

# A Sample Client for Lua-ML

Christian Lindig  
`lindig@eecs.harvard.edu`

November 10, 2025

This document demonstrates how to extend and embed a Lua interpreter into an OCAML application. We regard this document as a complement to the more detailed documents `lua.nw` and `newlib.nw` that document the general case while we focus on the simpler common case.

## 1 Prerequisites

The Lua interpreter comes as a library `lua-std.cma` and many interface files `lua*.cmi`. All must be found by the OCAML compiler; the compiler can be directed to the library using the `-I` flag. When compiling for native code, the Lua interpreter comes as the additional files `lua-std.cmxa` and `lua-std.a`.

When we assume that our application is implemented by a module `Luaclient` whose source code resides in `luaclient.ml`, our `Makefile` looks like this:

```
1  <Makefile 1>≡
    INTERP      = std
    OCAMLC     = ocamlc
    OCAMLC_FLAGS =
    luaclient: lua-$(INTERP).cma luaclient.cmo
                $(OCAMLC) $(OCAMLC_FLAGS) -o $@ unix.cma lua-$(INTERP).cma luaclient.cmo
```

Defines:  
  `luaclient`, used in chunk 7b.

The `Makefile` links together the `luaclient` binary from the library, the application code, and the Unix module that is required by the Lua code.

## 2 The Big Picture

The Lua interpreter is highly functorized and must be linked together before it can be used. Since we want to use Lua to control our application, we have to extend the Lua interpreter with new primitives that our application implements. These extensions, too, have to be linked into the interpreter. At run time, finally, we have to pass control to the newly created interpreter.

2    *<luaclient.ml 2>*≡  
      *<Linking the Interpreter I 3a>*

```
let main () =
    let argv    = Array.to_list Sys.argv in
    let args   = List.tl argv in
    let state  = I.mk () in (* fresh Lua interpreter *)
    let eval e = ignore (I.dostring state e) in
        (List.iter eval args
         ; exit 0
        )

let _ = main ()          (* alternatively use: module G = Lua.Run(I) *)
```

Defines:  
  **main**, never used.

The interpreter implementation resides in module **I**. Several active interpreters can co-exist because the global state for an interpreter is kept as an explicit value. In our simple **main** function we create a new interpreter (**state**) and evaluate all command line arguments inside. In a more realistic application we probably would evaluate some startup code from a file. In any case, the code we evaluate can use new primitives that we have added to the interpreter and therefore controls our application.

If we just want to type in Lua code interactively we don't even need to write our own **main**. Instead, we can use

```
module G = Lua.Run(I)
```

which adds a **main** function with a read-eval-print loop.

As an example for application-specific data types, we add two new Lua types to the interpreter. A character type (Lua only knows strings), and a polymorphic pair type. Both come with functions to create and observe them.

From a Lua user's point of view the two new types are so-called **userdata** types whose values are accessed from funtions in the tables **Char** and **Pair** that act as modules. For example, **x=Pair.mk("one",2)** creates a pair value

of a string and a number. Each component can be observed by `Pair.fst(x)` and `Pair.snd(x)`, respectively.

### 3 Linking together the interpreter

An interpreter is linked together from a parser and a core, which in turn takes our user-defined types `T` and a library module `L` that depends on them. We will almost always use the standard parser such that the main task is to construct new types and code that uses them.

3a     $\langle \text{Linking the Interpreter I} \ 3a \rangle \equiv$  (2)  
 $\quad \langle \text{user defined types T} \ 3b \rangle$   
 $\quad \langle \text{library module L} \ 3c \rangle$

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

Each user-supplied Lua type is implemented in a module of its own. We link all of them together into one module `T` that we pass into the `MakeCore` functor. The `T` module contains sub-modules, one for each argument, that we name for convenience.

