Continuations and Compilation

CS 611
Lecture 22
Andrew Myers

Meta-language vs language

- Standard semantics: how to translate a rich language (w/ sophisticated state, control constructs) into a lazy, functional meta-language
- Meta-language ≈ ML ≈ F_0
- Can view our denotational semantics as rules for translating F_x into F_0
  - (caveat: strictness in target language avoided only by explicitly introducing closures/thunks)
  - Translation function T : Exp → Exp
  - Target subset of Exp has useful properties: written in *continuation-passing style*

CPS conversion

Translation of program E: T(E) (fn out out)

T(x) = (fn k (k x)) (*i.e.*, T(x)k = (k x))
T(n) = (fn k (k n))
T(binop e1 e2) =
  (fn k (T(e1) (fn v1 (T(e2) (fn v2 (k (binop v1 v2))))))
T(if e0 e1 e2) =
  (fn k (T(e0) (fn b (if b (T(e1) k) (T(e2) k)))))
T(fn x e) =
  (fn k (k (fn k' (fn x (T(e) k')))))
T(call e1 e2) =
  (fn k (T(e1) (fn f (T(e2) (fn v ((f k) v))))))

CPS conversion, again

Written in the form used for earlier semantics:

T(x) k = (k x)
T(n) k = (k n)
T(binop e1 e2) k =
  (T(e1) (fn v1 (T(e2) (fn v2 (k (binop v1 v2)))))
T(if e0 e1 e2) k =
  (T(e0) (fn b (if b (T(e1) k) (T(e2) k))))
T(fn x e) k =
  (k (fn k' (fn x (T(e) k'))))
T(call e1 e2) k =
  (T(e1) (fn f (T(e2) (fn v ((f k) v)))))

Exposing continuations

- The (callcc e_f) construct from the previous lecture can be converted too
  - calls function f passing current contin. as function
  - can be used to implement new control constructs (e.g. threads, exceptions)

T(callcc e) k = (T(e) (fn f ((f k) (fn k' k))))

CPS grammar

T(x) k = (k x)
T(n) k = (k n)
T(binop e1 e2) k = (T(e1) (fn v1 (T(e2) (fn v2 (k (binop v1 v2)))))
T(if e0 e1 e2) k = (T(e0) (fn b (if b (T(e1) k) (T(e2) k))))
T(fn x e) k = k (fn k' (fn x (T(e) k'))))
T(call e1 e2) k = (T(e1) (fn f (T(e2) (fn v ((f k) v)))))

- Can think of RHS as expressions in different language F_{CPS}
- CPS conversion turns an F_0 expression into an F_{CPS} command that sends the expression's result: T : Exp → Cmd_{CPS}
- Any expression generated by conversion has this grammar:

$$c \in Cmd_{CPS} ::= x\ e \mid c\ e \mid \text{if } x\ c_1\ c_2$$
$$e \in Exp_{CPS} ::= x \mid n \mid \text{fn } x\ c \mid \text{binop } x_1\ x_2$$

attributes of F_{CPS}

$c \in Cmd_{CPS} ::= x\ e \mid c\ e \mid \text{if } x\ c_1\ c_2$

$e \in Exp_{CPS} ::= x \mid n \mid \text{fn } x\ c \mid \text{binop } x_1\ x_2$

- Can't use function call as an expression
- Can't build up expression trees CPS vs. ordinary F_0:
- Order of evaluation becomes explicit (left to right)
- No anonymous intermediate expression results
- Functions never return (except at final k_0 call)
- No implicit pending computation ⇒ no need for an implicit stack to allow computations to resume

F_{CPS} and compilation

- F_{CPS} is universal
- Makes various low-level constructs explicit:
  - Can't use function call as an expression
  - Can't build up expression trees CPS vs. ordinary F_0:
  - Order of evaluation becomes explicit (left to right)
  - No anonymous intermediate expression results
  - Functions never return (except at final k_0 call)
  - No implicit pending computation ⇒ no need for an implicit stack to allow computations to resume
- These are also attributes of machine language!
- CPS code is good intermediate language

