JFP 17 1–000, January 2008.
DOI: 10.1017/S0000000000000000 Printed in the United Kingdom

Embedding an Interpreted Language Using Higher-Order Functions and Types

Norman Ramsey
Department of Computer Science
Tufts 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 an API suited to that language. This paper presents Lua-ML, a new API that is suited to languages that provide higher-order functions and types. The API exploits higher-order functions and types to reduce the amount of *glue code* needed to use an embedded interpreter. Where embedding in C requires a special-purpose "glue function" for every function to be embedded, embedding in Lua-ML requires only a description of each function's type. Lua-ML also makes it easy to define a Lua function whose behavior depends on the number and types of its arguments.

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.
- The application can be controlled by an interactive loop that uses the embedded language.
- The application programmer need not write code to lex, parse, or evaluate command-line arguments or configuration files; these tasks are handled by 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; Ierusalimschy 2003). 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 shows that an embedded language can benefit you too, and it explains how to realize those benefits.

To create an embedded language, you must design not only the language 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 primary subject of this paper, which presents Lua-ML, a new API for embedding. Compared with other APIs, Lua-ML provides two significant advances:

- Type safety is guaranteed: it is impossible for an error in glue code to lead to an unexplained core dump. (Lua itself is also type safe.)
- 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. A programmer writing an application therefore writes significantly less glue code than a programmer writing a similar application in C.

Lua-ML is supported by two technical contributions: an adaptation of Danvy's (1996) type-indexed functions for partial evaluation (Section 4.1), which makes it easy to embed host-language functions (Section 4.2); and a programming convention that enables host-language functions to inspect or modify the state of an embedded interpreter (Section 4.3). Type-indexed embedding functions can be combined with ideas from parsing combinators (Fokker 1995) to make it easy to define a Lua function whose behavior depends on the number and types of its arguments (Section 4.4).

These ideas were first described in a workshop paper (Ramsey 2003). This revised paper has new examples, different treatment of related work, some notational improvements, and new technical material on exploiting dynamic typing.

To focus attention 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 *World of Warcraft* (Ierusalimschy, de Figueiredo, and Celes 2001; Ierusalimschy, de Figueiredo, and Celes 2007). 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. 2004), 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. (A similar API would be possible for Haskell, but significant changes would be called for: Haskell's type classes could be exploited to reduce glue code even further, but the API would be complicated by the need to use monads to describe Lua functions.)

A disadvantage of using Objective Caml as a host language—one which applies equally to other functional languages—is that Objective Caml comes with its own garbage collector, which expects to have complete control of the managed heap. Unfortunately, the original implementation of Lua also comes with its own garbage collector, which itself expects to have complete control of the managed heap. To get two such garbage collectors to share values is a problem beyond the current state of the art, so rather than being a thin layer over the existing implementation, the Lua-ML API is exported by a new implementation of Lua written in Objective Caml. While the duplication of effort is regrettable, it does create an opportunity to provide stronger type-safety guarantees than can be provided in the original (Ramsey 2005). And if the memory-management problem is solved in the future, the techniques described in this paper should work equally well with a thin layer.

One question that may puzzle some readers is which language is "on top," the host language or the embedded language? It doesn't matter; in practice, both orientations are useful. In a game like *Grim Fandango*, even though a tremendous amount of the game is written in Lua, the host language is "on top" in the form of an engine that repeatedly renders the world and then calls Lua code (Mogilefsky 1999); a user never sees Lua. By contrast, in Don Libes's `expect` tool (Libes 1994), the embedded language is "on top;" the main application is written as a library, and the user interface is simply the embedded language (here, Tcl) extended with this library.

2 Embedded scripting for functional programs

It might not be clear why an embedded interpreted language could be useful to an application written in ML. After all, languages like ML and Haskell usually have top-level interactive loops, and there are well-studied techniques for expressing embedded-language terms as Haskell terms (Leijen and Meijer 2000; Rhiger 2003), so why not just use ML or Haskell? Why have a scripting language? My reasons appear above, but to add depth, here I present examples of Lua-ML in action.

The major application in which Lua-ML has been used is a retargetable, optimizing compiler for the portable assembly language `C--` (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000). The compiler comprises over 20,000 lines of code written in Objective Caml, plus about 1300 lines of Lua code broken down approximately as follows:

150	Support for configuring the compiler
250	Debugging and utility code
350	The compiler driver
600	Descriptions of back ends, calling conventions, and stack-frame layout

There are significant advantages to having this code in Lua.

- It is easy to change the configuration of the optimizer, the compiler, or even the target machine by putting a simple phrase on the command line. For example, simply writing `qc-- backend=Backend.mips` allows us to cross-compile for our MIPS machine. Cross-compilation is better than compiling natively because our MIPS machine is very slow.

- The Lua code is invaluable for debugging. For example, we can turn off a pass of the compiler by a command-line assignment such as `backend.expand=nil`, and we can make the compiler emit "diagnostic assembly language" by using the assignment `backend.asm=Asm.cmm`.
- The "driver" code in Lua runs not only the C-- compiler but also preprocessors, front ends, the assembler, and the linker. It can therefore be used to compile a test case from start to finish, then do another test without restarting the compiler. The advantage is speed. For example, to test calling conventions, we compile a suite of several hundred test cases. Using the Lua driver, these cases compile three to five times faster than using a Perl script in which the compiler is started separately for each case.

Here's an example of interpreted Lua code from the driver. Like `cc`, `qc--` translates each file named on the command line. The translation puts each file through a sequence of forms, each of which is identified by a file's extension. For example, a C file might be put through the sequence `.c`, `.i`, `.c--`, `.s`, and `.o`; a Tiger file might be put through the sequence `.tig`, `.c--`, `.s`, and `.o`. Function `CMD.translate_files` loops through each of the files and calls `CMD.translate`, which takes a form in a sequence, performs a translation step, and returns a list containing the next form(s) in the sequence. If translation is supposed to stop on reaching a certain form, say a `.s` file, a flag is set in the table `Options.stop`, and the Lua function `CMD.translate_files` stops translation upon reaching such a form.

```
function CMD.translate_files(files)
  local i = 1
  while files[i] do
    local ext = CMD.canonical_extension(files[i])
    if not Options.stop[ext] then
      CMD.translate_files(CMD.translate(files[i]))
    end
    i = i+1
  end
end
```

To translate from one form to the next, `CMD.translate` looks at a filename's extension and calls the appropriate entry in a table called `CMD.compilertab`. Most entries make calls to external programs such as the assembler or linker; the interesting entry is this Lua function, which compiles a C-- program:

```
function CMD.compilertab[".c--"](file)
  local asm = CMD.outfilename(file, ".s")
  Compile.file(backend, Options.globals, file, asm)
  return { asm }
end
```

