Continuations for Compilation

sk and dbtucker

2002-10-21

1 Continuation-Passing Style

The interpreter we saw in the last lecture was clearly in a stylized form, somewhat like the style we used to implement state. Whereas store-passing style is only really useful for implementing state in a pure (state-less) language, the style used to implement continuations has very broad applications. We call that style *continuation-passing style*, or CPS for short.

We have already seen one such application, namely implementing continuations. In turn, continuations have a fairly momentous impact in several areas: I/O programming, especially on the Web; expressing standard patterns of control flow (such as exceptions); helping programmers define new control operators (such as the stream generators); and so on. But as we will see shortly, CPS itself has another very interesting use.

Before we discuss CPS in more detail, let's make a semantic distinction clear. CPS is a *style of program*; many programs, not only interpreters, can be in CPS. Any program that satisfies a certain pattern of code can be said to be in CPS. More to the point, any program that is not in CPS can be converted into CPS. We will examine that first. (We don't really have a verb for this conversion other than the rather verbose "conversion to CPS", so it's common to abuse language and use CPS itself as a verb: thus, one might say, "Let's CPS this program!" Remember, though, that CPS is itself a program form; it is not the name of a converter that translates programs into this form, except for convenience.)

1.1 Converting Programs to CPS

There are many ways of expressing the conversion of arbitrary programs to CPS. In these notes, we will employ one of the oldest and most venerable of these, due to Michael Fischer.

First, we will consider a stripped-down version of the language, to include only constants and variables, functions, and applications. In the lambda calculus lecture, we will see why this is a sufficient language for capturing all programs. The language is obviously rather spartan to program in, but it's sufficient for illustrating the CPS transformation.

Fischer's transformation has two operators, $\mathcal{F}_v$, for converting values, and $\mathcal{F}$, for transforming all other expressions. Fischer's transformer is defined as follows:

$$\begin{aligned}
\mathcal{F} &: \mathsf{FAE} \to \mathsf{FAE} \\
\mathcal{F}[Val] &= \texttt{\{fun \{k\} \{k } \mathcal{F}_v[Val]\texttt{\}\}} \\
\mathcal{F}[\texttt{\{}M\ N\texttt{\}}] &= \texttt{\{fun \{k\} \{}\mathcal{F}[M]\texttt{ \{fun \{mv\} \{}\mathcal{F}[N]\texttt{ \{fun \{nv\} \{\{mv k\} nv\}\}\}\}\}\}}
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}_v &: \textit{Values} \to \mathsf{FAE} \\
\mathcal{F}_v[var] &= var \\
\mathcal{F}_v[num] &= num \\
\mathcal{F}_v[\texttt{\{fun \{x\} } M] &= \texttt{\{fun \{k\} \{fun \{x\} \{}\mathcal{F}[M]\texttt{ k\}\}\}}
\end{aligned}$$

Fischer's transformation was defined thirty years ago, and since then there has been considerable research into building a better CPS transformation. The problem with this version is, while it is very easy to understand, it introduces a considerable amount of overhead: look at all the functions in the output that weren't in the source program! Indeed, apply the transformation defined above to the pre-CPS interpreter and compare the result against the one we wrote off-the-cuff. The difference will be clear.

Building a better transformer is not, however, merely an idle pursuit. The CPS form of a program is sometimes read by computers, but sometimes by humans also. Both benefit from getting better code from the transformer. We will see an instance of each end-user today itself. Therefore, the Fischer transformation is primarily of historical interest. We present it because it's easy to understand, but if we were to implement the transformation for practical use, we would use one of its modern variants.

1.2 Observations about Programs in CPS

First, let's study the interpreter in CPS. This is an instance of why the output of a CPS transformer should be human-readable, because the actual form of the code contains important information, and a dense transformation can obscure the insights waiting to be read from the output.

Let's study the interpreter in CPS form, case-by-case. For this lecture, we're going to assume a language without continuations (*CFAE*), which should demonstrate that a CPS interpreter is useful even for a language without continuations: though we performed the conversion on the road to adding continuations, it will prove to be useful otherwise.

Three cases correspond to values (numbers, identifiers and functions), so we'll get them out of the way quickly:

```
(define (interp expr env k)
  (cases CFAE expr
    [num (n) (k (numV n))]
    [id (v) (k (lookup v env))]
    [fun (param body)
         (k (closureV (lambda (arg-val dyn-k)
                          (interp body (aSub param arg-val env) dyn-k))))]
    ...))
```

In each of the remaining cases, the interpreter receives a continuation, *k*, but proceeds to build a larger continuation during the process of evaluating its given expression. Study these three examples carefully: do you spot a pattern?

Let's consider addition first. We have

```
[add (l r) (interp l env
                   (lambda (lv)
                     (interp r env
                             (lambda (rv)
                               (k (numV+ lv rv))))))]
```

The interpreter creates two intermediate continuations for evaluating the two sub-expressions, but when it has values for each one, it uses *the same continuation* originally fed into it.

