Embedding an Interpreted Language Using Higher-Order Functions and Types

Norman Ramsey
Division of Engineering and Applied Sciences
Harvard University

Abstract Using an embedded, interpreted language to control a complicated application can have significant software-engineering benefits. But existing interpreters are designed for embedding into C code. To embed an interpreter into a different language requires a suitable API. Lua-ML is a new API that uses higher-order functions and types to simplify the use of an embedded interpreter. A typical application-program function can be added to a Lua-ML interpreter simply by describing the function's type.

1 Introduction

Suppose you have an application written in a statically typed, compiled language such as C, C++, or ML. If the application is complicated, like a web server or an optimizing compiler, it 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 *embed* its interpreter into an application. 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 application programmer need not implement lexing, parsing, or evaluation; they come from the embedded interpreter.

To gain these benefits, the major effort required is the effort of writing the *glue code* that grants control of the host application to the embedded language.

The benefits above were first demonstrated by Tcl (Ousterhout 1990), which was followed by embedded implementations of other languages, including Python, Perl, and several forms of Scheme (Laumann and Bormann 1994; Benson 1994; van Rossum 2002; Jenness and Cozens 2002), as well as by another language designed expressly for embedding: Lua (Ierusalimschy, de Figueiredo, and Celes 1996a). But to use any of these embedded languages, you have to write your application in C (or C++). If you prefer a statically typed, functional language like ML, this paper explains how you too can reap the benefits of embedded languages.

To create an embedded language, you must design not only the language itself but also an interface that allows host-language application code to be scripted from within the embedded language. This interface—the *embedding API*—is the subject of this paper, which presents Lua-ML, a new API for embedding. Lua-ML provides two significant benefits:

- Type safety is guaranteed: it is impossible for an error in glue code to lead to an unexplained core dump.
- In almost all cases, glue code for a function is replaced by a simple description of the function's type—and this description is checked for correctness at compile time. Applications therefore require significantly less glue code than similar applications written in C.

Lua-ML is supported by two technical contributions: an adaptation of Danvy's (1996) type-indexed functions for partial evaluation (Section 3.1), which makes it easy to embed host-language functions (Section 3.2); and a programming convention that enables host-language functions to inspect or modify the state of an embedded interpreter (Section 3.3).

To focus attention clearly on the API, Lua-ML does not introduce a new host or embedded language; it uses existing languages. As an embedded language, I have chosen Lua, which is clean, flexible, efficient, and easy to implement. Lua enjoys a modest but growing following; its most visible users may be the developers of such popular games as *Grim Fandango*, *Baldur's Gate*, and *Escape from Monkey Island* (Ierusalimschy, de Figueiredo, and Celes 2001). Although convenient, Lua is not essential; it could be replaced by Tcl, Perl, or some other dynamically typed language.

As a host language, I have chosen Objective Caml (Leroy et al. 2001), a popular dialect of ML. Objective Caml provides algebraic data types, programming by pattern matching, higher-order functions, Hindley-Milner type inference, a sophisticated system of parameterized modules, and an object system that is compatible with type inference. Lua-ML uses higher-order functions and types in essential ways, but Objective Caml could be replaced by Standard ML or some other higher-order, typed language. (Preliminary experiments with Haskell show that a similar API should be possible for Haskell. Haskell's type classes reduce glue code even further, but the API is complicated by the need to use monads to describe Lua functions.)

IVME'03, June 12, 2003, San Diego, California, USA.

2 Lua-ML summarized and compared

The distinctive benefits of Lua-ML come from the part of the API used to integrate application-specific code into the embedded interpreter. Because this part would be hard to understand in isolation, especially for readers who have no experience using a similar API, I summarize the full API here. The summary looks at the Lua language, at the API's high-level view of an interpreter, at the transfer of values between host code and embedded code, and at the integration of application-specific code and data into the interpreter.

Because it is difficult to evaluate the merits of an API in isolation, I include comparisons with three established APIs: Tcl 7.3, Lua 2.5, and Lua 4.0. These comparisons point out not only the distinctive aspects of Lua-ML but also those aspects that are similar to related APIs.

2.1 What the language looks like

An API can be hard to understand unless you know a little about the language behind it. This section highlights the salient features of Lua and Tcl.

Lua 2.5 and 4.0 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 current version as of early 2003 is Lua 4.0; 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, 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 has three significant syntactic categories: expression; statement; and top-level *chunk*, which may be a statement or a function definition. Functions may be defined only at top level; Lua 2.5 has first-class, non-nested functions. Versions 4.0 and 5.0 (beta) offer nested functions, and each version has a different mechanism for giving a nested function access to parameters and variables of enclosing functions.

The Lua language has two unusual features. First, a Lua function may accept a variable number of parameters and return a variable number of results. Moreover, the number of actual parameters in a call need not match the number of formal parameters a function expects. If there is a mismatch, the parameters are *adjusted*: if a function receives more actual parameters than it expects, the extra actual parameters are dropped, and if a function receives fewer actual parameters than it expects, extra formal parameters are set to nil. A similar adjustment is applied to results.

Lua's other unusual feature is *fallbacks*, which provide a dynamic exception mechanism similar to a PL/I on-unit. A fallback is a Lua function that is called when a built-in operation fails, e.g., when Lua code tries to retrieve an index not present in a table. Fallbacks can be used to implement error handling, overloading, inheritance and other features. In Lua 4.0, fallbacks are replaced by a similar but more powerful mechanism called *tag methods*. Further details are not relevant to this paper.

