Programming in Lua – User-defined types

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Exposing data

- We have seen how to expose C functions to Lua
- But C libraries do not have just functions, they usually also define complex data structures, and their functions operate on these data structures
- We need a way to pass these data structures to Lua code, and get them back
- One possible way would be to marshall these data structures to Lua data structures, such as strings or tables, but doing this marshalling and unmarshalling anytime we call C and come back would be expensive!
- Fortunately, C functions can pass opaque pointers and binary blobs to Lua and get them back with no marshalling/unmarshalling, with *userdata*

Bit vectors

- Suppose we have a library for *bit vectors* for representing boolean arrays efficiently, this library has the following interface (`bv.h`):

```
typedef struct BitVector BitVector;
int  bv_bytes (int n); /* size needed for a bitvector of n elements */
void bv_set   (BitVector *bv, int i, int b);
void bv_clear (BitVector *bv);
int  bv_get   (BitVector *bv, int i);
/* sets and gets length of bit vector, so users can do bounds checking */
void bv_setn  (BitVector *bv, int n);
int  bv_getn  (BitVector *bv);
```

- The library does not do bounds-checking, this is the responsibility of the caller; we want to expose the following interface to Lua code:

```
v = bv.new(n)     -- create bit vector of n elements
bv.set(v, i, b)   -- set ith element to truth val of b
b = bv.get(v, i)  -- get boolean val of ith element
n = bv.len(v)     -- get length of bit vector
```

- The Lua API will be bounds-checked, and throw errors for out-of-bound access

Userdata

- Function `bv.new` must allocate a new bit vector, and return it to Lua after setting the length so the other functions can do bounds-checking, and clearing the vector

- `lua_newuserdata` allocates a block of memory, pushes an *userdata* for this block, and returns the address of the block:

```
static int newbv(lua_State *L) {
    int i; BitVector *bv;
    int n = luaL_checkinteger(L, 1);
    luaL_argcheck(L, n >= 1, 1, "size must be positive");
    bv = (BitVector*)lua_newuserdata(L, bv_bytes(n));
    bv_setn(bv, n);
    bv_clear(bv);
    return 1;
}
```

- We just need to register this function as `new` in our `bv` module

Back from Lua

- The other functions in our bv library take the bit vector userdata as the first argument; we can use the `lua_touserdata` function to take them out of the stack:

```
static int setbv(lua_State *L) {
    BitVector *bv = (BitVector*)lua_touserdata(L, 1);
    int i = luaL_checkinteger(L, 2) - 1;
    int n = bv_getn(bv);
    luaL_argcheck(L, 0 <= i && i < n, 2, "index out of range");
    luaL_checkany(L, 3);
    bv_set(bv, i, lua_toboolean(L, 3));
    return 0;
}
```

- Again we use `luaL_argcheck`, now to do bounds checking; `lua_toboolean` works with any Lua value, so we just need to check for the presence of a third argument with `luaL_checkany`

Safety

- The memory we allocate for userdata is managed by Lua, so we do not need to free it; Lua’s garbage collector takes care of it
- But our implementation is unsafe in another way: there is no checking to see if the user has actually passed a valid bit vector as a first argument
- If the user does not pass an userdata `lua_touserdata` returns `NULL`, and we could check for that
- But other libraries produce their own userdata, and our code will happily corrupt them
- We need a way to *tag* an userdata so our module can check to see if it is a bit vector or not

Metatables

- Userdata can have metatables, too, so we will tag our bit vector userdata with a shared metatable that we will keep in the registry

- The Lua API has three convenience functions to access these metatables:

```
/* creates an empty metatable, pushes it and adds to the registry */
int   luaL_newmetatable(lua_State *L, const char *name);
/* pushes the metatable from the registry */
void  luaL_getmetatable(lua_State *L, const char *name);
/* checks if the value at stack index i is an userdata with correct metatable */
void *lua_checkudata   (lua_State *L, int index, const char *name);
```

- Lua uses `name` as the registry key for the metatable, so prefix it with the module name

- Function `lua_setmetatable(L, i)` pops a table from the stack and sets it as the metatable of the value at index `i` (a table or userdata)

Safe bit vectors

- To tag our bit vectors, we need to create the metatable when loading the module:

```
int luaopen_bv(lua_State *L) {
    luaL_newmetatable(L, "bv.mt");
    luaL_newlib(L, bv);
    return 1;
}
```

