CPS and Compilers

sk and dbtucker

November 6, 2000

So far this semester we've covered two aspects of programming language design. First, we wrote interpreters so we could actually run programs written in our language. Second, we designed type systems in order to statically determine whether our programs contain any type errors.

What is the next step in making our programming language more useful? We need to speed up the execution of our programs, which we can do by writing a *compiler*. In this section of the course, we will study how to automatically translate programs in our language into low-level machine code.

Continuation passing style

Remember contexts? A context is a procedural representation of the computation remaining to be done using the value of the current expression. We wrote a version of our interpreter that passed the current context as an argument. This style of writing functions with a context parameter is known as continuation passing style (CPS).

We can write any program, not just our interpreter, in continuation passing style. What is the general algorithm for converting a program to CPS form? We'll work through the factorial function as an example:

```
(define !
  (λ (n)
    (if (= n 0)
        1
        (* n (! (- n 1))))))
```

Step 1: Add a parameter k. This parameter will represent the continuation.

```
(define !/k
  (λ (n k)
    ...))
```

Step 2: Send values to k. Instead of returning 1 as the value of the function, we need to apply the continuation k to 1 to perform the rest of the computation.

```
(define !/k
  (λ (n k)
```

```
    (if ...
        (k 1)
        ...)))
```

Step 3: Leave primitives alone when their arguments are values. In this function, the arguments to = are already values.

```
(define !/k
  (λ (n k)
    (if (= n 0)
        (k 1)
        ...)))
```

Step 4: For computations, first order the pieces, then perform the first one with the continuation (λ (*v*) *...rest v...*), where *...rest v...* is the CPS version of the remaining pieces, with *v* substituted for the first piece.

```
(define !/k
  (λ (n k)
    (if (= n 0)
        (k 1)
        (!/k (− n 1) (λ (v) (k (∗ n v)))))))
```

This is the final CPS version of factorial.

Why is the CPS form of a program useful? For one, we have not changed the meaning of the expression. Also, we have taken several decisions out of Scheme's hands. We have made the current context explicit by representing it as a closure and passing it as an argument to the function. We have also nailed down order of operation for our language, rather than relying on Scheme's conventions.

A nice property of CPS is that we can automatically transform any expression into CPS; that is, we can write a program to perform the steps outlined above.

Representation independence

Now that we have the ability to manipulate the context, let's abstract it away. We'll replace the use, extension, and creation of contexts with the functions *Pop*, *Push*, and *Stop* respectively. (The names are purposely chosen to connote a stack.) The factorial function and interface now look like this:

```
(define !/k
  (λ (n k)
    (if (= n 0)
        (Pop k 1)
        (!/k (− n 1) (Push n k)))))

(define !
  (λ (n)
    (!/k n Stop)))
```

And now we just fill in the three new functions with the expressions we had before:

```
(define Stop
  (λ (x) x))

(define Push
  (λ (n k)
    (λ (m) (k (* n m)))))

(define Pop
  (λ (k m)
    (k m)))
```

We haven't done anything interesting so far—we just put the context manipulation routines in separate functions.

Now we can represent contexts in a different manner, without using closures. For the factorial function, the context is just going to be some number of multiplications that need to be done, so we might as well represent the context as the list of numbers to multiply.

The *Stop* function above creates an identity context, which means there is no computation left to be done. In the list representation, the empty list means that there is no remaining computation. The *Push* function creates a new multiplication; using lists, this function will just *cons* the number onto the current list.

The interesting case is *Pop*. When *Pop* is invoked with the closure-based version of contexts, it starts a chain reaction of multiplications that computes the final result. We achieve the same effect with lists by multiplying the value (scheme1) by the first element of the list, and recursively calling *Pop*.

```
(define Stop
  empty)

(define Push
  cons)

(define Pop
  (λ (k m)
    (if (empty? k)
      m
      (Pop (rest k) (* (first k) m)))))
```

The above step (conversion to list representation) is *not* automatic—it relies on the fact that this particular computation is just multiplication of a list of numbers.

Factorial, accumulator style

For our final trick, we'll define our context manipulation functions as follows:

```
(define Stop
  1)

(define Push
  *)

(define Pop
  *)
```

Note that this is *exactly* the accumulator implementation of factorial. What's interesting is that we were able to derive it by converting our original factorial function to CPS, and then playing with the representation of contexts.

Filter

Remember the *filter* function? If not, here's its definition:

```
(define (filter l)
  (if (empty? l)
      l
      (if (f (first l))
          (cons (first l) (filter f (rest l)))
          (filter f (rest l)))))
```

Exercise: Write *filter* in CPS.