Tcl 7.3 Tcl 7.3 is described by Ousterhout (1994). Tcl has no type system or type checking: the only values are immutable strings and named hash tables, and a hash table is represented as a string containing the table's name. Whether a string stands for itself or the name of a table depends on the context in which it appears.

Tcl has only two syntactic categories: expression and command. The command category includes what in another language might be declarations, statements, and function definitions. For example, the `proc` command introduces a new "procedure" command, which executes its body (a sequence of commands) in a fresh environment with private local variables. Tcl's concrete syntax has almost no grammatical structure: it says only that a command is a name applied to a list of strings. The meaning of each string is determined by the command, which decides if a string represents a value, a command or expression that should be evaluated, or a syntactic keyword. For example, the `if` command evaluates its condition as an expression, evaluates its true or false branch as a command, and treats `then` and `else` as keywords.

2.2 How an interpreter appears in the API

At a high level, Lua-ML looks much like other APIs. An application can create many interpreters, each of which has mutable state; it can manipulate the global variables and functions of an interpreter; and it can evaluate Lua code in the context of an interpreter.

An interpreter The state of a Lua-ML interpreter is represented by a value in the host language, Objective Caml. The state includes a table of global variables; a table of fallbacks; and a summary of the call stack, which is printed in the event of a fatal error. An interpreter is created in two stages: compile-time and run-time. At compile time, the application supplies a (possibly empty) set of libraries to an ML module called `MakeInterp`, which returns an interpreter module we will call `Interp`.[1] At run time, calling `Interp.mk` creates a fresh instance of the interpreter; the instance has type `Interp.state`, which we write simply as `state`.

Lua 4.0 and Tcl 7.3 treat the interpreter similarly: its state is represented by a value in the host language (of type `lua_State*` or `Tcl_Interp`, respectively). In Lua 4.0, the visible state of an interpreter includes a stack of values. This stack can be manipulated only through the API; its representation is not exposed. In Tcl 7.3, an interpreter `i` includes a field `i->result`, which is used to communicate results and error messages from commands. The Lua 2.5 API is unusual in that the interpreter and its state are implicit; interpreters cannot be created or destroyed, so an application contains exactly one interpreter.

Variables, functions, and commands In Lua-ML, application code can add or remove global Lua variables or change their values, all by manipulating the global-variable table in the interpreter's state. Functions are treated the same as variables: a Lua function is simply a variable that has a function value. The Lua 4.0 and Lua 2.5 APIs are similar. Lua 4.0 also includes the capability of bundling a function value with other values to form a kind of closure.

The Tcl 7.3 API is only slightly different. It provides one set of API functions for manipulating values, which are strings, and another set of API functions for manipulating commands. A command is represented as a C function bundled with a value of type `ClientData`, which may be a pointer to mutable state. This representation enables a Tcl command to simulate a closure or an object.

[1] The compile-time composition of libraries to form the interpreter module is beyond the scope of this paper; Ramsey (2003) presents details.

Evaluation In each of these APIs, the host application is ultimately in control: It decides what embedded code is evaluated when. Evaluation requires code and an environment, and each system makes the environment part of the state of the interpreter. All four APIs provide functions that evaluate sequences of definitions (or commands), which may be located in strings or files. Most APIs can evaluate definitions at top level only, but Tcl 7.3 makes it possible to evaluate a command either in the top-level environment or in the environment of the currently active function.

Only the Tcl API provides functions that evaluate expressions. Such functions are necessary because a string passed to a Tcl command may represent a keyword, a value, an expression, or another command. It is up to the implementation of the command to know which is which and to call the appropriate API function.

2.3 How values cross the interface

The Lua-ML, Lua, and Tcl APIs differ most greatly in their treatment of values. We consider how an embedded value appears to host code, how memory is managed, and how values are converted between host and embedded forms.

What an embedded value looks like In Lua-ML, a Lua value is represented by a value in the host language (Objective Caml). Such a value has type `value`, which is exposed to an application as follows:

```
type value
  = Nil
  | Number   of float
  | String   of string
  | Function of srcloc * funty
  | Userdata of userdata
  | Table    of table
and funty = value list -> value list
and table = (value, value) Luahash.t
```

This declaration defines `value` to be one of the six alternatives listed above, and it defines `funty` and `table` to be a function type and a hash table, respectively. The type `funty` doesn't mention `state`, which should surprise you, because a Lua function can inspect and modify the state of its interpreter. All is revealed below, in Section 3.3. Types `srcloc` and `userdata` are abstract types, the declarations of which are not shown. The type `srcloc` represents the source location at which a function is defined. The type `userdata` represents application-specific data declared in libraries.

`Luahash.t` is a type constructor exported by a hash-table abstraction that is part of the Lua-ML API. It can be used independently of a Lua interpreter.

In Lua 4.0, a Lua value cannot be represented as a value in the host language: every Lua value is hidden inside an interpreter. Such a value is either on the interpreter's stack or is accessible through an opaque *reference*. API functions that manipulate values do so by referring to stack positions. A reference can be created from a value on a stack and can later be used to push that same value back on the stack.

In Lua 2.5, a value may be on the stack or hidden behind a reference, but it may also be converted to a C value of type `lua_Object`. Some API functions use stack positions and others use arguments or results of type `lua_Object`.

