Closures in Lua

Roberto Ierusalimschy* Luiz Henrique de Figueiredo† Waldemar Celes*

(version by lhf, April 24, 2013 at 12:50 A.M.)

Abstract

First-class functions are a very powerful language construct and a fundamental feature in functional languages. However, few procedural languages support this feature, due to their stack-based implementation. In this paper we discuss a new algorithm for the implementation of first-class functions used in the implementation of Lua 5.x. Unlike previous techniques, our algorithm does not need static analysis of the source code (a luxury that the on-the-fly Lua code generator cannot afford) and does not prevent the use of an array-based stack to store regular local variables.

1 Introduction

First-class functions are a hallmark of functional languages, but they are a useful concept in procedural languages too. Many procedural functions may be more generic if coded as higher-order functions. For instance, even in C the standard sort function (`qsort`) has its comparison function given as an external argument, in the form of a function pointer. Callback functions are another example of the usefulness of first-class functions in procedural languages. Moreover, procedural languages that support first-class functions allow programmers to borrow programming techniques from the functional world.

Object-oriented features can partially fulfill the needs of higher-order functions. After all, a first-class object carries functions (methods) with it. However, classes are often too cumbersome when all the programmer needs is a single function. So, even object-oriented languages offer other mechanisms for better support of higher-order functions, such as anonymous inner classes in Java [AGH05] and delegates and anonymous methods in C# [ISO06].

Lua [IdFC96, Ier03] is a procedural scripting language widely used in the game industry that uses first-class functions extensively. Its standard library contains several higher-order functions: for instance, `sort` takes a comparison function; `gsub`, which does pattern matching and replacement on strings, can receive a *replacement function* that receives the original text matching the pattern and returns its replacement. The standard library also offers *traversal functions* that receive a function and call it on every element of a collection (an imperative version of `map`).

*Department of Computer Science, PUC-Rio, Rio de Janeiro, Brazil, roberto,celes@inf.puc-rio.br
†IMPA–Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, Brazil, lhf@impa.br

In Lua, as in Scheme, all functions are anonymous dynamic values created at run time. Lua offers a conventional syntax for creating functions, like this:

```
function fact (n)
  if n <= 1 then return 1
  else return n * fact(n - 1)
  end
end
```

However, this syntax is simply sugar for an assignment of an anonymous function to a regular variable:

```
fact = function (n) ... end
```

First-class functions are also at the core of the Lua interpreter.[1] Unlike most other interpreted languages, which offer an *eval* primitive to evaluate an arbitrary piece of code represented as data, Lua offers a *compile* primitive, called `loadstring`, which returns an anonymous function equivalent to the given arbitrary piece of code. Of course, *eval* and *compile* are equivalent—given one of them, it is easy to implement the other. However, we consider that *compile* is a somewhat better primitive, because it has no side effects and it clearly separates the transformation from data into code from the actual execution of the resulting code.

Lua supports first-class functions with lexical scoping via closures [Lan64, Car84]. Since simplicity is one of the main goals of Lua, any acceptable implementation of closures for Lua must satisfy the following requirements:

- It should have no impact on the performance of programs that do not use non-local variables.
- It should provide acceptable performance for imperative programs, where side effects (e.g., assignments) are the norm.
- It should be compatible with the standard execution model for procedural languages, where variables live in activation records allocated in an array-based stack.
- It should be amenable to a one-pass compiler that generates code on the fly, without intermediate representations.

The first two requirements apply to any imperative language, where higher-order functions may play an important role but are not at the core of most programs. The third requirement also applies to imperative languages in general, but it is stronger in Lua, because of its simple compiler. The last requirement is specific to Lua, as explained below.

Lua has a tiny implementation. The whole interpreter, with the compiler and the standard libraries, comprises less than 17,000 lines of C code. Lua is used often on devices

[1] The evolution of this feature in Lua is described in [IdFC07].

with restricted memory, such as the Lego Mindstorms NXT (256K of flash memory plus 64K of RAM) [Hem08]. Moreover, Lua is frequently used for data representation. Lua programs representing data may have huge expressions, spanning many thousands lines of code. All those characteristics call for a small and fast compiler.

