component of such a signature is an `init` function, which when applied to an interpreter's state, extends the interpreter with new values or functions.
- If a code module requires that the interpreter include a particular new type, that module is represented as an ML functor; the functor takes as argument a *view* of the required type and produces a result that matches `USERCODE`. A view provides a type together with the ability to embed and project values of the type; it matches signature `TYPEVIEW` (Section 3.4). 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 view-like mechanism for sharing 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*: type modules can be composed to form a new type module, and 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 Linking

After being compiled separately, type modules and code modules are linked to form an interpreter.

1. Using Lua-ML's `Combine.T*` functors, type modules are composed into a single module `T`. The module `T` includes a view of each of its constituent type modules.
2. Each code module is specialized to `T`; for example, if a code module depends on one or more type modules, it is applied to the relevant views in `T`.
3. Using Lua-ML's `Combine.C*` functors, the specialized code modules are composed into a single module `C`.
4. Modules `T` and `C` are linked with a parser to form an interpreter:

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

The `Combine` functors and the relevant signatures are described in the rest of this section; an extended example appears in Section 4.

3.3 Elements of the design

Value and state

Both a Lua value and the state of a Lua interpreter are represented as explicit values in the host language, Objective Caml. A Lua interpreter includes a submodule that matches the `VALUE` signature, an abbreviated version of which 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
```

The `VALUE` signature represents a *family* of signatures; a member of the family is identified by giving a definition of `userdata'`. In an implementation, `userdata'` is defined by composing type modules. Constructor `userdata'` is a two-level type; its type parameter represents a `value`, as you can see from the definition of `userdata`, where the recursive knot is tied. Using this mechanism, the `value` type can be extended by libraries. The `state` type, by contrast, cannot be extended.

One example of a type constructor that could be used as `userdata'` is `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 functions 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, which has been independently discovered by Benton (2005), is inspired by Danvy (1996), who uses a similar family to implement partial evaluation.[1] We build our type-indexed family of functions as follows.

- 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, such as the function arrow `->`, we continue in a similar vein.

In Lua-ML, the functions that build embedding/projection pairs are part of the `VALUE` signature; the details appear elsewhere (Ramsey 2003). What is important here 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)

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

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. If a value being registered is already present, both register_globals and register_module raise an exception.

As an 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
```

The I/O library 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 2003). 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.

