Automating the CPS Transformation

sk and dbtucker

November 10, 2000

In a previous lecture, we claimed that we can automatically convert any program to continuation-passing style. Today we will write a function that turns any expression into CPS.

Alternate representation of CPS

For this exercise, we'll find it convenient to use an alternate representation of CPS. Instead of adding a k parameter to each function, we'll have the function return a closure which accepts k. For example, we can write factorial as follows:

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

The recursive call to *!/k* is curried—the function *!/k* is applied to $(-\ n\ 1)$, and the result is applied to the continuation. The interface function must also follow this convention:

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

Automatic conversion to CPS

The *cps* function consumes an expression in our language and produces the CPS version of that expression. Thus its type is:

$$cps : AFunExp \rightarrow AFunExp$$

Given some expression e in our language, the result of $(cps\ e)$ will be an *AFunExp* function that takes a context k as its parameter. The context k will be applied to the value of the expression e.

Here's the shell of the function:

```
(define (cps l)
  (cases AFunExp l
    [numE (n) ...]
    [varE (v) ...]
    [addE (le re) ...]
    [funE (param body) ...]
    [appE (fe ae) ...]))
```

Based on our discussion above, the *numE* case is easy; we return a function which accepts a continuation and applies it to the number:

```
[numE (n) (funE 'k (appE (varE 'k) (numE n)))]
```

For the sake of clarity, we'll rewrite this expression as the following equivalent expression using quasiquote and unquote[1]:

```
[numE (n) (parse `(fun (k) (k ,n)))]
```

The *varE* case is similar—we apply the continuation to the variable:

```
[varE (v) (parse `(fun (k) (k ,v)))]
```

You might wonder why we aren't looking up the variable in the environment. Remember, we are not *evaluating* the program; we are *translating* it to another (equivalent) program which will be evaluated by the interpreter. When the interpreter runs on the CPS version of the program, it will perform the necessary variable lookups.

The *funE* case should also be similar, since functions are values. The one difference is that we must convert the body of the function to CPS:

```
[funE (param body) (parse `(fun (k)
                                (k (fun (,param)
                                        ,(cps body)))))]
```

Now let's look at the *addE* case. First of all, we know the result must be a function of k. Second, since *le* and *re* are arbitrary expressions, we have to call *cps* on them. The result of (*cps le*) accepts a continuation k and applies k to the value of *le*. If we write:

```
[addE (le re) (parse `(fun (k)
                           (,(cps le) (fun (lv) ...))))]
```

then lv is bound to the value of *le* in the ellipsis. We can play the same trick with *re*:

```
[addE (le re) (parse `(fun (k)
                           (,(cps le) (fun (lv)
                                           (,(cps re) (fun (rv) ...)))))))]
```

[1] See the Appendix if you've never used quasiquote and unquote

Now lv is the value of the left expression, and rv is the value of the right expression. The function returned by *cps* is required to take a continuation k and apply it to the value of the expression. The value of the *addE* expression is (+ lv rv), so the final version is as follows:

```
[addE (le re) (parse `(fun (k)
                          (,(cps le) (fun (lv)
                                          (,(cps re) (fun (rv)
                                                          (k (+ lv rv))))))))]
```

Is the *appE* case exactly the same as *addE*?

```
[appE (fe ae) (parse `(fun (k)
                          (,(cps fe) (fun (fv)
                                          (,(cps ae) (fun (av)
                                                          (k (fv av))))))))]
```

No—this is incorrect. What will be the result of evaluating (fv av)? It will be the CPS version of the function's body, with the parameter bound to the value of the argument.

Exercise: Convince yourself of the above statement.

We want to evaluate the body with the continuation we were given; therefore, we apply the result of (fv av) to k:

```
[appE (fe ae) (parse `(fun (k)
                          (,(cps fe) (fun (fv)
                                          (,(cps ae) (fun (av)
                                                          ((fv av) k)))))))]
```

The final version of the *cps* function looks like this:

```
(define (cps l)
  (cases AFunExp l
    [numE (n) (parse `(fun (k) (k ,n)))]
    [varE (v) (parse `(fun (k) (k ,v)))]
    [addE (le re) (parse `(fun (k)
                               (,(cps le) (fun (lv)
                                               (,(cps re) (fun (rv)
                                                               (k (+ lv rv))))))))]
    [funE (param body) (parse `(fun (k)
                                    (k (fun (,param)
                                            ,(cps body)))))]
    [appE (fe ae) (parse `(fun (k)
                               (,(cps fe) (fun (fv)
                                               (,(cps ae) (fun (av)
                                                               ((fv av) k)))))))]))
```

Appendix

Here's a brief description of quasiquote and unquote. Say you want to write a function that takes a number x, and returns a list whose third element is the square of x. You could write:

```
(define (make-list x)
  (cons 1 (cons 2 (cons (* x x) (cons 4 (cons 5 empty))))))

> (make-list 10)
(1 2 100 4 5)
```

We would like to use quote notation to make lists, but the $(* \ x \ x)$ gets treated as a list of symbols:

```
(define (make-list x)
  '(1 2 (* x x) 4 5))

> (make-list 10)
(1 2 (* x x) 4 5)
```

The solution is to use quasiquote and unquote. You create the list using a quasiquote. Every element is treated as a symbol—except that if you unquote the element, is is evaluated. We write quasiquote with a back tick (‘) and unquote with a comma (,):

```
(define (make-list x)
  `(1 2 ,(* x x) 4 5))

> (make-list 10)
(1 2 100 4 5)
```