Notes on Continuations

Programming Languages

October 25, 2016

Recursion and Iteration

Let's revisit something I talked about a few weeks ago.

We discussed how we interpreted features of the object language (FUNC, for instance) using corresponding features of the metalanguage (in our case, Python). In particular, we interpreted recurison in FUNC using recursion in Python.

For instance, suppose we are interpreting a call `(sum 10)` where `sum` is defined as usual:

```
(defun sum (n) (if (zero? n) 0 (+ n (sum (- n 1)))))
```

If we evaluate `(sum 10)`, or more accurately the abstract representation of `(sum 10)`, we end up calling method `ECall.eval()` with function `sum` and argument `10`. During that call, we will end up recursively calling method `ECall.eval()` with function `sum` and argument `9`. Thus, recursion in the object language is interpreted via recursion in the metalanguage.

The problem with this approach in our specific case is that Python is *terrible* at recursion. If you try to evaluate `(sum 5000)` with our interpreter, chances are you'll get an error of some kind, or even crash Python.

Why is that?

Python implements recursion the way many imperative languages implement recursion, by using a stack (called a call stack) in which to save the context that existed at the time of a recursive call so that when the recursive call returns execution can pick up where it left off. (This context is saved into something called an *activation frame*, which holds, among other things, the local variables of the function.) Every recursive call adds an activation frame on the call stack. And Python does not let the call stack get too tall.

This is not specific to our interpreter. Consider the Python version of the above function sum:

```
def sum (n):
  if n == 0:
```

```
    return 0
  return n + sum(n-1)
```

This will also fail with `sum(5000)`.

This seems unavoidable. Recursion seems to require a call stack to grow, and Python has limitations on the size of the call stack.

Things are not hopeless, though. It turns out that we can bypass this limitation of Python, at least, in some instances. In order to do so, we need to study our object language a bit more closely.

If you look at the Python code for `sum`, and you have some experience with Python, you'll realize that this is a bad way to compute the sum of all numbers between 0 and `n`. A better way is to use iteration:

```
def sum_iter (n):
  result = 0
  while n > 0:
    result += n
    n = n-1
  return result
```

Note that this does not grow the call stack.

How can we lift the iteration idea to our object language? It turns out that in a functional language, we can write a form of iteration as follows:

```
(defun sum_iter (n result)
  (if (zero? n)
      result
      (sum_iter (- n 1) (+ n result))))
```

You can see the similarities. One disadvantage is that it requires the passing of two values, but that's easy enough to fix with a wrapper function `(function (n) (sum_iter n 0))`.

Clearly, the two functions `sum` and `sum_iter` compute the same thing. But they do so differently. To analyse them, we'll use *equational reasoning*. Roughly speaking, we'll reason algebraically about the behavior of the functions. You can think of the definition of a function as defining an equation, so that, for example:

$$\texttt{(sum n)} = \texttt{(if (zero? n) 0 (+ n (sum (- n 1))))}$$

and

$$\texttt{(sum\_iter n res)} = \texttt{(if (zero? n) res (sum\_iter (- n 1) (+ res n)))}$$

So let's use these equations to see what `(sum 6)` and `(sum_iter 6 0)` look like:

```
(sum 6) = (if (zero?  6) 0 (+ 6 (sum (- 6 1))))
        = (if false 0 (+ 6 (sum (- 6 1))))
        = (+ 6 (sum (- 6 1)))
        = (+ 6 (sum 5))
        = (+ 6 (if (zero?  5) 0 (+ 5 (sum (- 5 1)))))
        = (+ 6 (+ 5 (sum 4)))
        = ...
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum 0)))))))
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (if (zero?  0) 0 (+ 0 (sum (- 0 1)))))))))))
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))))    (*)
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1)))))
        = (+ 6 (+ 5 (+ 4 (+ 3 3))))
        = ...
        = (+ 6 15)
        = 21
```

The key thing to notice is that we can't start simplifying the expression until we've basically expanded out all the recursive `sum` calls, in line (*).

Contrast to the equational simplification of `(sum_iter 6 0)`:

```
(sum_iter 6 0) = (if (zero?  6) 0 (sum_iter (- 6 1) (+ 6 0)))
               = (if false 0 (sum_iter (- 6 1) (+ 6 0)))
               = (sum_iter 5 6)
               = (if (zero?  5) 6 (sum_iter (- 5 1) (+ 5 6)))
               = (sum_iter 4 11)
               = (if (zero?  4) 11 (sum_iter (- 4 1) (+ 4 11)))
               = (sum_iter 3 15)
               = (if (zero?  3) 15 (sum_iter (- 3 1) (+ 3 15)))
               = (sum_iter 2 18)
               = (if (zero?  2) 18 (sum_iter (- 2 1) (+ 2 18)))
               = (sum_iter 1 20)
               = (if (zero?  1) 20 (sum_iter (- 1 1) (+ 1 20)))
               = (sum_iter 0 21)
               = (if (zero?  0) 21 (sum_iter (- 0 1) (+ 0 21)))
               = 21
```

While the terms in the equational simplification of `(sum n)` get longer and longer as `n` gets larger, the terms in `(sum_iter n 0)` remain essentially constant size.