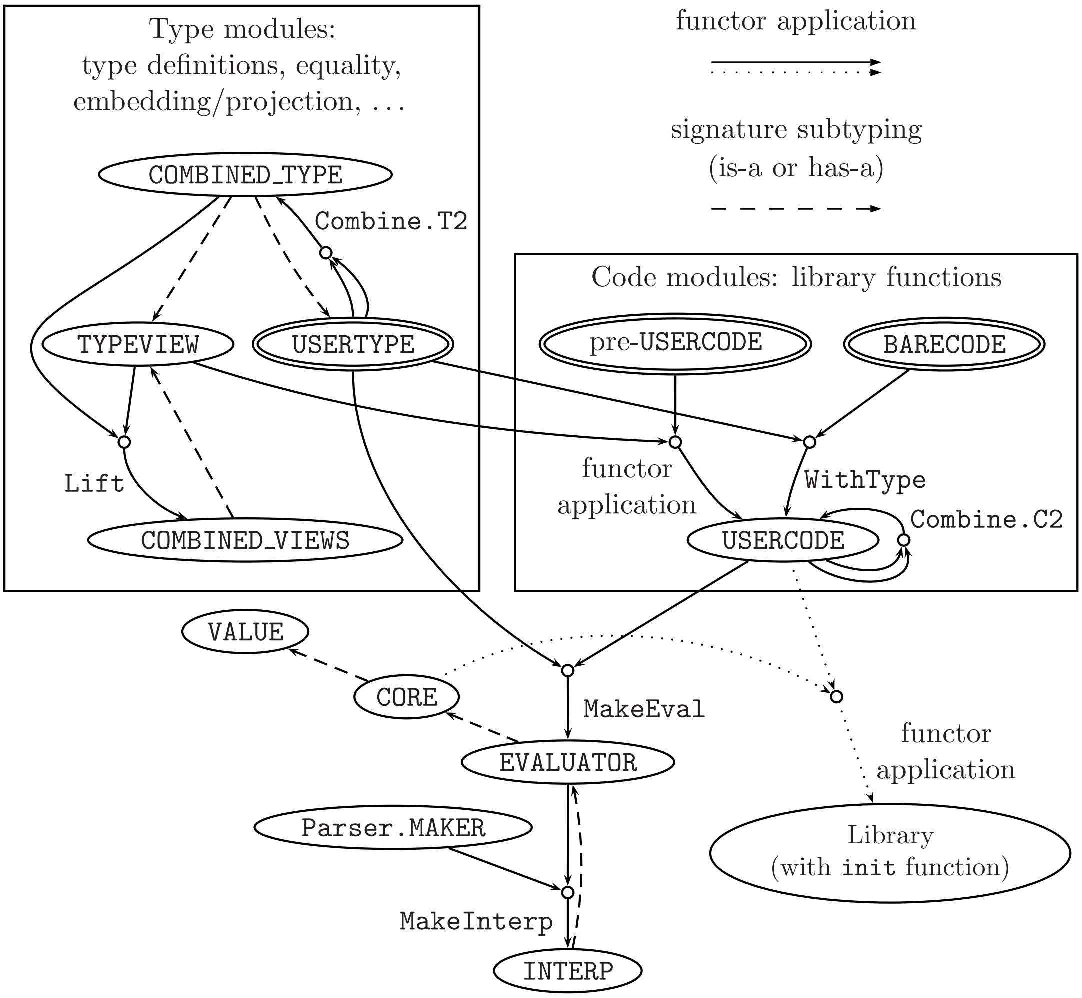
3.4 Components of an interpreter

Because so many modules are required to build a Lua-ML interpreter, I summarize their signatures and relationships in a figure. Figure 1 shows both a graphical view, which uses bubbles and arrows; and an algebraic view, which uses *informal* matching and subtype claims about modules and signatures. Either view suffices to summarize the system, so you can focus on the one you find more congenial.

- Type modules are described in the upper left box and in the middle group of algebraic claims. Code modules are described in the upper right box and in the bottom group of algebraic claims. Other components of an interpreter are described at the bottom of the graphical view and in the top group of algebraic claims.
- A module that is written by hand appears in the graphical view as a signature in a double-bordered oval and in the algebraic view as a phrase written in italics. A module that is supplied with Lua-ML or is built by applying a functor appears in the graphical view as a signature in a single-bordered oval and in the algebraic view as a name written in typewriter font.
- A possible functor application appears in the graphical view as a tiny circle that is connected with arrows. In most cases, the incoming arrows come from the functor's arguments, and the outgoing arrow, which is labeled with the functor's name, points to its result. In some cases, however, one incoming arrow comes from the functor and the other from its argument; the outgoing arrow, which is labeled "functor application," still points to the functor's result. Solid arrows represent a functor application in client code; dotted arrows represent a functor application that is done "behind the scenes" by one of Lua-ML's higher-order functors.

  A possible functor application appears in the algebraic view as an arrow in a signature. An example that appears in both views is `MakeEval`: it can be applied to a module matching `USERTYPE` and a module matching `USERCODE` to produce a module matching `EVALUATOR`.
- Figure 1 shows two forms of subtyping on signatures: "is-a" and "has-a." As an example of is-a subtyping, any module that matches `COMBINED_TYPE` also matches `USERTYPE`. This relation appears in the graphical view as a dashed arrow and in the algebraic view as the relation $\leq$. As an example of has-a subtyping, any module that matches `COMBINED_TYPE` contains submodules that match `TYPEVIEW`. This relation appears in the graphical view as a dashed arrow and in the algebraic view as the relation $\leq\!\!\cdot$.

The final result of applying all Lua-ML's functors is an interpreter, which matches signature `INTERP` and is shown at the bottom of the graphical view. Since an interpreter is our ultimate goal, we begin our explanation there.



