6.001 Recitation 24: Explicit Control Evaluator

RI: Gerald Dalley, dalleyg@mit.edu, 10 May 2007

http://people.csail.mit.edu/dalleyg/6.001/SP2007/

Announcements

- Next Monday/Tuesday: last tutorial
- Next Wednesday: last recitation (exam review)
- Next Thursday: last lecture
- Exam:
  - Tues. 22 May, 1:30-4:30pm, Johnson Athletic Center.
  - 3 pages of notes are allowed.
  - Topics on the final will cover the entire semester (basic procedures, recursion/iteration, orders of growth, HOPs, data structures), but significant parts will be from the last third of the course after Quiz 2 (OOP, evaluators, lazy evaluation with and without memoization, streams, register machines, *etc*ifnextchar...).

Explicit Control Evaluator: The Important Contracts

Function	Assumptions	Promises
`eval-dispatch`	• expression in `exp`, • environment in `env`, • return address in `continue`	• end up at `continue` • result in `val`
`apply-dispatch`	• procedure to be applied in `proc`, • list of arguments in `argl`, • return address at the top of the stack	• top of stack popped, • stack top → end address • result in `val`
`eval-sequence`	• sequence of expressions in `unev`, • environment in `env`, • return address at the top of the stack	• eval the expressions in sequence, • top of stack popped, • stack top → end address • result of the final expression in `val`

Calling Conventions

In general, a call to `eval-dispatch` should look like this:

```
    <save registers whose values are needed later>
    (assign exp <expression to evaluate>)
    (assign env <environment in which to evaluate expression>)
    (assign continue <label>)
    (goto eval-dispatch)
  <label>
    <restore any saved registers>
    <use the expression value stored in val>
```

But often `exp`, `env`, and/or `continue` already have the proper value.

`eval-sequence` and `apply-dispatch` use a different convention/contract for storing the return point (`continue`). this is just a performance optimization to minimize use of the stack (tail recursion).

Adding AND to the Explicit Control Evaluator

Let's add the special form and to the explicit control evaluator. For simplicity, we'll only worry about `and`s of two sub-expressions. Assume that we have primitive ops `and-first-exp` and `and-second-exp`. first we add a clause to `eval-dispatch`:

```
eval-dispatch
  ...
  (test (op and?) (reg exp))
  (branch (label ev-and))
  ...
```

Fill in the blanks below:

```
   ev-and
 1   (assign unev [                                ])
 2   (assign exp  [                                ])
 3   (save continue)
 4   (save env)
 5   (save unev)
 6   (assign continue eval-after-first)
 7   (goto [                                       ])
   eval-after-first
 8   (restore [                                    ])
 9   (restore [                                    ])
10   (test (op true?) (reg val))
11   (branch (label eval-second-arg))
12   (assign val #f)
13   (restore [                                    ])
14   (goto (reg continue))
   eval-second-arg
15   (assign exp  [                                ])
16   (assign continue after-second)
17   (goto [                                       ])
   after-second
18   [                                             ]
19   (goto (reg continue))
```

Hand evaluate the expression `(and #t (and #f #t))`

exp	env	continue	val	unev	stack (grows right)	pc
(and #t (and #f #t))	GE	halt				eval-dispatch

Does this `ev-and` routine handle tail recursion? Compare the behavior of regular scheme and our explicit control evaluator when evaluating the expressions below.

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (cdr x)))))
(define z (list 1))
(set-cdr! z z)
(list? z)
```

How could we change the code above so that it handles tail-recursion? *Hint: remove lines 16 through 19 and add two lines in their place.*

- new 16:
- new 17:

Can we get rid of the new line 16 by moving another line somewhere?

Could we remove line 12 without changing the value returned by the code? Why or why not?

How can we get rid of line 15 by changing another line?

To be continued...

Assume the Scheme expressions below are evaluated in sequence by `eval-dispatch`. Determine what label is in the `continue` register when `eval-dispatch` is called to evaluate the subexpressions. Assume that the `continue` register contains the label `halt` when each top-level expression is evaluated.

