CPS

T. METIN SEZGIN

Recursive vs. Iterative Control Behavior

- Consider

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

- The trace

```
   (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

Recursive vs. Iterative Control Behavior

- Consider

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

- The trace

```
  (fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

What is the key difference between the two versions?

- What do we do after each call?
- How does the control context grow?
- Continuation:
  - Captures the control context
  - Describes what needs to be done next!

A CPS Interpreter

- The environment grows as we evaluate expressions
- Now we need to keep around a list of things to do after the evaluation of each expression.
- Introduce apply-cont
  - Example:

FinalAnswer = *ExpVal*
apply-cont : *Cont* × *ExpVal* → *FinalAnswer*

```
(apply-cont (end-cont) val)
= (begin
    (eopl:printf "End of computation.~%")
    val)
```

Evaluation

- Value-of-program

value-of-program : *Program* → *FinalAnswer*

```
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of/k exp1 (init-env) (end-cont))))))
```

- Value-of/k

value-of/k : *Exp* × *Env* × *Cont* → *FinalAnswer*

```
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val (procedure var body env))))
      ...)))
```

Evaluation

- Letrec

```
(letrec-exp (p-name b-var p-body letrec-body)
  (value-of/k letrec-body
    (extend-env-rec p-name b-var p-body env)
    cont))
```

- Zero?

```
(zero?-exp (exp1)
  (value-of/k exp1 env
    (zero1-cont cont)))
```

```
(apply-cont (zero1-cont cont) val)
= (apply-cont cont
    (bool-val
      (zero? (expval->num val))))
```

Evaluation

- Let
  - Before

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))
```

  - After

```
(let-exp (var exp1 body)
  (value-of/k exp1 env
    (let-exp-cont var body env cont)))
```

```
(apply-cont (let-exp-cont var body env cont) val)
= (value-of/k body (extend-env var val env) cont)
```

Evaluation

- If

```
(if-exp (exp1 exp2 exp3)
  (value-of/k exp1 env
    (if-test-cont exp2 exp3 env cont)))
```

```
(apply-cont (if-test-cont exp2 exp3 env cont) val)
= (if (expval->bool val)
    (value-of/k exp2 env cont)
    (value-of/k exp3 env cont))
```

Example

```
(value-of/k <<letrec p(x) = x in if b then 3 else 4>>
   ρ0  cont0)
=  letting ρ1 be (extend-env-rec ... ρ0)
(value-of/k <<if b then 3 else 4>> ρ1  cont0)
=  next, evaluate the test expression
(value-of/k <<b>> ρ1  (test-cont <<3>> <<4>> ρ1  cont0))
=  send the value of b to the continuation
(apply-cont  (test-cont <<3>> <<4>> ρ1  cont0)
             (bool-val #t))
=  evaluate the then-expression
(value-of/k <<3>> ρ1  cont0)
=  send the value of the expression to the continuation
(apply-cont cont0  (num-val 3))
=  invoke the final continuation with the final answer
(begin (eopl:printf ...) (num-val 3))
```

Evaluation

- diff

```
(diff-exp (exp1 exp2)
  (value-of/k exp1 env
    (diff1-cont exp2 env cont)))
```

```
(apply-cont (diff1-cont exp2 env cont) val1)
= (value-of/k exp2 env
    (diff2-cont val1 cont))
```

```
(apply-cont (diff2-cont val1 cont) val2)
= (let ((num1 (expval->num val1))
        (num2 (expval->num val2)))
    (apply-cont cont
      (num-val (- num1 num2))))
```

Example

```
(value-of/k
  <<-(-(44,11),3)>>
  ρ0
   #(struct:end-cont))
=  start working on first operand
(value-of/k
  <<-(44,11)>>
  ρ0
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont)))
=  start working on first operand
(value-of/k
  <<44>>
  ρ0
  #(struct:diff1-cont <<11>> ρ0
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
=  send value of <<44>> to continuation
(apply-cont
  #(struct:diff1-cont <<11>> ρ0
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont)))
  (num-val 44))
=  now start working on second operand
(value-of/k
  <<11>>
  ρ0
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont))))
=  send value to continuation
(apply-cont
  #(struct:diff2-cont (num-val 44)
     #(struct:diff1-cont <<3>> ρ0
        #(struct:end-cont)))
  (num-val 11))
=  44 − 11 is 33, send that to the continuation
(apply-cont
  #(struct:diff1-cont <<3>> ρ0
     #(struct:end-cont))
  (num-val 33))
=  start working on second operand <<3>>
(value-of/k
  <<3>>
  ρ0
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont)))
=  send value to continuation
(apply-cont
  #(struct:diff2-cont (num-val 33)
     #(struct:end-cont))
  (num-val 3))
=  33 − 3 is 30, send that to the continuation
(apply-cont
  #(struct:end-cont)
  (num-val 30))
```

Evaluation

- Procedure application
  - Before

```
(call-exp (rator rand)
  (let ((proc1 (expval->proc (value-of rator env)))
        (val (value-of rand env)))
    (apply-procedure proc1 val)))
```

  - After

```
(call-exp (rator rand)
  (value-of/k rator env
    (rator-cont rand env cont)))
```

```
(apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env
    (rand-cont val1 cont))
```

```
(apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
    (apply-procedure/k proc1 val2 cont))
```

apply-procedure/k : $Proc \times ExpVal \times Cont \rightarrow FinalAnswer$

```
(define apply-procedure/k
  (lambda (proc1 val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body
          (extend-env var val saved-env)
          cont)))))
```