In Tcl 7.3, a value is always represented as a string and has type `char *`. Such strings contain no embedded nulls.

Hash-table abstractions are included in Lua 4.0 and Lua 2.5, but these abstractions can be used only in conjunction with an interpreter, because the hash-table operations use the interpreter's stack. By contrast, Tcl 7.3, like Lua-ML, provides a hash-table abstraction that can be used independently of an interpreter.

Memory management In Lua-ML, both embedded values and host values are managed by the host's garbage collector; the API need not mention memory management. The other APIs don't have the luxury of a built-in garbage collector, so they have to deal with memory management. Both Lua 4.0 and Lua 2.5 provide a special-purpose garbage collector for Lua values only. Of the two, Lua 4.0 has the simpler API for collection, because every root is either on the interpreter's stack or is pointed to by a reference. Tcl does not use garbage collection: Because every value is a string and therefore contains no pointers, it suffices for the API to specify whether application code or the Tcl implementation is responsible for freeing each string's memory, and similarly for allocating.

Conversion between host and embedded values Lua-ML exposes the representation of a Lua value, so functions that convert between Lua values and ML values are not required. Such functions are, however, extremely convenient. Lua-ML provides type-specific conversion functions in pairs: `embed` and `project`. The `embed` function is the mapping from Caml into Lua, which always succeeds; `project` is the mapping from Lua to Caml, which can fail (and raise an exception) if the Lua value has the wrong type. For example, one might convert a Caml floating-point value to a Lua value by calling `float.embed`, or convert a Lua number to a Caml floating-point value by calling `float.project`. The main innovation in Lua-ML is that it provides *higher-order* functions that can create an *unlimited* supply of conversion functions. The details are the topic of Section 3.1.

Lua 4.0 also provides conversion functions, but there are only six pairs: one for each basic Lua type. Because values are abstract, the API also provides a test function for each basic Lua type. For example, one might use `lua_isnumber` to see if a Lua value is a number, `lua_getnumber` to convert a Lua number to a C floating-point value, and `lua_pushnumber` to convert a C floating-point value to a Lua value. The conversion and testing functions, like other functions in the Lua 4.0 API, refer to Lua values on the Lua stack. The conversion interface in Lua 2.5 is similar, except that only conversion from C to Lua uses the stack; the testing functions and the conversion functions from Lua to C accept arguments of type `lua_Object`.

In Tcl 7.3, the API provides conversion procedures for integers, floating-point numbers, Booleans, and lists. To convert from a Tcl value to a C value, one uses a type-specific procedure that writes its result through a pointer and returns a termination code, which indicates success or failure. For example, the API procedure `int Tcl_GetDouble(Tcl_Interp *i, char *s, double *p)` tries to convert the Tcl value `s` to a floating-point value, stores the result in `*p`, and returns a termination code. To convert from a C value to a Tcl value, an application uses standard C procedures such as `sprintf`; no API procedures are needed.

2.4 Application-specific code and data

An API should make it easy to embed application-specific code or data into an interpreter; we compare the APIs.

2.4.1 Embedding a function or command

The *glue code* used to make host functions available in the embedded interpreter represents most of the work of using an embedded language. For each API we consider, we explain how to embed a function, and we show example glue code used to embed the library function `atan2`.

Lua-ML In Lua-ML, an application programmer can define a new Lua function by writing an ML function that takes a list of `values` as arguments and returns a list of `values` as results. (Such a function's access to interpreter state is discussed in Section 3.3.) It is much more convenient, however, to define an ordinary ML function and to convert it to a Lua function by using the `embed` member of an embedding/projection pair. For example, `my_atan2` is the Lua version of `atan2`.

```
let my_atan2 =   my_atan2 : value list -> value list
  let f = func (float **-> float **-> result float)
  in  f.embed atan2
```

On the second line, the expression `func (float **-> float **-> result float)` builds an embedding/projection pair for the ML type `float -> float -> float`, which takes two floating-point arguments and returns a single floating-point result. The Lua-ML operators `func`, `**->`, and `result` are explained in Section 3.2, and their types are shown in Table 2. The box at the top of the display gives the type of `my_atan2`.

The expression `(func` t`).embed` is so common that we provide an abbreviation `efunc` t. We could have written only `efunc (float **-> float **-> result float) atan2`. Thus, the only glue code needed is a description of `atan2`'s type. This lack of glue code is typical of the great majority of embedded functions.

Lua 2.5 and 4.0 In Lua 2.5, an application programmer defines a new Lua function by writing a C function that takes no arguments and returns no results. This function gets its arguments from the Lua stack, and it returns its results, of which there may be more than one, by pushing them onto the stack (on top of the arguments). A function may indicate an error by calling the API function `lua_error`, which uses `longjmp` to achieve the effect of raising an exception.

Here is `atan2` in Lua 2.5. A Lua function conventionally ignores extra arguments, so the number is not checked.

```
void my_atan2(void) {
  if (!lua_isnumber(lua_getparam(1)))
    lua_error("first arg not a number");
  if (!lua_isnumber(lua_getparam(2)))
    lua_error("second arg not a number");
  lua_pushnumber(
    atan2(lua_getnumber(lua_getparam(1)),
          lua_getnumber(lua_getparam(2))));
}
```

The example has a similar flavor in Lua 4.0, except that the state of the interpreter is passed explicitly throughout.