```
(if #f
    (+ 3 2)
    (* 3 2))
```

- What's the value in the `continue` register when `#f` is evaluated?
- What's the value in the `continue` register when `*` is evaluated?

```
(+ (* x x) (- 2))
```

- What's the value in the `continue` register when `(* x x)` is evaluated?

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
(fact 2)
```

- What's the value in the `continue` register when the `if` consequent `1` is evaluated?
- What's the value in the `continue` register when `2` is evaluated?

```
(define ifact
  (lambda (n product)
    (if (= n 1)
        product
        (ifact (- n 1) (* n product)))))
(ifact 2 1)
```

- What's the value in the `continue` register when the `if` consequent `product` is evaluated?
- What's the value in the `continue` register when `1` is evaluated?

Solutions

Announcements

- Next Monday/Tuesday: last tutorial
- Next Wednesday: last recitation (exam review)
- Next Thursday: last lecture
- Exam:
  - Tues. 22 May, 1:30-4:30pm, Johnson Athletic Center.
  - 3 pages of notes are allowed.
  - Topics on the final will cover the entire semester (basic procedures, recursion/iteration, orders of growth, HOPs, data structures), but significant parts will be from the last third of the course after Quiz 2 (OOP, evaluators, lazy evaluation with and without memoization, streams, register machines, *etc*ifnextchar...).

Explicit Control Evaluator: The Important Contracts

Function	Assumptions	Promises
`eval-dispatch`	• expression in `exp`, • environment in `env`, • return address in `continue`	• end up at `continue` • result in `val`
`apply-dispatch`	• procedure to be applied in `proc`, • list of arguments in `argl`, • return address at the top of the stack	• top of stack popped, • stack top → end address • result in `val`
`eval-sequence`	• sequence of expressions in `unev`, • environment in `env`, • return address at the top of the stack	• eval the expressions in sequence, • top of stack popped, • stack top → end address • result of the final expression in `val`

Calling Conventions

In general, a call to `eval-dispatch` should look like this:

```
    <save registers whose values are needed later>
    (assign exp <expression to evaluate>)
    (assign env <environment in which to evaluate expression>)
    (assign continue <label>)
    (goto eval-dispatch)
  <label>
    <restore any saved registers>
    <use the expression value stored in val>
```

But often `exp`, `env`, and/or `continue` already have the proper value.

`eval-sequence` and `apply-dispatch` use a different convention/contract for storing the return point (`continue`). this is just a performance optimization to minimize use of the stack (tail recursion).

Adding AND to the Explicit Control Evaluator

Let's add the special form and to the explicit control evaluator. For simplicity, we'll only worry about ands of two sub-expressions. Assume that we have primitive ops `and-first-exp` and `and-second-exp`. first we add a clause to `eval-dispatch`:

```
eval-dispatch
  ...
  (test (op and?) (reg exp))
  (branch (label ev-and))
  ...
```

Fill in the blanks below:

```
  ev-and
1   (assign unev [(op and-second-exp) (reg exp)] )
2   (assign exp [(op and-first-exp) (reg exp)] )
3   (save continue)
4   (save env)
5   (save unev)
6   (assign continue eval-after-first)
7   (goto [(label eval-dispatch)] )
  eval-after-first
8   (restore [unev] )
9   (restore [env] )
10  (test (op true?) (reg val))
11  (branch (label eval-second-arg))
12  (assign val #f)
13  (restore [continue] )
14  (goto (reg continue))
  eval-second-arg
15  (assign exp [(reg unev)] )
16  (assign continue after-second)
17  (goto [(label eval-dispatch)] )
  after-second
18  [(restore continue)]
19  (goto (reg continue))
```