The entire Lua compiler (scanner plus parser plus code generator) has less than 3,000 lines of C code. Lua 5.1 compiles four times faster than Perl 5.8 and 30 times faster than gcc 4.2 with no optimizations.[2] The absence of any intermediate representation is a key ingredient in achieving this performance, as the compiler does not waste time building and traversing data structures.

Lua supports restricted first-class functions since its first version, released in 1994. However, older versions did not support proper lexical scoping, because implementation techniques at the time did not fit those requirements. In 1998, Lua 3.1 introduced the concept of *upvalues*, which allowed nested functions to access the values of external variables, but not the variables themselves [IdFC07]. In 2003, Lua 5.0 introduced full support for first-class functions with proper lexical scoping, using a novel implementation. However, with the exception of a short description in [IdFC05], this implementation for closures has not been documented yet. This paper aims to close this gap, giving a detailed description of the implementation of closures in Lua.

In the next section we discuss some implementation techniques for first-class functions with lexical scoping. Section 3 describes our implementation. Section 4 evaluates the result, both analytically and with a few benchmarks. Finally, in Section 5 we draw some conclusions.

2 Closures in other Languages

In this section we discuss current implementations for first-class functions with lexical scoping and similar mechanisms in some languages. We shall first look at implementations for functional languages and then we shall cover imperative languages.

All functional languages have adequate support for first-class functions with lexical scoping; after all, this is one of the main features for a language to be called functional. Imperative languages, however, usually have no support or only partial support for closures.

Closures in functional languages

Functional languages have tackled the problem of efficient implementations for first-class functions with lexical scoping for a long time, with very good results. It is not our intention here to present a comprehensive survey of the vast literature on the implementation of functional languages. Instead, we shall briefly comment on some selected works, with a special emphasis on Scheme, because of its semantic similarities with Lua.

⋆ *não tem um survey ou livro?*

[2] Python trashed a Pentium 4 machine with 1.5GB of memory when compiling a function with one million lines of code.

Since Landin's seminal work [Lan64], the usual implementation for first-class functions in languages with strict semantics uses *closures*. A closure is a data structure that represents a function and the environment needed for its execution. However, this first model was somewhat inefficient both in space, as each closure kept a reference to the entire environment where it was created, and in time, as closures were represented by linked lists.

Rabbit [Ste78] was the first relevant compiler for Scheme. It already devoted great attention to closure creation: "A fair amount of analysis is devoted to determining whether environments may be stack-allocated or must be heap-allocated." [Ste78, p. 60] The Rabbit compiler classifies functions (*lambda-expressions*, in Scheme terminology) in three cases: The first case is when a function is used only in function-call position in the same environment where it was created; in this case the function needs no closure and can run in the same environment of its caller. The second case is when the function is used only in function-call position, but within other functions; then the compiler may generate a "partial closure". The last case is when the function "escapes", that is, it is used as an argument or return in a call. In that case, the compiler creates a full closure.

Cardelli [Car84] introduced the concept of *flat closures*. Unlike a traditional closure, which keeps a reference to the entire environment where it was created, a flat closure keeps individual references only to the variables it needs from the environment. This representation is compatible with the *safe-for-space complexity* rule that "any local variable binding must be unreachable after its last use within its scope" [SA00]. Nevertheless, this representation may be inefficient because of the excessive copies of variables from one closure to another [SA00].

Shao & Appel [SA94, SA00] use extensive compile-time control and data-flow information to choose the best closure representation for each closure (what they call *closure conversion*). Their representations allow closure sharing, thereby reducing the cost of copies, but still ensure safe-for-space complexity.

Unfortunately, most results from functional languages do not move easily into the procedural world in general, and into Lua in particular. The main restriction is that they treat non-local variables as read-only (as we did with the use of upvalues in Lua 3.1). In ML, assignable cells have no names, so non-local variables are naturally read only. In Scheme, most implementations do *assignment conversions* [Ste78, AKK$^+$86], that is, they implement assignable variables as ML cells. This is a smart decision for mostly-functional languages, where assignment is rare, but not for procedural languages, where assignment is the norm. Moreover, assignment conversion demands code analysis that does not fit into the compiler used by Lua, as mentioned before.

