Continuations and Control

Gul Agha
CS 421

Imperative Version of fact-iter

(define n-reg *)
(define a-reg *)
(define fact-iter
  (lambda (n)
    (set! n-reg n)
    (set! a-reg 1)
    (fact-iter-acc/reg)))

Imperative version (contd.)

(define fact-iter-acc/reg
  (lambda ( )
    (if (zero? n-reg)
      a-reg
      (begin (set! a-reg (* n-reg a-reg))
        (set! n-reg (– n-reg 1))
        (fact-iter-acc/reg)))))

Imperative Transform on CPS style code

• Additional complication:
  What environment will the continuation be evaluated in?

CPS form of Remove

(define remove
  (lambda (s  los  k)
    (if (null? los)
      (k '( ))
      (if (eq? s (car los))
        (remove s (cdr los) k)
        (remove s (cdr los)
          (lambda (v)
            (k (cons (car los) v)))))
    ))

Imperative Transform on remove-cps

(define s-reg *)
(define los-reg *)
(define k-reg *)
(define remove
  (lambda (s  los  k)
    (set! s-reg s)
    (set! los-reg los)
    (set! k-reg k)
    (remove/reg)))
(define remove/reg
  (lambda ()
    (if (null? los-reg)
        (k '( ))
        (if (eq? s-reg (car los-reg))
            (begin
              (set! los-reg (cdr los-reg))
              (remove/reg))
            (begin
              (set! k (lambda (v)
                          (k-reg (cons (car los-reg) v))))
              (set! los-reg (cdr los-reg))
              (remove/reg))))

Environment

- Continuations must save the bindings in (part of) the current environment that they need.
- Linked list of continuations

Wrong!! Why?

What cps does not achieve?

- Time complexity class of code is unchanged.
- Time or space requirement may change only by a constant factor.

Fibonacci

(define fib
  (lambda (n)
    (if (or (= n 0) (= n 1))
        n
        (+ (fib (– n 1)) (fib (– n 2))))))

(define fib
  (lambda (n k)
    (if (or (= n 0) (= n 1))
        (k n)
        (fib (– n 1)
             (lambda (v) (fib (– n 2)
                         (lambda (w) k (+ v w)))))))))

Exercise: convert to cps form!

Linear Recursive Fib

(define fib* (lambda (n)
               (cond
                 ((= n 0) (cons 0 0))
                 ((= n 1) (cons 0 1))
                 (else (let ((x (fib* (– n 1)))
                             (y (car x))
                             (z (cdr x))
                             (k (lambda (v) (+ v (fib (– n 2))))))
                          (cons (+ (car x) (cdr x))
                                (k x)))))))

CPS Fib

(define fib
  (lambda (n k)
    (if (or (= n 0) (= n 1))
        (k n)
        (fib (– n 1)
             (lambda (v)
                          (fib (– n 2)
                               (lambda (w) k (+ v w))))))))

Still exponential time!
First-class Continuations

- Continuations may be implicit or explicit in the interpreter.
- CPS interpreter carries explicit continuations.
- Scheme makes these continuations available to programmer.
  - call/cc stands for call with current continuation.
- Useful for exception handling and explicit control

Call/cc

- Packages up the current continuation
- Passes it as an argument
- Recall
  - Continuations are procedures which take one argument

Example

(call-with-current-continuation
  (lambda (exit)
    (for-each
      (lambda (x)
        (if (negative? x) (exit x)))
      '(54 0 37 -3 245 19))
  #'t))

==> -3

Breakpoint facility

1. Capture the top-level read-eval-print loop:
   (call/cc (lambda (x) (set! top x)))
2. Define break point:
   (define break
    (lambda (x)
      (call/cc (lambda (k)
          (begin (set! resume k)
              (writeln "Interrupt!
                Use (resume) to return"
          (top x) ))))))
3. Debug then use resume to restart computation.

Example Escape

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r (lambda (obj)
                      (cond ((null? obj) 0)
                             ((pair? obj)
                             (+ (r (cdr obj)) 1))
                             (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))

==> 4

(list-length '(a b . c))

==> #f
Co-routines

- Co-routines provide a symmetric control model.
  - Form of concurrency.
- Used in simulations and games (Simula)
  - Co-routine for each player or object.
  - Master unit calls one co-routine.
  - Co-routines yield to each other.

(define (make-coroutine
  (lambda (co-body)
    (letrec ((state (lambda ( ) (co-body resume)))
               (resume (lambda (that)
                           (call/cc (lambda (this)
                                      (set! state this)
                                      (that)))))
               (lambda ( )
                   (state))))))

Example Co-routine

(define A
  (make-coroutine
    (lambda (resume)
      (….)
      (resume B)
      (….)
      (resume B)
      (….)
      'finito)))

(define B
  (make-coroutine
    (lambda (resume)
      (….)
      (resume A)
      (….)
      (resume A)
      (….)
      'finito)))

Binding current continuation

(let/cc  k  body)

Is a macro for:

(call/cc (lambda (k)  body))

- Can there be more than one continuation stored?
- What is the extent of a continuation?
- What is (call/cc call/cc)?

What does this do?

(let ((cont  #f))
  (call/cc
    (lambda (k)
      (set!  cont  k)))
  (cont #f))

- Initial value of cont unimportant
- Current continuation is bound to k
- cont is bound to k
- cont is invoked with an unimportant value
- We have a time warp

Lexically Scoped Exceptions

- Also called block/break exceptions.
- Lexically surrounding block must catch exception.
- A structured form of labels and gotos
- Examples:
  - return, break, continue in C have single point of return.
  - Labeled break in Java.
True Exceptions

Also called catch/throw exceptions
(define find-symbol
 (lambda (id tree)
   (letrec ((find (lambda (tree)
                   (if (pair? tree)
                    (or (find (car tree))
                        (find (cdr tree)))
                    (if (eq? tree id)
                        (throw 'find t)
                        f)))))
   (catch 'find (find tree)) )
}

Escape in Common Lisp
(defun fact (n)
 (prog (r)
    (setq r 1)
    loop (cond ((= n 1) (return r))
                 (setq r (* n r))
                 (setq n (- n 1))
                 (go loop) )
   )

Escape from Nested Context in Common Lisp
(defun fact2 (n)
 (prog (r)
    (setq r 1)
    loop (setq r (* (cond ((= n 1) (return r))
                        ('else n) )
              r )
    (setq n (- n 1))
    (go loop) )
   )

Continuations with Indefinite Extent
(define (fact n)
 (let ((r 1)  (k   'void))
    (call/cc  (lambda (c)  (set!  k  c)  'void))
    (set!  r   (* r  n))
    (set!  n   (- n  1))
    (if  (= n 1)
        r
        (k   'recurse)) )

Continuations with Indefinite Extent
(define (fact n)
 (let ((r 1))
    (let ((k  (call/cc (lambda (c)
                        c))))
     (set!  r   (* r  n))
     (set!  n   (- n  1))
     (if  (= n 1)
         r
         (k   'recurse)) )
   )