Hand evaluate the expression `(and #t (and #f #t))`

exp	env	continue	val	unev	stack (grows right)	pc
(and #t (and #f #t))	GE	halt				eval-dispatch
(and #t (and #f #t))	GE	halt				ev-and
#t	GE	eval-after-first		(and #f #t)	halt GE (and #f #t)	eval-dispatch
#t	GE	eval-after-first		(and #f #t)	halt GE (and #f #t)	ev-self-eval
#t	GE	eval-after-first	#t	(and #f #t)	halt GE (and #f #t)	eval-after-first
#t	GE	eval-after-first	#t	(and #f #t)	halt	eval-second-arg
(and #f #t)	GE	after-second	#t	(and #f #t)	halt	eval-dispatch
(and #f #t)	GE	after-second	#t	(and #f #t)	halt	ev-and
#f	GE	eval-after-first	#t	#t	halt after-second GE #t	eval-dispatch
#f	GE	eval-after-first	#t	#t	halt after-second GE #t	ev-self-eval
#f	GE	eval-after-first	#f	#t	halt after-second GE #t	eval-after-first
#f	GE	after-second	#f	#t	halt	after-second
#f	GE	halt	#f	#t		halt

Does this `ev-and` routine handle tail recursion? Compare the behavior of regular scheme and our explicit control evaluator when evaluating the expressions below.

```
(define (list? x)
  (or (null? x)
      (and (pair? x) (list? (cdr x)))))
(define z (list 1))
(set-cdr! z z)
(list? z)
```

No. In line 18, we restore the `continue` register after the call returns. For tail recursion, we need the preceding call to `eval-dispatch` to return directly to `ev-and`'s caller, not to `after-second`.

How could we change the code above so that it handles tail-recursion? *Hint: remove lines 16 through 19 and add two lines in their place.*

- new 16: `(restore continue)`
- new 17: `(goto (label eval-dispatch))`

Can we get rid of the new line 16 by moving another line somewhere?

yes, move 13 after 9

Could we remove line 12 without changing the value returned by the code? Why or why not?

yes (assuming everything except `#f` is `true?`)

How can we get rid of line 15 by changing another line?

Change line 8 to `(restore exp)`.

Here's the code after all optimizations:

```
    ev-and
 1    (assign unev (op and-second-exp) (reg exp))
 2    (assign exp (op and-first-exp) (reg exp))
 3    (save continue)
 4    (save env)
 5    (save unev)
 6    (assign continue eval-after-first)
 7    (goto (label eval-dispatch))
    eval-after-first
 8    (restore exp)
 9    (restore env)
10    (restore continue)
11    (test (op true?) (reg val))
12    (branch (label eval-second-arg))
13    (goto (reg continue))
    eval-second-arg
14    (goto (label eval-dispatch))
```

To be continued...

Assume the Scheme expressions below are evaluated in sequence by `eval-dispatch`. Determine what label is in the `continue` register when `eval-dispatch` is called to evaluate the subexpressions. Assume that the `continue` register contains the label `halt` when each top-level expression is evaluated.

```
(if #f
    (+ 3 2)
    (* 3 2))
```

- What's the value in the `continue` register when `#f` is evaluated? `ev-if-decide`
- What's the value in the `continue` register when `*` is evaluated? `ev-appl-did-operator`

```
(+ (* x x) (- 2))
```

- What's the value in the `continue` register when `(* x x)` is evaluated? `ev-appl-accumulate-arg`

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
(fact 2)
```

- What's the value in the `continue` register when the `if` consequent `1` is evaluated? `ev-appl-accum-last-arg`
- What's the value in the `continue` register when `2` is evaluated? `ev-appl-accum-last-arg`

```
(define ifact
  (lambda (n product)
    (if (= n 1)
        product
        (ifact (- n 1) (* n product)))))
(ifact 2 1)
```

- What's the value in the `continue` register when the `if` consequent `product` is evaluated? `halt`
- What's the value in the `continue` register when `1` is evaluated? `ev-appl-accum-last-arg`