

# Continuations & CPS

Dr. Greg Lavender  
Department of Computer Sciences  
University of Texas at Austin

## What is a “Continuation”?

- A semantic concept used in denotational semantics to express generalized jumps
  - need to semantically express a transfer of control out of a command sequence
    - If  $C_2$  is a jump out of a command sequence, what is the semantic meaning of  $\mathcal{A}[[C_1; C_2; \dots; C_n; ]]$  ?
- A continuation “k” is a function that represents the “rest of the computation” following the current point of control
  - continuations are most often implicit
  - we can program with them explicitly if “first-class”

2/7/05

2

## How are continuations used?

- non-local jumps
- exception handling
- continuation passing style (CPS)
- lightweight processes in thread packages
- stack unwinding in parsers
- callbacks for event handlers
- backtracking in search and theorem proving
- breakpoints in a debugger

2/7/05

3

## A Simple “While” Language

### Syntactic Categories:

$L \in Id$	Identifiers
$C \in Cmd$	Commands
$E \in Exp$	Expressions
$F \in Fn$	Some Primitive Commands

### Command Syntax:

$C ::= F \mid \mathbf{dummy} \mid$   
 $C_0; C_1 \mid E \rightarrow C_0, C_1 \mid \mathbf{while} E \mathbf{do} C \mid$   
 $\mathbf{goto} E \mid \mathbf{begin} L_0; C_0; L_1; C_1; \dots L_{n-1}; C_{n-1} \mathbf{end} \mid$   
 $\mathbf{resultis} E$

### Expression Syntax:

$E ::= L \mid \mathbf{true} \mid \mathbf{false} \mid E_0 \rightarrow E_1, E_2 \mid \mathbf{valof} C$

2/7/05

4

## While Semantics

- Domain  $\mathbf{T}$  of truth values
- Domain  $\mathbf{S}$  of states or ‘stores’
  - $\sigma, \sigma' \in \mathbf{S}$
  - $\mathbf{S} \rightarrow \mathbf{S}$  denotes a store transform
- Domain  $\mathbf{C}$  of commands
- Domain  $\mathbf{D}$  of denotations
  - $\varepsilon \in \text{Env} = [\text{Id} \rightarrow \mathbf{D}]$
  - $\varepsilon$  stands for an “environment” which is a semantic function that maps identifiers to their meaning

2/7/05

5

## Jumps

- A difficulty arises in dealing denotationally with sequence of commands with jumps
- Normally, the denotation of a command sequence is a nice clean composition of jump-free store transforms
  - $\mathcal{A}[[C_1; C_2; \dots; C_n]](\varepsilon) =$   
 $[[C_n]](\varepsilon) \circ [[C_{n-1}]](\varepsilon) \circ \dots \circ [[C_1]](\varepsilon)$
  - but what is the meaning if some  $C_i$  is a **goto** command that jumps out of this sequence?

2/7/05

6

## Command Continuation

- Define for each command  $C_i$  a function that transforms the store from the state of the store at that command to the end of the program, i.e., the “rest of the computation”
  - $\mathcal{P}[[C]](\varepsilon)(\kappa) = \kappa \circ [[C]](\varepsilon)$ 
    - we call  $\kappa$  the (*command*) *continuation*
- For jump-free commands the “normal” continuation does what you expect, but for jump-dependent commands, it is thrown away
  - $\mathcal{P}[[\text{goto } E]](\varepsilon)(\kappa)$  will ignore  $\kappa$  and use the continuation obtained from  $E$

2/7/05

7

## The Continuation Function

- The function  $\kappa$  takes any state as an argument and transforms the store to the final state of the program
  - $\sigma_n = \kappa(\sigma_i)$
- What about expressions?
  - we can simply pass the value of an expression as an additional argument to the continuation function and that value will be bound in the environment of the new command that is the target of the continuation

2/7/05

8

## Setjmp/longjmp in C

- Primitive machine-specific library calls
  - Implement continuation capture and non-local jumps. Must capture the current stack activation frame context, program counter, registers, cc, etc.,

```
static jmp_buf k;

void foo(...) { ...; longjmp(k,1); ...; return; }
int main() {
if (setjmp(k) == 0) // set current continuation
    foo(...);
else // setjmp returns 1 from longjmp
    ... // handle non-local return
```

2/7/05

9

## Exceptions in C++

- Defined as part of the language
  - more intuitive syntax
  - safer than setjmp/longjmp
  - ability to transfer data objects with non-local jump
  - proper stack unwinding
    - but can still have memory leaks of pointers into heap

```
void foo(...) { ...; throw Exception(); ...; }
...
try { foo(...); } // like setjmp==0
catch (Exception e) { ... } // like setjmp==1
```

2/7/05

10

## Continuations in Scheme

- In Scheme, continuations are “first-class” objects that can be used explicitly in programs
- call-with-current-continuation
  - (*call/cc procedure*)
  - calls *procedure* with the current continuation at the point of the call/cc within some expression