The examples above show simple interpreted codes that could equally well have been written in a non-embedded interpreter such as `/usr/bin/perl` or `/bin/sh`. The next function, `Compile.file`, also written in Lua, is the first to use embedded ML values, which are highlighted with gray backgrounds. Most important of these values are the functions that parse the input, typecheck it, translate it with the standard optimizer `opt`, and emit it as assembly language.

```
function Compile.file(backend, emitglobals, file, out)
  local ast = Driver.parse(file)
  local opt = { apply  = Compile.run_optimizer
              , action = Opt.standard(backend)
              }
  Driver.compile(backend.target, opt, emitglobals, ast, backend.asm)
  backend.emit(backend.asm, out)
end
```

The driver code is convenient to write in Lua but is seldom changed. The advantages of being able to change configuration on the command line come from describing a back end in Lua. The description is a record of *stages*, each of which is essentially a function from procedures to procedures. Here is a record for the Intel $x86$, written using Lua's curly-brace-and-comma syntax for records:

```
Backend.x86 =
  { target    = Targets.x86
  , widen     = Backplane.seq
                   { Widen.x86_floats, Widen.store_const(32),
                     Stages.assert("proper widths") }
  , placevars = Placevar.context(Placevar.x86, "x86")
  , preopt    = Optimize.remove_nops
  , expand    = Expander.x86
  , liveness  = Liveness.liveness
  , ralloc    = Ralloc.color
  , freeze    = X86.layout
  , rmvfp     = Stages.replace_vfp
  , asm       = Asm.x86
  , emit      = Driver.assemble
  }
```

As shown by highlighting, almost every stage in the Lua record is actually implemented in the host language, Objective Caml—but the stages are composed using Lua. The advantage is that by putting an appropriate Lua assignment on the compiler's command line, we can omit or change any stage, including the assembly-language emitter.

I hope these examples suggest to you that an embedded interpreter is useful. But the main concern of this paper is not with the advantages of an embedded interpreter but with the code needed to make ML values accessible to it. Such code is called *glue code*. When an interpreter is embedded into C, it is common to need a hand-written "glue function" for each function to be embedded. The glue function is actually the middle step of a three-step process:

1. Write (or import) an underyling C function
2. Wrap that function in a glue function that can be embedded.
3. "Register" the wrapper to make it available to the embedded language, often as the value of a global variable. Code for registration is also considered glue code.

For example, the following C code is adapted from Jeske's (1998) HZ game engine. The underlying C functions are highlighted; the two functions given here are wrappers. Although the underlying C functions have different types, every wrapper function has the same type: `void (*)(void)`.

```
static void C_setpalette(void) {
  lua_Object file = lua_getparam(1);
  if (lua_isstring(file))
    I_vid_setpalette(lua_getstring(file));
}

static void C_followsprite() {
  lua_Object objnum = lua_getparam(1);
  if (!lua_isuserdata(objnum))
    lua_error("incorrect argument 1...");
  mainViewPort->followSprite(
    (Sprite *)lua_getuserdata(objnum));
}
```

Finally, the registration step assigns these functions to the values of the *Lua* global variables `C_setpalette` and `C_followsprite`; this code is also written in C:

```
void register_luafunctions() {
  lua_register("C_setpalette",   C_setpalette);
  lua_register("C_followsprite", C_followsprite);
}
```

It is characteristic of an interpreter embedded in C that each function needs a hand-written glue function, and that each glue function needs hand-written code to check and convert each parameter and result. For comparison, here is some glue code that uses the Lua-ML API. It is excerpted from the `Driver` module used in the Quick C-- example above. As indicated by the type of `register_module`, which is shown in the box, registration puts values into a named global table rather than directly into global variables; the code is written in Objective Caml:

```
register_module : string -> (string * value) list -> state -> unit
```

```
register_module "Driver"
  [ "assemble", efunc (asm **->> unit)          (fun a -> a#emit);
    "parse"   , efunc (string **->> ast)        Driver.parse;
    "compile" , efunc (target **-> optimize proc **-> bool **->
                       ast **-> asm **->> unit)  Driver.compile;
  ] session
```

This simplicity is typical. In Lua-ML, the wrapper for an embedded function f is the function `efunc` applied to a description of f's type and to f itself. As in C, each wrapper has the same type (`value`), but it is not necessary to write glue functions by hand, and we can embed more functions and more complicated functions using far less code than is needed using the C API. (The code required to register the resulting wrappers, by contrast, is essentially the same.) A more thorough comparison appears in Section 5, but first I explain what Lua-ML looks like and how it achieves its simple embeddings.

3 The Lua language and Lua-ML API

The distinctive benefits of Lua-ML come from the part of the API that is used to integrate application-specific code into the embedded interpreter. To put these benefits in context, I summarize the Lua language and the Lua-ML API.

3.1 Sketch of the language and its interpreter

The Lua language is typical of small, dynamically typed languages, but it is more carefully crafted than most. 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. As of this writing, the most recent official version is Lua 5.0, which was released in Spring of 2003.

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. By convention, tables are also used to represent lists, sets, and arrays. 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 may be mutable at the application's discretion. Lua's datatypes are fairly typical of small languages, except that reserving a type for extensions (userdata) is unusual.

Like most languages in its class, Lua is an imperative language that distinguishes statements from expressions. In addition to statements and expressions, there is one other significant syntactic category: 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. (Lua 5.0 has nested functions.)

Lua has one unusual feature that complicates embedding: 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.

The use of Lua-ML in an application is typical of embedded languages. Given an embedded interpreter, an application can create many *sessions*, each of which has its own private mutable state. This state is represented by a value in the host language, Objective Caml. The state includes a table of global variables. In Lua-ML, an interpreter is created 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`. A session is created at run time by calling `Interp.mk`. The session's type is `Interp.state` or simply `state`.

Application code can add and remove global Lua variables and change their values, all by manipulating the global-variable table in the session's state. Functions are treated the same as variables: a Lua function is simply a variable whose value is a function.

An interpreter is passive: it evaluates code only at the request of the host application. Evaluation requires both code and a session. The API provides functions that evaluate sequences of top-level chunks, which may be located in strings or files.

3.2 How values cross the interface

In Lua-ML, a Lua value is represented by a value in the host language (Objective Caml). Both embedded values and host values are managed by the host's garbage collector, so the API need not mention memory management. An embedded 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 Lua types, 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 a session. All is revealed in Section 4.3. Type constructor `Luahash.t` is exported by a hash-table abstraction that is part of the Lua-ML API. 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; it is used for debugging. The type `userdata` represents application-specific data, such as an open file handle (Ierusalimschy, de Figueiredo, and Celes 2000) or a set of bigrams used in spam filtering (Siefkes et al. 2004). In Lua-ML, the `userdata` type is defined by composing application-specific libraries; the design provides extensibility and separate compilation while preserving type safety, but the details are beyond the scope of this paper (Ramsey 2005).