Tcl 7.3 In Tcl 7.3, an application programmer defines a new Tcl command by writing a C function that is passed an interpreter `i` and a list of arguments. Each argument is a string, and the list is represented using the C convention of `argc` and `argv`. An argument may represent a value or it may represent *syntax*, which may be evaluated to produce a value; the intepretation of each argument is up to the command procedure. A command procedure is also passed an argument of type `ClientData`, which makes it possible to write a command that simulates a method of an object or that simulates a closure.

When executed, a command procedure has a side effect on `i->result` and returns a termination code. The code may indicate successful termination, an error, a nonlocal exit such as `break`, or an application-specific termination condition. The termination code, like `longjmp`, provides a way to work around the lack of exceptions in C.

Here is `atan2` in Tcl 7.3.

```
int my_atan2(ClientData d, Tcl_Interp *i,
             int argc, char *argv[])
{
  double x, y;
  if (argc != 3) {
    i->result = "wrong # of args";
    return TCL_ERROR;
  }
  if (Tcl_GetDouble(i, argv[1], &x) != TCL_OK)
    return TCL_ERROR;
  if (Tcl_GetDouble(i, argv[2], &y) != TCL_OK)
    return TCL_ERROR;
  sprintf(i->result, "%f", atan2(x,y));
  return TCL_OK;
}
```

The types of Tcl's conversion procedures enforce an assembly-language style of programming, in which each intermediate result must be named. The Lua 2.5 and 4.0 APIs allow more natural programming, because the result of calling a conversion procedure may be passed directly to another procedure. Lua-ML enables the most natural style: each conversion procedure is used declaratively to describe a function's type, and the ML code for the function itself need not use any conversion procedures.

2.4.2 Embedding application-specific data

Application-specific code often operates on data of application-specific types. Each of the APIs we consider uses a different mechanism to convert an application-specific host value to a value in the embedded language.

In Lua-ML, each application-specific type is declared in a library. The library supplies a definition of the type, a function used to print a value of the type, and a function used to compare two values for equality. Libraries are compiled separately and combined using the ML modules system (Ramsey 2003). The combined libraries define the `userdata` type used in the interpreter, and they provide an embedding/projection pair for each application-specific type. The design provides extensibility and separate compilation while preserving type safety; the details are beyond the scope of this paper.

In Lua 4.0 and 2.5, a value of application-specific type must be represented by a C value of type `void *` and by an accompanying *tag*, which is a small integer. The tag can be used to distinguish different application-specific types. A tag and pointer may be converted to a Lua value of type userdata, from which the same tag and pointer can be extracted. Type safety is ultimately left up to the programmer, but unsafe code can easily be isolated in an application-specific conversion routine. An example appears in Appendix A.

In Tcl 7.3, a value of application-specific type must be represented as a string. Tcl lacks the equivalent of Lua's tag: the API provides no help in distinguishing an application-specific string from any other string, and making sure such strings are unique and are used safely is entirely up to the application. An application programmer is advised to give every value a unique name, to keep a hash table in private state, and to use the hash table to map the name to the value (Ousterhout 1994, p. 283). Knowing when to use this hash table is up to the programmer.

3 Technical contributions of Lua-ML

Lua-ML's advantages stem from its handling of functions.

- A function can be embedded with almost no glue code because embedding can be extended to an unbounded number of types, including function types (Section 3.1).

- Objective Caml and Lua use different models of functions, and each language reacts differently to a function call in which arguments are missing. These differences are cleverly hidden by the embedding/projection pairs for function types (Section 3.2).
- Although almost 90% of embedded Caml functions have no need to inspect or modify the state of a Lua interpreter, some do. Lua-ML supports both kinds of functions without messing up the API (Section 3.3).

3.1 Embedding and projection

This section describes the implementation of embedding and projection functions. To represent an embedding/projection pair, we define 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
```

One example pair is `float`, which has type `float map` and is introduced in Section 2.3 above. The value `float.embed` is the function `(fun x -> Number x)`, which takes the Caml number `x` into the corresponding Lua value, which is built by applying the `Number` constructor to `x`.[2]

Defining `float.project`, which converts from a Lua value to a floating-point number, is more complicated, because in Lua, a string may be used where a floating-point value is expected, provided the string represents a floating-point number. The value `float.project` is the function

```
function
| Number x -> x
| String s when is_float_literal s ->
    float_of_string s
| v -> raise (Projection (v, "float"))
```

This function maps a Lua number to the same Caml number. It also maps a Lua string `s` to the floating-point number represented by `s`, *provided* that `s` satisfies `is_float_literal`, which checks to see that `s` is an appropriate string. If it gets any other kind of value, it raises the `Projection` exception, indicating that the value cannot be converted to a floating-point number. In Lua-ML, every dynamic type error raises `Projection`.

To provide a small set of conversion functions is not new. What Lua-ML adds is the ability to create pairs of conversion functions for *arbitrarily many* ML types. In other words, 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. Danvy (1998) credits Andrzej Filinski and Zhe Yang with originating this family, which has also been independently adapted by Benton (2003) for use in embedded interpreters.

We build a 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 unary type constructor, 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. Such constructors are rare, except for the arrow constructor, which describes a function type. The arrow needs careful treatment because Lua and Caml treat partial application differently.

[2] In Objective Caml, unlike in Haskell or Standard ML, a datatype constructor cannot be used as a function, so the η-expansion is necessary.

