Preparation for Lua Coroutines: The Programming Model

N. Ramsey and E. Pailes, for COMP 250RTS

Wednesday, November 1, 2017

Reading

Please read the pedagogical chapter in Roberto's book as well as selected sections of the scholarly article that appeared in *TOPLAS*.

- Roberto Ierusalimschy, *Programming in Lua*, Chapter 9 (Coroutines)
- Moura, A.L.D. and Ierusalimschy, R., 2009. Revisiting coroutines, *ACM Transactions on Programming Languages and Systems (TOPLAS)* 31(2), article No. 6, sections 1, 2, 3.1, and 4

Hints from the operational semantics

If you're up to the task of decoding a small-step semantics that is expressed using evaluation contexts, I recommend a look at section 3.2 of the TOPLAS paper. Here are my key findings:

- Rules 1 to 7 are standard small-step semantics, with evaluation context C used to control order of evaluation. (You may be used to seeing E for a context, but you see C here because this is intended primarily as a semantics of *commands*. I would have written E anyway.)
- The evaluation contexts look a bit off because the order of evaluation is nonstandard—right to left.
- Rule 10 reveals where the bodies are buried. A semantic *subcontext* C' corresponds to a thread's call stack. Rules 8 and 10 reveal that when a thread is not running, its stack lives on the heap. Rule 9 pulls the stack off the heap and starts running it.
- The left-hand side of rule 10 shows the dynamic structure of a running Lua machine: the largest subcontext C'_2 corresponds to a currently running

thread, whereas the enclosing context C_1 corresponds to a stack of threads executing in (I hope) "normal" state.

Exercise to complete ahead of class

Before class, we'll all complete a short programming exercise using Lua coroutines. Most of the classic problems are solved in the book or the paper. But here's one that isn't: without materializing any intermediate data structures, tell if two binary trees have the same *fringe*. I'm formulating the problem in *not quite* the classic way: I define the fringe of a binary tree as the list of values produced by an inorder traversal. This handout shows you a simple sequential solution in Lua; your mission is to adapt the solution using coroutines, so that no list of fringe values is never materialized. Bonus points if you minimize traversal.

Sequential solution

Representation of binary trees

A binary tree is one of the following:

- `nil`
- A table with keys `left`, `right`, and `v`, where `left` and `right` are binary trees and `v` is any value

My examples use binary *search* trees; a binary search tree is a binary tree in which values `v` satisfy the standard order invariant.

Insertion into a binary search tree

This insertion uses destructive update; it is imperative code.

```
function insert(t, v)
  if t == nil then
    return { left = nil, v = v, right = nil }
  elseif t.v == v then
    return t
  elseif v < t.v then
    t.left = insert(t.left, v)
    return t
  else
    t.right = insert(t.right, v)
    return t
  end
end
```

Search tree computed from a list.

A binary-search tree represents a set, and this function converts a list of values to a set represented as a binary-search tree.

```
function set(vs)
  local t = nil
  for _, v in ipairs(vs) do
    t = insert(t, v)
  end
  return t
end
```

Inorder traversal

Here's a higher-order traversal. It takes as arguments a function_`f` and tree `t`, and it applies `f` to each `v` value in `t`, in order. Function `f` is applied only for side effect.

```
function apply_inorder(f, t)
  if t == nil then
    return
  else
    apply_inorder(f, t.left)
    f(t.v)
    apply_inorder(f, t.right)
  end
end
```

Sequential computation of the fringe

Convert fringe to a list by allocating a fresh, empty list `vs`, and in order, insert each element into the list.

```
function fringe(t)
  local vs = { }
  apply_inorder(function(v) table.insert(vs, v) end, t)
  return vs
end
```

Comparing two fringes for equality

Insist on equal lengths, then compare element-wise.

```
function eqlists(vs, ws)
  if #vs == #ws then
    for i = 1, #vs do
      if vs[i] ~= ws[i] then
        return false
      end
    end
    return true
  else
    return false
  end
end
```

Two trees have the same fringe if the fringes, as lists, are equal.

```
function eqfringes(t, u)
  return eqlists(fringe(t), fringe(u))
end
```

Testing

Here are some simple unit tests.

```
local vs = { 1, 2, 3, 4, 5, 6 }
local permutation = { 3, 2, 1, 4, 5, 6 }

assert(not eqlists(vs, permutation))

assert(eqlists(vs, fringe(set(permutation))))

assert(eqlists(fringe(set(permutation)), fringe(set { 6, 5, 4, 3, 1, 2})))

assert(eqfringes(set(permutation), set(vs)))
assert(not eqfringes(set(permutation), set { 1, 2, 3, 4, 5, 7 }))
```

Coroutine solution

Your mission is to define a function `coeqfringes` which takes two trees and arguments and returns a Boolean indicating whether the two trees have the same fringe, *without ever materializing any fringe as a list*. You may find it helpful to define an auxiliary function that uses coroutines for traversal or to test your understanding by using coroutines to build a list representation of a fringe.