Because Lua-ML exposes the representation of a Lua value, functions that convert between Lua values and ML values are not required—but they are very convenient. Lua-ML provides conversion functions in type-specific pairs: `embed` and `project`. The `embed` function maps from Caml to Lua, and it always succeeds; `project` maps from Lua to Caml, and it fails (by raising 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 4.1.

The primary reason to use Lua-ML is to embed application-specific code and data into a Lua interpreter. In using any embedded language, most of the work is in writing the *glue code* that makes host-language functions available in the embedded interpreter, or more precisely, in a given session. In Lua-ML, an application programmer could 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 a session's state is discussed in Section 4.3.) But it is much more convenient to define an ordinary ML function and to convert it to a Lua function by using the `embed` member of an embedding/projection pair, as is done at the end of the previous section by using `efunc` to embed such functions as `Driver.parse` and `Driver.compile`.

4 Technical contributions of Lua-ML

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

- Because embedding can be extended to an unbounded number of types, including function types, a function can be embedded with almost no glue code (Section 4.1).
- 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 4.2).
- Although about 95% of embedded Caml functions ignore the state of the Lua session in which they are embedded, a few need access to this state. Lua-ML supports both kinds of functions without complicating the API (Section 4.3).
- Lua-ML supports a form of type-based dispatch, which makes it relatively easy to define a Lua function whose behavior depends on the number and types of its arguments (Section 4.4).

4.1 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` to 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 mentioned in Section 3.2 above. The value `float.embed` is the function `(fun x -> Number x)`, which takes the Caml number `x` to the corresponding Lua value, which is built by applying the `Number` constructor to `x`.[1]

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

```
float.project : value -> float
```

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

[1] The η-expansion is necessary because in Objective Caml, unlike in Haskell or Standard ML, a datatype constructor cannot be used as a function.

```
type 'a map = { embed : 'a -> value; project : value -> 'a }

val float    : float    map
val int      : int      map
val bool     : bool     map
val string   : string   map
val userdata : userdata map
val unit     : unit     map
val value    : value    map
val table    : table    map
val list     : 'a map -> 'a list   map
val option   : 'a map -> 'a option map
val default  : 'a -> 'a map -> 'a map
```

Table 1. *Constructors for embedding/projection pairs*

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 Zhe Yang (1999) and Andrzej Filinski with originating this family, which has also been independently adapted by Benton (2005) for use in embedded interpreters.

We build a type-indexed family of functions using a library of combinators:

- 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.

Table 1 gives the types of these functions.

To use a particular member of the type-indexed family, we pick a type—for example, list of integers—and to get an embedding/projection pair, we write a function application whose structure follows the structure of the type. Because of the syntactic rules of ML, the syntax of such an application can be startling: for example, if we pick the type `int list`, the function application is written `list int`. In both cases we have the same structure: a type constructor or function `list` is applied to a type or value `int`. But in the type language, application is written *backwards*, with the argument first and the function in postfix position. In the term language, by contrast, application is written with the function first, in prefix position.

The arrow type constructor, written `->` in the type language, is neither prefix nor postfix; it is infix and right associative. In order that the corresponding function in the term language also be infix and right associative, I have given it the name `**->`. (The rules of Caml require that any infix, right-associative operator have a name beginning with `**`. Had I used Standard ML, I could have chosen a name like `-->` and made it infix and right-associative via an explicit *fixity* declaration.) In some contexts, as explained in Section 4.2, we use a different form of the arrow constructor, named `**->>`.

The implementations of the functions in Table 1 are more interesting than you might expect, because the domains of values used in Caml and Lua are substantially different. For example, Lua lacks the `int`, `bool`, `list`, and `option` types, and Lua's most important data type, the table, is seldom used in Caml functions. To account for such differences requires suitable programming conventions, and the conventions are embodied in embedding/projection pairs. By embodying conventions in pairs, we make 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 A, are part of the idiom of Lua 2.5 and 5.0. 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. For example, the value `Some None` of type `'a option option` would embed as nil and then project as `None`.

To build an embedding/projection pair for type `'a option`, we need such a pair (here called `t`) for type `'a`:

```
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));
 }
```

The `option` function has type `'a map -> 'a option map`.

Another convention helps embed and project 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 **->> list value) List.rev
```

reverses any Lua list, no matter what Lua values the list contains. It instantiates `List.rev` at the type `value list -> value list`. As discussed above, this type has the same structure as the phrase `list value **->> list value` in the term language; only the syntax is different.

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

4.2 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, the expression `String.index("hello", 'e')` is not the same as `String.index("hello")('e')`[2], which is equivalent to `String.index("hello", nil)('e')`. Although curried calls such as the last two expressions are permitted in Lua, the first, uncurried form is conventional.

[2] This syntax is available in Lua 5.0, which provides first-class, nested functions, but in Lua 2.5, which does not allow nested functions, such an expression is not even syntactically correct.

```
type 'a mapf (* map to an uncurried function of type 'a *)