```
type 'a map =
  { embed   : 'a -> value ;
    project : value -> 'a }
float    : float    map
int      : int      map
bool     : bool     map
string   : string   map
userdata : userdata map
unit     : unit     map
value    : value    map
table    : table    map
list     : 'a map -> 'a list   map
option   : 'a map -> 'a option map
default  : 'a -> 'a map -> 'a map
```

Table 1: Constructors for embedding/projection pairs

Table 1 gives the types of these functions.

The implementations of these functions are more interesting than you might expect, because Caml and Lua are substantially different. For example, Lua lacks the `int`, `bool`, `list`, and `option` types, and Lua's showcase, the table type, is seldom used in Caml functions. Resolving such inconsistencies requires suitable programming conventions, and the conventions are embodied in embedding/projection pairs. This implementation strategy makes it easy to add new conventions and to use consistent conventions throughout a program.

One such convention is shown above: a string can represent a floating-point number. Here are some others:

- Any Lua value can be interpreted as a Boolean; nil represents falsehood, and every non-nil value represents truth. This convention is embodied by the `bool` pair, which has type `bool map`.

  `bool : bool map`

  ```
  let bool =
    { embed   = (fun b -> if b then Number 1.0
                          else Nil);
      project = (fun v -> v <> Nil)
    }
  ```

- A number may be used where a string is expected.
- A list should be represented as a Lua table, where the elements of a list of length n are stored with keys $1, 2, \ldots, n$.

These conventions, code for which is shown in Appendix B, are part of the idiom of Lua 2.5 and 4.0. Some, like the Boolean and list conventions, have syntactic and semantic support in the Lua language. Another common convention is that a function may allow nil to stand for a default argument. We support this convention with the `default` function, which has type `'a -> 'a map -> 'a map`; the pair `default` v `t` behaves just like the pair `t`, except it projects nil to v.

For Lua-ML, we also invented new conventions. For example, ML has a built-in type constructor `option`. A value of type `'a option` may be `None`, which means the absence of any value, or it may be `Some x`, which means the value `x`, where `x` has type `'a`. In our convention, the Lua value nil stands for `None`, and any other value stands for `Some` of that value. (This convention fails if a value v of type `'a` is itself embedded in Lua as nil, since the convention projects nil as `None`, not as `Some` v.) To build an embedding/projection pair for type `'a option`, we need an embedding/projection pair for type `'a`, which here is called `t`:

```
option : 'a map -> 'a option map
```

```
let option t =
  { embed   = (function None   -> Nil
                      | Some x -> t.embed x);
    project = (function Nil -> None
                      | v   -> Some (t.project v))
  }
```

As shown in the box, `option` has type `'a map -> 'a option map`.

Another convention involves embedding and projection of polymorphic functions. For example, Objective Caml's list-reversal function, `List.rev`, has type `'a list -> 'a list`: It is a polymorphic function that can reverse a list containing any type of value. But Lua does not have parametric polymorphism, so what is the embedding/projection pair that corresponds to the type variable `'a`? It is the `value` pair, which embeds and projects using the identity function. The Lua function

```
efunc (list value **-> result (list value)) List.rev
```

reverses any Lua list, no matter what Lua values the list contains.

Most programming conventions are easily embodied in simple embedding/projection pairs such as are shown above. The big exception is the convention for functions.

3.2 Conventional uses of functions

In Objective Caml, a function of multiple arguments is conventionally defined in its Curried form, i.e., as a function that returns another function. For example, the library function `String.index` has type `string -> (char -> int)`. We normally write such a type as `string -> char -> int`, because the type arrow is right-associative. To apply such a function, we write `(String.index "hello") 'e'`, or because function application is left-associative, simply `String.index "hello" 'e'`. In Objective Caml, there is no real difference between a function that takes two arguments and a function that takes one argument and returns a new function. But in Lua, there is a big difference! The difference can be explained by considering what happens when a function is applied to only some of its arguments, i.e., when it is *partially applied*.

In Caml, a partially applied function, such as `String.index "hello"`, creates a closure, which represents a new function that is returned. This new function, when itself applied to an argument such as `'e'`, behaves as would `String.index` applied to the two arguments `"hello"` and `'e'`. In Lua, a partially applied function is *adjusted*, which means that any "missing" arguments are filled in with nils. In Lua, therefore, `String.index("hello", 'e')` is different from `String.index("hello")('e')`[3], which is equivalent to `String.index("hello", nil)('e')`. By convention, the first, unCurried form is expected.

When embedding a multi-argument Caml function into Lua, we have to convert it from Curried to unCurried form. (One can retain the Curried form by using the Lua-ML operator `-->`, which has type `'a map -> 'b map -> ('a -> 'b) map`, but we have never used this operator in a real program, and its implementation has no interesting features.) We convert a function by describing its type using the `**->` operator.[4] The `**->` operator combines the type of each argument with a *result type*. The result type is not an ordinary embedding/projection pair but a special value that has the abstract type `'a mapf` ("map to function"), where `'a` is the type of the value returned by the Caml function.

[3]This syntax is available in Lua 4.0, which provides first-class functions, but in Lua 2.5 such an expression is not even syntactically correct; it would have to be written `x = String.index("hello"); x('e')`.

[4]The operator's name begins with `**` because that is the only way to make a user-defined operator infix and right-associative.

