Delayed (Lazy) Evaluation & Haskell

- ordinary fns vs special forms
- applicative order
  - all args eval before fn is called
- special forms
  - and (&&)
  - or (||)
  - if then else (?:) ← "short circuit" in C/C++ terminology

  both in fn & procedural lang.
- some semantics simpler with delayed eval

Terminology:

nonstrict	strict
normal order	applicative order
lazy	eager

Algol 60 – delayed eval in its pass by name params

```
ex: fn p (x:bool; y:int):int
    begin
      if x then 1
           else y
    end;
```

p(true, 1 div 0) returns 1 does not eval y

```
fn sum (a:float; i:int; n:int):float
    s:int = 0;
    while i<n do (s+=a[i]; i+=1)
    return s
}
```

call a:array s:= sum(a[1], i, n)

c++ f([](int a){ret 2*a;})

Delayed & Haskell (2)

- passes unevaluated in a thunk.

Scheme
```
(define (p x y)(if x 1 (y)))
(p #t (lambda () (/ 1 0)))
```

Ocaml
```
let p x y = if x 1 (y ())
  p : int -> (() -> int) -> int
p true (fun x -> if x then 1 else y ())
```

Inefficient

```
(def (delsa x)(* (force x)(force x)))
```

param x is evaluated twice for call-by-name

- so use memoization
- pass expr in a thunk (pass by need)
- evaluate first time called then save value.
- just use value other times called.

then special forms like if are just functions

Delay allows params unevaluated

ex: take 1st n items from list

```
let take n l = match l with
      | [] -> []
      | x::xs -> if n <= 0 then []
                 else x :: take (n-1) xs
```

- need consider 2 spec cases
  list is [] and n = 0
  - errors? n < 0
    n > length list ?
  - raise exn or default behavior

Suppose we have a list generator:

```
let gen n m =
    if n > m then []
             else n :: gen (n+1) m
```

So:

```
take 5 (gen 100 10000)
```

- will generate a list of 9901 elements then use only 5

Lazy Stream

Delayed & Haskell (4)

```
(* stream is a lazy list *)
type 'a stream = End | Stream of 'a * 'a stream thunk

let lgen n m =
    if n > m then End
        else Stream (n,
            ref (Delay (fun() -> lgen (n+1) m)))
#int -> int -> int stream
let naturals = lgen 0 maxint
```

lazy list of 0 ... 4,611,686,018,427,387,903 Quintillions (4.6×10^{18})

but takes little memory

```
let lazytake n sl = match n, sl with
    | _, End -> End
    | n, _ when n<=0 -> End
    | _, Stream(car, cdr) ->
        Stream (car,
            ref (Delay (fun() -> lazytake (n-1)
                (force cdr))))
#int -> 'a stream -> 'a stream
```

so: take 5 naturals

returns [0;1;2;3;4] without generating 10^{18} values

How to do a thunk?

```
type 'a promise =
      | Value of 'a
      | Excep of exn
      | Delay of (unit -> 'a)

type 'a thunk = 'a promise ref

let force thunk = match !thunk with
    | Value v -> v
    | Excep e -> raise e
    | Delay d -> [redacted]
        try let v = [redacted] d ()
            in (thunk := Value v; v)
        with [redacted] -> (thunk := Excep e;
                            raise e)
 # 'a promise ref -> 'a
```

1. program makes a promise with Delay
   - is an unevaluated fun of form $() \to$ 'a
2. first force evaluates it,
   memoizes result,
   returns result
3. more forces just returns result

Lazy evaluation

in a fn langugge can implicitly delay & force:

1. all args to user fns are delayed
2. all bindings in *let* are delayed
3. all args to constructors are delayed
4. all args to arith fns are forced (e.g. *, +, !, ...)
5. all fn-valued args are forced
6. cond & selection *if* first arg are forced

so long lists only eval as far as needed

why not?

