

Introduction to CPS

CS 502

Lecture 5

9/11/08

Review

- We're interested in defining intermediate representations that capture a program's control and dataflow properties.
 - Problems:
 - How do we pick a good representation?
 - Tradeoffs between simplicity and generality
 - How do we translate from our high-level program (e.g., Mini-ML) to this representation?
-

Continuation-Passing Style

- Starting point:
 - How do we represent a program's control-flow?
 - Loops
 - Procedure call and return
 - Mini-ML abstracts these notions as primitives:
 - We don't see the "instruction sequence" that represents a procedure call or a loop iteration.
 - Can we make these sequences explicit?
-

Example

Consider a factorial function:

```
fun fact(n:int):int = if n = 0
                        then 1
                        else n * fact(n-1)
```

Each call to fact is made with a "promise" that the value returned will be multiplied by the value of n at the time of the call.

Example (cont)

Now, consider:

```
let fun fact-iter(n:int):int =  
  let fun loop(n:int,acc:int):int =  
    if n = 0  
      then acc  
      else loop(n - 1, n * acc)  
  in loop(n,1)  
end
```

There is no promise made in the call to loop by fact-iter, or in the inner calls to loop: each call simply is obligated to return its result.

Unlike fact, no extra control state (e.g., promise) is required; this information is supplied explicitly in the recursive calls.

What is the implication of these different approaches?

- Recursive vs. iterative control
-

Tail position

- An expression in tail position requires no additional control-information to be preserved.
 - Intuitively, no state information needs to be saved.
 - Examples:
 - The true and false branches of an if-expression.
 - A loop iteration.
 - A function call that occurs as the last expression of its enclosing definition.
 - Tail recursive implementations can execute an arbitrary number of tail-recursive calls without requiring memory proportional to the number of these calls.
-

Continuation-passing style

- Is a technique that can translate any procedure into a tail recursive one.
- Example:

$4 * 3 * 2 * \text{fact}(1)$

- Define the context of $\text{fact}(1)$ to be

$\text{fn } v \Rightarrow 4 * 3 * 2 * v$

The context is a function that given the value produced by $\text{fact}(1)$ returns the result of $\text{fact}(4)$

Example revisited

```
fun fact-cps(n:int, k: int -> int): int =  
  if n = 0  
  then k(1)  
  else fact-cps(n-1, fn v => k (n * v))
```

The k represents the function's continuation: it is a function that given a value returns the "rest of the computation"

By making k explicit in the program, we make the control-flow properties of `fact` also explicit, which will enable improved compiler decisions.

Observe that $k(\text{fact}(n)) = \text{fact-cps}(n, k)$ for any k .

Example revisited

fact-cps(4,k) -->

fact-cps(3, fn v => k(4,v))

fact-cps(2, fn v => (fn v => k(4 * v))(3 * v))

fact-cps(2, fn v => k (4 * 3 * v))

fact-cps(1, fn v =>

 (fn v => k (4 * 3 * v))

 (2 * v))

fact-cps(1, fn v => k (4 * 3 * 2 * v))

....

fact-cps(0, fn v => k (4 * 3 * 2 * 1 * v))

(fn v => k (4 * 3 * 2 * 1 * v)) 1

k 24

by def. of fact-cps
by application

The initial k supplied to fact-cps represents the "context" in which the call was made.

Our Goal

- Take a mini-ML program and convert it to CPS form.
 - Issues:
 - Where do we insert continuations?
 - How do we record the “rest of the computation” that a continuation is to represent?
 - How do we distinguish between continuations that
 - Represent the return point of an arbitrary procedure call (e.g., the outer call to fact-cps).
 - Represent iterative computation (e.g., the inner recursive calls in fact-cps)
-

First cut

- Start with a very simple language:
 - Variables, functions, applications, and conditionals.
 - Define a translation function:
 - $C : \text{Exp} \times \text{Cont} \rightarrow \text{Exp}$
 - A continuation will be represented as a function that takes a single argument, and perform "the rest of the computation"
 - The translation will ensure that
 - Functions never directly return -- they always invoke their continuation when they have a value to provide.
-

The Initial Algorithm

$$C[x]k = k\ x$$

Returning the value of a variable simply passes that value to the current continuation.

$$C[\lambda x.e]k = k(\lambda x\ k'. C[e]k')$$

A function takes an extra argument which represents the continuation(s) of its call point(s), and its body is evaluated in this context. In the following, $\lambda x.e$ means $\text{fn } x \Rightarrow e$, and $\lambda x_1\ x_2. e$ means $\text{fn } (x_1, x_2) \Rightarrow e$

$$C[e_1(e_2)]k = C[e_1]\lambda v. C[e_2]\lambda v'.v(v', k)$$

An application evaluates its first argument in the context of a continuation that evaluates its second argument in the context of a continuation that performs the application and supplies the result to its context.

Initial Algorithm (cont)

$C [\text{if } e_1 \text{ then } e_2 \text{ else } e_3] k =$

$C[e_1] \lambda v. \text{if } v \text{ then } C[e_2]k \text{ else } C[e_3]k$

Evaluate the test expression in a context that evaluates the true and false branch in the context of the conditional.

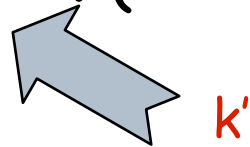
Note that k is duplicated in both branches. We would like to avoid this.

Example

$C [(x_1(x_2) x_3)] k \rightarrow$

$C [x_1(x_2)] \lambda v_1 . C [x_3] \lambda v_2 . v_1(v_2, k) \rightarrow$

$C [x_1(x_2)] \lambda v_1 . (\lambda v_2 . v_1(v_2, k)) x_3 \rightarrow$



$C [x_1] \lambda v_3 . C [x_2] \lambda v_4 . v_3(v_4, k') \rightarrow$

$(\lambda v_3 . (\lambda v_4 . v_3(v_4, k')) x_2)$

$x_1)$

Example (cont)

$C [x1] \lambda v3 . C [x2] \lambda v4 . v3(v4, k') \quad \rightarrow$

$(\lambda v3 . (\lambda v4 . v3(v4, k') x2) x1) \quad \rightarrow$

$(\lambda v3 . (\lambda v4 . v3(v4, k') x2) x1) \quad \rightarrow$

$x1(x2, k') \quad \rightarrow$

$x1(x2, (\lambda v1 . (\lambda v2 . v1(v2, k)) x3)) \quad \rightarrow$

$x1(x2, (\lambda v1 . v1(x3, k)))$

Issues

- Specifying the translation is pretty simple, but the resulting code is very complex:
 - Lots of functions and function calls that need to be eliminated.
 - Haven't distinguished between different uses of continuations:
 - Loops and known jump points
 - Unknown return points
 - Will explore a more refined translation algorithm next time.
-