```
type 'a mapf (* abstract type *)
**->   : 'a map  -> 'b mapf -> ('a -> 'b) mapf
result : 'a map  -> 'a mapf
func   : 'a mapf -> 'a map
```

Table 2: Embedding and projection for functions

Table 2 shows the types of the function-conversion operations. To convert a Caml function into unCurried, Lua form, we take the embedding/projection pair for the result, convert it to a result type using `result`, add the argument types with `**->`, and finally convert the result type back to an embedding/projection pair using `func`. For example, if we have a Caml function of type `t -> u -> v -> w`, we turn it into a Lua function of three arguments by using the embedding/projection pair produced by `func (t **-> u **-> v **-> result w)`.

The representation of type `'a mapf`, which is not exposed in the API, is an embedding/projection pair between `'a` and `value list -> value list` (thus the nickname "map to function"). The function-conversion operations that work with `mapf` are a bit tricky. The simplest is `func`: embedding adds `srcloc` and applies `Function`, while projection strips `Function` and ignores `srcloc`.

```
type 'a mapf = ('a, value list -> value list) ep
func : 'a mapf -> 'a map
```

```
let func (arrow : 'a mapf) : ('a map) =
 { embed = (fun (f : 'a) ->
            Function (caml_fun, arrow.embed f));
   project =
    (function
     | Function (_, f) -> (arrow.project f : 'a)
     | v -> raise (Projection (v, "function")))
 }
```

Value `caml_fun` of type `srcloc` identifies the function as an embedded function. A function translated from Lua source code would have a `srcloc` field indicating its source-code location.

The details of `**->` and `result` are a bit technical, but because the resulting embedding and projection functions are novel, they are worth presenting anyway. The `**->` operation converts between Curried Caml functions and unCurried Lua functions. It builds an embedding/projection pair inductively from `firstarg`, which is an embedding/projection pair for the first argument, and from `lastargs`, which is an embedding/projection pair for a function that takes one less argument. To build `firstarg **-> lastargs`, we need an embedding (`apply`) and a projection (`unapply`).

```
**->    : 'a map -> 'b mapf -> ('a -> 'b) mapf
apply   : ('a -> 'b) -> (value list -> value list)
unapply : (value list -> value list) -> ('a -> 'b)
```

```
let ( **-> ) (firstarg : 'a map) (lastargs : 'b mapf)
        : ( ('a -> 'b) mapf ) =
  let apply (f : 'a -> 'b) = fun actuals ->
    let v, vs = match actuals with []     -> Nil, []
                                 | h :: t -> h, t in
    let f_v = f (firstarg.project v) in
    lastargs.embed f_v vs
  in
  let unapply (f_lua : value list -> value list) =
    fun (v : 'a) ->
      lastargs.project
      (fun vs -> f_lua (firstarg.embed v :: vs))
  in
    { embed = apply; project = unapply }
```

The apply function takes a Caml function f of type 'a -> 'b and converts it to a Lua function of type value list -> value list. This converted function takes its actual arguments actuals, puts the first argument in v, and puts any remaining arguments in vs. (This code also implements *adjustment*: if the list of arguments is empty, it is as if the first argument had been Nil.) Because the Caml function f is Curried, it can be partially applied to the first argument v to produce f_v, which has type 'b. Function f_v is then converted to a Lua function (by lastargs.embed) and applied to the remaining arguments.

The projection function unapply takes a Lua function f_lua and converts it to a Caml function of type 'a -> 'b. The Caml function takes its first argument v and must return a function of type 'b. The Caml function therefore converts v to Lua using firstarg.embed, then builds a new, anonymous Lua function. This anonymous function takes the remaining arguments vs and applies f_lua to all the arguments. The anonymous Lua function is then converted to a Caml function of type 'b by using lastargs.project.

The base case for the conversion of functions is a pair for a function that takes no arguments and returns a result of type 'a. Such a pair is produced by result. In turn, result uses an embedding/projection pair r that converts between a result of type 'a and a Lua value.

`result : 'a map -> 'a mapf`

