Comp 411
Principles of Programming Languages
Lecture 23
Explaining `letcc` and `error`

Corky Cartwright

November 14, 2012

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*. We will use Scheme rather than Scala to illustrate these ideas since Scheme actually includes our example reification primitives and the syntax is simpler.

An Example of Reification in Scheme

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 a Scheme 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 continuations and use the (implicit) continuation. New continuations (non tail-calls) are created in the code for applications. The other two clauses (raw returns) shown use the current continuation by passing it a value.

A CPSed Interpreter for LC

We now use the standard technique for transforming Scheme code to transform the interpreter into CPS:

```
(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 cont.

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**

Explaining `error` in Scheme code

We intentionally left the fall-through case of the **`cond`** expressions in the **`cond`** procedures empty. 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`**; it does NOT abort. 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 rely on `letcc` the construct in Scheme, which explains nothing. The relevant clause in an `Eval` function for JAM 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 a 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))
```

Implemening `call/cc` or `letcc`

Assume you are writing a compiler for a language supporting `call/cc` or `letcc`. How hard is it to implement this construct? If records are heap allocated (as in Standard ML), the implementation is trivial because the current continuation is explicitly available in the heap as the pointer to the current activation record. (You simply generate code for a unary procedure that returns its argument to the context represented by the current activation record.)

What if you runtime is represented by a stack? The brute-force approach is to copy the stack into the heap (assuming that free variables in procedures are heap allocated). Ugh. Kent Dybvig has written a several papers (in the context of Chez Scheme) on how you can support continuations efficiently without affecting the performance of programs that do not use continuations (unlike SML). The key idea is to copy the stack lazily into the continuation representation. If the saved continuation is actually applied, the current stack is blown away. But in the worst case, we really do need to make a copy of the stack at the time of continuation creation. (Assume the continuation is stored in a global variable, control returns to top level, and the continuation is applied.)