Continuation-Passing Interpreters

Gul Agha
CS 421 Spring 2006

Representing and Manipulating Control Contexts

• Let interpreter explicitly pass continuations
• Add language facilities to manipulate continuations:
  – Exceptions
  – First-class continuations
  – Threads
• Add a new parameter to eval-expression
  – expression, environment, continuation

Review

• Environment passing interpreters
  – Simple expressions
  – Recursive definitions
  – Conditional expressions
  – Primitive applications
  – Procedure applications
  – Let expressions
  – Assignments

Environment Passing Interpreter

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum )
      (var-exp (id) ..)
      (proc-exp (id body) ...
        (letrec-exp (proc-names idss bodies letrec-body) ..)
        (if-exp (test-exp true-exp false-exp) ...
          (primapp-exp (prim rands) ...) ..
          (app-exp (rator rands) ...) ...
          (let-exp (ids rands body) ...) ...
          (varassign-exp (id rhs-exp) ....)))))

Evaluating simple expressions

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum )
      (var-exp (id) (apply-env env id))
      (proc-exp (ids body)
        (closure ids body env))))

Evaluating Recursive Definitions

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (letrec-exp (proc-names idss bodies letrec-body) ...
        (eval-expression letrec-body
          (extend-env-recursively
            proc-names idss bodies env)))
      (app-exp (rator rands) ...) ...
      (let-exp (ids rands body) ...) ...
      (varassign-exp (id rhs-exp) ....))))
Evaluating Conditional Expressions

(if-exp (test-exp true-exp false-exp)
  (if (true-value?
       (eval-expression test-exp env))
       (eval-expression true-exp env)
       (eval-expression false-exp env)))

Evaluating Primitive Applications

(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args))))))

Evaluating Procedure Applications

(app-exp (rator rands)
  (let ((proc (eval-expression rator env))
         (args (eval-rands rands env)))
   (if (procval? proc)
       (apply-procval proc args)
       (eopl:error 'eval-expression
                    "Attempt to apply non-procedure ~s" proc))))

Evaluating Let Expressions

• Retain call-by-value behavior for let expressions

(let-exp (ids rands body)
  (let ((args (eval-let-exp-rands rands env)))
    (eval-expression body
                     (extend-env ids args env))))

Evaluating Assignments

(varassign-exp (id rhs-exp)
    (begin
      (setref!
       (apply-env-ref env id)
       (eval-expression rhs-exp env))
    1))

Continuation Passing Interpreter

• Add continuations
• Pass continuation to each expression
• What about top level continuation?
  – Read-eval-print loop
  – Introduce a continuation to which the result of evaluating the initial (main body) expression goes
  – This is the “final” continuation…
  – What should this continuation do?
Top level Evaluation

\[(\text{define eval-program})\]
\[\lambda (\text{pgm})\]
\[\text{(cases program pgm)}\]
\[\text{(a-program (exp)}\]
\[\text{(eval-expression)}\]
\[\text{exp (init-env) (halt-cont))])))\]

Applying the Final Continuation

- \[(\text{apply-cont (halt-cont) val})\]
  = \[(\text{begin (write val) (newline))}\]

Top level continuation

\[(\text{define halt-cont})\]
\[\lambda ( )\]
\[\lambda (\text{val})\]
\[\text{(begin (write val) (newline)))}\]

Evaluating Expressions

 Instead of expression and environment mapped to a value we have:

\[exp * env * cont \rightarrow expval\]

Simple Expressions I

\[(\text{define eval-expression})\]
\[\lambda (\text{exp env cont})\]
\[\text{(cases expression exp)}\]
\[\text{(lit-exp (datum)}\]
\[\text{(apply-cont cont datum))}\]
\[\text{(var-exp (id)}\]
\[\text{(apply-cont cont)}\]
\[\text{(apply-env env id))}]\]

More Simple Expressions II

\[(\text{define eval-expression})\]
\[\lambda (\text{exp env cont})\]
\[\text{(cases expression exp)}\]
\[\text{(proc-exp (ids body)}\]
\[\text{(apply-cont cont)}\]
\[\text{(closure ids body env))}]\]

...
Recursive Procedure Definitions

• Creates a new environment without calling eval-expression
• Evaluates the body in the new environment
• Returns the value of the body
  – Control context of the body is the same as that of the entire expression

Recursive procedures

(letrec-exp
  (proc-names idss bodies letrec-body)
  (eval-expression letrec-body
   (extend-env-recursively
     proc-names idss bodies env)
   cont)) ;not halt continuation

Evaluation of Conditionals

(if-exp (test-exp true-exp false-exp)
  (eval-expression test-exp env
   (test-cont true-exp
    false-exp
    env
    cont)))

Continuation for Tests

(define test-cont
  (lambda (true-exp false-exp env cont)
    (lambda (val)
      (if (true-value? val)
        (eval-expression
         true-exp env cont)
        (eval-expression
         false-exp env cont))))

Applying Continuations

(define apply-cont
  (lambda (cont v)
    (cont v)))

Data Structure Representation of Continuations

(define-datatype continuation continuation?
  (halt-cont)
  (test-cont
   (true-exp expression?)
   (false-exp expression?)
   (env environment?)
   (cont continuation?)))
Applying Continuations in Records

```scheme
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      (halt-cont ( )
        (begin (write val) (newline))
      ...) )
  )
```

