Comp 411
Principles of Programming Languages
Lecture 25
Exposing the Low-Level Meaning of Function Calls Via the CPS Transformation

Corky Cartwright

March 30, 2020

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 the 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 **`Pi`** report an error?

```
let
  PiAcc := map l,acc to
    if l = null then acc
    else PiAcc(rest(l), first(l)*acc);
  Pi2 := map l to PiAcc(l, 1);
in Pi2(...)
```

Does **`Pi2`** preserve the order of evaluation in **`Pi`**?

Guidance From an Example cont.

Suppose `Pi2` is passed a corrupt list. In that case, `PiAcc` multiplies *all the numbers* found until the erroneous input is encountered. Can we avoid these wasted multiplications? Yes. By making `acc` into a suspension (thunk), which requires wrapping `acc` in a function with a dummy parameter `?`.

```
let PiLAcc := map l,lacc to
              if null?(l) then lacc(true)
              else if number?(first(l)) then
                PiLAcc(rest(l), map ? to first(l)*lacc(true))
              else first(l)  // return the rogue element;

    Pi3 := map l to PiLAcc(l, map ? to 1)
```

This program avoids unnecessary multiplications. But like `Pi2`, it changes the order of multiplications from `Pi`. If the `*` primitive were not associative, the transformation used to create `Pi2` would not work.

Systematically Avoiding Nested Function Calls

We failed to preserve the evaluation order in **`Pi`** in constructing **`Pi2`** and **`Pi3`** 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`** is:

```
let Pik := map l,k to  //function k performs the rest of the computation
             if null?(l) then k(1)
             else if number?(first(l)) then
               Pik(rest(l), map prod to k(first(l)*prod))
             else first(l); // abort on an illegal input
     Pi4 := map l to Pik(l, map x to x)
```

Why does the first **`else`** clause work? Because **`Pi4`** and **`Pk`** are tail-recursive. **`P4`** is only called at top-level; it tail-calls the tail-recursive function **`Pik`**. Hence, the value returned by an **`else`** clause is guaranteed to return directly to the top-level caller of **`Pi4`**. CPS converts all functions to tail-call-form (*every call on a program-defined function can only appear as the last function call or operation in the execution of the function body*). A CPSed program can be executed without the support of a call stack! All subroutine call operations become jumps. Argument values can be passed in a fixed block of registers (limiting arity to block size).

The CPS Transformation

A *primitive expression* is constructed from constants, variables, operators and primitive functions. Assume Jam programs are restricted to a form where **`let`** only appears at the top-level and the body of a function is either :

- an *ordinary expression*, which has the same definition as a primitive expression except that may contain calls on program-defined functions; or
- a *conditional expression* **`if-then-else`** chain) 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 definition **`map`**.
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 **`map res to 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 this step (3b) until no unconverted function calls, including continuations, remain.

       A continuation corresponds to a *reification* (packaging some program behavior as a function) performing the rest of the computation as described by the control stack (in an Algol-style runtime) The top-level **`let`** creates an activation record and each function call creates a new activation.

       The generated continuation functions have the same restricted form as the original program.
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

Assume that Jam includes binary trees with **`int`** leaves (the **`BT`** type from Lecture ??). Then:

```
let treePi := map t to
                if leaf?(t) then t
                else treePi(left(t))*treePi(right(t))
```

In the first iteration in creating the CPS version is:

```
let treePik :=
  map t,k to
    if leaf?(t) then k(t)
    else treePik(left(t), map res to k(res*treePi(right(t)))
```

The preceding program in not in *tail-call-form* because the continuation function in the recursive call on **`treePik`** is not in tail-call-form. In addition is still contains a call on **`treePi`**

Second Iteration

After the first iteration, the generated continuation (which is a program `map`) is *not* in tail-call-form, so we must transform it as well.

```
let tree-Pi-k :=
  map t,k to
    if leaf?(t) then k(t)
    else tree-Pi-k(left(t),
                   map r1 to tree-Pi-k(right(t),
                                       map r2 to k(r1*r2))
```

which is in tail-call-form.

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 the polymorphic λ-calculus (System F).

Of course, primitive operations are much easier to process than program functions because they typically do not abort (a few operations like division and object accessors are exceptions) or otherwise discard the pending continuation. *Modular 2's complement arithmetic* (other than division) is a good example.

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 subroutine calls.

CPSing Within Compilers

The CPS transformation is often performed by compilers for “higher - order” languages (those that support functions as data values), 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 subtrees in a concrete representation of the abstract syntax of a program.

Reviewing the CPS Transformation

Assume Jam/Scheme programs are restricted to a form where the body of a function is either:

- a *primitive* expression constructed from constants, variables, operators, and primitive functions, and program-defined functions; or
- 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 of the new clause a call on a reshaped version of the program-defined function that takes an extra argument of the form **`map res to body`**, called the continuation. 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 3(b) until no unconverted function calls remain.

Review: Another Example

```
let
  treeSum :=
    map t to if leaf?(t) then t
             else Tree-Sum(left(t)) + Tree-Sum(right(t))
In treeSum( ... )
```

Then first iteration in creating the CPS version, `treeSumK`, is

```
let
  treeSumK :=
    map t,k to if leaf?(t) then k(t)
               else treeSumK(left(t),
                             map res to k(res + treeSum(right(t)))
in treeSumK( ... , map x to x)
```

Second Iteration

```
let
  treeSumK := map t,k to                          // rule 1
    if leaf?(t) k(t)                              // rule 3a
    else treeSumK(left(t),                        // rule 3b
                map r1 to treeSumK(right(t),
                                   map r2 to k(r1 + r2)))
in treeSumK( ...,  map x to x)
```

Comprehensive Formulations of the CPS Transformation

The rules for performing the CPS transformation are more complex in the context of explicit binding constructs like **`lambda`**, **`let`**, and **`letrec`** (recursive **`let`**). In principle, these extensions do not add anything new, but they complicate the detailed structure of environments and the CPS transformations eliminates explicit environments (other than local variables) by encoding environments (represented using the stack in algol-like run-times) as closures (continuations) in the heap.

Study the rules for Assignment 6, which constitute one possible way to handle the Jam recursive **`let`** and **`map`** constructs. Good CPS translations are concise. The rules for Assignment 6 produce reasonably concise CPS translations but they could be improved at the cost of more complexity.