Comp 411
Principles of Programming Languages
Lecture 24
The Low-Level Meaning of Function Calls

Corky Cartwright
March 29, 2019
Machine-level Semantics

Interpreters written in a clean functional metalanguage (like functional Scheme or Haskell) provide clear definitions of meaning for programming languages. We can even tolerate occasional use of imperative features (e.g., mutation operations on shared data in Scheme) and still claim that interpreters clearly define the behavior of programs (provided we define the semantics of our imperative extension of metalanguage).

But these interpreters do not describe how to implement programming languages efficiently in terms of conventional machine instructions. What is missing? A description of the meaning of function calls and error operations used in the interpreter metalanguage. Primitive functions (e.g., addition of 2's complement integers) are implemented by machine instructions or short sequences of machine instructions. In principle, we would like to know how to hand translate our interpreters to machine code, or even better, how to compile source programs directly into machine code.
Guidance From an Example

Consider the following Jam procedure, which computes the product of a list of numbers:

```
let Pi := map l to
    if l = null then 1 else first(l)*Pi(rest(l))
in Pi(...)
```

What if the input list may be corrupt (contain non-numbers)? How can $\text{Pi}$ report an error?

```
let
    Pi-acc := map l,acc to
        if l = null then acc
        else if Pi-acc(rest(l), first(l)*acc);
    Pi-2 := map l to Pi-acc(l, 1);
in Pi-2(...) 
```

Does $\text{Pi-2}$ preserve the order of evaluation in $\text{Pi}$?
Guidance From an Example cont.

Suppose \texttt{Pi-2} is passed a corrupt list. In that case, \texttt{Pi-acc} multiplies all the numbers found until the erroneous input is encountered. Can we avoid these wasted multiplications? Yes. By making \texttt{acc} into a suspension (thunk).

\begin{verbatim}
(define Pi/acc
  (lambda (l acc)
    (cond ((null? l) (acc))
          ((number? (first l))
            (Pi/acc (rest l) (lambda () (* (first l) (acc))))))
          (else (first l))))

(define Pi-3 (lambda (l) (Pi/acc l (lambda () 1))))
\end{verbatim}

This program avoids unnecessary multiplications. But like \texttt{Pi-2}, it changes the order of multiplications from \texttt{Pi}. If the \texttt{*} primitive were not associative, the transformation used to create \texttt{Pi-2} would not work.
Systematically Avoiding Nested Function Calls

We failed to preserve the evaluation order in \texttt{Pi} in constructing \texttt{Pi-2} and \texttt{Pi-3} because updating an accumulator reverses the order of operations on a list. Is there a systematic way to avoid nesting function calls while preserving evaluation order? Yes! It is called transformation to continuation-passing style (CPS). The CPS transform of \texttt{Pi} is:

\begin{verbatim}
(define Pi-k
  (lambda (l k) ; k is rest of the computation reified as a function
    (cond ((null? l) (k 1))
          ((number? (first l))
           (Pi-k (rest l) (lambda (prod) (k (* (first l) prod)))))
          (else (first l)))) ;; escape on an illegal input!

(define Pi-4 (lambda (l) (Pi-k l (lambda (x) x))))
\end{verbatim}

Why does the (else ...) clause work? Because \texttt{Pi-4} and \texttt{Pi-k} are tail-recursive. \texttt{Pi-4} is only called at top-level; it tail-calls the tail-recursive function \texttt{Pi-4}. Hence, the value returned by the else clause is guaranteed to return directly to the top-level caller of \texttt{Pi-4}. CPS converts all functions to tail-form (no nested function calls).
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; (ii) an *ordinary expression*, which has the same definition as a primitive expression except that may contain calls on program-defined functions; (iii) 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. For each function body that is a conditional expression, each predicate (test expression) is unchanged and each result clause is treated separately as follows:
   a) for each result clause \( b \) composed from primitive operations and constants, write \((k \ b)\).
   b) for each clause (which we call *body*) 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 \((\text{lambda (res) body})\). The original contents of that clause are placed in the *body*, enclosed in a call on the continuation \( k \), with the selected recursive call replaced by \( \text{res} \). Repeat this step 3b until no unconverted function calls remain.
   **Note:** each continuation corresponds to a reification of the enclosing computation performed in the enclosing static chain (assuming an Algol-stack implementation of the original program where the top level let creates an activation record and each function call creates a new activation record and let constructions other than the top level let do not.
4. For each function body that is an ordinary expression (but not a primitive expression), convert it to CPS form using the process described in 3b above.
Another Example

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

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

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