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

Corky Cartwright

April 5, 2021

Continuations and Evaluation Contexts

For the sake of simplicity assume that every redex (phrase being evaluated) in a program corresponds to a continuation, which is equivalent to performing the cps conversion process on every node of a syntax tree including constants (which is absurd in practice because almost every constant never escapes [discards the passed continuation]). 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 invoke implicit continuations (returning into the stack). New implicit continuations are created in the code for `Apply`. The other two clauses shown invoke the current implicit continuation by returning a value.

A CPSed Interpreter for LC cont.

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))))))
          ...)))
```

Explaining `letcc` in JAM code

We want to add a `letcc` construct to LC with syntax `(letcc x M)` where `x` is a variable name and `M` is an LC expression. This construct simply creates an extended environment (like other `let` constructs) with bound to the current continuation (reified context) and evaluates the expression M in the extended environment. Assume that have introduced an abstract syntax structure for in our JAM interpreter (written in Scheme) including the operations `letcc?`, `body-of`, and `var-of`. In our direct (non CPSed) interpreters written in Scheme, the only way we can define `letcc` is 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. On the other hand, CPSing an interpreter written in Java is a daunting task. In Assignment 6, we use a different approach. When we CPS programs, then it is easy to support `letcc` in a framework without support for continuations because we actually construct program text defining the continuation for each node in the AST.

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`**. Note that the continuations for the two recursive calls on **`Eval`** in original interpreter are different. Why?

Explaining `letcc` in JAM code, cont.

What if we had *performed the CPS transformation on the Jam code instead of our Scheme interpreter for Jam*? Then we can write a continuation-based interpreter `Eval'` by invoking `Apply` from a conventional (not CPSed!) Jam interpreter as a help function. We simply define

```
(define (Eval' M-k) (Apply (Eval M-k Empty-Env) (lambda (x) x)))
```

where `M-k` is the CPS conversion (a lambda-expression) of the source program `M` and `Empty-Env` is bound to the empty environment. If eliminating the associated stack space and stack manipulation overhead is important, we could either write an interpreter without any recursion (or even any procedure calls), exploiting the fact that all calls on program-defined functions occur in tail position. But even a conventional meta-interpreter using standard stack allocation (like your Java interpreters for Jam) can still define the correct behavior for `letcc` if the input programs (*e.g.*, Jam source programs) have been CPSed because the CPSing process can support `letcc`. When the CPS interpreter implements the clause for `letcc` with binding variable `k'`, the interpreter (`Eval`) can simply extend the environment by binding `k'` to the value of `k` and recursively calling the interpreter (`Eval`) with the extended environment.

`<fn-name>(arg`$_1$`, . . ., arg`$_n$`, k)`

so the interpreter (`Eval`) can call `Apply` with three arguments `f`, `a`, `k` just like `Eval/k` does after evaluating the `n+1` arguments passed to the called function `<fn-name>`.