Applying Continuation (contd.)

```scheme
(test-cont
  (true-exp false-exp env cont)
  (if (true-value? val)
    (eval-expression
      true-exp env cont)
    (eval-expression
      false-exp env cont))
)
```

Evaluating Local Bindings

```scheme
(let-exp (ids rands body)
  (eval-rands rands env
    (let-exp-cont
      ids  env  body  cont))
)
```

Continuations for Let Expressions

```scheme
(define-datatype continuation continuation?
  ......
  (let-exp-cont
    (ids (list-of symbol?))
    (env environment?)
    (body expression?)
    (cont continuation?)
  )
)
```

Applying Let Continuations

```scheme
(define apply-cont
  (lambda (cont val)
    (cases continuation cont
      ...
      (let-exp-cont (ids env body cont)
        (let ((new-env (extend-env ids val env)))
          (eval-expression body new-env cont)))
        ...
      ))
  )
```

Primitive Operations

```scheme
(primapp-exp (prim rands)
  (eval-rands
    rands
    env
    (prim-args-cont prim cont)))
```


Applications

(app-exp (rator rands)
 (eval-expression rator env
  (eval-rator-cont rands
   env
   cont)))))

Evaluating Operands

(define eval-rands
 (lambda (rands env cont)
  (if (null? rands)
    (apply-cont cont '())
    (eval-expression
     (car rands)
     env
     (eval-first-cont rands env cont)))))

Continuations for Arguments

(define-datatype continuation continuation?
  ...
  (eval-rator-cont
   (rands (list-of expression?))
   (env environment?)
   (cont continuation?))
  (eval-rands-cont
   (proc expval?)
   (cont continuation?))

Arguments in Application (contd.)

(eval-first-cont
 (exps (list-of expression?))
 (env environment?)
 (cont continuation?))
 (eval-rest-cont
 (first-value expval?)
 (cont continuation?)))

Applying the Continuation

(eval-first-cont (exps env cont)
  (eval-rands (cdr exps) env
   (eval-rest-cont first cont)
   (let ((rest val))
    (apply-cont cont
     (cons first rest)))))

Continuation for Operands

(eval-rands-cont (proc cont)
  (let ((args val))
    (if (procval? proc)
      (apply-procval proc args cont)
      (eopl:error .... )))))
**Procedures**

```
(define-datatype procval procval?
  (closure
    (ids (list-of symbol?))
    (body expression?)
    (env environment?)))
```

**Applying Procedures**

```
(define apply-procval
  (lambda (proc args cont)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body
          (extend-env ids args env)
          cont)))))
```

**Primitive Arguments Continuation**

```
(define-datatype continuation continuation?
  ...
  (prim-args-cont
    (prim primitive?)
    (cont continuation?))
  ...
  (define apply-cont
    (lambda (cont val)
      ...
      (prim-args-cont
        (prim cont)
        (let ((args val))
          (apply-cont
            cont (apply-primitive prim args)))))..)
```

**An Imperative Interpreter**

- Shared variables in place of bindings
- `eval-expression exp env cont`
- `eval-rands rands env cont`
- `apply-procval proc args cont`
- `apply-cont cont val`
- Add seven global registers
- Use assignments to the register

**Exception Handling**

```
<expression> \rightarrow
<expression> handle <expression>
<expression> \rightarrow raise <expression>
```

- Evaluate the second expression and install it as an exception handler.
- If an exception is raised invoke the most recent exception handler.

**Record Representation for Exceptions AST**

```
try-exp (body-exp handler-exp)
raise-exp (exp)
```
Example

• list-index returns the first occurrence of a given number in a list of numbers

letrec
index (n, l) =
  if null? (l)
    then sub1 (0)
    else if equal? (n, car(l))
      then 0
      else let p = (index n cdr(l))
        in if equal? (p, sub1 (0))
          then sub1 (0)
          else add1 (p)
  in ...

List Index using Exception Handling

let index = proc (n, l)
letrec
  loop (l) =
    if null? (l)
      then raise sub1 (0)
      else if equal? (n, car(l))
        then 0
        else let p = (index n cdr(l))
          in if equal? (p, sub1 (0))
            then sub1 (0)
            else add1 (p)
      in ...

Implementing Exception Handling

• Add two new continuation handlers
  (handler-cont
    (body expression?)
    (env environment?)
    (cont continuation?))
  (try-cont
    (handler expval?)
    (cont continuation?))

Applying Continuations

(tr-expr (body-expr handler-exp)
  (eval-expr handler-expr env
    (handler-cont body-expr env cont)))
  (apply-cont (handler-cont body-expr env cont)
    handler-valu)
  = (if (procval? handler-valu)
        (eval-expr body-expr env
          (try-cont handler-valu cont))
        (eopl: error 'eval-exp "Error handler not procedure ...")

Normal Execution

• If the body returns normally:
  (apply-cont (try-cont handler cont) val)
  = (apply-cont cont val)

Exception Raised

• If an exception is raised, search through the continuation for the nearest handler.
  In eval-expression:
    (raise-exp (exp)
      (eval-expression exp env (raise-cont cont)))
  In continuations:
    (apply-cont (raise-cont cont) val)
    = (find-handler val cont)