val **->    : 'a map  -> 'b mapf -> ('a -> 'b) mapf
val result  : 'a map  -> 'a mapf
val func    : 'a mapf -> 'a map
val efunc   : 'a mapf -> 'a
val **->>   : 'a map  -> 'b map  -> ('a -> 'b) mapf
val results : ('a -> value list) -> (value list -> 'a) -> 'a mapf
```

N.B. t `**->>` $t' \triangleq t$ `**->` `result` t' and `efunc` $t \triangleq$ (`func` t)`.embed`.

Table 2. *Embedding and projection for functions*

Other languages in which currying is supported but not conventional include Perl and Scheme.

When embedding a multi-argument Caml function into Lua, we have to convert it from curried to uncurried form. We convert a function by describing its type using the `**->` and `result` operators, whose types are shown in Table 2. (One can retain the curried form by using the Lua-ML operator `-->`, which is infix, left-associative, and has type `'a map -> 'b map -> ('a -> 'b) map`, but we use this operator rarely. Its implementation is shown in Appendix A.)

The conversion inductively builds a *map to an uncurried function*, or `mapf`. The `mapf` type constructor is abstract; a value of type $(\mathtt{t}_1$ `->` $\cdots$ `->` $\mathtt{t}_n$ `-> t) mapf` represents the ability to uncurry a function of type $\mathtt{t}_1$ `->` $\cdots$ `->` $\mathtt{t}_n$ `-> t`. The inductive conversion works right to left. In the base case, `result` is applied to an embedding/projection pair for `t`, the result type of the function. For example, if the function to be converted returns a value of type `int`, the value `result int` has type `int mapf`. Each induction step uses the `**->` operator to combine the embedding/projection pair for a $\mathtt{t}_i$ with the current map to an uncurried function. During the induction, i goes from n down to 1. When `**->` has been applied to $\mathtt{t}_1$, the induction is complete, and the `func` operator is used to convert the value from type $(\mathtt{t}_1$ `->` $\cdots$ `->` $\mathtt{t}_n$ `-> t) mapf` to type $(\mathtt{t}_1$ `->` $\cdots$ `->` $\mathtt{t}_n$ `-> t) map`. This value is an embedding/projection pair that converts between curried and uncurried forms.[3]

In practice, the conversion is simple. 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)`. In examples and programs, we use the abbreviation `v **->> w` as a shorthand for `v **-> result w`. We also use the abbreviation `efunc m` as a shorthand for `(func m).embed`. In this section, however, we show the underlying mechanism by making `result` and `.embed` explicit.

The representation of a value of type `'a mapf`, which is not exposed in the API, is an embedding/projection pair between `'a` and `value list -> value list`. The function-conversion operations that work with `mapf` are a bit tricky. The simplest is `func`: embed-

[3] The types in Table 2 suggest that `func` and `result` might be inverses or half-inverses, but they are not. I have been unable to discover any interesting laws relating `func` and `result`.

ding adds a source-code location `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 has a `srcloc` field indicating its source-code location.

A value of type `'a mapf` that is passed to `func` must have been built inductively using `**->` and `result`. The details 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 value 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 value 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 results. In Lua, it is possible to return a *list* of results, but in Caml, it is not. If a Caml function wants to return multiple results, it must wrap them in a tuple, and if the function wants to return zero results, it must return the empty tuple. To deal with this mismatch in languages, the base case for conversion of a function requires conversions between the Caml return type 'a and the Lua return type value list.

```
results : ('a -> value list) -> (value list -> 'a) -> 'a mapf
```

```
let results a_to_values a_of_values =
  { embed   = (fun (a:'a) -> fun lua_args -> a_to_values a);
    project = (fun f_lua -> (a_of_values (f_lua []) : 'a));
  }
```

To embed a Caml result as a no-argument Lua function, we take the result a and produce a Lua function that ignores its arguments, converts a to a list of Lua values, and returns the list. To project a Lua function f_lua as a result, we apply f_lua to the empty list of arguments, take the list of Lua results, and convert that list to a Caml value.

In practice, a Caml function almost always corresponds to a Lua function that returns exactly one result. The Lua-ML API supports this common case with two abbreviations: the result combinator takes an embedding/projection pair r for a single result of type 'a, and the **->> combinator applies **-> to a value returned by result.

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

```
let result r = results (fun v -> [r.embed v]) (fun l -> r.project (take1 l))
let ( **->> ) t t' = t **-> result t'
```

Function take1 returns the first element from a list of Lua values, unless that list is empty, in which case it returns Lua's Nil. This computation is another example of adjustment.

It is easy to use results to provide other cases, such as a function that returns a Caml pair as a list of two Lua values or a function that returns the Caml unit as an empty list of Lua values. But unlike the n-argument case for functions, the n-result case for tuples cannot be programmed inductively. Instead, it is necessary to write a different application of results for every n. In our application, this necessity has not been burdensome, because the only case of interest has been pairs ($n = 2$), for which we provide a convenience function. Aside from functions that return pairs, only seven functions use results, and none of them fit the simple return-tuple model. Four of the seven perform some kind of evaluation, so they return the same list of results as the Lua code they evaluate. The other three are the library functions next, nextvar, and strfind, each of which may return a different number of results depending on the arguments with which it is called.

The typical client of the Lua-ML API uses the three functions func, **->, and **->> (result is typically used only to define **->>). These functions 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 awkward bit is having to use **->> instead of **-> to identify the result of a function. Programmers soon learn the difference, however, because if the **-> arrow is used by mistake, Caml's type checker complains.

4.3 Functions and the interpreter's state

A Lua function can modify the state of a session, for example, by changing the value of a global variable. In a pure language, such a function would have a type like `state -> value list -> state * value list`. In an impure language, a more natural type would be `state -> value list -> value list`. At a call site, one would apply the function to a state and to arguments, and the function would return results, possibly also having side effects on the state. But surprisingly, the type `state -> value list -> value list` does not work well with embedding and projection.

The source of the difficulty is twofold: most embedded functions don't use `state`, and we don't hand-write a glue function for each embedded function.

- Our compiler has 194 embedded functions, of which only 10 use `state`. Another 8 functions use an embedding/projection pair that needs access to `state`. Most of the state users either interpret Lua code in the context of a `state`, or they read or set global variables. If we ignore Lua-library functions and consider only application-specific functions, only 1 of 105 functions uses `state` directly.
- If each embedded function requires a hand-written glue function, then it is easy for the hand-written function to ignore `state` as required. But it is not so easy to do so when creating embedding/projection pairs by applying higher-order functions.

If a Caml function `f` does not expect a `state`, when we embed `f` into a Lua function of type `state -> value list -> value list`, we can change the body of `func` to use not the function `arrow.embed f` but instead the function `fun s -> arrow.embed f`. Passing `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 Caml function that uses `state`, we can provide an operation `impure_func` of type `'a mapf -> (state -> 'a) map`.) The sticky part is to *project* a Lua function into a Caml function that does not expect a state. 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 must partially apply `fl'` to some state, then project the result. But no suitable state is available!

In an early implementation of Lua-ML, I tried to solve this problem by applying `fl'` to the empty state; I thought that because the underlying Caml function did not expect a state, any state would do. This code worked for a surprisingly long time, but I had overlooked higher-order functions. For example, suppose we embed `List.map`, which has type `('a -> 'b) -> 'a list -> 'b list`. `List.map` expects no state: 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 just because `List.map` does not expect a state, there is no reason to think that its argument expects no state. In fact, every function compiled from Lua source *does* expect a state, and when we pass such a function to `List.map`, things go wrong.

An obvious way to correct the projection problem is to pass the state explicitly 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 correctly, but the loss of symmetry is discouraging. And passing state explicitly makes the code much uglier. When only one in a hundred application-specific functions needs state, such ugliness cannot be justified.

The solution is to represent a Lua function internally not as a Caml value of type `state -> value list -> value list` but as a value of type `value list -> value list`. For the rare Caml function that expects a state, we partially apply the function to the relevant state *before* the function is converted to a Lua `value`. 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 the state of the session in which it is running, so it partially applies the closure as soon as it is built.
- If a function is defined in Caml, it can't be used until it is *registered* with a session. 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 a session'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 sessions, because each partial application creates a closure that captures a different state.

By capturing the state in a closure instead of passing it explicitly at a call site, we realize several benefits: 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 same trick 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 stage," in which case it uses the state of the session to find a name by which the stage should be known.

4.4 Beyond static typing

The combinators shown above provide plenty of tools for embedding existing ML functions into Lua. But existing ML functions don't take full advantage of Lua's dynamic typing. In a dynamically typed language, there's no need to restrict a function to just one type, or even to a fixed number of arguments. The Lua I/O library exploits this capability:

- The I/O library sends output to a "current output" file, which is changed by `writeto`.
  - — Passing a string to `writeto` opens the file named by that string, makes that file the current output, and returns a "handle" on that file.
  - — Passing a handle to `writeto` makes the handled file the current output.
  - — Passing `nil` to `writeto` makes standard output the current output.

