CS 565 August 25, 1996
Jens Palsberg Readings: EOPL, Secs 9.1–2, 10.1–3

Supplement to Lecture Note 6:

More Continuations

Converting the Interpreter to CPS

Source interpreter:

```
(define eval-exp
  (lambda (exp env)
    (record-case exp
      (lit-exp (constant) constant)
      (var-exp (id) (apply-env env id))
      (proc-exp (formals body)
        (make-closure formals body env))
      (if-exp (test-exp exp1 exp2)
        (if (true-value? (eval-exp test-exp env))
          (eval-exp exp1 env)
          (eval-exp exp2 env)))
      (app-exp (rator rands)
        (let ((proc (eval-exp rator env))
              (args (eval-rands rands env)))
          (apply-proc proc args)))
      (else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))
```

```
(define eval-rands
  (lambda (rands env)
    (if (null? rands)
      '()
      (cons (eval-exp (car rands) env)
            (eval-rands (cdr rands) env)))))
```

```
(define apply-proc
  (lambda (proc args)
    (record-case proc
      (primitive-proc (primop) (apply-primop primop args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error 'apply-proc "Bad Procedure: ~s" proc)))))
```

The CPS Interpreter

```
(define eval-exp
  (lambda (exp env k)
    (record-case exp
      (lit-exp (constant) (k constant))
      (var-exp (id) (k (apply-env env id)))
      (proc-exp (formals body)
        (k (make-closure formals body env)))
      (if-exp (test-exp exp1 exp2)
        (eval-exp test-exp env
          (lambda (v)
            (if (true-value? v)
              (eval-exp exp1 env k)
              (eval-exp exp2 env k)))))
      (app-exp (rator rands)
        (eval-exp rator env
          (lambda (proc)
            (eval-rands rands env
              (lambda (args)
                (apply-proc proc args k))))))
      (else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))
```

```
(define eval-rands
  (lambda (rands env k)
    (if (null? rands)
      (k '())
      (eval-exp (car rands) env
        (lambda (first-arg)
          (eval-rands (cdr rands) env
            (lambda (other-args)
              (k (cons first-args other-args))))))))))
```

```
(define apply-proc
  (lambda (proc args k)
    (record-case proc
      (primitive-proc (primop) (k (apply-primop primop args)))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env) k))
      (else (error 'apply-proc "Bad Procedure: ~s" proc)))))
```

Restoring Representation Independence: Continuation Builders

```
; the initial continuation
(lambda (val) val)

; continuation for tests
(lambda (v)
  (if (true-value? v)
      (eval-exp exp1 env k)
      (eval-exp exp2 env k)))

; continuation for evaluation of operator
(lambda (proc)
  (eval-rands rands env
    (lambda (args)
      (apply-proc proc args k))))

; continuation for evaluation of an argument
(lambda (first-arg)
  (eval-rands (cdr rands) env
    (lambda (other-args)
      (k (cons first-args other-args)))))

; nested continuations
(lambda (args) (apply-proc proc args k))

(lambda (other-args)
  (k (cons first-args other-args)))
```

Procedural Representation of Continuations

```
(define make-final-value-cont
  (lambda ()
    (lambda (val) val)))

(define make-test-value-cont
  (lambda (exp1 exp2 env k)
    (lambda (v)
      (if (true-value? v)
        (eval-exp exp1 env k)
        (eval-exp exp2 env k)))))

(define make-proc-value-cont
  (lambda (rands env k)
    (lambda (proc)
      (eval-rands rands env
        (make-args-cont proc k)))))

(define make-args-cont
  (lambda (proc k)
    (lambda (args)
      (apply-proc proc args k))))
```

```
(define make-first-arg-cont
  (lambda (rands env k)
    (lambda (first-arg)
      (eval-rands (cdr rands) env
        (make-other-args-cont first-arg k)))))

(define make-other-args-cont
  (lambda (first-arg k)
    (lambda (other-args)
      (apply-continuation k
        (cons first-arg other-args)))))

(define apply-continuation
  (lambda (k val) (k val)))
```

Representation-Independent Interpreter

```
(define eval-exp
  (lambda (exp env k)
    (record-case exp
      (lit-exp (constant) (apply-continuation k constant))
      (var-exp (id) (apply-continuation k (apply-env env id)))
      (proc-exp (formals body)
        (apply-continuation k (make-closure formals body env)))
      (if-exp (test-exp exp1 exp2)
        (eval-exp test-exp env
          (make-test-value-cont exp1 exp2 env k)))
      (app-exp (rator rands)
        (eval-exp rator env
          (make-proc-value-cont rands env k)))
      (else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))
```

```
(define eval-rands
  (lambda (rands env k)
    (if (null? rands)
      (apply-continuation k '())
      (eval-exp (car rands) env
        (make-first-arg-cont rands env k)))))

(define apply-proc
  (lambda (proc args k)
    (record-case proc
      (primitive-proc (primop)
        (apply-continuation k (apply-primop primop args)))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env) k))
      (else (error 'apply-proc "Bad Procedure: ~s" proc)))))
```