This is not surprising. In a way that can be made precise (but which I won't do here), the size of the terms in the equational simplifications above correspond precisely to the growth of the call stack in the Python execution model!

(Because of this feature, a function such as `sum_iter` is usually said to implement an *iterative process* in the context of a functional language. Note that it is still syntactically recursive: it is still a function that refers to itself in its body. But when it executes, it doesn't grow the stack. In contrast, a function such as `sum` is said to implement a *recursive process*.)

By this correspondence,`sum_iter` in our object language doesn't seem to grow the stack. It's enlightening to go back to Python and see how that manifests itself there. Here's `sum_iter` in Python, using recursion:

```
def sum_iter (n,result):
  if n == 0:
    return result
  return sum_iter(n-1,result+n)
```

If you execute `sum_iter(5000)`, you once again blow the stack! So what's going on? I claim that the stack doesn't grow based on the equational simplification above, yet Python still can't deal with.

That's because the actual claim is that the stack *doesn't need to grow*. If Python was a tiny bit more clever in its execution model and implemented something called *tail-call optimization*, then `sum_iter` would execute just fine. Tail-call optimization is the realization that if the very last thing that a function A does before returning is calling another function B, then we don't need to put a new activation frame for B on the call stack, but instead we can just reuse the activation frame for A. (An activation frame is meant to record the context of a call so that the system know what to return to after the call is done, but if the next thing to do when function B returns is to return from A itself, then we don't need to record the context of A — it doesn't serve any role.)

If Python implemented tail-call optimization, then we could execute `sum_iter(5000)` without growing the stack, and it would execute without problem.

That's a problem for us because if you look at, for instance, the `eval()` method in `ECall`, you see the following:

```
    def eval (self,env):
        f = self._fun.eval(env)
        if f.type != "function":
            raise Exception("Runtime error: trying to call a non-function")
        args = [ e.eval(env) for e in self._args]
        if len(args) != len(f.params):
```

```
            raise Exception("Runtime error: argument # mismatch in call")
        new_env = zip(f.params,args) + f.env
        return f.body.eval(new_env)
```

and note the last line: the last thing `eval()` does before returning is call `f.body.eval()`. If Python implemented tail-call optimization, it would not need to grow the stack here. If you do a bit of reasoning and scribble some diagrams, you can convince yourself that such a thing would allow you to interpret `(sum_iter n)` in our object language for very large `n`s.

Most functional languages implement tail-call optimization. This allows them to execute recursive functions in which every recursive call is in tail position (that is, where every recursive call is the last thing that the function does before returning — those functions often called *tail recursive*) without growing the stack, as an iterative process.

But Python does not implement tail-call optimization. So if we want to be able to execute an iterative process implemented as a recursive function in our object language, we have to work harder. We can modify our interpreter so that `eval()` for `ECall` does not use an activation frame for evaluating the body of the function being called. (Something similar can be done for `EIf` and `ELet` — can you see why?)

The trick is to basically pull evaluation *out* of the objects, and make it a single global function. Here's the code. Can you figure out why this works?

```
def eval_iter (exp,env):
    current_exp = exp
    current_env = env
    while True:

        # these are all the special forms which directly
        # return the result of evaluating an expression

        if current_exp.expForm == "ECall":
            f = current_exp._fun.eval(current_env)
            if f.type != "function":
                raise Exception("Error: calling a non-function")
            args = [ e.eval(current_env) for e in current_exp._args]
            if len(args) != len(f._params):
                raise Exception("Error: argument # mismatch in call")
            new_env = zip(f._params,args) + f._env
            current_exp = f._body
            current_env = new_env

        elif current_exp.expForm == "EIf":
            v = current_exp._cond.eval(current_env)
            if v.type != "boolean":
```

```
                raise Exception ("Error: condition not a Boolean")
            if v.value:
                current_exp = current_exp._then
            else:
                current_exp = current_exp._else

        elif current_exp.expForm == "ELet":
            current_env = [ (id,e.eval(current_env))
                                  for (id,e) in current_exp._bindings] +
                          current_env
            current_exp = current_exp._e2

        else:
            return current_exp.real_eval(current_env)
```

This relies on a couple of things:

1. every expression node has a field `expForm` that tells you the type of expression it represents;
2. the eval() method in every expression node simply calls `eval_iter()` (so that we can still just call `exp.eval()` and get evaluation — we don't need to change the code that uses our abstract representation);
3. every expression node has a method `real_eval()` that does the actual work of evaluating the expression (for those cases that don't require tail-call optimization).