```
val dots_arrow : 'a map -> 'b map -> ('a list -> 'b) mapf
val ( *****-> ): 'a map -> 'b map -> ('a list -> 'b) mapf  (* synonym *)

type alt (* alternative variant of a function *)
val alt     : 'a mapf -> 'a -> alt
val choose  : alt list -> value

val ( <|> ) : 'a map -> 'a map -> 'a map
val ( <@  ) : 'a map -> ('a -> 'b) -> 'b map
```

Table 3. *Constructors for defining dynamically typed Lua functions*

- The `write` function writes any number of strings.
  — When `write`'s first argument is a file, it writes the remaining arguments to that file.
  — When `write`'s first argument is not a file, it writes all arguments to the current output file.

In effect, functions such as `write` and `writeto` are overloaded based on the types of their arguments. Because Lua-ML is dynamically typed, the implementations of such functions must dispatch dynamically on the types of their arguments. Such code can be written by pattern matching on the ML type `value`, but writing glue functions to do pattern matching is just what we are trying to avoid! Lua-ML therefore provides three mechanisms to help programmers define Lua functions that make decisions based on the number and types of their parameters.

- To define a function that accepts a variable number of arguments, Lua-ML provides the `dots_arrow` combinator. Because an application of `dots_arrow` is easier to read if it is infix, we usually abbreviate it using the name `*****->>`.
- To choose among several potential actions based on types and number of arguments, Lua-ML provides the `alt` and `choose` combinators.
- To enable a single ML argument to be represented by multiple Lua types, Lua-ML provides the `<|>` and `<@` combinators, which are based on monadic parsing combinators.

These combinators and their types are summarized in Table 3. I show examples of each, then their implementations.

A variable number of arguments Like Icon, Lisp, Perl, and Python, Lua provides a mechanism by which a function may receive its first few arguments in named parameters and the remaining arguments in a list. Assuming that every "remaining" argument has the same ML type `'a`, we can use `dots_arrow` for a function that accepts a variable number of arguments.

```
val dots_arrow : 'a map -> 'b map -> ('a list -> 'b) mapf
```

The result of type `('a list -> 'b) mapf` can be extended with `**->` and converted with `func` or `efunc` as usual.

One function that uses `dots_arrow` is `format`, which is the Lua analog of C's `sprintf`. The ML implementation of `format` has ML type `string -> value list -> string`, and it is embedded into Lua as follows:

```
register_globals : (string * value) list -> state -> unit
format           : string -> value list -> string
```

```
let ( *****->> ) = dots_arrow in
register_globals
  [ "format", efunc (string **-> value *****->> string) format ] session
```

The `dots_arrow` combinator is also useful in conjunction with type-based dispatch.

Type-based function dispatch The combinators `alt` and `choose` enable a programmer to define a Lua function by giving a list of variant implementations. Each variant may accept a different number of parameters, parameters of different types, or both. When the Lua function is called, the embedding/projection pair selects the first variant that is appropriate to the Lua types of the actual parameters. A variant is represented by a value of type `alt`.

```
type alt  (* an alternative variant of a function *)
val alt    : 'a mapf -> 'a -> alt
val choose : alt list -> value
```

In this interface, `alt` resembles `efunc`. Like `efunc`, `alt` takes a type description of type `'a mapf` and a curried function of type `'a`, but instead of returning a `value` it returns an `alt`. To get a `value`, one applies `choose` to a list of `alt`s. This `value` represents a function that, when called, selects the first of the `alt`s whose type description matches the dynamic types of the actual parameters. If none of the `alt`s matches, the function raises an exception.

As an example, here is the implementation of `writeto`. The field `io.currentout`, which is mutable, represents the current output; `outfile` is the embedding/projection pair for an output file.

```
writeto   : value
to_file   : out_channel -> out_channel
to_string : string      -> out_channel
to_nil    : unit        -> out_channel
to_other  : 'a          -> 'b
```

```
let writeto =
  let to_file file   = (io.currentout <- file; file) in
  let to_string name = to_file (open_out name) in
  let to_nil ()      = (close_out io.currentout; to_file stdout) in
  let to_other _     = ⟨fail with an error message⟩ in
  choose
  [ alt (outfile **->> outfile) to_file;
    alt (string  **->> outfile) to_string;
    alt (unit    **->> outfile) to_nil;
    alt (value   **->> value)   to_other ]
```

As another example, `write` combines type-based dispatch with a variable number of arguments. If the first argument is a file, it writes to that file; otherwise it writes to the current output.

```
write_strings : out_channel -> string list -> int
write         : value
```

```
let write_strings f l =
  (List.iter (output_string file) l; flush f; 1)
```

```
let write = choose
  [ alt (string *****->> int) (fun l -> write_strings io.currentout l);
    alt (outfile **-> string *****->> int) write_strings ]
```

We write the first alternative in η-expanded form because `io.currentout` refers to the contents of a mutable field, which should be evaluated when `write` is called, not when `write` is defined.

Multiple Lua types for a single argument When our C-- compiler places parameters in machine registers, it chooses a register (e.g., floating-point or integer) based on the results of applying predicates to the *kind* and *width* of that parameter (Olinsky, Lindig, and Ramsey 2006). The kind is a string that classifies the parameter's type, and the width is an integer giving the parameter's size in bits. For example, on the SPARC, a kind of `"float"` and a width of 64 might satisfy the predicate for placement in a pair of 32-bit floating-point registers.

In our ML code, we define predicates for common cases: checking for equality of a kind or a width. The always-true predicate is also common. Each predicate has type `string -> int -> bool`.