3b     $\langle \text{user defined types T} \ 3b \rangle \equiv$  (3a)  
 $\quad \langle \text{module LuaChar} \ 4 \rangle$   
 $\quad \langle \text{module Pair} \ 5a \rangle$

```
module T = (* new types *)
  Lua.Lib.Combine.T3 (* T3 == link 3 modules *)
    (LuaChar) (* TV1 *)
    (Pair) (* TV2 *)
    (Luaiolib.T) (* TV3 *)

  module LuaCharT = T.TV1
  module PairT = T.TV2
  module LuaiotT = T.TV3
```

The primitive types and functions supplied by the standard interpreter are themselves split across several modules. Thus, we could build an extra-small interpreter by omitting what we don't use. Usually we want all we can get and link `L` together like here:

3c     $\langle \text{library module L} \ 3c \rangle \equiv$  (3a)  
 $\quad \langle \text{new primitives} \ 5b \rangle$

```
module W = Lua.Lib.WithType (T)
module C  =
  Lua.Lib.Combine.C5 (* C5 == combine 4 code modules *)
    (Luaiolib.Make(LuaioT))
    (Luacamllib.Make(LuaioT))
    (W (Luastrlib.M))
    (W (Luamathlib.M))
    (MakeLib (LuaCharT) (PairT))
```

The IO, math, and string library are standard; our own code resides in `MakeLib` and is parametrized over the new Lua types (`LuaCharT`, `PairT`) that we have introduced. Because the string and math libraries have signature `Lua.BARE` they need to be extended with a type (any will do), before they can be combined with others.

## 4 New primitive types and functions

Most of the code above provides necessary infrastructure. The real work is implementing new primitive Lua types and functions.

Most often we want to add not just new functionality but add also an application-specific type to the Lua interpreter. Each type is represented by a module of module type `Lua.USERDATA`. As an example, we add a new type that represent characters. The `LuaChar` module provides: an OCAML representation for the new type, a name of the type, an equality predicate, and a function to represent a datum as a string.

```

4      ⟨module LuaChar 4⟩≡                                (3b)
        module LuaChar = struct
            type 'a t          = char
            let tname         = "char"
            let eq _           = fun x y -> x = y
            let to_string     = fun _ c -> String.make 1 c
        end

```

As a somewhat more complicated example we also add a polymorphic pair that works with all Lua values. Although we don't know the representation of Lua values here, we do know that the type parameter '`a`' of `t` represents the actual value data type of the interpreter. Therefore, our representation is simply a polymorphic pair. The foresight of the interpreter's designer also helps us with the problem of printing values in the `to_string` function: the first parameter `f` to `to_string` is a function that prints any value, such that `(f x)` gives us the string of value `x`.

Usually we have functions to work with the new types. We can implement them outside or inside the module that provides the type. In the case of `Pair`, we added `mk`, `fst`, and `snd`.

```
5a  ⟨module Pair 5a⟩≡ (3b)
  module Pair = struct
    type 'a t      = 'a * 'a
    let tname     = "pair"
    let eq _       = fun x y -> x = y
    let to_string = fun f (x,y) -> Printf.sprintf "(%s,%s)" (f x) (f y)
    let mk x y    = (x,y)
    let fst       = fst
    let snd       = snd
  end
```

The approved way to link together the modules that extend an interpreter is to write a `MakeLib` functor. It has an argument for each new type, where each but the first one comes with a sharing constraint for the `combined` type. Intuitively, these constraints ensure that all modules use the same representation for values in the interpreter.

Note, that the arguments to `MakeLib` are *not* the modules `LuaChar` and `Pair` that we just have defined, but the ones re-exported by the `Lua.Lib.Combine.Tn` functor.

```
5b  ⟨new primitives 5b⟩≡ (3c)
  module MakeLib
    (CharV: Lua.Lib.TYPEVIEW with type 'a t      = 'a LuaChar.t)
    (PairV: Lua.Lib.TYPEVIEW with type 'a t      = 'a Pair.t
                                and type 'a combined = 'a CharV.combined)
    : Lua.Lib.USERCODE with type 'a userdata' = 'a CharV.combined = struct

      type 'a userdata' = 'a PairV.combined
      module M (C: Lua.Lib.CORE with type 'a V.userdata' = 'a userdata') = struct
        module V = C.V
        let ( **-> ) = V.( **-> )
        let ( **->> ) x y = x **-> V.result y
        ⟨register new functions in interpreter 6a⟩
      end (* M *)
    end (* MakeLib *)
```

Finally we have to register the new functions in the interpreter. The most important aspect is the conversion back and forth between the value representation in the interpreter, and our (much simpler) representation that we have provided in `LuaChar` and `Pair`. It is good practice to collect these

conversion functions into one module `Map` with a function for each type.