How about conditionals?

```
[if0 (test then else)
     (interp test env
             (lambda (tv)
               (if (numV-zero? tv)
                   (interp then env k)
                   (interp else env k))))]
```

Again, though the interpreter created a new continuation, when it finally has a value (that of either the then or else case), it passes this to *the same continuation* originally fed to it.

Finally, in the case of application,

```
[app (fun-expr arg-expr)
     (interp fun-expr env
             (lambda (fun-val)
               (interp arg-expr env
                       (lambda (arg-val)
                         (cases CFA-value fun-val
                           [closureV (c)
                                     (c arg-val k)]
```

```
                              [else
                               (error 'interp "can only apply functions")])))))]))
```

we again see that once all the sub-expressions are done evaluating, the dynamic continuation fed to the representation of procedures is *k*, not some extension of it.

All this says that *k*, the continuation, performs a very specific and stylized purpose, so it's worth understanding it more broadly. What does it do, exactly? The interpreter uses that argument position to keep track of the tasks remaining to be done. In particular, it uses it to track, for instance, the interpretation of sub-expressions, and then as the target of results. What in a conventional machine architecture keeps track of pending computation and then sends the results to that computation? It's the machine *stack*!

In other words, the continuation *is* the stack. More precisely, it's the stack closed over its lexical environment, but from now on we'll take that as a given and assume static scope. Even more precisely, then, it's a *representation of* the machine's stack. In particular, by converting the interpreter to CPS, we have exposed something normally hidden during the evaluation process, and made it accessible for manipulation. This is the principle of *reflection*. And when we add continuations to the language, we go a step further: we *reify* the stack, i.e., turn it into a value over which the programmer has control.

A brief digression: Reflection and reification are very powerful programming concepts. Most programmers encounter them only very informally or in a fairly weak setting. For instance, Java offers a very limited form of reflection (a programmer can, for instance, query the names of methods of an object), and some languages reify very low-level implementation details (such as memory addresses in C). Few languages reify truly powerful computational features such as continuations; the ones that do enable entirely new programming patterns that programmers accustomed only to more traditional languages usually can't imagine. Truly smart programmers sometimes create their own languages with the express purpose of reifying some useful hidden element, and implement their solution in the new language, to create a very powerful kind of reusable abstraction.

Back to stacks. What we're seeing here is that the stack is used solely to store intermediate results. But it plays no part in the actual invocation of a function. The interpreter is very explicit on this account: the function consumes a dynamic continuation (we now know that this is asking for a reference to the "current stack"—or the *stack pointer*), which is the same as that at the time evaluation of the function application began. In the net, the stack has not been affected by the fact that we are invoking a function. This sets on its head everything you have probably been taught about stacks until now. This is such an important, and perhaps startling, point that we feel compelled to make it stand out clearly:

Stacks are not necessary for calling functions.

The stack only plays a role in *evaluating the argument to the function*; once that argument has been reduced to a value (in an eager language), the stack has no further role with respect to invoking a function. The actual invocation itself is merely a jump to an address holding the code of the function: it's a "goto".

While this view may seem unorthodox, the interpreter does not lie, and this is indeed the lesson we learn from studying its structure. Yet the result is surprising to those of us who've been indoctrinated to think in terms of the role of stacks for function calls. So let's work through a few examples. For convenience, we'll use Scheme programs, because we already have some familiarity with such programs in CPS.

2 Example: Factorial

Here's our initial program:

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

You should be able to convince yourself that the following is the same program in CPS:

```
(define fact/k
  (lambda (n k)
    (if (= n 0)
        (k 1)
        (fact/k (- n 1) (lambda (v) (k (* n v)))))))

(define fact
  (lambda (n)
    (fact/k n (lambda (x) x))))
```

To make the stack more explicit, we'll give special names to each of our continuation creations and uses:

```
(define (fact/stack n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/stack (- n 1)
                  (Push stack (lambda (val) (Pop stack (* n val)))))))

(define (Pop stack value)
  (stack value))

(define (Push stack receiver)
  (lambda (v) (receiver v))) ;; same as receiver

(define (fact n) (fact/stack n EmptyStack))

(define (EmptyStack value) value)
```

Merely by assigning names to the same continuations, we've made it possible to change their representations without affecting the original factorial code: that is, we have gained *representation independence*. Because we know the stack is really a sequence of stack records, which is easy to represent as a list, we'll use lists to denote stacks. Notice that we only need to change the continuation abstractions; the two factorial procedures (*fact* and *fact/k*) stay unchanged:

```
(define (Pop stack value)
  ((first stack) value))

(define (Push stack receiver)
  (cons receiver stack))

(define EmptyStack (cons (lambda (value) value) empty))
```

Now we have a version where the stacks themselves are lists, but the elements of the lists—the *activation records*—are still functions. We'll create meaningful names for the functions and use them as constructors of a datatype.

```
(define-datatype stack-record stack-record?
  [stack-rec-mult (n number?)]  ;; combine data with rest of stack using mult
  [stack-rec-empty])  ;; signals bottom of stack

(define (Pop stack value)
  (local ([define top-rec (first stack)])
    (cases stack-record top-rec
      [stack-rec-mult (n) (Pop (rest stack) (* n value))]
      [stack-rec-empty () value])))

(define (Push stack new-record)
  (cons new-record stack))

(define EmptyStack (cons (stack-rec-empty) empty))

(define (fact/stack/rec n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/stack/rec (− n 1) (Push stack (stack-rec-mult n)))))

(define (fact n) (fact/stack/rec n EmptyStack))
```

Notice how CPSing the program (there we go: we've begun to use it as a verb, finally!) and transforming the representation has slowly taken us closer and closer to the very program a compiler might generate! This is a promising direction to consider.