2/7/05

11

## Continuation Points

- Consider a conditional
  - (if (null? x) '() (cdr x))
  - first we evaluate the condition (null? x) then decide what is the “next computation”
  - the “next computation” is the continuation
  - so at any point in a program there are many continuation points,  $k_0, k_1, \dots, k_n$  each of which represents the “rest of the program” beyond the current point in the execution
  - the “current” continuation is some  $k_i$  at the current point of execution

2/7/05

12

## Call/cc example

- Call/cc constructs the current continuation and passes it as the procedure argument `k`
  - the current continuation is itself represented by a procedure, which if applied to a value passes that value to the continuation

```
(+ 2 (call/cc (lambda (k) (* 5 4)))) => 22
```

```
(+ 2 (call/cc (lambda (k) (* 5 (k 4)))) => 6
```

2/7/05

13

## Naïve Recursion

- Recursive control flow
  - requires  $O(n)$  stack space
  - requires execution time overhead for procedure call & activation frame mgmt

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

2/7/05

14

# Tail Recursion

- Tail call optimization
  - tail call position
    - eliminate all recursive calls in an operand position by moving them to an operator position
    - allows a recursive definition but results in iterative execution
    - converts  $O(n)$  stack space usage to  $O(1)$  stack space
    - execution time savings since no procedure call overhead

```
(define (tfact x)
  (let fact ((n x) (r 1))
    (if (zero? n)
        r
        (fact (- n 1) (* r n)))))
```

2/7/05

15

# Continuation Passing Style

- CPS transform
  - use a continuation to “carry forward” future execution without having to “return”
    - no need to maintain a return context
  - can be used to turn a double recursion into tail recursion
    - quicksort is doubly recursive so not amenable to typical tail call optimization
    - but we can use a continuation instead

2/7/05

16

# CPS Factorial

```
(define cps-fact
  (lambda (n k) ;; k is the continuation
    (if (zero? n)
        (k 1)
        (cps-fact (- n 1) ;; tail call
                   (lambda (r) (k (* n r)) ;new k
                       )))))
```

```
(define (cfact n)
  (cps-fact n (lambda (x) x) ))
```

2/7/05

17

# CPS Factorial Evaluation

```
;; for n = 1
((lambda (r)
  ((lambda (x) x)
   (* 1 r)))
 1)
```

```
;; for n = 2
((lambda (r)
  (lambda (r)
    ((lambda (x) x)
     (* 2 r)))
   (* 1 r)))
 1)
```

```
;; for n = 3
((lambda (r)
  ((lambda (r)
    ((lambda (r)
      ((lambda (x) x)
       (* 3 r)))
     (* 2 r)))
   (* 1 r)))
 1)
```

```
;; and so on for
;; n = 4, 5, ...
```

2/7/05

18

## Recursive Quicksort

```
(define partition
  (lambda (op pivot ls)
    (filter (lambda (e) (op e pivot)) ls)))
```

```
(define quicksort
  (lambda (ls)
    (if (null? ls)
        '()
        (let* ((pivot (car ls))
               (rest (cdr ls))
               (left (partition < pivot rest))
               (right (partition >= pivot rest)))
            (append (quicksort left)
                    (list pivot)
                    (quicksort right))))))
```

2/7/05

19

## CPS Quicksort

```
(define qsort
  (lambda (ls)
    (if (null? ls)
        '()
        (let ((pivot (car ls)))
            (cps-qsort pivot
                      (cdr ls)
                      (lambda (left right)
                        (append (qsort left)
                                (list pivot)
                                (qsort right))))
            ))))
```

2/7/05

20

# CPS Quicksort

```
(define cps-qsort
  (lambda (pivot ls k) ;; k is the continuation
    (if (null? ls)
        (k '() '())
        (cps-qsort pivot (cdr ls) ;; tail call
                    (lambda (left right)
                      (let ((x (car ls)))
                        (if (< x pivot)
                            (k (cons x left) right)
                            (k left (cons x right))))))))))
```

2/7/05

21

# For Further Study

1. Peter Landin, "A Generalization of Jumps and Labels"  
– <http://www.kluweronline.com/issuetoc.htm/1388-3690+11+2+1998>
2. Christopher Strachey & Christopher P. Wadsworth,  
"Continuations: A Mathematical Semantics for Handling Full  
Jumps"  
– <http://www.kluweronline.com/issuetoc.htm/1388-3690+11+2+1998>
3. Michael J. Fischer, "Lambda-calculus Schemata"  
– <http://www.cs.yale.edu/homes/fischer/pubs/lambda.pdf>
4. John C. Reynolds, "The Discoveries of Continuations"  
– <http://www.brics.dk/~hosc/local/LaSC-6-34-pp233-248.pdf>
5. ReadScheme.org, Bibliography of Continuations and Continuation  
Passing Style  
– <http://library.readscheme.org/page6.html>

2/7/05

22