```
let result (r : 'a map) : ('a mapf) =
 { embed   = (fun (v:'a) -> fun args -> [r.embed v]);
   project = (fun fl -> r.project (take1 (fl [])))
 }
```

To embed a result as a no-argument Lua function, we take the result v and produce a Lua function that ignores its arguments and returns the singleton list holding the result of embedding v. To project a Lua function fl as a result, we apply fl to the empty list of arguments, take the first element of the list of results, and project that Lua value into a Caml value. (If the list of results is empty, take1 returns Lua's Nil, which is another example of adjustment.)

The three functions func, **->, and result cooperate to create a natural mapping between Caml functions and Lua functions. Using this mapping, Caml code defines and uses functions in Curried style, which is natural for Caml. Lua code defines and uses functions in unCurried style, which is natural for Lua. The only aspect that is slightly unnatural is having to apply result to the embedding/projection pair for each result type. Programmers soon learn to apply result, however, because if it is mistakenly omitted, Caml's type checker complains.

3.3 Functions and the interpreter's state

A Lua function can modify the state of its interpreter, e.g., by changing the value of a global variable. Such a function might naturally be expected to have the type state -> value list -> value list. But we actually use value list -> value list. How and why to do so is not obvious at first, and the explanation is one of the contributions of this paper.

We first present two designs that use the type state -> value list -> value list, and we show the flaws in each design. We then present our preferred design.

The incorrect but obviously attractive design Our first design used the function type state -> value list -> value list. As in our current design, the embedding/projection pairs built with result and **-> were designed for pure Caml functions, i.e., those that do not manipulate the state of a Lua interpreter. We therefore needed a slightly different embedding function to be produced by func: instead of the function arrow.embed f, we used the function fun s -> arrow.embed f. Passing the pure Caml function f to arrow.embed converts f to a function of type value list -> value list, and wrapping this function in fun s -> ... yields a function of type state -> value list -> value list. To embed the rare impure Caml function, i.e., one that actually depends on a state, we provided an operation impure_func of type 'a mapf -> (state -> 'a) map.

The sticky part of this design is projecting a function. We are given a function fl' of type state -> value list -> value list. To project fl' to a Caml function that does not expect a state, we need to apply fl' to some state, then project the result (which has type value list -> value list). We figured that if we were projecting a pure function, it wouldn't use its state, so we could apply fl' to an arbitrary state: for example, the empty state. This design worked for a surprisingly long time.

The problem we overlooked occurs in the higher-order case. Suppose we embed a function like List.map, which has type ('a -> 'b) -> 'a list -> 'b list. List.map is pure: if we apply it to function f and list l, it returns a new list containing the results of applying f to each element of l. But when we *embed* List.map, we create a function that *projects* each of List.map's arguments from the type value to the Caml type that List.map expects for that argument. And when we project a Lua function fl', things go wrong: just because List.map is pure, there is no reason to expect its argument to be pure. We were bitten by this bug the first time we applied a higher-order Caml function to a function defined in Lua. The Lua function couldn't find any global variables; even the predefined function print was missing. The globals were missing because our projection had applied fl' to the empty interpreter state, which does not contain any global variables, not even print!

The obviously correct but unattractive design An obvious way to correct the projection problem is to pass the state to each projection function: with each embedding function of type 'a -> 'b, we can pair a projection function of type state -> 'b -> 'a. But the mapf type becomes horrifying:

```
type 'a mapf = (* don't try this at home *)
 { embedf : 'a -> (state -> value list -> value list);
   projectf :
    state -> (state -> value list -> value list) -> 'a
 }
```

This design works and is correct, but the loss of symmetry is discouraging. Passing states around requires extra bookkeeping, which is doubly offensive because states are rarely used: in our largest application, we embed 138 functions into the interpreter, and only 17 of these make any use of state. (And we could cut this number in half by making a small change in the way errors are handled.) If we ignore Lua-library functions and consider only application-specific functions, only 2 of 83 functions use state. In short, the implementation is ugly.

The less obvious, correct, attractive design The type value list -> value list leads to nice embedding/projection pairs, and it is the type of the vast majority of our functions. But in the general case, the type of a Lua function is state -> value list -> value list. Instead of treating every function as an instance of the general case, our best design treats every function as the special case, and it resolves the mismatch by changing the way functions are called.

When a function is called in one of the flawed designs above, it is applied to state and arguments. But any one interpreter has a unique, mutable state. It is therefore possible to *partially apply* a function to the state at an earlier time, save the resulting closure, and when the function is called, apply the closure only to the argu-

```
strindex : int map
init     : state -> unit
```

```
let strindex =
  { embed   = (fun n -> int.embed (n + 1));
    project = (fun v -> int.project v - 1)
  }

