∼ Lecture IX ∼

Keywords:

call-by-value, call-by-name, and call-by-need evaluation; lazy datatypes: sequences, streams, trees; lazy evaluation; sieve of Eratosthenes; breadth-first and depth-first traversals.

References:

- [MLWP, Chapter 5]

Call-by-name evaluation

To compute the value of `F(E)`, first compute the value of the expression `F` to obtain a function value, say `f`. Then compute the value of the expression obtained by substituting the expression `E` for the formal parameter of the function `f` into its body.

NB: *Call-by-need* is similar, but duplicated expressions are only evaluated once. (Haskell is the most widely used call-by-need purely-functional programming language.)

Example: Consider the following function definitions.

```
fun pred n
  = if n = 0 then []
    else n :: pred(n-1) ;
fun lsum( [] , l ) = 0
  | lsum( h::t , l ) = h + lsum( t , l ) ;
```

```
val pred = fn : int -> int list
val lsum = fn : int list * 'a -> int
```


1. Call-by-value evaluation.

```
lsum( pred(2) , pred(10000) )
    | pred(2) ⇝ 2::pred(1) ⇝ 2::1::pred(0) ⇝ 2::1::[]
    | pred(10000) ⇝ 10000::pred(9999) ⇝ ···
    |     ··· ⇝ 10000::9999::...1::[]
⇝ lsum( 2::1::[] , 10000::9999...1::[] )
⇝ 2 + lsum( 1::[] , 10000::9999...1::[] )
⇝ 2 + ( 1 + lsum( [] , 10000::9999...1::[] ) )
⇝ 2 + ( 1 + 0 )
⇝ 2 + 1
⇝ 3
```

2. Call-by-name evaluation.

```
lsum( pred(2) , pred(10000) )
      | pred(2) ⇝ 2::pred(2-1)
⇝ 2 + lsum( pred(2-1) , pred(10000) )
      | pred(2-1) ⇝ 1::pred(1-1)
⇝ 2 + ( 1 + lsum( pred(1-1) , pred(10000) ) )
      | pred(1-1) ⇝ []
⇝ 2 + ( 1 + 0 )
⇝ 2 + 1
⇝ 3
```


Lazy datatypes

Lazy `datatype`s are one of the most celebrated features of functional programming. The elements of a lazy `datatype` are not evaluated until their values are required. Thus a lazy `datatype` may have infinite values, of which we may view any finite part but never the whole.

In a *call-by-value* functional language, like ML, we implement lazy `datatype`s by explicitly *delaying evaluation*. Indeed, to delay the evaluation of an expression `E`, we can use the nameless function `fn() => E` instead, and we force the evaluation of this expression by the function application `(fn() => E)()`.

Examples:

1. Sequences (= finite and infinite lists).

```
datatype 'a seq
  = nil | cons of 'a * ( unit -> 'a seq ) ;
```

2. Streams (= infinite lists).

```
datatype 'a stream
  = cons of unit -> 'a * 'a stream ;
```

3. Finite and infinite non-empty finitely-branching trees.

```
datatype 'a infFBtree
  = W of 'a * (unit -> 'a infFBtree list) ;
```


Lazy evaluation in ML

Example: Consider the following function definitions in ML.

```
fun seqpred n
  = if n = 0 then nil
    else cons( n , fn() => seqpred(n-1) ) ;
fun seqlsum( nil , l ) = 0
  | seqlsum( cons(h,t) , l )
      = h + seqlsum( t() , l ) ;

val seqpred = fn : int -> int seq
val seqlsum = fn : int seq * 'a -> int
```

$\boxed{!}$ Evaluate `seqlsum( seqpred(2) , seqpred(10000) )` and compare the process with the *call-by-name* evaluation of `lsum( pred(2) , pred(10000) )` !

Sequence manipulation

```
datatype
  'a seq = nil | cons of 'a * ( unit -> 'a seq ) ;
```

1. Head, tail, and null testing.

```
exception Empty ;
fun seqhd nil = raise Empty
  | seqhd( cons(h,t) ) = h ;
fun seqtl nil = raise Empty
  | seqtl( cons(h,t) ) = t() ;
fun seqnull nil = true
  | seqnull _ = false ;
val seqhd = fn : 'a seq -> 'a
val seqtl = fn : 'a seq -> 'a seq
val seqnull = fn : 'a seq -> bool
```


