MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – November 12, 1996

Variations on a Scheme—Nondeterministic evaluation

In todays lecture, we are going to discuss another variation on the idea of an evaluator, in this case introducing the idea of nondeterminism.

Nondeterminism can be introduced into Scheme with the new special form `amb`. The idea is that when the interpreter reaches an expression of the form `(amb` e_1 e_2`)` it nondeterministically selects one of expressions e_1 or e_2 and continues the evaluation with that expression. Since the language with `amb` is nondeterministic, a given expression can have many different possible executions and many different possible values. In general, the special form `amb` can take any number of arguments, one of which is nondeterministically selected at run time.

As an example consider

```
(define (a-number-between low high)
  (cond ((= low high)
         low)
        (else
         (amb low
              (a-number-between (+ low 1) high)))))
```

The expression `(a-number-between 1 10)` has ten different possible values, each of which is an integer between 1 and 10.

Consider the following procedure.

```
(define (a-pythagorean-triple-between low high)
  (let ((n (a-number-between low high))
        (m (a-number-between low high))
        (p (a-number-between low high)))
    (cond ((= (+ (* n n) (* m m)) (* p p))
           (list n m p))
          (else (amb)))))
```

The expression `(a-pythagorean-triple-between 1 10)` has 1000 different possible executions. All but four of these executions fail. The four non-failing executions have values `(3 4 5)`, `(4 3 5)`, `(6 8 10)`, and `(8 6 10)`. In general, if `(n m p)` is a non-failing value of the expression `(a-pythagorean-triple-between low high)` then `n`, `m`, and `p` are numbers between `low` and `high` such that $\mathtt{n}^2 + \mathtt{m}^2 = \mathtt{p}^2$. In other words, there exists a right triangle whose sides have length `n`, `m`, and `p`.

One way to solve such problems is to exhaustively search out all possibilities, filtering out those that violate some constraint. We can express the requirement that a particular predicate expression `p` is true by `require`ing its truth:

```
(define (require p)
  (cond ((not p) (amb))
        (else 'ok)))

(define (a-pythagorean-triple-between low high)
  (let ((n (a-number-between low high))
        (m (a-number-between low high))
        (p (a-number-between low high)))
    (require (= (+ (* n n) (* m m)) (* p p)))
    (list n m p)))
```

We are often interested in determining whether or not a given nondeterministic expression has a nonfailing value. One might try running the probabilistic interpreter over and over again hoping to find a nonfailing value. A much better approach is to *systematically search* the space of all possible executions. We will play with a modified version of the evaluator which we call the *search evaluator*. The search evaluator systematically searches the space of all possible executions of a given expression. When the search evaluator encounters an application of `amb` it initially selects the first argument. If this selection does not result in a nonfailing execution, then the evaluator "backs up" and tries the second argument. The search evaluator is used to implement a read-eval-print loop with some unusual properties. The read-eval-print loop reads an expression and prints the value of the first nonfailing execution. For example, consider the following interaction.

```
SEARCH==>> (a-number-between 1 10)
Starting a new problem
1
SEARCH==>> (a-pythagorean-triple-between 1 10)
Starting a new problem
(3 4 5)
```

The read-eval-print loop allows the user to ask for alternative executions of an expression. If the first execution is not desired, or if one simply wants to see the value of the next successful execution, one can ask the interpreter to back up and attempt to generate a second nonfailing execution.

An example where searching over possible evaluations is useful is given by the following elementary logic puzzle:

Multiple Dwelling [1]

> Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors.
> Baker does not live on the top floor.
> Cooper does not live on the bottom floor.
> Fletcher does not live on either the top or the bottom floor.
> Miller lives on a higher floor than does Cooper.
> Smith does not live on a floor adjacent to Fletcher's.
> Fletcher does not live on a floor adjacent to Cooper's.
> Where does everyone live?