Imperative languages

Older conventional procedural languages avoid the need for closures either by restricting the use of functions as values or by restricting the access to external variables:

- C does not offer nested functions, and so a function cannot access local variables from another function.

- Pascal offers nested functions but restricts how functions may be manipulated: functions may be passed as arguments to other functions, but can neither be returned nor stored in variables, and so cannot escape the function that creates them.

- Modula offers nested functions and first-class functions, but not together. Only functions defined at the top level are first-class functions.

Several newer imperative languages offer some support for first-class functions, but with some restrictions:

- Java does not offer nested functions, but it offers *anonymous inner classes*, so that a method in an anonymous class is nested inside another method. However, a nested method can access only *final* variables from its enclosing method [AGH05]. Because a final variable has a constant value, the Java compiler simply adds a copy of that variable into the nested class, and the nested method accesses it as a regular instance variable. (In other words, it avoids the need for assignment conversion by forbidding assignment.)

- Python offers nested functions, but although an inner function can access the value of an outer variable, it cannot assign to such variables [Lut06].

- Perl offers first-class functions with lexical scoping. Perl allocates all its variables in the heap, and so it is easy to implement closures in the language [Sri97]. But the overall performance of all function calls (not only closures) pays a high price for this feature. ⋆ *ref?*

3 Closures in Lua

Lua uses a register-based virtual machine [IdFC05].[3] Each function call creates an activation record on a stack, wherein live the registers used by the function. So, each function has its own set of registers. The compiler allocates registers following a stack discipline. In a function call, the caller places the arguments in the highest free registers of its activation record, which then become the lowest registers of the called function.[4] (See Figure 1.)

Local variables and temporary values are always allocated into registers. Each function may allocate up to 250 registers; this limit is imposed by the format of the virtual machine instructions which use eight bits to denote a register. If the function needs more registers (e.g., because it has too many local variables or intermediate values) the compiler simply emits an error of "function or expression too complex". In the very few cases where a programmer hits this limit, it is easy to refactor the code to avoid the problem (e.g., by creating limited scopes via explicit blocks or functions).

Lua uses a flat representation for closures [Car84], as shown in Figure 2. The *GC*

[3] As far as we know, Lua's was the first register-based virtual machine to have wide use.

[4] In other words, Lua uses a software version of a *register window*.

```
function f(x,y)                 GETGLOBAL       2 -1    ; R[2] = print
  print(x+y,x-y,x,y)            ADD             3 0 1   ; R[3] = R[0] + R[1]
end                             SUB             4 0 1   ; R[4] = R[0] - R[1]
                                MOVE            5 0     ; R[5] = R[0]
                                MOVE            6 1     ; R[6] = R[1]
                                CALL            2 5 1
                                RETURN          0 1
```

Figure 1: The software register window used by the Lua VM. Parameters `x` and `y` are allocated to registers 0 and 1. Register 2 holds the function to be called. Registers 3, 4, 5, and 6 hold the arguments to the call.

header contains garbage-collection information that does not concern us here. After that, the closure has a pointer to a *prototype*, which keeps all the static information about the function: Its main component is the compiled code for the function, but the prototype also keeps the number of parameters, debug information, and similar data. Finally, the closure has zero or more pointers to *upvalues*, with each upvalue representing one non-local variable used by the closure. This flat representation, with individual upvalues for each variable, favors closures that use few (or no) non-local variables, which are common in non-functional languages. Moreover, this representation allows a *safe-for-space* implementation [SA94].

The key component in the implementation of Lua closures is the *upvalue* structure, which intermediates the connection between a closure and a variable. Figure 3a shows the structure of an upvalue. Again, the first field is a (irrelevant for us) garbage-collection header. The next field is a pointer to the cell that holds the value of this upvalue. With that structure, the interpreter can access the cell holding a non-local variable with a single indirection. Roughly, the following C code gives the address of the non-local variable with index i:

```
closure->upvalue[i]->cell
```

There are two instructions to access upvalues: `GETUPVALUE`, which copies the value of an upvalue into a register, and `SETUPVALUE`, which copies the value of a register into an upvalue.

