Data and Program Structure

Non-Deterministic Computing

(SICP 4.3)

Lecture IX

Ahmed Rezine

Linköpings Universitet

TDDA69, VT 2014

Outline

Outline

Non-deterministic programs: Choosing among alternatives

Solving many problems, such as:

- Finding a path in a graph
- Matching a pattern of some sort
- Finding a solution placement for the eight queens problem

amounts to exploring the possible choices, for example in depth first order with "chronological backtracking" until a solution is found or all alternatives are exhausted.

Non-deterministic programs : the **amb** operator

```
(amb expr1 expr2 ... exprn)
;; chooses one of expr1, ... exprn
```

-
  ```
  (amb 1 2 3 4 5)
  ;; can evaluate to 1, 2, 3, 4, or 5
  ```

-
  ```
  (list (amb 1 2) (amb 'a 'b))
  ;; can evaluate to (1 a) (1 b) (2 a) (2 b)
  ```

If we backtrack to amb after a failure we should get a new value. It is not enough to randomly choose a possibility.

Non-deterministic programs: requirements and failure

```
(amb)
;; expresses FAILURE of the current path
```

```
(define (require p)
  (if (not p) (amb)))
;; require expresses requirement.
;; Their violation amounts to a failure
```

```
(require (= a b))

(require (sorted? l))
```

Non-deterministic programs: **an-element-of**

```
(define (an-element-of items)
  (require (not (null? items)))
  (amb (car items)
       (an-element-of (cdr items))))

(an-element-of '(1 3 5 7))
;; can evaluate to 3, another element or ``
   failure''
```

Modelling with **require** and **an-element-of**

```
(define (prime-sum-pair list1 list2)
  (let ((a (an-element-of list1))
        (b (an-element-of list2)))
       (require (prime? (+ a b)))
       (list a b)))
```

```
> (prime-sum-pair '(1 3 5 8) '(20 35 100))
> (1 100)

> try-again
> (3 20)

> try-again
> (3 100)

> try-again
;;; There are no more values of ...
```

Outline

Continuation-passing style

- How to implement backtracking when a path leads to a failure or because of a **try-again** ?
- We will need to get back to the latest choice point and restore the same state as if the last choice did not take place and we are about to choose a new choice.
- With streams, we instead generated all possibilities and filterd them in order to enumerate the solutions
- Here we are going to choose a new style of programming that makes use of "continuations"

Continuation-passing style

- **Continuations** save enough information in order to continue the computation. They are a way to formalize what is there left to do in a computation
- E.g., **(lambda (r) (+ r 5))** is a continuation of **(* 3 4)** in **(+ (* 3 4) 5)**
- A function written in CPS takes its continuation as argument
- The continuation is applied on the function result

```
(+ (* 3 4) (- 6 2))
;; becomes, where (define (times a b f) (f (* a b))):
(times 3 4
       (lambda (product)
         (minus 6 2
                (lambda (diff)
                  (plus product diff
                        (lambda (sum) sum))))))
```

Continuation-passing style (cont.)

Many languages have first class continuations with different names (e.g. callcc in Ruby).
Scheme supplies the **call-with-current-continuation** (or **call/cc**) operator to manipulate the flow of control:

```
(define return #f)

(+ 1 (call/cc
          (lambda (cont)
               (set! return cont)
               1)))
```

```
> (return 22)
23
>(return 16)
17
```

Continuation-passing style (cont.)

Many languages have first class continuations with different names (e.g. callcc in Ruby).
Scheme supplies the **call-with-current-continuation** (or **call/cc**) operator to manipulate the flow of control:

```
(define foo #f)



(define (count)
  (let ((c 0))
    (set! c (+ 1 c))
    (call/cc
      (lambda (k)
         (set! foo k)))
    (set! c (+ 1 c))
    c ))
```

```
> (count)
2
>(foo)
3
>(foo)
4
>(define bar foo)
>(count)
2
>(foo)
3
>(bar)
5
```

Continuation-passing style (cont.)