let init interp = register_globals
["strlen",   efunc (string **-> result int)
             String.length;
 "strlower", efunc (string **-> result string)
             String.lowercase;
 "strupper", efunc (string **-> result string)
             String.uppercase;
 "ascii",
   efunc (string **-> default 0 strindex
                             **-> result int)
   (fun s i -> Char.code (String.get s i));
 "strsub",
   efunc (string **-> strindex **-> option strindex
                              **-> result string)
   (fun s start optlast ->
     let maxlast = String.length s - 1 in
     let last = match optlast with
       | None -> maxlast
       | Some n -> min n maxlast in
     let len = last - start + 1 in
     String.sub s start len);
 ... (* many more functions omitted *)
] interp
```

Figure 1: Example embeddings from the Lua string library

ments. The closure keeps its reference to the state throughout its lifetime.

Exactly when to partially apply a function to the state depends on how the function is defined.

- If a function is defined in Lua, the interpreter reads the function's definition and builds a closure of type `state -> value list -> value list`. The interpreter has access to its own state, so it can partially apply the closure at that time.
- If a function is defined in Caml, it can't be used until it is *registered* with the interpreter. Registration might involve putting the function in a global variable, or in a table that is stored in a global variable, or indeed in any Lua data structure—but it always requires access to the interpreter's state. So in the rare, general case, a function can be partially applied to the state at the time that it is registered. Such a function can easily be registered with multiple interpreters, because each partial application creates a closure that captures a different state.

Either way, once a function gets into the interpreter, it has type `value list -> value list`. The `state` parameter, if any, is hidden in a closure.

The benefits of this design are that the API matches the common case, the code for embedding and projection is clean, the design is correct, and the general case is accomodated easily. The trick used to handle general functions can also be used to build embedding/projection pairs that have access to state. For example, our optimizing compiler can project a Lua function into an "optimization pass," in which case it uses the state of the interpreter to find a name by which the function should be known.

4 Experience and discussion

We have used Lua-ML to configure and control an optimizing compiler. The glue code for almost every application-specific function is just a type description, as for `atan2` in Section 2.4.1. The glue code for the Lua libraries is more elaborate, because we use the Caml libraries to implement the Lua libraries, and the semantics don't always match. For example, Figure 1 shows the embedding of some representative functions from the Lua string library.

Figure 1 begins with `strindex`, an embedding/projection pair that embodies a programming convention for strings: Lua-strings are 1-indexed, while Caml strings are 0-indexed. Function `init` is the registration function. The first three functions registered require no glue code, because the Lua and Caml versions match exactly. The fourth function, `ascii`, has no Caml version, but it is easy to write, especially using `default` to handle the default parameter. The last function, `strsub`, requires lots of glue code, because in Caml, the third parameter is a length, but in Lua, it is an optional position. This example is atypical and is about as bad as it gets—a cost of choosing existing, incompatible host and embedded languages.

Higher-order functions and types provide great flexibility to the designer of an API for an embedded language. We have exploited that flexibility to make embedding most functions as easy as writing their types. The main idea is that Danvy's (1996) type-indexed family of functions can be adapted to convert values. Making it work requires some trickery in the embedding of functions, plus careful handling of functions that need access to an interpreter's state.

These ideas don't require much code. The parts of Lua-ML discussed here take about 400 lines of Objective Caml; the whole system fits in 3,400 lines. In size, Lua-ML is comparable to the C implementation of Lua 2.5, which is about 6,000 lines.

Others have avoided writing glue code by generating it automatically. For example, `toLua` (Celes 2003) reads a "cleaned" version of a C header file and generates glue code for the functions declared in that file. "Cleaning" must be done by hand. SWIG (Beazley 1996) is more ambitious; version 1.3.16 generates glue code for nine scripting languages. These program generators offer some of the benefits of Lua-ML, but at much greater cost. The `toLua` tool is 8,000 lines of C, and the SWIG system is about 30,000 lines of C; its C parser alone is 4,500 lines. Eliminating glue code using higher-order functions and types takes a fraction of this effort and is easier for users to extend.

Acknowledgements

João Dias, Simon Peyton Jones, Sukyoung Ryu, and anonymous referees helpfully criticized drafts of this paper.

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

References

Beazley, David M. 1996. SWIG: An easy to use tool for integrating scripting languages with C and C++. In USENIX, editor, *Proceedings of the fourth annual Tcl/Tk Workshop*, pages 129–139, Berkeley, CA.

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.

Benton, Nick. 2003. Embedded interpreters. See `http://research.microsoft.com/~nick`.

Celes, Waldemar. 2003 (March). toLua—accessing C/C++ code from Lua. See http://www.tecgraf.puc-rio.br/~celes/tolua.

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.

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2001. *The Objective Caml system release 3.04: Documentation and user's manual*. INRIA. Available at http://pauillac.inria.fr/ocaml/htmlman.

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.

Ramsey, Norman. 2003 (March). ML module mania: A type-safe, separately compiled, extensible interpreter. Available from http://www.eecs.harvard.edu/~nr/pubs/mania-abstract.html.

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

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`.

```
typedef struct mine *Mine;
int mine_tag;
int ismine(lua_Object o) {
  return lua_isuserdata(o)
      && lua_type(o) == mine_tag;
}

Mine getmine(lua_Object o) {
  assert(ismine(o));
  return (Mine) lua_getuserdata(o);
}

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

The cast in function `getmine` cannot be checked by the C compiler, but it is safe as long as `mine_tag` is unique. In Lua 2.5, the uniqueness of the tag is up to the application programmer, but Lua 4.0 contains an API function that allocates a unique tag.

B More conversion functions

This appendix presents implementations of more of the conversion functions defined by Lua-ML.

A number may be used where a string is expected.

`string : string map`

```
let string =
  { embed = (fun s -> String s);
    project =
      function String s -> s
             | Number x -> string_of_float x
             | v -> raise (Projection (v, "string"))
  }
```

A list of length n is represented as a table with keys $1..n$.

`list : 'a map -> 'a list map`

```
let list (ty : 'a map) =
  let table l = (* convert list to table *)
    let n = List.length l in
    let t = Table.create n in
    let rec set_elems next = function
      | [] -> ()
      | e :: es ->
         ( Table.bind t (Number next) (ty.embed e);
           set_elems (next +. 1.0) es )
    in (set_elems 1.0 l; Table t)
  in
  let untable (t:table) = (* convert table to list *)
    let n = Luahash.population t  in
    let get_i i =
      Table.find t (Number (Pervasives.float i)) in
    let rec elems i =
      if i > n then []
      else ty.project (get_i i) :: elems (i + 1) in
    elems 1
  in { embed = table;
       project =
         (function
           | Table t -> untable t
           | v -> raise (Projection (v, "list")))
      }
```

We frequently allow nil to stand for the empty list, so we define a convenience function `optlist` for this case.

`optlist : 'a map -> 'a list map`

```
let optlist ty = default [] (list ty)
```

If desired, the `-->` operator can be used to create a Curried Lua function.

`--> : ('a map -> 'b map) -> ('a -> 'b) map`

```
let ( --> ) arg res =
  { embed =
      (fun f -> caml_func (fun args ->
         [res.embed (f (arg.project (take1 args)))]))
  ; project =
     (function
       | Function (_, f) ->
           (fun x ->
             res.project (take1 (f [arg.embed x])))
       | v -> raise (Projection (v, "function")))
  }
```