Comp 411
Principles of Programming Languages
Lecture 26
Explaining `letcc` and `error`

Corky Cartwright

March 31, 2014

Continuations and Evaluation Contexts

One of our goals is to produce a tail-recursive interpreter, which can serve as a guide to implementing an interpreter in machine code or writing a compiler to translate source programs to machine code.

To recap our discussion of CPS, during the evaluation of a program, every phrase is surrounded by some computation that is waiting to be performed (and, typically, that depends on the value of this phrase). In a rewrite-rule semantics, the program text for the remaining computation is simply the surrounding text; it is called an *evaluation context*. Turning the meaning of this evaluation context into a program function is the act of making the continuation explicit. This process is called *reification*.

An Example of Reification

For instance in

```
(+ (* 12 3) (- 2 23))
```

the evaluation context of the first sub-expression (assuming it is evaluated first) is

```
(+ _ (- 2 23))
```

(where we pronounce _ as ``hole''), so the program function corresponding to this context is

```
(lambda (x) (+ x (- 2 23)))
```

A CPSed Interpreter for LC

Let us consider our interpreter for LC:

```
(define Eval
  (lambda (M env)
    (cond ((var? M) (lookup M env))
          ((lam? M) (make-closure M env))
          ((app? M)
           (Apply (Eval (app-rator M) env)
                  (Eval (app-rand M) env)))
          ((add? M) ...)
          ...)))
```

In this interpreter, we both create new implicit continuations (growing the stack) and use implicit continuations (returning into the stack). New implicit continuations are created in the code for applications. The other two clauses shown use the current implicit continuation by returning a value.

A CPSed Interpreter for LC II

We now use the standard technique for transforming Scheme code to transform the interpreter into CPS, making implicit continuations explicit:

```
(define Eval/k
  (lambda (M env k)
    (cond ((var? M) (k (lookup M env)))
          ((lam? M) (k (make-closure M env)))
          ((app? M)
           (Eval/k (app-rator M)
                   env
                   (lambda (rator-v)
                     (Eval/k (app-rand M)
                             env
                             (lambda (rand-v)
                               (Apply/k rator-v rand-v k))))))
          ...)))
```

A CPSed Interpreter for LC III

Similarly

```
(define Apply
  (lambda (f a)
    (cond ((closure? f)
           (Eval (body-of f)
                 (extend (env-of f) (param-of f) a)))
          (else ...))))
```

becomes

```
(define Apply/k
  (lambda (f a k)
    (cond ((closure? f)
           (Eval/k (body-of f)
                   (extend (env-of f) (param-of f) a) k))
          (else ...))))
```

where **extend** is treated as a primitive operation like **body-of**. Note that the continuations for the two recursive calls on **Eval** in original interpreter are different. Why?

Explaining `error` in Scheme code

We intentionally left the fall-through case of the `cond` expressions in the `cond` procedures empty (which can generate meta-errors). However, there should be a call to `error` in that slot. In the CPSed form, we can return an error without relying on an error-throwing mechanism in the metalanguage!

```
(define-struct error (msg))

(define Apply/k
  (lambda (f a k)
    (cond ((closure? f)
           (Eval/k (body-of f)
                   (extend (env-of f) (param-of f) a) k))
          (else
            (make-error "Attempted to apply non-closure")))))
```

Note that the error clause discards the continuation `k`. We could also include a similar error clause in `Eval/k`.

Explaining `letcc` in JAM code

In our direct interpreters, the only way we could define `letcc` was to use the `letcc` construct in Scheme, which explains nothing. The relevant clause in `Eval` would be

```
((letcc? M)
 (letcc k (Eval (body-of M)
                (extend env (var-of M) k))))
```

given the expression syntax

```
(define-struct letcc (var body)).
```

In our CPSed interpreter, we can define `letcc` without any special support from the metalanguage:

```
((letcc? M) (Eval/k (body-of M)
                    (extend env (var-of M) k)
                    k))
```

Note that we can now easily implement `letcc` in interpreters written in languages (like Java) without continuations.