- runtime complexity big-O much harder
- side effects may be unpredictable

Haskell

Delayed & Haskell (7)

- fully curried lazy lang with overloading.
- named after Haskell Curry
- current Haskell 98
- fn overloading
- monads to maintain state
- otherwise is pure fn lang (no state)

Syntax: simpler than Ocaml (sometimes)

ex:
```
fac 0 = 1
fac n = n * fac (n-1)
len [] = 0
len (_:xs) = 1 + len xs
sq x = x * x
```

Curried sections

```
plus2 = (2+)   ⟹  (λx→2+x)
times3 = (*3)  ⟹  (λx→x*3)
(+) 2 3 ⟹ 5
(λx→x*x) 3 ⟹ 9
```

note: use \ for λ

HOF & List comprehensions

Delayed & Haskell (8)

map type is $(a \to b) \to [a] \to [b]$

[a] means 'a list

ex:

```
sqlist = map (\x -> x*x)
Num a, Num b [a] -> [b]
```

(.) is func composition:

ex: ((*3) . (2+)) 5 => (*3) 7
=> 21

List comprehensions

```
sqlist l = [x*x | x <- l]
-- semantics: same as above

possq l = [x*x | x <- l, x > 0]
                  ↑ "and"   ↑ condition
```

unsugared:

```
possq l = map (\x -> x*x) (filter (>0) l)
```

where:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (h:t) = if p h then h:filter p t
                        else filter p t
```

```
qsort [] = []
qsort (h:t) = qsort [x | x <- t, x <= h]
              ++ [h]
              ++ qsort [x | x <- t, x > h]
```

- but beware of bad pivot.
- big-O is difficult when not eager

Lazy & Infinite lists

```
f x = 2
-- x never evaluated
myif b x y = if b then x else y
-- work same as built in if

ones = 1 : ones
-- ∞ list of 1's
[n..] means ints n..∞
[n, n+2 ..] means n, n+2, n+4, n+6 .... ∞
```

lazy take/drop

```
take 0 _ = []
take _ [] = []
take n (h:t) = h : take (n-1) t
drop 0 lis = lis
drop _ [] = []
drop n (_:t) = drop (n-1) t
```

what if n<0 ?

Τό Κόσκινον Ἐρατοσθένους (Sieve of Eratosthenes)

Delayed & Haskell (10)

```
sieve (p:lis) = p : sieve [n | n <- lis, mod n p /= 0]
primes = sieve [2..]
```

The sieve is the first number followed by all #'s in list not multiples of it

so: take 6 primes

[2,3,5,7,11,13]

Type Classes

```
sq x = x*x
```

sq 2 ⇒ 4

sq 3.5 ⇒ 12.25

sq :: Num a ⇒ a → a

says that sq is an (a → a) but only if a is of typeclass Num

ex: class Num a where

(+), (-), (*) :: a → a → a

negg, abs :: a → a

... etc. ...

```
 1: (* $Id: lazythunk.ml,v 361.1 2006-03-02 18:48:23-08 - - $ *)
 2:
 3: open Printf
 4:
 5: (* re-implementation of module Lazy *)
 6:
 7: type 'a promise =
 8:      | Value of 'a
 9:      | Excep of exn
10:      | Delay of (unit -> 'a)
11:
12: type 'a thunk = 'a promise ref
13:
14: let force thunk = match !thunk with
15:      | Value value -> value
16:      | Excep excep -> raise excep
17:      | Delay delay -> (try  let value = delay ()
18:                            in  (thunk := Value value; value)
19:                       with excep -> (thunk := Excep excep;
20:                                      raise excep))
21:
22: let (!!) = force                                    !!x
23:
24: (* stream and lazy stuff *)
25:
26: type 'a stream = End | Stream of 'a * 'a stream thunk
27:
28: exception End_stream
29:
30: let (@::) car cdr = Stream (car, cdr)
31:
32: let head stream = match stream with
33:      | End -> raise End_stream
34:      | Stream (car, _) -> car
35:
36: let tail stream = match stream with
37:      | End -> raise End_stream
38:      | Stream (_, cdr) -> !!cdr
39:
40: let rec take n stream = match n, stream with
41:      | _, End -> End
42:      | n, _ when n <= 0 -> End
43:      | _, Stream (car, cdr) ->
44:           Stream (car, ref (Delay (fun () -> take (n - 1) !!cdr)))
45:
46: let rec list_of_stream stream = match stream with
47:      | End -> []
48:      | Stream (car, cdr) -> car :: list_of_stream !!cdr
49:
50: let rec drop n stream = match n, stream with
51:      | _, End -> End
52:      | n, _ when n <= 0 -> stream
53:      | _, Stream (car, cdr) -> drop (n - 1) !!cdr
54:
55: let rec iter fn stream = match stream with
56:      | End -> ()
57:      | Stream (car, cdr) -> (fn car; iter fn !!cdr)
58:
```