Interpreter

- INTERP ≤ EVALUATOR ≤ CORE ≤ VALUE
- MakeInterp : Parser.MAKER → EVALUATOR → INTERP
- Parser.MakeStandard : Parser.MAKER
- MakeEval : USERTYPE → USERCODE → EVALUATOR

Type modules

- *type module for application-specific type* τ : USERTYPE$_\tau$
- Combine.T2 : USERTYPE$_{\tau_1}$ → USERTYPE$_{\tau_2}$ → COMBINED_TYPE$_{\tau_1+\tau_2}$
- COMBINED_TYPE$_\tau$ ≤ USERTYPE$_\tau$
- COMBINED_TYPE$_{\tau_1+\tau_2}$ ≤ TYPEVIEW$_{\tau_i}$
- COMBINED_VIEWS ≤ TYPEVIEW
- Lift : COMBINED_TYPE → TYPEVIEW → COMBINED_VIEWS

Code modules

- *code module using application-specific type* τ : TYPEVIEW$_\tau$ → USERCODE
- *code module using no application-specific types* : BARECODE
- USERCODE = CORE → sig val init : state -> unit end
- WithType : USERTYPE → BARECODE → USERCODE
- Combine.C2 : USERCODE → USERCODE → USERCODE

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

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 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. We omit the parser signatures `Luaparser.S` and `Parser.MAKER`, which are of little interest.

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 interpreter, 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 typically 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 3.3, 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
```

Here 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`, which appears as a subscript in Figure 1, is a two-level type; when the recursive knot is tied by the definition of `userdata`, the type parameter `'a` will be `value`. The operations `eq` and `to_string` are required because in Lua it must be possible to compare any two values for equality and to convert any value to a string. Because comparing values of type `'a t` may require comparing values of type `'a`, for example, these operations are defined as higher-order functions. Finally, Lua-ML names each type, so if projection fails it can issue an informative error message.

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 (Ramsey 2003).

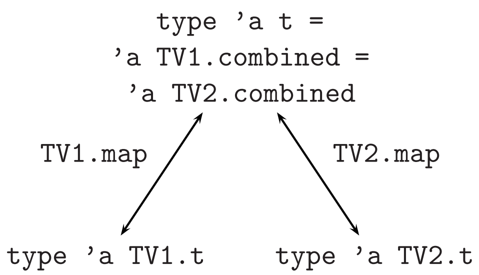


Fig. 2. Views of combined types in `COMBINED_TYPE`

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.T2`, shown below, which takes two `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 matches the `TYPEVIEW` signature, which is defined approximately as follows:

```
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." Such a module is the composition of two 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
end
```

Each view's combined type is equal to the type `'a t` 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 either of the constituent types, but projection might raise an exception.

The `Combine` module provides functor `Combine.T2`, which combines two type modules and returns appropriate views. Because `COMBINED_TYPE` is a subtype of `USERTYPE`, the results of applying `Combined.T2` can themselves be passed to `Combine.T2`:

```
module Combine : sig
  module T2 (T1 : USERTYPE) (T2 : USERTYPE)
    : COMBINED_TYPE with type 'a TV1.t = 'a T1.t
                    with type 'a TV2.t = 'a T2.t
  ...
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.

The idea behind `Combine.T2` is very similar to the idea behind the `OR` type constructor of Liang, Hudak, and Jones (1995). Since Liang, Hudak, and Jones are using Haskell, they define embedding and projection for `OR` types by using type classes, not functors.

Defining and composing code modules

A code module is a library module that initializes an interpreter by registering 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 in Figure 1 as `CORE`, supertype of `EVALUATOR`.

```
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 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 constructor. A definition is suitable if it is consistent with the embedding and projection functions. In other words, we would really like to describe a *family* of signatures, like `BARECODE`, but parameterized over the `userdata'` type constructor in the functor parameter `C`.

We address the second requirement first.

The standard way to parameterize a family of signatures over a constructor like `userdata'` is to make `userdata'` abstract and then specialize it using the `with type` constraint (Harper and Pierce 2005, §8.7). 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.[2]

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

A hand-written code module is unlikely to implement `USERCODE` directly. Instead, it is likely to depend on particular *views*. Because such a module takes one or more views and returns a module matching `USERCODE`, we call it a "pre-`USERCODE`" module. Applying a pre-`USERCODE` code module establishes two type identities:

- The view's application-specific type, `'a t`, is equal to the type on which the code module depends.
- The view's `combined` type constructor is equal to the `userdata'` type constructor in the `USERCODE` module that results from the application.

[2] To remove this limitation, along with several others, Ramsey, Fisher, and Govereau (2005) have proposed some extensions to Caml's signatures language.

As an example, here is a synopsis of the interface to 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
```

Like type modules, code modules can be composed:

```
module Combine : sig
  ...
  module C2 (C1 : USERCODE)
            (C2 : USERCODE with type 'a userdata' = 'a C1.userdata')
    : USERCODE             with type 'a userdata' = 'a C1.userdata'
end
```

Code modules can be composed only if they share one definition of `userdata'`.

