Comp 411
Principles of Programming Languages
Lecture 22
The Low-Level Meaning of Function Calls

Corky Cartwright

November 9, 2012

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 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. 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 procedure, which computes the product of a list of numbers:

```
def pi(l: List[Int]): Int = l match {
    case List() => 1
    case head :: tail => head * pi(tail)
  }
```

What if the input list may contain zero and we want to avoid useless work?

```
def piAcc(l: List[Int]) = piAccHelp(l, 1)
def piAccHelp(l: List[Int], acc: Int): Int = l match {
  case List() => acc
  case first :: rest =>
    if first = 0 then 0 else piAccHelp(rest, first*acc)
```

Does `piAcc` preserve the order of evaluation in `pi`? No. `pi` multiplies from right-to-left while `piAcc` multiplies from left-to-right.

Guidance From an Example cont.

Does **piAcc** avoid all unnecessary multiplications? Suppose **piAcc** is passed a list containing **0**. In that case, **piAcc** multiplies *all the numbers* found until **0** is encountered. Can we avoid these wasted multiplications? Yes. By making **acc** into a suspension which we can perform in Scala simply by passing **acc** by name.

```
def piAccBN(l: List[Int]) = piAccHelp(l, 1
def piAccBNHelp(l: List[Int], => acc: Int): Int = l match {
  case List() => acc
  case first :: rest =>
    if first = 0 then 0 else piAccBNHelp(rest, first*acc)
```

This program avoids unnecessary multiplications. But like **piAcc**, it changes the order of multiplications from **pi**. If the * operation were not associative, the transformation used to create **piAcc** would not work.

Systematically Avoiding Nested Function Calls

We failed to preserve the evaluation order in **pi** in constructing **piAcc** and **piAccBN** 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 **pi** , enhanced with short-circuiting on **0**, is

```
def piCPS(l: List[Int], k: (Int) => Int = ((i:Int) => i)): Int =
  l match {
    case List() => k(1)
    case head :: tail =>
      if (head == 0) k(0)
      else piCPS(rest, (prod: Int) => k(head * prod))
  }
```

Why does the **if** clause work? Because **piCPS** is tail-recursive; it is only called at top-level. Hence, the value returned by the else clause is guaranteed to return directly to the caller of `Pi-4`. CPS converts all functions to tail-form (no nested function calls).

The CPS Transformation

Assume Scala 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/pattern match 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:

a)1. Add an extra parameter `k` to every function.

2. For each function body `b` that is a primitive expression, write `(k b)`.

3. Given a conditional with a primitive test, 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 `(res: T) => 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 `res`.
   – Repeat step (3b) until no unconverted function calls remain.

Another Example

```
sealed trait BinaryTree[T]
case class Leaf[T](value: T) extends BinaryTree[T]
case class Branch[T](left: T, right: T) extends BinaryTree[T]

def pi(t: BinaryTree[Int]): Int =
  t match {
    case Leaf(i) => i
    case Branch(left,right) => pi(left)*pi(right)
  }
```

Then first iteration in creating the CPS version **piCPS** is

```
def piCPS(t: BinaryTree[Int], k: (Int => Int)) =   // rule 1
  t match {
    case Leaf(i) => k(i)                 // rule 3a
    case Branch(left,right) =>           // rule 3b
      piCPS(left, (res1:Int) => k(res1 * pi(right))
  }
```

Second Iteration

```
def piCPS(t: BinaryTree[Int], k: (Int => Int)) =   // rule 1
 t match {
   case Leaf(i) => k(i)                 // rule 3a
   case Branch(left,right) =>           // rule 3b
     piCPS(left,
           (res1:Int) =>
             piCPS(right, (res2:Int) => k(res1 * res2)))
 }
```