

# More on CPS

---

CS 502

Lecture 6

9/18/08

# Possible Optimizations

---

- First mark all continuations
  - Normalize terms according to
    - $\beta$ : removes unnecessary applications
      - $((\lambda x. M) N) \rightarrow M[x/n]$
    - $\eta$ : removes unnecessary tail calls
      - $\lambda x. M x \rightarrow M$  (if  $x$  not in  $FV(M)$ )
  - Intermediate reductions still quite large
  - Still does not distinguish among continuations used in different ways.
-

# A More Refined Translation

---

Source language:

$S ::= x \mid S + S$

$E ::= S \mid E + E \mid E(E) \mid$

$\text{if } E \text{ then } E \text{ else } E \mid$

$\text{let } x = E \text{ in } E \mid$

$\text{fn } x \Rightarrow E \mid$

$f(E)$

---

# Continuation and Procedure Kinds

---

- Will introduce three kinds of procedures:
    - Proc : used as the translation of source-level procedures. These procedures take a continuation as an argument. Invoked as an ordinary procedure.
    - Ret: continuations used as return points for calls to Procs.
    - Jump: continuations used within the current procedure instead of being passed to another one. Useful when two arms of a conditional have to rejoin at a common point and for bodies of loops. Can be compiled as a goto.
-

# Translation Algorithm

---

$F : \text{Exp} \times \text{Cont} \rightarrow \text{Exp}_c$

Simple Expressions

$F[x]k = k(x)$

$F[x + y]k = k(x + y)$

$F[\text{fn } x \Rightarrow E]k = k(\text{fn\_p } (x, k') \Rightarrow F[E]k')$

(fn\_p stands for a "Proc" procedure)

---

# Translation Algorithm

---

## Ret continuations

Fix order of evaluation for let expressions. Here, we assume the  $E_i$  are not simple (variables, or primitive operations).

$F[\text{let } x = E_1 \text{ in } E_2] k =$

$F[E_1] \text{fn\_r}(x) \Rightarrow F[E_2]k$

$F[E_1 + E_2] k =$

$F[E_1] (\text{fn\_r}(y) \Rightarrow (F[E_2] \text{fn\_r}(z) \Rightarrow F[y+z]k))$

$F[E_1(E_2)] k =$

$F[E_1] (\text{fn\_r}(y) \Rightarrow (F[E_2] \text{fn\_r}(z) \Rightarrow F[y(z)]k))$

---

# Conditional Expressions

---

$F[\text{if } E1 \text{ then } E2 \text{ else } E3] k =$

$F[E1] \text{fn}_r(t) \Rightarrow \text{if } t \text{ then } F[E2] k$   
 $\text{else } F[E3] k$

$F[\text{if } E1 \text{ then } E2 \text{ else } E3] \text{fn}_r(y) \Rightarrow E' =$

$F[E1] \text{fn}_r(t) \Rightarrow \text{let } x = \text{fn}_j(y) = E'$   
 $\text{in if } t$   
 $\text{then } F[E2] x$   
 $\text{else } F[E3] x$

What is the rationale for this rule?

---

# Application

---

Call jump lambdas directly. Don't return via the continuation.

$F[x(y)] k = \text{let } v = x(y) \text{ in } z(v)$

if  $k = z$  and "let  $z = \text{fn\_j}(\dots) \Rightarrow \dots$ " in the translated program

$F[x(y)] k = x(y, k)$  otherwise

---

# Example

---

Count the number of times zero appears in a sequence

```
fun f(limit) =  
  loop | ( i = 0, c = 0) in  
    if i = limit  
      then c  
    else let x = f(i)  
          in let c' = if x = 0  
                    then c + 1  
                    else c  
          in | (i+1,c')
```

---

# Translation to CPS

---

```
let fun f(limit,k) =  
  let fun _j l(i,c) =  
    if i = limit  
    then k(c)  
    else f(i,fn _j(x) =>  
      let j = fn _j(c') => l(i+1,c')  
      in if x = 0  
        then j(c + 1)  
        else j(c)  
      in l(0,0)
```

---

# Translation

---

- Introduce 3 kinds of calls:
  - Tail: An application that is in the “return” position of an expression is a tail call. The continuation of the call is the continuation of the caller.
  - Tail calls do not build stack

# Translation

---

- Non-tail calls:
  - Given a procedure, argument, and a continuation label, evaluate the procedure with the continuation argument of the callee.
  - The call results in a new stack frame that holds the argument of the called procedure.

# Translation

---

- Return: given a return value, pop the current stack frame, bind the return value as the continuation argument, and evaluate the continuation expression.

# Translation

---

- Goto: Local transfer of control. Given a continuation label, and value, jump to the continuation, bind the value to the continuation argument.
- Only one kind of continuation
  - continuation variables serve to hold return values, and temporaries.

# Example

---

```
let fun f(x) = x in (f(13),f(true)) end
```

Compiled Expression:

-----

```
letrec f(x) = Return:(x) in Goto: L6 (13) end
```

:Continuations:

```
cont: (L6,V6) Goto: L5 (f)
cont: (L5,V5) NonTail: V5 (V6, L0)
cont: (L0,V0) Goto: L4 ((true))
cont: (L4,V4) Goto: L3 ((V4))
cont: (L3,V3) Goto: L2 (f)
cont: (L2,V2) NonTail: V2 (V3, L1)
cont: (L1,V1) Return:((V0, V1))
```

Program terminated with value: (13,(true))

# Example

---

```
let fun fact(n) = if n < 1 then 1 else n * fact(n-1) in fact(4) end
```

Compiled Expression:

-----

```
letrec fact(n) = Goto: L7 (n = 0) in Goto: L13 (4) end
```

:Continuations:

```
cont: (L13,V13) Goto: L12 (fact)
```

```
cont: (L12,V12) Tail: V12 (V13)
```

```
cont: (L7,V7) if V7 then Return:(1) else Goto: L8 (n)
```

```
cont: (L8,V8) Goto: L11 (n - 1)
```

```
cont: (L11,V11) Goto: L10 (fact)
```

```
cont: (L10,V10) NonTail: V10 (V11, L9)
```

```
cont: (L9,V9) Return:(V8 * V9)
```