4 Putting it all together

Lua-ML's library support may 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 normally 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 type modules using* `Combine.T2`*, and call the result* `T`*.* For interpreters that use more than two type modules, Lua-ML actually provides `Combine.T*` functors in arities up to 10, which has two benefits: in source code, less notation is needed to combine multiple types, and at run time, there is less allocation and pointer-chasing in the implementations of embedding and projection.
2. *From* `T`*, which matches* `COMBINED_TYPE`*, extract and rename each submodule matching* `TYPEVIEW`*.* This step is not strictly necessary, but the submodules have names like `T.TV4`, and renaming them enables subsequent code to use more readable names.
3. *Arrange for code modules to agree among themselves (and with* `T.t`*) on the definition of* `userdata'`*.* Agreement is arranged by specializing each code module to work with `T`:
   - A code module that is pre-`USERCODE` is applied to the relevant views from step 2.

- A code module matching `BARECODE` is associated with `T` by having the functor `WithType (T)` applied to it:

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

4. *Once code modules are specialized, combine them using* `Combine.C2`*, and call the resulting combination* `C`*.* As for type modules, Lua-ML provides `Combine.C*` functors in arities up to 10.
5. *Apply* `MakeEval` *and* `MakeInterp`*:*

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

The `I` module contains everything a client needs to create an interpreter and evaluate Lua code with respect to the interpreter's state.

As an example, here are some excerpts from our C-- compiler. The compiler defines many type modules. Here 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 by following the five steps above. For step 1, we combine type modules. To illustrate nested composition of 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 nested applications of `Combine.T*` functors create a slight complication: module `T1` provides views that map between a child type and its parent type `T1.t`, but what are needed are views that map between a child type and its grandparent type `T.t`. 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 result of `Lift` matches `COMBINED_VIEWS`, which is just like `COMBINED_TYPE` except it does not include `USERTYPE`.

Given `Lift`, the renaming is straightforward.

```
module T1' = Lift (T1) (T.TV1)
module DocTV        = T1'.TV1      module BackplaneTV  = T.TV2
module LuaioTV      = T1'.TV2      module EnvTV        = T.TV3
module AsmTV        = T1'.TV3      module ProcTV       = T.TV4
module AstTV        = T1'.TV4      module TargetTV     = T.TV5
module ColorgraphTV = T1'.TV5      module BlockTV      = T.TV6
```

In steps 3 and 4, we specialize code modules and combine the results using `Combine.C7`. These steps are best done together in one big functor application:

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

Finally, in step 5, 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 embedding and projection functions. In ML, we define the sum constructor as a functor (e.g., `Combine.T2`), not as a type constructor, and we compose embedding and projection functions manually, by functor application; otherwise the designs are similar. Whether you view manual composition as a cost or a benefit depends on your views about implicit computation and on your skills with Haskell's automatic mechanism.

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 significant if, for example, multiple libraries want to use the same control-flow graph.

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, and similar mechanisms for sharing and composing state would be unnecessary, because in practice, types and state are used very differently:

- 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 look at 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

[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 added to Lua-ML.

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 number of 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. 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 extending 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 EML (Millstein, Bleckner, and Chambers 2002). Because EML can distinguish among multiple extensible types, and because it can check for exhaustive pattern matching over an extensible type, its mechanism looks more attractive than simply extending ML's `exn` type, but the mechanism 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, one cannot 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 as 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`.

- *Polymorphic variants.* One could define `userdata'` using polymorphic variants, which allow multiple cases to be combined into a single union without an explicit type declaration (Garrigue 1998). The implementation would be very similar to the implementation using functors, but there would be a few different tradeoffs. Extensions would be combined at the term level (Garrigue 2000); linking would involve defining `eq`, `to_string`, and `tname`. There would be no predefined limit on the number of types that could be combined, and embedding and projection would be simpler. But because the code would depend on the *names* of the variants, it could not be written once and reused, as it is in Lua-ML. On the whole, polymorphic variants would require a bit less work from the implementor of Lua-ML and a bit more work from clients.

Each of these mechanisms enables a solution in which extensions can be independent and in which types need not be composed explicitly, 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 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.

