Comp 311
Principles of Programming Languages
Lecture 27
The CPS Transformation

Corky Cartwright

November 12, 2010

CPS Granularity

In pure form, the CPS transformation is typically given for the untyped λ-calculus (see the optional notes on the CPS Transformation in OCaml). But this characterization (like most formalisms based on the untyped λ-calculus) is misleading in practice because it does not address the issue of processing primitive operations (the untyped λ-calculus has no primitive operations!). Neither does System F.

Of course, primitive operations are much easier to process than program functions because they generally cannot abort (a few operations like division are exceptions) or otherwise discard the pending continuation.

But primitive operations can be treated like program functions provided the libraries implementing are re-shaped so that every such operation takes an extra continuation argument. The designation of which operations are primitive has a huge impact on the final form of the CPSed code. If primitive operations are CPSed, then the CPSed code is much more complex. In practice, CPSing primitives is generally not advisable since CPSing adds overhead (extra function arguments and extra function calls) and we typically only need to CPS the operations that correspond to machine-level subroutine calls.

CPSing Within Compilers

The CPS transformation is often performed by compilers for “higher -order” languages (those that support functions as data), because CPSing exposes all of the operations that are implicitly performed on the stack in standard code (which uses an algol-like stack run-time).

But there are less severe alternative transformations (notably *A-normal form*) that perform much the same function. In A-normal form, every non-trivial intermediate result is explicitly stored in a local variable. An application is trivial iff the rator is a primitive operation.

If no operation is treated as primitive, then A-normal form conversion is very similar to a much older representation used in optimizing compilers called value-numbering. In value-numbering hashing is used to avoid duplicating values.

Review: The CPS Transformation

Assume Jam/Scheme programs are restricted to a form where the body of a function is either (*i*) a primitive expression constructed from constants, variables and primitive functions, and program-defined functions; or
(*ii*) a conditional where the predicates are *primitive* expressions and the result clauses are *ordinary* expressions (primitive expressions augmented by program-defined functions) . Then the CPS transformation of such a program is defined as follows:

1. Add an extra parameter `k` to every function.
2. For each function body `b` that is a primitive expression, write `(k b)`.
3. Each clause in a conditional is treated separately:
   a) For each result clause `b` composed from primitive operations and constants, write `(k b)`.
   b) For each clause containing calls on program-defined functions, pick the call that will be evaluated first. Make the body for the new clause a call that takes an extra argument, which is of the form `(lambda (res) body)`. The original contents of that clause are placed in the `body`, enclosed in a call on the continuation `k`, with the selected call replaced by `res`.
   c) Repeat preceding step 3b until no unconverted function calls remain.

Review: Another Example

```
(define Pi
  (lambda (t)
    (cond
      ((leaf? t) t)
      (else (* (Pi (left t))
               (Pi (right t)))))))
```

Then first iteration in creating the CPS version, `Pi-k`, is

```
(define Pi-k
  (lambda (t k)                          ;; rule 1
    (cond ((leaf? t) (k t))              ;; rule 3a
          (else                          ;; rule 3b
            (Pi-k (left t)
                  (lambda (res)
                    (k (* res (Pi (right t)))))))))))
```

Second Iteration

```
(define Pi-k
  (lambda (t k)                        ; rule 1
    (cond ((leaf? t) (k t))            ; rule 3a
          (else                        ; rule 3b
            (Pi-k (left t)
                  (lambda (r1)
                    (Pi-k (right t)
                          (lambda (r2) (k (* r1 r2)))))))))))
```