```
 59: let rec iter2 fn stream1 stream2 = match stream1, stream2 with
 60:     | End, _ -> ()
 61:     | _, End -> ()
 62:     | Stream (car1, cdr1), Stream (car2, cdr2)
 63:         -> (fn car1 car2; iter2 fn !!cdr1 !!cdr2)
 64:
 65: let rec iter3 fn stream1 stream2 stream3 =
 66:     match stream1, stream2, stream3 with
 67:     | End, _, _ -> ()
 68:     | _, End, _ -> ()
 69:     | _, _, End -> ()
 70:     | Stream (car1, cdr1), Stream (car2, cdr2), Stream (car3, cdr3)
 71:         -> (fn car1 car2 car3; iter3 fn !!cdr1 !!cdr2 !!cdr3)
 72:
 73: let rec map2 fn stream1 stream2 = match stream1, stream2 with
 74:     | End, _ -> End
 75:     | _, End -> End
 76:     | Stream (car1, cdr1), Stream (car2, cdr2)
 77:         -> Stream (fn car1 car2,
 78:                    ref (Delay (fun () -> map2 fn !!cdr1 !!cdr2)))
 79:
 80: (* stuff that uses streams and Nums *)
 81:
 82: let rec range head limit =
 83:     if head > limit
 84:     then End
 85:     else let next = head + 1
 86:          in  Stream (head, ref (Delay (fun () -> range next limit)))
 87:
 88: let naturals = range 0 max_int
 89:
 90: let fac n =
 91:     let rec fac' n m = match n with
 92:         | 0 -> m
 93:         | n -> fac' (n - 1) (n * m)
 94:     in  if n < 0 then invalid_arg "fac"
 95:                  else fac' n 1
 96:
 97: let printfac n = printf "%2d! = %10d\n" n (fac n)
 98:
 99: let printfacs n = iter printfac (take n naturals)
100:
101: (* lazy let fib = 0 : 1 : map2 (+) fib (tail fib) *)
102:
103: let fibstream =
104:     let rec stream0 = Stream (0, stream1)
105:         and stream1 = ref (Delay (fun () -> Stream (1, stream2)))
106:         and stream2 = ref (Delay (fun () -> map2 (+) stream0 !!stream1))
107:     in  stream0
108:
109: let printfib n nfib nfib' =
110:     printf "fib(%3d) = %11d, %20.15f\n"
111:            n nfib (float_of_int nfib /. float_of_int nfib')
112:
113: let printfibs n = iter3 printfib naturals
114:                                 (take n fibstream)
115:                                 (take n (drop 1 fibstream))
116:
```

```
 1: type 'a promise = Value of 'a | Excep of exn | Delay of (unit -> 'a)
 2: type 'a thunk = 'a promise ref
 3: val force : 'a promise ref -> 'a
 4: val ( !! ) : 'a promise ref -> 'a
 5: type 'a stream = End | Stream of 'a * 'a stream thunk
 6: exception End_stream
 7: val ( @:: ) : 'a -> 'a stream thunk -> 'a stream
 8: val head : 'a stream -> 'a
 9: val tail : 'a stream -> 'a stream
10: val take : int -> 'a stream -> 'a stream
11: val list_of_stream : 'a stream -> 'a list
12: val drop : int -> 'a stream -> 'a stream
13: val iter : ('a -> 'b) -> 'a stream -> unit
14: val iter2 : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> unit
15: val iter3 :
16:   ('a -> 'b -> 'c -> 'd) -> 'a stream -> 'b stream -> 'c stream -> unit
17: val map2 : ('a -> 'b -> 'c) -> 'a stream -> 'b stream -> 'c stream
18: val range : int -> int -> int stream
19: val naturals : int stream
20: val fac : int -> int
21: val printfac : int -> unit
22: val printfacs : int -> unit
23: val fibstream : int stream
24: val printfib : int -> int -> int -> unit
25: val printfibs : int -> unit
```