⋆ *why here?*

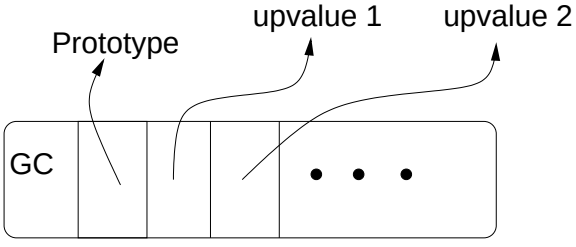

Figure 2: The representation of a closure in Lua.

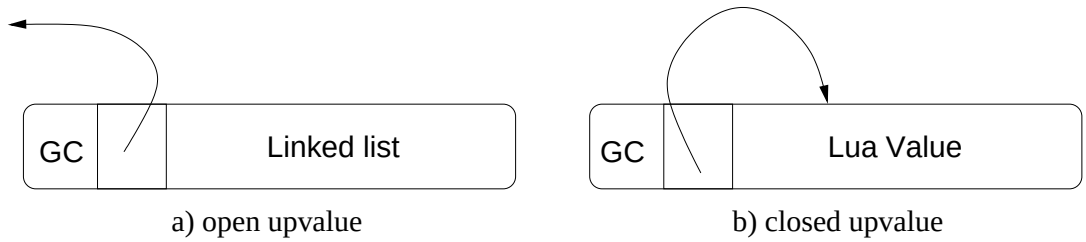

Figure 3: The upvalue structure.

An upvalue may be in two states: open or closed. When the upvalue is created, it is open, and its pointer points to the corresponding variable in the Lua stack. When Lua *closes* an upvalue, the value is moved to the upvalue itself and the pointer is corrected accordingly. Figure 3b shows a closed upvalue.

For the moment, let us assume that a function can access only external variables that are local to its immediately enclosing function. With this restriction, when Lua creates a closure, all its external variables are in the activation record of the running function (which is its enclosing function). So, when Lua has to create a new closure, it could create one new upvalue structure for each non-local variable used by the closure, link each upvalue with its corresponding variable in the stack, and link the closure with these new upvalues. When the scope of a variable ends, the interpreter *closes* its corresponding upvalue, moving the value of the variable to a reserved field in the upvalue itself and making its pointer points to this field (Figure 3b).

Let us now see how to handle external variables that are not in the immediately enclosing function. As an example, consider this simple code of a curried addition function with three arguments:

```
function add3 (x)         -- f1
  return function (y)     -- f2
    return function (z) return x + y + z end    -- f3
  end
end
```

The inner function `f3` needs three variables: `x`, `y`, and `z`. Variable `z` will be in the activation record of `f3` when the function executes. Variable `y` will be in an upvalue in the closure representing `f3`: When the closure was created, that variable was accessible in the activation record of the running function, `f2`.

Variable `x`, however, might have already vanished by the time the closure for `f3` is created. To solve this difficulty, the compiler adds `x` as an upvalue in the intermediate function `f2`. Roughly, it handles the code as if it was like this:

```
function add3 (x)
  return function (y)
    local dummy = x
    return function (z) return dummy + y + z end
```