The expression problem

Type safety, separate compilation, and extensibility are elements of what Wadler (1998) has called the *expression problem*. The expression problem demands two kinds of extensibility: it should be possible to add new operations on existing unions, as functional languages are good at, and it should also be possible to add new cases to existing unions, as object-oriented languages are good at. The expression problem is discussed by many authors; I found Zenger and Odersky (2005) especially helpful.

Although Lua-ML does make it possible to use a functional language to add new cases to an existing union (`value`), Lua-ML does not solve the expression problem: it is not possible to add a new operation on `value`s without recompiling existing code.

ML module mania

Lua-ML's use of Objective Caml modules is aggressive—perhaps even maniacal. In particular, Lua-ML uses higher-order 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 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—for which purpose it would need additional API functors—and then apply each USERCODE functor to this CORE. The notational burden would be modest, but the disruption to the API is troubling; Leroy (1995, §2.4) has also observed that hoisting applications outside of functors can disrupt the modular structure of a program. The higher-order functor, although still not essential, is even more valuable than in the previous case.
- 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 higher-order functors can help express pleasing modular structures, they are not needed to build a type-safe, separately compiled, extensible interpreter.

- Although higher-order functors enable a cleaner API, 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`.
- We could even do without functors entirely—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 polymorphic variants or type dynamic, and we could 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 higher-order functors aggressively? To me, the most surprising result is that the only higher-order 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 higher-order functors. While at first I found this conclusion discouraging, on reflection I am neither discouraged nor surprised; after all, although I normally use higher-order *functions* heavily, I manage without them when I program in C. And like higher-order functions, higher-order functors make programming a lot more fun. I hope that designers of future functional languages will include them in their powerful modules systems.

Acknowledgement

João Dias, Simon Peyton Jones, and Sukyoung Ryu helped smooth some rough spots in the manuscript. The referees for the workshop were unusually thorough and energetic in suggesting improvements and related work. Todd Millstein kindly explained some of the fine points of his work. Matthew Fluet and Aleks Nanevski carefully read a very late draft.

This work is part of the C-- project and was supported by NSF grants CCR-0096069, CCR-0311482, ITR-0325460; by a Sloan Research Fellowship; and by a gift from Microsoft. Code can be downloaded from www.cminusminus.org.

References

Arthur I. Baars 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.

Don Batory 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.

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

Nick Benton. 2005 (July). Embedded interpreters. *Journal of Functional Programming*, pages 503–542.

Edoardo Biagioni, 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.

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

Olivier Danvy. 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.

Jacques Garrigue. 1998 (September). Programming with polymorphic variants. In *ACM SIGPLAN Workshop on ML*.

Jacques Garrigue. 2000 (November). Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering (FOSE)*, Sasaguri, Japan.

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

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

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

Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 1996 (November)b. *Reference Manual of the Programming Language Lua 2.5*. TeCGraf, PUC-Rio. Available from the author.

Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. 2001 (May). The evolution of an extension language: A history of Lua. In *V Brazilian Symposium on Programming Languages*, pages B14–B28. (Invited paper).

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

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

Daan Leijen 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.

Xavier Leroy. 1995. Applicative functors and fully transparent higher-order modules. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 142–153.

Sheng Liang, 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.

Todd Millstein, 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.

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

Emir Pasalic and Tim Sheard. 2004 (September). Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587.

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

Norman Ramsey. 2003 (June). Embedding an interpreted language using higher-order functions and types. In *Proceedings of the ACM Workshop on Interpreters, Virtual Machines, and Emulators*, pages 6–14. A revised version of this paper will appear in the *Journal of Functional Programming*.

Norman Ramsey, Kathleen Fisher, and Paul Govereau. 2005 (September). An expressive language of signatures. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 27–40.

Norman Ramsey 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.

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

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

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

Philip Wadler. 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.

Philip Wadler. 1998 (November). The expression problem. Unpublished note on the `java-genericity` mailing list, November 12, 1998; archived at `http://www.daimi.au.dk/~madst/tool/papers/expression.txt`.

Matthias Zenger and Martin Odersky. 2005 (January). Independently extensible solutions to the expression problem. In *The Twelth International Workshop on Foundations of Object-Oriented Languages (FOOL)*. See `http://homepages.inf.ed.ac.uk/wadler/fool/program/10.html`.