- We will not use **call/cc** in the implementation of the **amb** evaluator
- Still, we will adopt a CPS of programming, by always calling procedures we got as parameters and never returning from them
- This is possible because the construction makes use of tail recursion and the stack does not explode

Outline

Implementation of the **amb-evaluator**

- We modify the analyzing evaluator of lecture VIII (Sect. 4.1.7) (it is also possible to modify the original evaluator of lecture VI)
- The difference will be in the execution procedures and not in the analysis
- The execution procedures of the the **amb** evaluator take four arguments :
  - the expression to be evaluated
  - an environment
  - a <u>success</u> continuation
  - a <u>failure</u> continuation

Implementation of the **amb-evaluator**

- Functions do not return like in the direct style of programming.
- Every function terminates by calling one of the two continuations
- If the evaluation results in a value, the success continuation is called with that value
- If a dead end is instead discovered, the failure continuation is called.
- The failure continuation is responsible for trying another choice or branch (corresponding to an **amb**-expression).
- Backtracking is captured by the construction and the mechanisms for calling appropriate continuations.

Implementation of the **amb-evaluator** (cont.)

- Failure continuations are constructed by
  - **amb** expressions in order to choose other alternatives in case the current one leads to a dead end
  - the top level driver in order to report failure when no more choices are possible
  - assignments to undo assignments during backtracking
- Failure continuations are initiated when
  - **(amb) is evaluated**
  - **try-again** is enterred at the top level
- Failure continuations are called, when
  - **amb** does not have any more choices to pick from
  - after undoing the effect of an assignment

Failure continuations are procedures without arguments:

```
(lambda () ...)
```

Implementation of the **amb-evaluator** (cont.)

A success contination is a procedure with two arguments: the value of the earlier evaluation and a failure continuation:

```
(lambda (value fail) ... )
```

All execution procedures look as follow:

```
(lambda (env succeed fail) ... )
;; succeed is a (lambda (value fail) ... )
;; fail is a (lambda () ...)
```

Implementation of the **amb-evaluator** (cont.)

A possible call from the top level

```
(ambeval
  exp
  the-global-environment
  (lambda (value fail) value)
  (lambda () 'failed))
```

```
(lambda (value fail) value)
```

is the top level success procedure, and if no failure is triggered, this procedure will be the last procedure to get evaluated and it will “return” its value.

Implementation of the **amb-evaluator** (cont.)

In the book, **ambeval** is implemented by modifying the analyzing evaluator.

```
(define (ambeval exp env succeed fail)
  ((analyze exp) env succeed fail))
```

Execution procedures are modified in order to pass and manipulate continuations

Implementation of the **amb-evaluator** (sicp p.429)

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

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

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

Implementation of the **amb-evaluator** (cont.)

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

(require (= (+ (an-element-of 1 2)
               3 ;; suppose we are evaluating 3 in:
               (an-element-of 4 5)))
            10)
```

- We first evaluate (an-element-of 1 2) before evaluating 3.
- The success continuation (succeed) adds 3 to the list of the already evaluated arguments and continues the evaluation
- The failure continuation (fail) is not affected by the evaluation of 3: in case of a failure, the compuations that needed to be done before the evaluation of 3 are the same as the ones after.
- The earlier argument created a failure-continuation that generated the value 2 in case the value 1 led to a dead end.

Conditional expressions (sicp p.430)

```
(...
  (if (= a (an-element-of 1 2 3))
    (+ 2 (an-element-of 4 5))
    10 ) ... )

(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
             ;; success continuation for evaluating the
             ;; predicate to obtain pred-value
             (lambda (pred-value fail2)
               (if (true? pred-value)
                   (cproc env succeed fail2)
                   (aproc env succeed fail2)))
             ;; failure continuation go evaluating the
                predicate
             fail))))
```

The **amb**-expression (sicp p.434)

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

Sequences (sicp p.432)

```
(define (analyze-sequence exps)
  (define (sequentially a b)
    (lambda (env succeed fail)
      (a env
         ;; success continuation for calling a
         (lambda (a-value fail2)
           (b env succeed fail2))
         ;; failure continuation for calling a
         fail)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs)
        first-proc
        (loop (sequentially first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error 'analyze-sequence "Empty sequence"))
    (loop (car procs) (cdr procs))))
```

Assignments (sicp p.432)