With these changes to the interpreter, we can evaluate `(sum_iter 5000)` and more wihtout any problem:

```
Lecture 7 - REF Language (defun, define, tail calls)
#quit to quit, #abs to see abstract representation
ref/tail> (defun sum_iter (n result)
              (if (zero? n) result (sum_iter (- n 1) (+ result n))))
sum_iter defined
ref/tail> (sum_iter 5000 0)
12502500
ref/tail> (sum_iter 50000 0)
1250025000
ref/tail>
```

Continuation-Passing Style

Since tail-recursive functions (recursive functions where every recursive call is the last thing that the function does before returning) can be evaluated without growing the stack, it makes sense to ask whether every recursive function can be rewritten in a way that is tail recursive, like we did above with `sum_iter` as a tail-recursive version of `sum`.

It turns out that yes, we can always do such a rewriting, and moreover, there is a mechanical way to do that kind of transformation. Moreover, the transformation has other interesting side effects that we will see later when we look at compilation.

I'll define the translation using the surface syntax, but it could (and probably should) be done at the abstract representation level. Because we already know that we can translate all of FUNC to simply functions, function calls, values, identifiers, and conditionals, that is what we'll take as the language to transform.

The idea is to pass a new argument, called a *continuation*, to every function. That continuation is a function that represents *the rest of the computation after the function has computed its value*, and expects one argument, the result of the function call. The idea is that the function, instead of returning its value, simply calls the continuation with the result.

Here's an example. Consider the translation of the expression `(+ 1 (* 3 4))`. Let K be a function that expects the result of this evaluation (for instance, to print it at the shell at the end of the computation). Then the translation of the expression could be written:

`(* 3 4 (function (x) (+ 1 x` K`)))`

where `*` and `+` have been modified so that they take that extra continuation argument and call that argument when they have computed their result.

This requires some amount of mental processing. Let's look at a larger example. Consider the recursive `sum` function we saw earlier. Here is one possible transformation into a version which uses continuations, in a context where every primitive operations takes an extra continuation argument that it calls with its result:

```
(defun sumC (n k)
  (zero? n
    (function (x) (if x (k 0)
                        (- n 1
                           (function (y) (sumC y
                                               (function (z) (+ n z k)))))))))
```

To call this function, you pass it a continuation that returns immediately with the result, such as:

```
  (sum 5000 (function (x) x))
```

(If you've ever programmed in JavaScript in the NodeJS framework, this may be familiar to you: basically, every function takes a callback that captures what to do with the result of the function call. Same thing here.)

Note that this new function `sumC` is tail recursive. In fact, every function call in `sumC` is in tail position. Technically speaking, we do not need to grow the stack to execute such a function. (And indeed, if we write and execute such a function in our interpreter modified to do tail-call optimization above, this function executes without a hitch even for large values of `n`.)

This form of programming is called *continuation-passing style* (CPS), and it turns out that we can mechanically define the transformation of standard code into continuation-passing style. This transformation is called a CPS transformation.

I'm going to define it mathematically — technically, it is defined recursively on the structure of expressions. In what's below, I write

$$[\![exp]\!]K$$

for the result of transformation of expression `exp` in the context of a continuation K (also an expression) expecting the result of evaluating the expression.

$$
\begin{aligned}
[\![v]\!]K &= \texttt{(}K\ v\texttt{)} \\
[\![\texttt{(if } c\ t\ e\texttt{)}]\!]K &= [\![c]\!]\texttt{(function (x) (if x } [\![t]\!]K\ [\![e]\!]K\texttt{))} \\
[\![id]\!]K &= \texttt{(}K\ id\texttt{)} \\
[\![\texttt{(function (x ...) } e\texttt{)}]\!]K &= \texttt{(}K\ \texttt{(function (x ... k) } [\![e]\!]\texttt{k))} \\
[\![\texttt{(}e\texttt{)}]\!]K &= [\![e]\!]\texttt{(function (f) (f } K\texttt{))} \\
[\![\texttt{(}e\ e_1\texttt{)}]\!]K &= [\![e]\!]\texttt{(function (f) } [\![e_1]\!]\texttt{(function (a) (f a } K\texttt{)))} \\
[\![\texttt{(}e\ e_1\ e_2\texttt{)}]\!]K &= [\![e]\!]\texttt{(function (f)} \\
&\qquad\qquad [\![e_1]\!]\texttt{(function (a) } [\![e_2]\!]\texttt{(function (b) (f a b } K\texttt{))))} \\
[\![\texttt{(}e\ e_1\ e_2\ e_3\texttt{)}]\!]K &= \ldots
\end{aligned}
$$

(You should be able to generalize to function calls with arbitrary many arguments.)

This translates reasonably easily into actual code. See the sample code provided for the lecture. Every Expression node in the abstract representation has a method `cps()` taking a continuation (again, just another expression evaluating to a function) and returning some new abstract representation expression representing the result of the CPS transformation for that Expression node. The one subtlety is that all the parameters to the functions introduced by the translation need to be *fresh*, that is, not appearing anywhere in the code being translated. That is in order to avoid accidental capture of existing identifiers.