```
let is_kind  k' = fun k w -> k = k'
let is_width w' = fun k w -> w = w'
let is_any      = fun k w -> true
```

These functions are exported into Lua-ML, but in a large specification, applications of `is_kind`, `is_width`, and `is_any` introduce syntactic noise. We therefore use the following programming convention:

- If a string s is supplied where a predicate is expected, that string is taken to represent the predicate `is_kind` s.
- If an integer n is supplied where a predicate is expected, that integer is taken to represent the predicate `is_width` n.
- If the value nil is supplied where a predicate is expected, it is taken to represent the predicate `is_any`.

And of course, a function can be supplied where a predicate is expected.

To implement this convention, we define an embedding/projection pair for predicates such that we can project a string, an integer, nil, or a function into a predicate. We could, of course, write such a pair directly, but again it would require the sort of pattern matching on `value` that we are trying to avoid. Instead we introduce two new combinators for pairs: an "or" combinator and a continuation combinator.

```
val ( <|> ) : 'a map -> 'a map -> 'a map
val ( <@  ) : 'a map -> ('a -> 'b) -> 'b map
```

The idea is borrowed from parsing combinators; the notation is that of Fokker (1995). The choice operator `<|>` combines two maps. To project, the map `t <|> t'` projects using `t` if possible; otherwise the map projects using `t'`. To embed, the map `t <|> t'` embeds using `t'`.

We use the continuation operator `<@` to apply a function to a value after projection. To project, the map `t <@ f` applies `f` to the result of projecting with `t`. Because function `f` cannot in general be inverted, the map `t <@ f` is not capable of embedding. It is therefore useful only on the left-hand side of the `<|>` operator.

Using these two operators, we implement our choice-predicate convention:

```
choice_pred : (string -> int -> bool) map
```

```
let choice_pred =
  (string <@ is_kind)  <|>
  (int    <@ is_width) <|>
  (unit   <@ (fun () -> is_any)) <|>
  func (string **-> int **->> bool)
```

Implementation The implementations of the functions in Table 3 are straightforward. In structure, the function `dots_arrow` resembles `results`, but it also uses ideas from `**->`. Because it consumes all the arguments it sees, it is simpler than either: it does not have to "peel off" one argument or implement Lua's *adjustment*.

```
let dots_arrow (varargs : 'a map) (answer : 'b map) : ('a list -> 'b) mapf =
  let apply (f : 'a list -> 'b) =
    fun (args : value list) ->
      [answer.embed (f (List.map varargs.project args))] in
  let unapply (f' : value list -> value list) =
    fun (args : 'a list) ->
      answer.project (take1 (f' (List.map varargs.embed args))) in
  { embed = apply; project = unapply }
```

The implementations of `choose` and `<|>` require a new tactic: try each alternative in turn until you find one that works. But to implement this tactic requires that the `'a map` abstraction contain additional information beyond just the functions `embed` and `project`. (You might think you could apply the `project` function and then try the next alternative if the `Projection` exception is raised, but the problem is that you don't know *where* `Projection` is raised, so you don't know if it indicates that you've chosen the wrong alternative or that something is wrong elsewhere.) The additional information takes the form of an `is` predicate. The full definitions of type constructors `map` and `mapf` are therefore as follows:

```
type 'a map =
  { embed   : 'a -> value;
    project : value -> 'a;
    is      : value -> bool;
  }
type 'a mapf =
  { embed   : 'a -> (value list -> value list);
    project : (value list -> value list) -> 'a;
    is      : value list -> bool;
  }
```

The `is` predicate for a value, in type `'a map`, tells whether that value can be projected into ML type `'a`. The predicate for a function, in type `'a mapf`, tells whether a list of arguments would be accepted by that function. Writing the predicates is straightforward; Figure 1 shows the `is` predicates for all the embedding/projection pairs used above.

```
let string     = { is = (function String _ -> true | Number _ -> true
                                   | _ -> false); ... }
let bool       = { is = (fun _ -> true); ... }
let option t   = { is = (function Nil -> true | v -> t.is v); ... }
let func arrow = { is = (function Function(_, _) -> true | _ -> false); ... }

let results r  = { is = (function [] -> true | _ :: _ -> false); ...  }
let dots_arrow varargs answer = { is = List.for_all varargs.is; ... }

let ( **-> ) firstarg lastargs =
  { is = fun args ->
      let h, t = match args with [] -> Nil, [] | h :: t -> h, t in
      firstarg.is h && lastargs.is t; ... }
```

Fig. 1. Definitions of `is` for combinators that build embedding/projection pairs

A programmer must remember that an `is` predicate dispatches on the *Lua* type of a value, not on the ML type that a function might be expecting. For example, because all Lua functions have the same Lua type, it is impossible to dispatch on the exact type of a function; the best one can do is dispatch on the distinction between function and non-function. A similar limitation applies to lists.

Given the `is` predicate, the implementation of type-based dispatch is simple. A value of type `alt` represents an alternative variant of a function. It is represented by the function itself, of type `value list -> value list`, together with a predicate that says when that function should be used. If no predicate is satisfied, `choose` raises an exception.

```
type alt = (value list -> value list) * (value list -> bool)
let alt t f = (t.embed f, t.is)
let choose alts =
  let run args =
    let f = try fst (List.find (fun (_, is) -> is args) alts)
            with Not_found ->
              let args = (list value).embed args in
              raise (Projection (args, "arguments matching alts")) in
    f args in
  Function (caml_fun, run)
```

The implementations of `<|>` and `<@` are even more straightforward.

```
let ( <|> ) t t' =
  { project = (fun v -> if t.is v then t.project v else t'.project v);
    embed   = t'.embed;
    is      = (fun v -> t.is v || t'.is v);
  }