6a    *⟨register new functions in interpreter 6a⟩* ≡ (5b)

```

    module Map = struct
        let pair = PairV.makemap V.userdata V.projection
        let char = CharV.makemap V.userdata V.projection
    end

    let init g =
        ⟨register Pair 6b⟩
        ⟨register Char 6c⟩
        ⟨register Example 7a⟩

```

Defines:

**init**, never used.

Once we have `Map`, we can provide a mapping between a Lua name like `Pair.mk` and its OCAML implementation. The `register_module` function takes a list of (name, value) pairs, where a value can be a function. The conversion between the interpreter's internal representation and our's is provided by a clever infix function `**->` that makes the conversion function look like a function type. The `Map` module is here essential to name the user-defined argument types.

```

6b   ⟨register Pair 6b⟩≡ (6a)
      C.register_module "Pair"
          [ "mk", V.efunc (V.value **-> V.value **->> Map.pair) Pair.mk
            ; "fst", V.efunc (Map.pair **->> V.value)                  Pair.fst
            ; "snd", V.efunc (Map.pair **->> V.value)                  Pair.snd
          ] g;

```

The registration of the `Char` module shows how to deal with errors in conversions. `Char.mk` expects a string whose first character is used to create the new character value. But what if this string is empty? We catch this problem here where we have the core interpreters's `error` function available, rather in `LuaChar`, where we don't. The argument `g` is the global interpreter state that must be passed to `error`. State `g` is an argument to `init` inside whose body we are. Error reporting isn't a problem for `Pair` because all functions in `Pair` are total.

```

6c   <register Char 6c>≡ (6a)
      C.register_module "Char"
          [ "mk", V.efunc (V.string **->> Map.char)
            (function
              | "" -> C.error "Char.mk: empty string"
              | s -> s.[0]

```

```

)
]
g;
```

Sometimes we want to add functionality for existing types without adding a new type. This case is easy because we simply can add the new functions without having to define extra modules for types. As an example, we provide some functions from the OCAML standard library. To avoid name space pollution we introduce an extra layer `Example`.

7a  $\langle \text{register Example} \rangle \equiv$  (6a)  
`C.register_module "Example"
 ["argv", (V.list V.string).V.embed (Array.to_list Sys.argv);
 "getenv", V.efunc (V.string **-> V.string) Sys.getenv;
 ]
g;`

With all explanations the client of our interpreter looks quite big. In fact, it is just about 100 lines long. Taking a look at the `luaclient.ml` gives us the more linear perspective of the compiler which is also instructive to understand this code.

## 5 Running the interpreter

After we have compiled and linked our client with the `lua-std.cma` library we can run it. Our `main` function simply evaluates all command line arguments from left to right. The example below shows, that our extensions are indeed part of the interpreter.

7b  $\langle \text{run 7b} \rangle \equiv$   
`% ./luaclient 'c=Char.mk("x")' 'print(c)'
x

% ./luaclient 'x=Pair.mk("one",2)' 'print(x)' 'print(Pair.fst(x))'
(one,2)
one

% ./luaclient 'print(Caml.Filename.chop_extension("foo.bar"))'
foo`

Uses `luaclient` 1.

If your Lua code is in a file, a `dofile("file.lua")` will make the interpreter read and execute it. This is useful for repetitive testing.

## 6 Further reading

This document provides a first recipe to embed the Lua interpreter into an application but it cannot explain every detail. The next document you should try to understand is Section 1.1 Values in `lua.nw` that documents the Lua API. In particular, Section 1.1 lists all conversion functions that are available to map values between their OCAML and Lua representation. The rest of `lua.nw` explains the multiple ways libraries can be combined and is important when you want to combine more than 10 libraries or types, or you want to go to the limits in other ways.