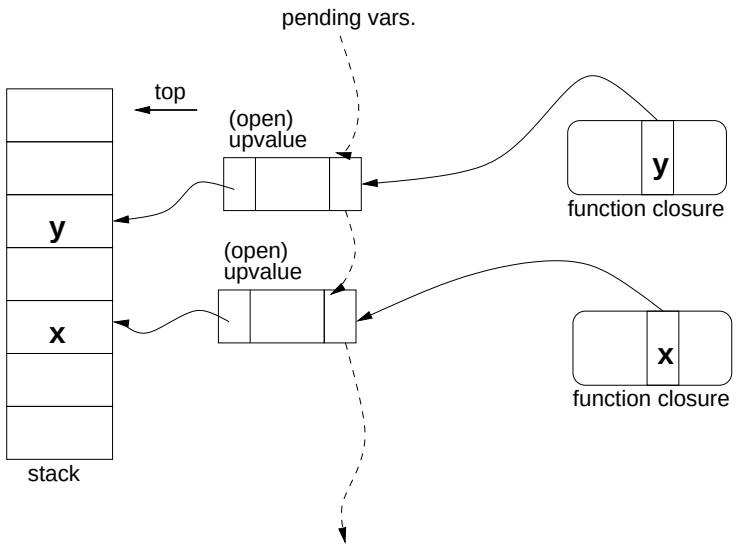

Figure 4: List of open upvalues.

```
  end
end
```

Of course, the compiler does not generate code for this dummy assignment: instead, it adds an upvalue for `x` in the intermediate function. So, when the closure for `f2` is created, the interpreter also creates an upvalue for variable `x`. Now, when the inner function `f3` is created, the interpreter simply copies the upvalue for `x`, which is in the running closure, to the new closure.

Sharing and closing upvalues

The previous scheme does not provide sharing. If two closures need a common external variable, each would have its own upvalue to access it. When these upvalues close, each closure would access a different copy of the variable. If one closure updates the variable, the other would not see this update.

To avoid this problem, the interpreter must ensure that each variable has at most one upvalue pointing to it. To do this, the interpreter keeps a linked list of all open upvalues of a stack (see Figure 4). This list is ordered by the level of the corresponding variables in the stack, starting from the top. When the interpreter needs an upvalue for a local variable, it first traverses this list: If it finds one, it reuses it, therefore ensuring sharing. Otherwise, it creates a new upvalue and links it in its appropriate place in the list.

Because the list is ordered and there is at most one upvalue for each variable, the maximum number of elements to be traversed when looking for a variable in this list can be known statically. This maximum is the number of variables that escape to

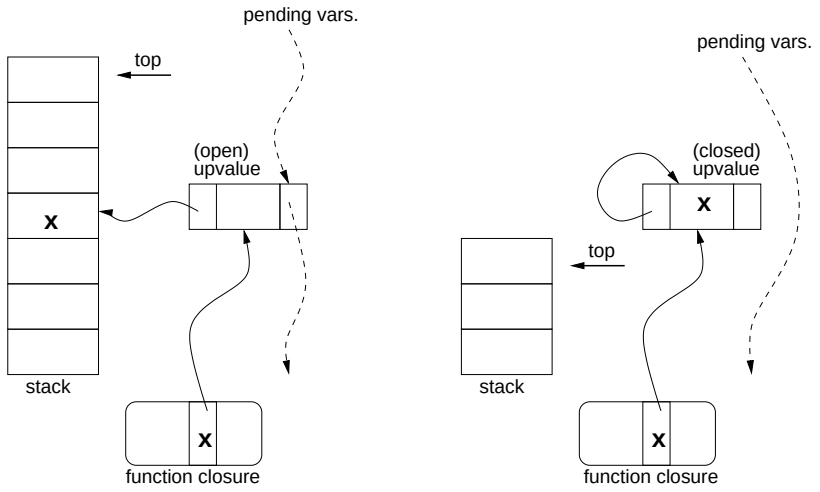

Figure 5: Closing an upvalue.

inner closures and that are declared between the closure and the external variable. For instance, consider the following fragment:

```
function foo ()
  local a, b, c, d
  local f1 = function () return d + b end
  local f2 = function () return f1() + a end
  ...
```

When the interpreter instantiates `f2`, it will traverse exactly three upvalues before realizing that `a` has no upvalue yet: `f1`, `d`, and `b`, in that order.

When a variable goes out of scope, its corresponding upvalue (if there is one) must be closed. The list of open upvalues is also useful for this task. When Lua compiles a block that contains variables that escape, it emits a `CLOSE` instruction, meaning "close all upvalues up to this level", at the end of the block. To execute this instruction, the interpreter simply traverses the list of open upvalues up to the given level, moving the variable contents from the stack to the upvalue and removing the upvalues from the list (see Figure 5).

To illustrate how the list of open upvalues ensures a correct sharing, let us consider the following code fragment:

```
local a = {}      -- an empty array
local x = 10
for i = 1, 2 do
  local j = i
  a[i] = function () return x + j end
end
x = 20
```

When the fragment starts, the list of open upvalues is empty. So, when the interpreter creates the first closure in the loop, it will create upvalues for `x` and `j` and insert them in the list. At the end of the loop body, there is a `CLOSE` instruction signaling that variable `j` (which escaped) is going out of scope. When the interpreter executes this instruction, it closes the `j` upvalue and removes it from the list. In the second iteration, when the interpreter creates the closure, it finds an upvalue for `x` and reuses it, but it cannot find one for `j`, so it creates a new one. At the end of the loop body, the interpreter again closes this second upvalue for `j`.