2. Constant sequences.

```
fun Kseq x = cons( x , fn() => Kseq x ) ;
val Kseq = fn : 'a -> 'a seq
```

3. Traces.

```
fun trace f s
  = case s of
      NONE => nil
    | SOME x => cons( x , fn() => trace f (f x) ) ;

fun from n = trace (fn x => SOME(x+1)) (SOME n) ;
val trace = fn :
      ('a -> 'a option) -> 'a option -> 'a seq
val from = fn : int -> int seq
```


4. Sequence display.

```
exception Negative ;
fun display n s
  = if n < 0 then raise Negative
    else if n = 0 then []
         else (seqhd s) :: display (n-1) (seqtl s) ;
val display = fn : int -> 'a seq -> 'a list
```

5. Append and shuffle.

```
fun seqappend( nil , s ) = s
  | seqappend( s , t )
      = cons( seqhd s ,
              fn() => seqappend( seqtl s , t ) ) ;
val seqappend = fn : 'a seq * 'a seq -> 'a seq
```



```
fun shuffle( nil , s ) = s
  | shuffle( s , t )
      = cons( seqhd s ,
              fn () => shuffle( t , seqtl s ) ) ;
val shuffle = fn : 'a seq * 'a seq -> 'a seq
```

6. Functionals: filter, map, fold.

```
fun seqfilter P nil = nil
  | seqfilter P s
      = let val h = seqhd s in
          if P h
          then cons( h , fn() => seqfilter P (seqtl s) )
          else seqfilter P (seqtl s)
        end ;
val seqfilter = fn : ('a -> bool) -> 'a seq -> 'a seq
```

```
fun seqmap f nil = nil
  | seqmap f s
      = cons( f( seqhd s ) ,
              fn() => seqmap f (seqtl s) ) ;
val seqmap = fn : ('a -> 'b) -> 'a seq -> 'b seq
fun seqnfold n f x s
  = if n < 0 then raise Negative
    else if n = 0 then x
    else if seqnull s then raise Empty
    else seqnfold (n-1) f ( f(seqhd s,x) ) (seqtl s) ;
val seqnfold = fn :
      int -> ('a * 'b -> 'b) -> 'b -> 'a seq -> 'b
```


Generating the prime numbers

Streams

```
datatype 'a stream = cons of unit -> 'a * 'a stream ;

fun head (cons f)
  = let val (h,_) = f() in h end ;

fun tail (cons f)
  = let val (_,t) = f() in t end ;

val head = fn : 'a stream -> 'a
val tail = fn : 'a stream -> 'a stream
```



```
fun filter P s
  = let
      fun auxfilter s
        = let
            val h = head s
          in
            if P h
            then cons( fn() => ( h , auxfilter(tail s) ) )
            else auxfilter( tail s )
          end
    in
      auxfilter s
    end ;
val filter = fn : ('a -> bool) -> 'a stream -> 'a stream
```


Sieve of Eratosthenes (I)

```
fun sieve s
  = let
      val h = head s
      val sift = filter (fn n => n mod h <> 0) ;
    in
      cons( fn() => ( h , sieve( sift (tail s) ) ) )
    end ;
val sieve = fn : int stream -> int stream
fun from n = cons( fn () => ( n , from(n+1) ) ) ;
val primes = sieve( from 2 ) ;
val from = fn : int -> int stream
val primes = cons fn : int stream
```

Sieve of Eratosthenes (II)

```
fun sieve s
= case head s of
    NONE => cons( fn() => ( NONE , sieve( tail s ) ) )
  | SOME h
    => let fun sweep s = itsweep s 1
       and itsweep s n
           = cons( fn() =>
                   if n = h then ( NONE , sweep (tail s) )
                   else ( head s , itsweep (tail s) (n+1) ) )
       in
       cons( fn() => ( SOME h , sieve( sweep (tail s) ) ) )
       end ;

val sieve = fn : int option stream -> int option stream
```


Infinite-tree manipulation

```
datatype
  'a infFBtree
    = W of 'a * ( unit -> 'a infFBtree list ) ;
```

1. Computation trees.

```
fun CT f s
  = W( s , fn() => map (CT f) (f s) ) ;

val CT = fn : ('a -> 'a list) -> 'a -> 'a infFBtree
```


2. Breadth-first traversal.

```
fun BFseq [] = nil
  | BFseq( W(x,F):: T )
      = cons( x , fn() => BFseq( T @ F() ) ) ;

val BFseq = fn : 'a infFBtree list -> 'a seq
```

3. Depth-first traversal.

```
fun DFseq [] = nil
  | DFseq( W(x,F)::T )
      = cons( x , fn() => DFseq( F() @ T ) ) ;

val DFseq = fn : 'a infFBtree list -> 'a seq
```