```
(define (analyze-assignment exp)
  (let ((var (assignment-variable exp))
        (vproc (analyze (assignment-value exp))))
    (lambda (env succeed fail)
      (vproc env
             (lambda (val fail2)        ; *1*
               (let ((old-value
                      (lookup-variable-value var env)))
                 (set-variable-value! var val env)
                 (succeed 'ok
                          (lambda ()    ; *2*
                            (set-variable-value! var
                                                 old-value
                                                 env)
                            (fail2)))))
             fail))))
```

The failure continuation (*2*) restores the old value of the variable before continuing the failure (fail2).

Procedure applications (sicp p.433)

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env succeed fail)
      (fproc env
             (lambda (proc fail2)
               (get-args aprocs
                         env
                         (lambda (args fail3)
                           (execute-application
                            proc args succeed
                                fail3))
                         fail2))
             fail))))
```

Procedure applications (sicp p.433)

```
(define (get-args aprocs env succeed fail)
  (if (null? aprocs)
      (succeed '() fail)
      ((car aprocs) env
       ;; success continuation for this aproc
       (lambda (arg fail2)
         (get-args (cdr aprocs)
                   env
                   ;; success continuation for
                   ;; rec. call to get-args
                   (lambda (args fail3)
                     (succeed (cons arg args)
                              fail3))
                                fail2))
       fail)))
```

The driver loop

```scheme
(define (driver-loop)
  (define (internal-loop try-again)
    (prompt-for-input input-prompt)
    (let ((input (read)))
      (if (eq? input 'try-again)
          (try-again)
          (begin
            (newline)
            (display ";;; Starting a new problem ")
            (ambeval input
                     the-global-environment
                     ;; ambeval success
                     (lambda (val next-alternative)
                       (announce-output output-prompt)
                       (user-print val)
                       (internal-loop next-alternative))
                     ;; ambeval failure
                     (lambda ()
                       (announce-output
                        ";;; There are no more values of")
                       (user-print input)
                       (driver-loop)))))))
```

The driver loop

```
  (internal-loop
   (lambda ()
     (newline)
     (display ";;; There is no current problem")
     (driver-loop))))
```

What have we used and done ?

- avoided interpr. overhead and created execution procedures
- used continuation passing style. No "return", always forward
- passed two continuation procedures: for success and failure
- we pass around code and environments because the execution procedures result procedure objects that amount to closures.
- backtracking captured by the procedure objects executing failure continuations
- only **amb** and assignments construct failre procedures
- only **amb** and the driver loop trigger failure procedures
- failure procedures are called only by **amb** and after undoing assignments

Outline

More on continuations

call/cc can be used for abnormal terminations, e.g. in a **catch** and **throw**

Example:Test if the elements of a list are positive. If not, stop directly and return the first negative number.

```
;; call/cc takes a procedure (exit-cont) as argument
;; it gives the actual context in which call/cc was called
;; the control ``jumps back'' directly to the calling site
;; when exit-cont is called

(call/cc
 (lambda (exit-cont)
   (for-each
    (lambda (x)
      (if (negative? x)
  (exit-cont x)))
     '(54 0 37 -3 245 19))
   #t))
```

More on continuations

Example: Compute the length of a proper list but return #f in case of an "improper" list.

```scheme
(define list-length
  (lambda (obj)
    (call/cc
     (lambda (return)
       (define (iter lst)
         (cond ((null? lst) 0)
               ((pair? lst) (+ 1 (iter (cdr lst))))
               (else (return #f))))
       (iter obj)))))

; (list-length '(1 2 3 4)) gives 4
; (list-length '(a b . c)) gives #f
```

More on continuations

Every function takes an extra argument: a continuation

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

; can be rewritten to:
(define (factorial-cps n cont-fn)
  (if (= n 0)
      (cont-fn 1)
      (factorial-cps (- n 1)
                     (lambda (val)
                        (cont-fn (* n val))))))
```

More on continuations

Using the substitution model :

```
> (factorial-cps 2 (lambda (res) res))

(factorial-cps 1
  (lambda (val)
    ((lambda (res) res) (* 2 val))))

(factorial-cps 0
  (lambda (val)
    ((lambda (val)
       ((lambda (res) res) (* 2 val))) (* 1 val))))

((lambda (val)
  ((lambda (val)
    ((lambda (res) res) (* 2 val))) (* 1 val))) 1)

(((lambda (val) ((lambda (res) res) (* 2 val))) 1)

((lambda (res) res) (* 2 1))

2
```