After the loop, the program has two closures that share an upvalue for `x`, each with its own private copy of `j`. Upvalue `x` is still open, that is, its value is still stored in the stack; so the last assignment `x=20` changes the value used by both closures.

The compiler

We now describe how the compiler works. The main point here is to show how to generate code for the runtime we just described on the fly, with a one-pass compiler.

As we have seen, all local variables live in registers. When the compiler sees a local-variable definition, it allocates a new register for that variable. Inside the current function, all accesses to that variable generate instructions that refer to that particular register, even if that variable later escapes into a nested function.

The compiler compiles nested functions recursively. When it sees a nested function, it stops generating code for the outer function and generates the entire code for the new function. As it compiles the nested function, it builds a list with all upvalues (external variables) that the function needs; this list will be reflected in the final closure structure. When the function ends, the compiler returns to the outer function and generates an instruction to create the closure with the needed upvalues.

When the compiler sees a variable name, it does a search for that variable. This search has three possible outcomes: the variable is global, the variable is local, or the variable is external, that is, local to an outer function. The compiler first searches the current local variables. If it finds the name, it knows the variable is located in a fixed register in the current activation record. If it cannot find the name, it recursively searches the variable in the outer function. If the name is not found, the variable is global.[5] Otherwise, the variable is local to an enclosing function. In this case, the compiler includes (or reuses) a reference to the variable in the list of upvalues of the current function. Once the variable is in the upvalue list, the compiler uses its position in the list to generate a `GETUPVALUE` or an `SETUPVALUE` instruction.

The process of searching for a variable is recursive. Therefore, any variable that is inserted in the upvalue list of a function either is local to the direct enclosing function or it has been inserted in the upvalue list of the enclosing function. As an example, consider the compilation of the `return` statement of function `C` in the code in Figure 6. The compiler finds the variable `c` in the current local list, and so handles it as local. The compiler cannot find `g` in the current function, and so searches in the enclosing

[5] As in some Scheme implementations, free variables default to global in Lua.

```
function A (a)
  local function B (b)
    local function C (c)
      return c + g + b + a
    end
  end
end
```

Figure 6: Access to different kinds of variables.

function `B`, and then recursively searches in `A` and in the anonymous function enclosing the chunk; as none of these searches find `g`, the compiler handles it as a global variable. The compiler cannot find variable `b` in the current function, and so it again searches in `B`; in this case, it finds `b` in `B`'s local list, anb then it inserts `b` in `C`'s upvalue list. Finally, the compiler cannot find `a` in the current function, and so it searches in `B`. It cannot find `a` in that function either, but it finds it in the outer function `A`. So, the compiler inserts the variable in `B`'s upvalue list and finally inserts the variable in `C`'s upvalue list.

⋆ *falar sobre reuso de upvalues?*

Each entry in the upvalue list has a flag and an index. The flag tells whether that variable is local to the enclosing function. If the variable is local to the enclosing function, the index is its location in the activation register. If the variable is an upvalue in the enclosing function, the index is its location in the upvalue list. In both cases, the entry has enough information for locating the variable in the enclosing function.

Back to the example above (Figure 6), at the end of its compilation function `C` will have two upvalues in its upvalue list: the first one (`b`) is local to its enclosing function with index 1, as it will be stored in register 1 of function `B`; the second one (`c`) is an upvalue in its enclosing function, also with index 1, as it is the first (and only) upvalue in that function.

When the compiler finishes compiling a function, it goes back to its outer function and generates a `CLOSURE` instruction, which will create the closure. This instruction has a reference to the newly created prototype for the inner function plus a compact representation of its upvalue list.?? Changed in 5.2 To execute this instruction, the interpreter first creates a closure object with the appropriate number of upvalues and then fills each upvalue field according to the upvalue list. If the upvalue is an upvalue of the running function, the interpreter simply copies the pointer to the upvalue object from the running closure to the new one. If the upvalue is a local variable of the running function, the interpreter creates (or reuses) an upvalue object pointing to the corresponding variable in the stack.