let ( <@ ) t k =
  { project = (fun v -> k (t.project v));
    embed   = (fun _ -> assert false);
    is      = t.is;
  }
```

5 Related Work

Lua-ML's embedding and projection combinators are a type-indexed family of functions. Type-indexed families have been used to address a variety of other problems; Section 5.1 points to some of the most closely related work. Section 5.2 compares and contrasts Lua-ML with three other, related APIs for embedded languages. Finally, Section 6 briefly compares Lua-ML with related tools that work by generating glue code.

5.1 Type-indexed families of functions

As mentioned in Section 4.1, other researchers developed the technique of using higher-order functions to create type-indexed families (Danvy 1996; Danvy 1998; Yang 1999), and Benton (2005) independently applied this technique to embedding and projection. Haskell enables another technique: one can create an indexed family using type classes. For example, Liang, Hudak, and Jones (1995) use type classes to code embedding and projection functions for "extensible union types." Unfortunately, the mapping based on type classes supports only single-argument, single-result functions. My students and I have attempted to generalize this mapping, but we have been unable to devise type classes that implement the currying and uncurrying transformations described in Section 4. Compared with explicit higher-order functions, type classes can also make it hard to see what is going on; for this reason, some readers prefer explicit functions.

Blume (2001) uses an elaborate type-indexing scheme to make representations of C data structures available to Standard ML programs. This scheme defines a family of ML functions indexed by C types; the functions are used to read and mutate C data structures. The family includes some very clever encodings; for example, Blume presents ML type and value constructors that are used only to code the (integer) size of a C array as an ML type.

5.2 Comparable APIs

Lua-ML is motivated by earlier work with embedded interpreters. To see how well the ideas fit with this earlier work, this section compares Lua-ML with three other APIs: Lua 2.5, Lua 5.0, and Tcl 7.3. Lua 2.5 implements the same language as Lua-ML, but in C. Lua 5.0 implements a more recent version of that language. Tcl 7.3 implements a very different language, but is worth comparing because Tcl was the first language designed to be embedded.

All four APIs have many similarities: an interpreter supports multiple sessions, each of which is represented as an abstract value in the host language;[4] each API provides evaluation functions, which the host program uses to control evaluation; and each API provides some conversion and testing functions for base types.

The APIs also have many differences. The differences that matter for embedding and projection are in the parts of the API that specify the treatment of values in the embedded language: how memory is managed, to what degree values and their representations are exposed, what different kinds of values there are, how embedded host functions get their arguments and results, and how application-specific data is embedded and projected.

[4] There is one exception: in Lua 2.5, there is only one session, and this session and its state are implicit.

Memory management Choices about memory management influence every part of an API, including embedding and projection. The key choice, which is made for both host and embedded language, is whether memory for values is managed automatically. In Lua-ML, the host garbage collector manages both host and embedded values, so the API need not mention memory management. In Lua 2.5 and Lua 5.0, C host values are not managed automatically, but embedded Lua values are managed by a garbage collector. The API must make it possible to keep track of roots and internal pointers. In Tcl 7.3, memory for both host and embedded values is managed explicitly with `malloc` and `free`. This choice adds significantly to the complexity of the API, which must specify who allocates and deallocates each embedded value.

Exposure of values A choice that affects both what embedding and projection functions must be provided and what those functions' interfaces look like is the degree to which values are exposed.

- The least exposed choice is to store all values in the state of a session and to allow them to be manipulated only through API functions, including functions for embedding, projection, and type-testing (like `is`). In Lua 5.0, which uses this choice, values are stored on the session's stack, and an API function refers to a value by its stack index.
- A more exposed choice is to allow values to escape the interpreter but to give them an abstract type, as is done in Lua 2.5. Escaping values can be passed among host-language functions and saved in host-language data structures, but they cannot otherwise be manipulated except through API functions. Functions for embedding, projection, and type-testing are required, but they have the convenience of receiving and returning values, not indices.
- The most exposed choice is not only to allow values to escape but also to expose their representations, as is done in Lua-ML and Tcl 7.3. Functions for embedding, projection, and type-testing are not required; they are mere conveniences.

Exposure should be influenced by memory management. In particular, if embedded values are managed automatically and host values are not, the less exposure, the better. When values are hidden in the interpreter's state, it is easy to implement garbage collection, because all roots are part of this state. When values escape, some mechanism must be provided to identify escaped values that should be treated as roots for garbage collection. In Lua 2.5, this mechanism adds significant complexity to the API and to host programs.

Kinds of values An API's functionality for embedding and projection is affected by the kinds of values that are available. The simplest case is Tcl 7.3, which has only one kind of value: the string. Tcl's API includes functions that project a string to an integer, a floating-point number, or a Boolean. It includes no embedding functions; perhaps to avoid issues of allocation, embedding is done with `sprintf`. Because there is only one kind of value, there are no type-testing functions: instead, it is up to the client to know from context what each string represents.

The Lua language has three different kinds of values: immutable, atomic values, such as numbers and strings; mutable, structured tables; and functions. Atomic values are easily supported and are treated similiarly in Lua 2.5, Lua 5.0, and Lua-ML: in each case, the API provides an embedding function, a projection function, and a type-testing (`is`) function.

Tables present more interesting choices. In Lua-ML, if it is necessary to preserve identity in the presence of mutation, a table must be projected into (and embedded from) a Caml value of type `table`. If a table will not be mutated, or if mutations can be ignored, other projections are possible; for example, Lua-ML provides a convenience function, of type `'a map -> (string * 'a) list map`, that projects a table as a list of key-value pairs. The Lua 2.5 and Lua 5.0 APIs provide no embedding or projection functions for tables, only functions that manipulate elements of tables.

The most interesting value to embed and project is a function. As shown below, the critical differences between APIs manifest in the way such a function deals with arguments and results.

Arguments and results of embedded functions In Lua-ML, a host Caml function that is embedded into Lua gets its parameters and results in the usual host-language way. In Lua 2.5, Lua 5.0, and Tcl 7.3, by contrast, a function can be embedded and projected only by means of a glue function that gets arguments from the interpreter using the interpreter's API. Section 2 above shows some examples of hand-written glue functions. Here, we provide an apples-to-apples comparison of glue code required to embed the two-argument arc-tangent function. In Lua-ML, we use the combinators described in Section 4:

`embedded_atan2 : value`

```
let embedded_atan2 = efunc (float **-> float **->> float) atan2
```

In Lua 2.5, we require a hand-written glue function that gets its arguments from the Lua stack and returns its results 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 the glue function written in C:

```
void embedded_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))));
}
```

In Lua 5.0, the example has a similar flavor, except that the state of the session is passed explicitly throughout.

In Tcl 7.3, a glue function is passed a list of string-valued arguments in the form of C variables `argc` and `argv`. The glue function has a side effect on the `result` component of the interpreter's state, and it returns a termination code, which provides a way to work around the lack of exceptions in C. Here is the glue function used to embed `atan2` into Tcl 7.3, again written in C:

```
int embedded_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;
}
```

In these four APIs, we see three ways of dealing with arguments and results:

- In Tcl, each conversion procedure returns a termination code, not a value. A result from one conversion procedure cannot usefully be passed to another conversion procedure, so they cannot be composed. Instead, the types enforce an assembly-language style of programming, in which each intermediate result must be named and procedures are executed for side effect.
- In Lua 2.5 and 5.0, the result of calling a conversion procedure may be passed directly to another procedure. Thus, conversion procedures can be composed at a call site where they are applied.
- In Lua-ML, conversion procedures can be composed *before* being applied. Such a composition is used declaratively to describe a function's type, and the ML code for the function itself need not contain any calls to conversion procedures.

Embedding application-specific data Lua-ML, Lua 2.5 and 5.0, and Tcl use very different techniques for embedding data of application-specific types. In Lua-ML, each application-specific type is declared in a library. Libraries are compiled separately and combined using ML modules (Ramsey 2005). 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 2.5, a value of application-specific type is represented by a C value of type `void *` and by an accompanying *tag*, which is a small integer. The tag is 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. In Lua 5.0, a value of application-specific type is associated not with a tag but with a *metatable*, but the same programming techniques work.

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

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

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

let init = register_globals
[ "strlen",   efunc (string **->> int)    String.length;
  "strlower", efunc (string **->> string) String.lowercase;
  "strupper", efunc (string **->> string) String.uppercase;
  "ascii",    efunc (string **-> default 0 strindex **->> int)
              (fun s i -> Char.code (String.get s i));
  "strsub",   efunc (string **-> strindex **-> option strindex **->> 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 *)
]
```

Fig. 2. Example embeddings from the Lua string library

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.