In this case, we can consider possible assignments of the characters to floors in a building. We will also need to be able to tell when a list is made up of distinct entries (no two are the same):

[1] This puzzle is typical of a large class of puzzles. This particular one is quoted from *Superior Mathematical Puzzles*, by Howard P. Dinesman, 1968, Simon and Schuster, New York.

```
(define (distinct list)
  (cond ((null? list) true)
        ((null? (cdr list)) true)
        ((member (car list) (cdr list)) false)
        (else (distinct (cdr list)))))
(define (multiple-dwelling)
  (let ((baker    (amb 1 2 3 4 5))
        (cooper   (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller   (amb 1 2 3 4 5))
        (smith    (amb 1 2 3 4 5)))
    ;; Beginning of filtration
    (require (distinct (list baker cooper fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require (not (= (abs (- smith fletcher)) 1)))
    (require (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Indeed, we can try it:

```
SEARCH==>> (multiple-dwelling)
((baker 3) (cooper 2) (fletcher 4) (miller 5) (smith 1))
SEARCH==>> (next)
There are no more values of (multiple-dwelling)
```

Below is code to implement this idea, which we will discuss. All of this is taken from the notes.

```
(define (analyze exp)
  (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
        ((quoted? exp) (analyze-quoted exp))
        ((variable? exp) (analyze-variable exp))
        ((assignment? exp) (analyze-assignment exp))
        ((definition? exp) (analyze-definition exp))
        ((if? exp) (analyze-if exp))
        ((lambda? exp) (analyze-lambda exp))
        ((begin? exp) (analyze-sequence (begin-actions exp)))
        ((amb? exp) (analyze-amb exp))
        ((cond? exp) (analyze (COND->IF exp)))
        ((let? exp) (analyze (LET->combination exp)))
        ((application? exp) (analyze-application exp))
        (else
         (error "Unknown expression type -- ANALYZE" exp))))
```

```
(define (analyze-self-evaluating exp)
  (lambda (env succeed fail)
    (succeed exp fail)))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env succeed fail)
      (succeed qval fail))))

(define (analyze-variable exp)
  (lambda (env succeed fail)
    (succeed (lookup-variable-value exp env)
             fail)))

(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env succeed fail)
      (succeed (make-procedure vars bproc env)
               fail))))
(define (analyze-definition exp)
  (let ((var (definition-variable exp))
        (vproc (analyze (definition-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail)
               (define-variable! var val env)
               (succeed 'done fail))
             fail))))
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail)          ; *1*
               (let ((oval (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'done
                          (lambda ()     ; *2*
                            (set-variable-value! var oval env)
                            (fail)))))
             fail))))
```

```
(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env succeed fail)
      (pproc env
             (lambda (pval fail)
               (if (true? pval)
                   (cproc env succeed fail)
                   (aproc env succeed fail)))
             fail))))
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
         (lambda (va fail)
           (b env
              succeed
              fail))
         fail)))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error
         "BEGIN requires subexpressions -- ANALYZE" exps))
    (define (loop first rest)
      (if (null? rest)
          first
          (loop (sequentially first (car rest))
                (cdr rest))))
    (loop (car procs) (cdr procs))))
```

The `amb` special form is the key element in the nondeterministic language. Here we see the essence of the interpretation process. The execution procedure for `amb` defines a loop `try-next` that cycles through the execution procedures for the alternative values of the `amb` expression. An alternative execution procedure is called with the success continuation of the `amb` expression, and a failure continuation that tries the next alternative. When there are no more alternatives to try, the `amb` expression fails.

```
(define (analyze-amb exp)
  (let ((alternatives (map analyze (operands exp))))
    (lambda (env succeed fail)
      (define (try-next alternatives)
        (if (null? alternatives)
            (fail)
            ((car alternatives)
             env
             succeed
             (lambda ()
               (try-next (cdr alternatives))))))
      (try-next alternatives))))
```

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail)
               (evlist aprocs
                       env
                       (lambda (args fail)
                         (exapply proc args succeed fail))
                       fail))
             fail))))
(define (exapply proc args succeed fail)
  (cond ((primitive-procedure? proc)
         (succeed (apply-primitive-procedure proc args)
                  fail))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))
          succeed
          fail))
        (else
         (error "Unknown procedure type -- RTAPPLY"
                proc))))
(define (evlist operands env succeed fail)
  (if (null? operands)
      (succeed '() fail)
      ((car operands)
       env
       (lambda (arg fail)
         (evlist (cdr operands)
                 env
                 (lambda (args fail)
                   (succeed (cons arg args)
                            fail))
                 fail))
       fail)))
```