Changing Representation Further

Strictly as an aside for this particular problem, consider the following representations of the stack primitives:

```
(define EmptyStack
  1)

(define Push
  ∗)

(define Pop
  ∗)
```

If we change *fact/stack/rec* to print values as it proceeds, here's what we observe:

```
(define (fact/stack/rec n stack)
  (begin
    (printf "n: ~s  stack: ~s~n" n stack)
    (if (zero? n)
        (Pop stack 1)
        (fact/stack/rec (− n 1) (Push stack n)))))

> (fact 10)
n: 10  stack: 1
n: 9  stack: 10
n: 8  stack: 90
n: 7  stack: 720
n: 6  stack: 5040
n: 5  stack: 30240
n: 4  stack: 151200
n: 3  stack: 604800
n: 2  stack: 1814400
n: 1  stack: 3628800
n: 0  stack: 3628800
3628800  ;; final value
```

In this particular case, we got lucky—by knowing enough about factorial (and associativity), we were able to translate the recursive Scheme program down into something very close to its hand-optimized assembly equivalent! We won't always get that lucky with respect to the optimization, but it is interesting that by compsing CPS with a series of transformations to reduce the level of abstraction, we were able to convert a source program into something that looks quite low-level, making the stack representation very concrete and language-independent. That's promising!

3 Example: Tree Sum

Now that we're done warming up, let's consider an interesting example: a procedure that consumes a tree of numbers and sums all the numbers in the tree. Nothing surprising about the source program:

```
(define-datatype tree Tree?
    [empty-tree]
    [node (n number?)
          (left Tree?)
          (right Tree?)])

(define (tree-sum atree)
  (cases tree atree
    [empty-tree () 0]
    [node (n left right) (+ n
                            (tree-sum left)
                            (tree-sum right))]))
```

Converting it to CPS:

```
(define (tree-sum/k atree k)
  (cases tree atree
    [empty-tree () (k 0)]
    [node (n left right)
          (tree-sum/k left (lambda (lv)
                             (tree-sum/k right (lambda (rv)
                                                 (k (+ n lv rv))))))]))

(define (tree-sum2 atree)
  (tree-sum/k atree (lambda (x) x)))
```

Applying the same transformations as before gives us this:

```
(define-datatype stack-record StackRecord?
    [rec-bottom]
    [rec-add-left (node-val number?)
                  (right-tree Tree?)]
    [rec-add-right (node-val number?)
                   (left-value number?)])

(define (tree-sum/rec atree stack)
  (cases tree atree
    [empty-tree () (Pop stack 0)]
    [node (n left right)
          (tree-sum/rec left (Push stack (rec-add-left n right)))]))

(define (Pop stack value)
  (local ([define top-rec (first stack)])
    (cases stack-record top-rec
      [rec-bottom () value]
      [rec-add-left (node-val right)
                    (tree-sum/rec right (Push (rest stack)
                                              (rec-add-right node-val value)))]
      [rec-add-right (node-val lv) (Pop (rest stack) (+ node-val lv value))])))

(define (Push stack record)
  (cons record stack))

(define EmptyStack (cons (rec-bottom) empty))

(define (tree-sum atree)
  (tree-sum/rec atree EmptyStack))
```

Note in particular that the *Pop* procedure here executes a *Push*. This is more typical of the general case—factorial is unusual in respect.

4 Consolidation

What we've seen today is a sequence of transformations: conversion to CPS; adding representation independence for the stack; converting the stack, or continuation, into a list of functions representing activation records; then converting those functions to first-order representations of the form found in just about any programming language (and easy to simulate in assembly). The result of all this is a program quite different in both style and substance from the original. In style, it has changed its form considerably, while in substance, its representations are all much lower-level. As a result, we are slowly converting high-level Scheme into programs that can run in just about any low-level language. Hopefully you can see where this is going!

Notice what we *not* doing: we are not explicitly writing a compiler. Instead, we are showing you how a compiler would operate: each transformed program is the result of a transformation we would code into the compiler. Actually writing all those transformations is a little more work, and doing so would distract from the mission of this course; courses on compiler construction force you to confront the details that arise in implementing these steps.

Task

For homework, please convert and transform the following program all the way to a concrete representation of the continuation, because we will begin next class assuming that you have done so:

```
(define (filter-pos L)
  (cond [(empty? L) empty]
        [(cons? L)
         (cond [(> (first L) 0)
                (cons (first L) (filter-pos (rest L)))]
               [else (filter-pos (rest L))])]))
```