Summary The crucial properties of Lua-ML's API are that memory for embedded values is managed automatically, embedding/projection pairs can be composed, and the compiler checks that the type of an embedded host-language function is consistent with the corresponding glue code. It is convenient that the API allows embedded values to escape, but there is no significant benefit to exposing their representation.

What may be surprising is that Lua-ML's design *could* be carried over into an API written in C. It would be possible to take a description of a C function's type and use that description to create a glue function dynamically. (Because Lua values are managed by a garbage collector, encoding a closure would be straightforward.) Such a glue function could get at its arguments using C's `stdarg.h` (varargs) mechanism. The main difficulty is that the type of the C function would have to be consistent with its description, and it would be impossible to guarantee this consistency, even with run-time checks. Because errors would be so difficult to diagnose, I expect that embedding and projection with higher-order functions would not be very useful in C.

6 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 5.2. The glue code for the Lua libraries is more elaborate, because we use the Caml libraries to implement the Lua libraries, and the semantics can differ. For example, Figure 2 shows the embedding of some representative functions from the Lua string library.

Figure 2 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 their incarnations in the Lua and Caml libraries have the same semantics. The fourth Lua function, `ascii`, has no counterpart in the Caml library, but it is easy to implement in Caml, 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.

The type-dispatch techniques described in Section 4.4 are ample for our needs. By using these techniques, we removed from the ML code all but one case-dispatch on `value`; the single remaining dispatch is in the implementation of the function `type` from the Lua library, which returns a string that represents the type of a Lua value. Although satisfying, combinators for type-based dispatch are not essential for an embedded interpreter. Indeed, perhaps because our interfaces were designed for ML and not for Lua, we use these combinators rarely; most uses appear as examples in this paper.

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. The same ideas can be applied in other contexts in which a statically typed language needs to manipulate data whose type is not known until run time, such as Blume's (2001) foreign-function interface.

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 3,800 lines. In size, Lua-ML is comparable to the C implementation of Lua 2.5, which is about 6,000 lines.

Performance is also roughly comparable. Although details of performance are beside the point of this paper, it should reassure readers to in end-to-end measurements of individual Lua programs executed by both Lua 2.5 and Lua-ML, the the C code outperforms the Caml version, but not by much. Mostly what matters is hash-table performance.

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

A preliminary version of this paper was presented at the ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME 03) in June, 2003.

Conversations with João Dias helped illuminate design alternatives, and João also gathered statistics about the functions we actually embed. Olivier Danvy, João Dias, Simon Peyton Jones, and Sukyoung Ryu helpfully criticized drafts of this paper. Insightful questions from anonymous referees spurred many improvements; special thanks to the referee who caught an embarrassing error in the type of `embedded_atan2`.

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`. A snapshot is available as a Web appendix to this paper.

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. Libscheme: Scheme as a C library. In *Proceedings of the USENIX Symposium on Very High Level Languages*, pages 7–19.

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

Blume, Matthias. 2001. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First workshop on multi-language infrastructure and interoperability*.

Celes, Waldemar. 2003. `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.

———. 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.

Fokker, Jeroen. 1995. Functional parsers. In Jeuring, Johan and Erik Meijer, editors, *Advanced Functional Programming*, Vol. 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer Verlag.

Ierusalimschy, Roberto. 2003. *Programming in Lua*. Lua.org. ISBN 85-903798-1-7.

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

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

———. 2000. *Reference Manual of the Programming Language Lua 4.0*. TeCGraf, PUC-Rio. Available from `http://www.lua.org`.

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

———. 2007. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, pages 2-1–2-26.

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

Jeske, David. 1998. Hz — a real-time action strategy engine. Unpublished software available from `http://pulp.fiction.net/~jeske/Projects/HZ`.

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

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

Leroy, Xavier, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2004. *The Objective Caml system release 3.08: Documentation and user's manual*. INRIA. Available at `http://pauillac.inria.fr/ocaml/htmlman`.

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

Libes, Don. 1994. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. pub-ORA:adr: O'Reilly & Associates, Inc.

Mogilefsky, Bret. 1999. Lua in Grim Fandango. At `http://www.grimfandango.net/?page=articles&pagenumber=2`, as of October 2007.

Olinsky, Reuben, Christian Lindig, and Norman Ramsey. 2006. Staged allocation: A compositional technique for specifying and implementing procedure calling conventions. In *Proceedings of the 33rd ACM Symposium on the Principles of Programming Languages*, pages 409–421.

Ousterhout, John K. 1990. 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.

Peyton Jones, Simon L., Norman Ramsey, and Fermin Reig. 1999. `C--`: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, Vol. 1702 of *LNCS*, pages 1–28. Springer Verlag.

Ramsey, Norman. 2003. 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.

———. 2005. ML module mania: A type-safe, separately compiled, extensible interpreter. In *ACM SIGPLAN Workshop on ML*, pages 172–202.

Ramsey, Norman and Simon L. Peyton Jones. 2000. 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 (May): 285–298.

Rhiger, Morten. 2003. A foundation for embedded languages. ACM Transactions on Programming Languages and Systems 25 (May): 291–315.

Siefkes, Christian, Fidelis Assis, Shalendra Chhabra, and William S. Yerazunis. 2004. Combining winnow and orthogonal sparse bigrams for incremental spam filtering. In *8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Vol. 3202 of *Lecture Notes in Computer Science*, pages 410–421. Springer.

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

Yang, Zhe. 1999. Encoding types in ML-like languages. Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, *in* SIGPLAN Notices 34 (January): 289–300.

A More conversion functions

This appendix presents implementations of more of Lua-ML's conversion functions.

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"));
    is = (function String _ | Number _ -> true | _ -> false);
  }
```

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 -> Function (caml_fun, (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")));
    is = (function Function (_, _) -> true | _ -> false);
  }
```

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")));
       is = (function Table t -> true | _ -> false);
     }
```

We frequently allow nil to stand for the empty list, for which convention we define a convenience function `optlist`.

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

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