- When we create the userdata for a new bit vector, we need to tag it, changing the end of newbv to:

```
luaL_getmetatable(L, "bv.mt");
lua_setmetatable(L, -2);
return 1;
```

- Finally, we change the other functions to use luaL_checkudata instead of lua_touserdata:

```
BitVector *bv = (BitVector*)luaL_checkudata(L, 1, "bv.mt");
```

Userdata objects

- Now that we have a metatable, we can add metamethods to it; we can decide what kind of interface we want: `v:get(i)`, `v:set(i, b)`, and `v:len()`, or `v[i]`, `v[i] = b`, and `#v`

- For the first one, we can point `__index` to the metatable itself, as we did for classes, and then add the `get`, `set`, and `len` functions to the metatable:

```
static const struct luaL_Reg bv_m[] = {
    {"set", setbv}, {"get", getbv}, {"len", lenbv}, {NULL, NULL}
};
int luaopen_bv(lua_State *L) {
    luaL_newmetatable(L, "bv.mt");
    lua_pushvalue(L, -1);
    lua_setfield(L, -2, "__index");
    luaL_setfuncs(L, bt_m, 0);
    luaL_newlib(L, bv);
    return 1;
}
```

```
v:get(5) -> v.get(v, 5) ->
getmetatable(v).__index.get(v, 5) ->
bv.get(v, 5)
```

- We do not need to change the `setbv`, `getbv`, or `lenbv` functions, as they already take the userdata as the first parameter

Userdata objects (2)

- For the second interface, we can just set the `__index` metamethod to `getbv`, `__newindex` to `setbv`, and `__len` to `lenbv`:

```
v[5] -> getmetatable(v).__index(v, 5) -> bv.get(v, 5)
v[5] = true -> getmetatable(v).__newindex(v, 5, true) -> bv.set(v, 5, true)
#v -> getmetatable(v).__len(v) -> bv.len(v)
```

- Using `luaL_setfuncs` to initialize the metatable is straightforward:

```
static const struct luaL_Reg bv_m[] = {
    {"__newindex", setbv}, {"__index", getbv}, {"__len", lenbv}, {NULL, NULL}
};
int luaopen_bv(lua_State *L) {
    luaL_newmetatable(L, "bv.mt");
    luaL_setfuncs(L, bt_m, 0);
    luaL_newlib(L, bv);
    return 1;
}
```

External resources

- Suppose the bit vector library manages its own memory, exposing the following interface:

```
typedef struct BitVector BitVector;
BitVector *bv_new (int n);
void        bv_free(BitVector *bv);
void        bv_set (BitVector *bv, int i, int b);
void        bv_clear (BitVector *bv);
int         bv_get (BitVector *bv, int i);
int         bv_getn(BitVector *bv);
```

- Now bv_new allocates a cleared bit vector, and sets is length; when the user is done with the bit vector he should call bv_free to reclaim its space

- How do we expose these bit vectors to Lua?

Indirect userdata

- The userdata for bit vectors can be *pointers* to the bit vectors, instead of the bit vectors themselves:

```
static int newbv(lua_State *L) {
    int i; BitVector **ud;
    int n = luaL_checkinteger(L, 1);
    luaL_argcheck(L, n >= 1, 1, "size must be positive");
    ud = (BitVector**)lua_newuserdata(L, sizeof(BitVector*));
    *ud = bv_new(n);
    luaL_getmetatable(L, "bv.mt");
    lua_setmetatable(L, -2);
    return 1;
}
```

- The other functions need to deal with the extra level of indirection:

```
BitVector *bv = *((BitVector**)luaL_checkudata(L, 1, "bv.mt"));
```

Finalizers

- There is still a problem: when should we call `bv_free`? The memory for storing the pointer is managed by Lua, but the bit vector itself is not
- To solve this, Lua has *finalizers*: a finalizer is a `__gc` metamethod in a userdata (in Lua 5.2 tables can have finalizers, too)
- Lua calls this metamethod just before garbage-collecting the userdata, passing the userdata to it; we just need to connect the `__gc` metamethod to a function that will call `bv_free`:

```
static int gcbv(lua_State *L) {
    BitVector *bv = *((BitVector**)luaL_checkudata(L, 1, "bv.mt"));
    bv_free(bv);
}
```

Quiz

- Suppose the bit vector library we are exposing to Lua did not have the `bv_setn` and `bv_getn` functions; how could we implement bounds checking in our C module?

SEE BV QUIZ FOR ANSWER!