An F_0 compiler

- Assume a simple register machine with an unbounded number of registers
- Statements:

mov r_1, r_2	mov a, r_1 op r_2
if r then s_1 else s_2	jump r
mov a, { s_1; …; s_n }	l:

- Compiler: A[e,k] yields code for expression e that puts result in a, transfers control to k
- Can read rules for CPS conversion as rules for machine code generation

Compilation rules

T(x) k = (k x) A(x, k) = mov a, x; jump k

T(*n*) k = (k *n*) A(*n*, k) = mov a, n; jump k

T(binop e1 e2) k = (T(e1) (fn v1 (T(e2) (fn v2 (k (binop v1 v2)))))

A(binop e1 e2, k) =
T(e1)
mov v1, a
T(e2)
mov v2, a
mov a, v1 op v2
jump k

- For simplicity & to avoid extra jumps, define A(x, k) so that it doesn't generate the jump to k itself (under assumption that k immediately follows the code it generates).

Compilation rules

T(x) k = (k x) A(x, k) = mov a, x

T(*n*) k = (k *n*) A(*n*, k) = mov a, n

T(binop e1 e2) k = (T(e1) (fn v1 (T(e2) (fn v2 (k (binop v1 v2)))))

A(binop e1 e2, k) =
A(e1, l1); l1: mov v1, a;
A(e2, l2); l2: mov v2, a;
mov a, v1 op v2

T(if e0 e1 e2) k = (T(e0) (fn b (if b (T(e1) k) (T(e2) k))))

A(if e0 e1 e2, k) =
A(e0,l0); l0: mov b, a;
if b then A(e1, k); jump k else A(e2, k)

T(call e1 e2) k = (T(e1) (fn f (T(e2) (fn v ((f k) v)))))

A(call e1 e2, k) =
A(e1,l1); l1: mov f, a;
A(e2, l2); l2: mov v, a;
mov ra, k; mov a, v; jump f

Functions and cwcc

T(fn x e) k = k (fn k' (fn x (T(e) k'))))

A(fn x e, k) = mov a, { mov k', ra; mov x, a; A(e, k'); jump k' }

T(callcc e) k = (T(e) (fn f ((f k) (fn k' k))))

A(callcc e, k) =
A(e, l1); mov f, a;
mov ra, k
mov a, { mov k', ra; jump k }
jump f

Oops: DCG

- Instructions of the form mov a, { s_1; ...; s_n } imply dynamic code generation if any s_i mention free variables from containing context
- Problem: F_{CPS} allows unrealistically powerful lexical scoping construct
- Optimization: rather than generating code for whole function every time the expression { s_1; ...; s_n } is encountered, only change bindings of free variables.
- Function represented by pair f = (L, ρ): a closure of code L with free variable bindings (environment) ρ
- Variables free in each piece of code are extracted from the accompanying environment tl(f) : static link
- Jump to function goes to hd(f) : function address

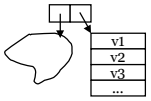
Using closures instead

k (fn k' (fn x (T(e) k'))))

A(fn x e, k) = mov a, (L, FVE(x,e))

L: mov k', ra; mov x, a; A(e, k'); jump k'

FVE(x,e) = let {v1, v2,...} = FV(e) - {x} in ['v1'->v1,'v2'->v2,...]



(T(e) (fn f ((f k) (fn k' k))))

A(cwcc e, k) = A(e, l0); l0: mov f, a; mov ra, k; mov a, (L, [k->k]); jump f

L: mov k', ra; jump k

Summary

- CPS conversion converts a programming language into continuation-passing style
- Style makes order of evaluation, intermediate results, control flow explicit
- CPS conversion corresponds fairly closely to compilation to a register machine