To generate `CLOSE` instructions, the compiler keeps track of whether any variable inside a block is used by an inner function. Therefore, when the compiler arrives at the end of a block, it already knows whether it needs to generate a `CLOSE` instruction at the end of that block. However, commands that interrupt a block may pose a problem. For instance, consider the code below:

```
local x
co = coroutine.create(function ()
       local y
       local f = function () return x + y end
       ...
     end)
coroutine.resume(co)
```

Figure 7: A closure accessing variables in multiple stacks.

```
function foo (x)
  while true do
    if something then return end
    globalvar = function () return x end
  end
end
```

Function `foo` may run the loop twice, creating a closure in the first iteration and then returning in the second iteration. However, when the compiler creates code for the `return` statement it still does not know that `x` escapes. So, a `RETURN` instruction always traverses the list of open upvalues, closing all upvalues up to the base level of the function. In the frequent case that the function does not have open upvalues, this "traversal" results in at most two simple tests (that the list is empty or its initial element has a level lower than the base of the function).

Besides the `return` statement, the only other statement that can exit a block in Lua is the `break` statement. Lua does not have `goto` statements. In Lua, a `break` statement can only break the innermost loop containing the statement. So, variables declared outside the loop do not go out of scope with the break; variables declared inside the loop can escape before a break only if that use appears textually before the break (as there are no back jumps inside the loop). Therefore, when compiling the `break` the compiler already knows whether it is exiting the scope of a variable that has escaped to an inner function. If that is the case, it emits a `CLOSE` instruction before the jump out of the loop.

In case of errors, the error handling mechanism closes all open upvalues in the list, up to the level of the error-recovery function.

Coroutines

Lua offers support for *stackful coroutines* [dMRI04, dMI09], that is, coroutines with independent stacks. Therefore, closures in Lua may form "cactus" structures, wherein a closure may access local variables from several different stacks. The code in Figure 7 exemplifies this condition. The call to `coroutine.create` creates a new coroutine with the given body, and the call to `coroutine.resume` runs it. When the coroutine runs, it

creates a closure `f`. This closure refers both to variable `x`, which is still alive in the stack of the main coroutine, and to variable `y`, which is still alive in the stack of coroutine `co`. If that situation recurs (e.g., if function `f` creates a new coroutine with a new body), a closure may access variables from any number of stacks.

Our mechanism needs no modifications to work with coroutines. All it needs is a separated list of open upvalues for each stack. When Lua creates a new closure, all its upvalues come either from the stack of the running coroutine or from upvalues from outer closures.

⋆ *add wrap-up par*

4 Evaluation

As we discussed before, the use of higher-order functions in procedural languages is not pervasive, being instead reserved for a few but important tasks. Therefore, our first consideration on performance is about the overhead of our implementation on programs that do not use higher-order functions.

Overhead of closures when not using them

There are two points to consider here: creation and invocation of functions.

The only change in the creation of new functions is that now each function has an extra structure, the closure, that sits between a reference to a function and its prototype. So, each function in Lua pays the price of this extra closure, both in memory and in time.

Of course, the number of functions created by a program that does not use higher-order functions is limited by its source. Such programs do not create new functions dynamically; there is only one instance of each function in its source. Therefore, any extra cost in creating functions is limited. For each function prototype, there will be exactly one closure.

A prototype is much larger than a closure with no upvalues. The bare minimal size for a prototype, the prototype for an empty function, uses around 80 bytes, while a closure with no upvalues needs 20 bytes. So, the space overhead is negligible.

For the time overhead for the creation of functions we may follow a similar reasoning. The compilation of a function creates several structures: the prototype itself, an array with VM instructions, an array with constants used by the function, two auxiliary tables used by the parser, etc. Compilation is a somewhat expensive process when compared with most runtime operations. Compared to that, the time overhead of creating a single structure to represent a closure is again negligible.