I encourage you to use functions `coroutine.create`, `coroutine.resume`, `coroutine.yield`, and `coroutine.status`. *Avoid* `coroutine.wrap`; it adds no expressive power, and it will make it harder to understand what's happening in the stack.

Tests for my coroutine code

As an analog of `fringe`, I built `cofringe`.

```
assert(eqlists(fringe(vs), cofringe(vs)))
assert(eqlists(fringe(permutation), cofringe(permutation)))
assert(not eqlists({1, 2, 3, 4, 5}, cofringe(permutation)))
```

Function `cofringe` is *not* used to implement `coeqfringes`.

```
assert(coeqfringes(set(vs), set(permutation)))
assert(coeqfringes(set(permutation), set(vs)))
```

```
assert(not coeqfringes(set(permutation), set { 1, 2, 3, 4, 5 }))
assert(not coeqfringes(set(permutation), set { 1, 2, 3, 4, 5, 6, 7 }))
```

Summary of key coroutine functions

`coroutine.create (f)`

Creates a new coroutine, with body `f`. `f` must be a Lua function. Returns this new coroutine, an object with type `"thread"`.

`coroutine.resume (co [, val1, ···])`

Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1`, · · · are passed as the arguments to the body function. If the coroutine has yielded, `resume` restarts it; the values `val1`, · · · are passed as the results from the yield. If the coroutine runs without any errors, `resume` returns **true** plus any values passed to `yield` (if the coroutine yields) or any values returned by the body function (if the coroutine terminates). If there is any error, `resume` returns **false** plus the error message.

`coroutine.status (co)`

Returns the status of coroutine `co`, as a string: `"running"`, if the coroutine is running (that is, it called `status`); `"suspended"`, if the coroutine is suspended in a call to `yield`, or if it has not started running yet; `"normal"` if the coroutine is active but not running (that is, it has resumed another coroutine); and `"dead"` if the coroutine has finished its body function, or if it has stopped with an error.

`coroutine.yield (···)`

Suspends the execution of the calling coroutine. The coroutine cannot be running a C function, a metamethod, or an iterator. Any arguments to `yield` are passed as extra results to `resume`.

Discussion questions for class

Code and its execution

(1) Compare your coroutines code with code written by other students in your group. What code does the group identify as seeming most idiomatic?

(2) If anybody in your group implemented `cofringe`, walk through an execution of `cofringe` on the tree below, and explain what you think is happening with Lua *call* stacks. (Assume that each thread gets its own call stack.)

```
   1
  / \
nil 3
   / \
  2  nil
```

If not, try the same question, but on an execution of `coeqfringes` which compares two copies of this tree.

Building a model of the implementation

(3) As noted in the reference manual, a Lua coroutine ("thread") may be in one of these states: `running`, `suspended`, `normal`, or `dead`. As a stepping stone to studying the implementation, analyze the state transitions:

   (a) Refine the state by adding some information about the thread's internal state. A good start would be the number of activation records on its call stack. For your first cut, ignore the presence or absence of activation records for C code.

   (b) Draw a state diagram with the given states. Show what happens at the following events:

   - A running thread calls a function.
   - A running thread returns from a function.
   - A running thread calls `coroutine.yield`.
   - A running thread calls `coroutine.resume`.[1]
   - A running thread calls `coroutine.create`.

   Keep in mind that even if we're ignoring C code, not every action is permissible in every state.

   (c) Refine your model by extending the thread state with field `nny` ("number of non-yieldable calls") and refining the action of "calls a function" into these two actions:

   - Calls a C function
   - Calls a Lua function

(4) Ignoring issues of C code, use Cormack's API (`emit`, `p`, `v`, `talk`, and `listen`) to implement Lua's coroutine API. (Because Lua threads are garbage-collected, you won't need `absorb`.)

Comparisons

(5) How do coroutines compare with Cormack's threads or Cilk's parallelism?

   (a) How difficult or easy would it be to implement `coeqfringes` using Cormack's threads?

   (b) How difficult or easy would it be to implement `coeqfringes` using Cilk?

   (c) What broader points of comparison, if any, do you see between coroutines, threads, and Cilk? In particular, which of these models can be made parallel?

[1] May cause state transitions in two different threads.

Bonus questions

(6) Would mailboxes and message passing be a useful abstraction for use with for coroutines? How would you implement them?

(7) The last example in Roberto's book chapter is very suggestive. Following this example, could you use coroutines to implement asynchronous computation in the style of Cilk? That is, you `spawn` an asynchronous computation, continue with your own work, and eventually wait for the asynchronous computation to complete (`sync`)?