Representing Continuations as Data Structures

We'll use list-of-frames representation with endmarker.

```
⟨cont⟩ ::=  ((exit-frame)) | (⟨frame⟩ . ⟨cont⟩)
⟨frame⟩ ::=(test-value-cont-frame ⟨expression⟩ ⟨expression⟩
                ⟨environment⟩)
  |        (proc-value-cont-frame ⟨expression-list⟩ ⟨environment⟩)
  |        (args-cont-frame ⟨value⟩)
  |        (first-arg-cont-frame ⟨expression-list⟩ ⟨environment⟩)
  |        (other-args-cont-frame ⟨value⟩)
```

```
(define-record exit-frame ())
(define-record test-value-cont-frame (exp1 exp2 env))
(define-record proc-value-cont-frame (rands env))
(define-record args-cont-frame (proc))
(define-record first-arg-cont-frame (rands env))
(define-record other-args-cont-frame (first-arg))
```

```
(define make-final-value-cont
  (lambda ()
    (list (make-exit-frame))))

(define make-test-value-cont
  (lambda (exp1 exp2 env k)
    (cons (make-test-value-cont-frame exp1 exp2 env) k)))

(define make-proc-value-cont
  (lambda (rands env k)
    (cons (make-proc-value-cont-frame rands env) k)))
```

```
(define make-args-cont
  (lambda (proc k)
    (cons (make-args-cont-frame proc) k)))

(define make-first-arg-cont
  (lambda (rands env k)
    (cons (make-first-arg-cont-frame rands env) k)))

(define make-other-args-cont
  (lambda (first-arg k)
    (cons (make-other-args-cont-frame first-arg) k)))
```

```
(define apply-continuation
  (lambda (k val)
    (let ((frame (car k))(k (cdr k)))
      (record-case frame
        (exit-frame () val)
        (test-value-cont-frame (exp1 exp2 env)
          (let ((v val))
            (if (true-value? v)
              (eval-exp exp1 env k)
              (eval-exp exp2 env k))))
        (proc-value-cont-frame (rands env)
          (let ((proc val))
            (eval-rands rands env (make-args-cont proc k))))
        (args-cont-frame (proc)
          (let ((args val))
            (apply-proc proc args k)))
        (first-arg-cont-frame (rands env)
          (let ((first-arg val))
            (eval-rands (cdr rands) env
              (make-other-args-cont first-arg k))))
        (other-args-cont-frame (first-arg)
          (let ((other-args val))
            (apply-continuation k (cons first-arg other-args))))
        (else (error 'apply-cont
                "Bad Continuation Frame: ~s" frame))))))
```

Imperative Interpreter

```
(define eval-toplevel
  (lambda (init-exp init-env)
    (set! exp init-exp)
    (set! env init-env)
    (set! k (make-final-value-cont))
    (eval-exp)))

(define eval-exp
  (lambda ()              ; uses exp, env, k
    (record-case exp

      (lit-exp (constant)
        ;; (apply-continuation k constant)
        (set! val constant)
        (apply-continuation))

      (var-exp (id)
        ;; (apply-continuation k (apply-env env id))
        (set! val (apply-env env id))
        (apply-continuation))
```

```
(proc-exp (formals body)
  ;; (apply-continuation k (make-closure formals body env))
  (set! val (make-closure formals body env))
  (apply-continuation))

(if-exp (test-exp exp1 exp2)
  ;; (eval-exp test-exp env
  ;;    (make-test-value-cont exp1 exp2 env k))
  (set! exp test-exp)
  (set! k
    (make-test-value-cont exp1 exp2 env  k))
  (eval-exp))

(app-exp (rator rands)
  ;; (eval-exp rator env
  ;;    (make-proc-value-cont rands env k))
  (set! exp rator)
  (set! k (make-proc-value-cont rands env k))
  (eval-exp))

(else (error 'eval-exp "Bad abstract syntax: ~s" exp)))))
```

```
(define eval-rands
  (lambda ()                             ; uses rands, env, k
    (if (null? rands)
      (begin
        ;; (apply-continuation k '())
        (set! val '())
        (apply-continuation))
      (begin
        ;; (eval-exp (car rands) env
        ;;   (make-first-arg-cont rands env k))
        (set! exp (car rands))
        (set! k (make-first-arg-cont rands env k))
        (eval-exp)))))

(define apply-proc
  (lambda ()                             ; uses proc, args, k
    (record-case proc
      (primitive-proc (primop)
        ;; (apply-continuation k (apply-primop primop args))
        (set! val (apply-primop primop args))
        (apply-continuation))
      (closure (formals body env1) ; need local name for env
        ;; (eval-exp body (extend-env formals args env) k)
        (set! exp body)
        (set! env (extend-env formals args env1))
        (eval-exp))
      (else (error 'apply-proc "Bad Procedure: ~s" proc)))))
```

```
(define apply-continuation
  (lambda ()                                   ; uses k, val
    (let ((frame (car k)))
      (set! k (cdr k))
      (record-case frame
        (exit-frame () val)

        (test-value-cont-frame (exp1 exp2 env1)
          ;; (let ((v val))
          ;;  (if (true-value? v)
          ;;    (eval-exp exp1 env k)
          ;;    (eval-exp exp2 env k)))
            (if (true-value? val)
              (set! exp exp1)
              (set! exp exp2))
            (set! env env1)
            (eval-exp))

        (proc-value-cont-frame (rands env1)
          (let ((proc val))
            ;; (eval-rands rands env
            ;;   (make-args-cont proc k))
            (set! rands rands)
            (set! env env1)
            (set! k (make-args-cont proc k))
            (eval-rands)))
```

```
(args-cont-frame (proc)
  (let ((args val))
    ;; (apply-proc proc args k)
    (set! proc proc)
    (set! args args)
    (apply-proc)))

(first-arg-cont-frame (rands env1)
  (let ((first-arg val))
    ;;   (eval-rands (cdr rands) env
    ;;    (make-other-args-cont first-arg k))
    (set! rands (cdr rands))
    (set! env env1)
    (set! k (make-other-args-cont first-arg k))
    (eval-rands)))

(other-args-cont-frame (first-arg)
  (let ((other-args val))
    ;;   (apply-continuation k
    ;;     (cons first-arg other-args))
    (set! val (cons first-arg other-args))
    (apply-continuation)))

(else
  (error 'apply-cont
    "Bad Continuation Frame: ~s" frame))))))
```