Now let us consider function invocation. When the function is called, the interpreter needs one extra indirection to get the function prototype, as function references point to closures instead of pointing directly to prototypes. During the interpretation of VM instructions there is no overhead: all variables live in registers in the stack; there are no `CLOSE` instructions when the code does not define nested functions. Only the `RETURN` instruction needs to check whether there are upvalues to be closed (as we discussed in

Section 3). When no function uses external variables, the list of open upvalues is always empty; therefore, that check is a simple check in C for an empty list.

Costs of using closures

Let us now analyze the costs involved with the use of closures, starting with memory allocation.

?? 5.2 Lua does not use any kind of closure sharing. Each closure needs an exclusive entry for each external variable that it accesses. This design simplifies the compiler at the cost of extra memory plus the cost of copying values to new closures. ?? Rever... This extra cost is not too high in a procedural language, where closure creation is not as frequent as in a functional language.

Besides the entry itself, a closure also needs memory for the upvalue structure. This structure is shared among closures accessing a common variable. In the worst case, only one closure accesses a given external variable, so we may consider that each closure needs a new upvalue structure for each external variable that it uses. So, the overall memory needed for a closure is roughly $32+28n$ bytes, where n is the number of external variables used by the closure.

Access to upvalues is practically as fast as access to local variables, because the cost of the indirection is small compared to the overhead of the interpreter. Nevertheless, the use of upvalues may have a significant cost in Lua. Because Lua uses a register-based virtual machine, several instructions can access local variables directly in the stack frame. As an example, consider the statement `var = var + 1`. If `var` is a local variable, stored in register 5 for instance, that statement generates a single instruction:

```
ADD  R5, R5, 1
```

Upvalues, however, must be loaded into a register before being operated on. So, if `var` is non local, that same statement `var = var + 1` needs three instructions:

```
GETUPVAL   R6, 4
ADD        R6, R6, 1
STOREUPVAL R6, 4
```

The machine must move the upvalue into a register, perform the operation, and store the result back in the upvalue. Of course, it is possible to decrease or even to eliminate this overhead by adding instructions that manipulate upvalues directly, but such addition would complicate the virtual machine.

⋆ *how? how much? and the compiler?*

Benchmarks

5 Final Remarks

References

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Prentice Hall PTR, fourth edition, 2005.

[AKK+86] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. ORBIT: an optimizing compiler for Scheme. *SIGPLAN Notices*, 21(7), July 1986. (SIGPLAN'86).

[Car84] Luca Cardelli. Compiling a functional language. In *LFP'84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217. ACM, 1984.

[dMI09] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems*, 31(2):6.1–6.31, 2009.

[dMRI04] Ana Lucia de Moura, Noemi Rodriguez, and Roberto Ierusalimschy. Coroutines in Lua. *Journal of Universal Computer Science*, 10(7):910–925, July 2004. (SBLP 2004).

[Hem08] Ralph Hempel. Porting Lua to a microcontroller. In Luiz H. de Figueiredo, Waldemar Celes, and Roberto Ierusalimschy, editors, *Lua Programming Gems*, chapter 26. Lua.org, 2008.

[IdFC96] Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. Lua—an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[IdFC05] Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. The implementation of Lua 5.0. *Journal of Universal Computer Science*, 11(7):1159–1176, 2005. (SBLP 2005).

[IdFC07] Roberto Ierusalimschy, Luiz H. de Figueiredo, and Waldemar Celes. The evolution of Lua. In *Third ACM SIGPLAN Conference on History of Programming Languages*, pages 2.1–2.26, San Diego, CA, June 2007.

[Ier03] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, Rio de Janeiro, Brazil, 2003.

[ISO06] ISO. *Information Technology — Programming Languages — C#*, 2006. ISO/IEC 23270:2006(E).

[Lan64] P. B. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Lut06] Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[SA94] Shong Shao and Andrew W. Appel. Space-efficient closure representations. *SIGPLAN Lisp Pointers*, VII(3):150–161, 1994.

[SA00] Shong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, January 2000.

[Sri97] Sriram Srinivasan. *Advanced Perl Programming.* O'Reilly & Associates, Inc., 1997.

[Ste78] Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical Report AITR-474, MIT, Cambridge, MA, 1978.