You Can Do More If You're Lazy!

Handout #48

CS251 Lecture 38
May 8, 2007

Franklyn Turbak

Wellesley College

Overview of Today's Lecture

- A quick introduction to Haskell, a language with lazy parameter-passing and data structures.
- Modularity problems involving lists, and their solution with lazy lists in Haskell
- Lazy trees
- Lazy data in other languages

Sample Haskell Expressions

```
Prelude> 2*(3+4)
14

Prelude> head [1,2,3,4]
1

Prelude> tail [1,2,3,4]
[2,3,4]

Prelude> map (2*) [1,2,3,4]
[2,4,6,8]

Prelude> foldr (+) 0 [1,2,3,4]
10

Prelude> take 2 [10,20,30,40,50]
[10,20]

Prelude> drop 2 [10,20,30,40,50]
[30,40,50]
```

More Haskell Expressions

```
Prelude> fst (1,2)
1

Prelude> snd (1,2)
2

Prelude> zip [1,2,3] [10,20,30,40]
[(1,10),(2,20),(3,30)]

Prelude> unzip [(1,10),(2,20),(3,30)]
([1,2,3],[10,20,30])

Prelude> (\ x -> x*x) (1+2)
9

Prelude> (\ x y -> x*x) (1+2) (3/0) -- illustrates laziness
9

Prelude> (\ x y -> x*x) (3/0) (1+2)
Program error: primDivDouble 3.0 0.0
```

Haskell Types

```
Prelude> :type map
map :: (a -> b) -> [a] -> [b]

Prelude> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> bw2

Prelude> :type zip
zip :: [a] -> [b] -> [(a,b)]

Prelude> :type unzip
unzip :: [(a,b)] -> ([a],[b])

Prelude> :type "foo"
"foo" :: String

Prelude> :type "foo" == "bar"
"foo" == "bar" :: Bool
```

Qualified Types in Haskell

```
Prelude> :type 1+2
1 + 2 :: Num a => a

Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]

Prelude> :type 1 == 2
1 == 2 :: Num a => Bool

Prelude> :type \ x -> x*x
\ x -> x * x :: Num a => a -> a
```

Haskell Definitions

In HUGS, definitions *must* be in a file, not a top-level!

```
a = 2 + 3 -- declare variable a

sq = \ x -> x * x -- sugared form: sq x = x * x

fact 0 = 1 -- recursive factorial
fact n = n * fact (n-1)

factIter n = loop n 1 -- iterative factorial
  where loop 0 ans = ans
        loop num ans = loop (num-1) (num*ans)

isEven 0 = True  -- Mutually recursive functions isEven and isOdd
isEven m = isOdd (m - 1)

isOdd 0 = False
isOdd n = isEven (n - 1)

mymap f [] = []
mymap f (x:xs) = (f x):(mymap f xs)
```

A Modularity Problem

Consider infinite sequences of integers, such as:

- *powers of 2:* 1, 2, 4, 8, 16, 32, 64, ...
- *factorials:* 1, 1, 2, 6, 24, 120, 720, ...
- *Fibonacci numbers:* 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- *primes:* 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ...

Suppose we want answers to questions like the following:

- What are the first n elements?
- What is the first element greater than 100?
- What is the (0-based) index of the first element greater than 100?
- What is the first consecutive pair whose difference is more than 25?
- What is the least index i for which the sum of elements 0 through i is more than 1000?

Challenge: can we answer these questions in a modular way?

Non-Modular Haskell Solutions

```
-- returns list of first n Fibonacci numbers
fibsPrefix :: Integer -> [Integer]
fibsPrefix num  =  gen  0 0 1
  where gen n a b =
          if n >=  num  then []
          else a : (gen  (n + 1) b (a + b))

-- returns least Fibonacci number greater than lim
leastFibGt :: Integer -> Integer
leastFibGt lim  = least 0 1
  where least a b = if a > lim then a
                    else least b (a + b)))

-- returns (0-based) index i such that first i Fibonacci
-- numbers have a sum greater than lim
fibSumIndex :: Integer -> Integer
fibSumIndex lim  = index 0 0 0 1
  where index i sum a b =
          if sum > lim  then i
          else index (i+1) (sum+a) b (a + b)
```

A More Modular Approach: Infinite Lists

Idea: Separate the generation of the sequence elements from subsequent processing. Since we don know how many elements we need, generate *all* of them — *lazily*!

```
nats = genNats 0 where genNats n = n : genNats (n + 1)
-- Can  also be written: nats  = [0..]

poss = tail nats -- the positive integers
-- Can  also be written: poss  = [1..]

powers n = genPowers 1
  where genPowers x = x : (genPowers (n * x))

facts = genFacts 1 1
  where genFacts ans n = ans : (genFacts (n * ans) (n + 1))

fibs = genFibs 0 1
  where genFibs a b = a : (genFibs b (a + b))
```

Processing Infinite Lists

Note: Assume the following functions are invoked only on infi nite lists. Then we can ignore the base case of an empty list! Each function could be extended to handle the empty list as well.

```
-- Returns a list of the first n elements of a given list.
-- (Note: the take function is part of standard Haskell)
take n (x:xs) = if (n == 0) then [] else x : (take (n-1) xs)

-- Returns first element satisfying predicate p
firstElem p (x:xs) = if (p x) then x else firstElem p xs

-- Returns first contiguous pair satisfying predicate p
firstPair  p (x:y:zs) =
  if (p(x,y)) then (x,y) else firstPair p (y:zs)

-- Returns (0-based) index of first elt satisfying pred p
indexOf p xs = ind 0 xs
  where ind i (x:xs) =
    if (p x) then i else (ind (i+1) xs)
```

Modular Infinite List Processing Examples

```
take 10 fibs
```

```
firstElem (\ x -> x > 100) (powers 2)
```

```
indexOf (\ x -> x > 1000) facts
```

```
firstPair (\ (x,y) -> (y - x) > 25) fibs
```

Scanning

Scanning accumulates partial results of `foldl` into a list.

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f ans (x:xs) = ans : scanl f (f ans x) xs

scanl (+) 0 (powers 2) -- be careful of initial zero!

-- alternative definition of facts
facts = scanl (*) 1 poss

-- Like scanl, but uses first elt as initial answer
scanl1 :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs) = scanl f x xs

indexOf (\ s -> s > 1000) (scanl1 (+) fibs)
```

Higher-order Infinite List Generation

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

-- another way to generate the nats
nats  = iterate (1 +) 0

iterate2 :: (a -> a -> a) -> a -> a -> [a]
iterate2 f x1 x2 = x1 : iterate2 f x2 (f x1 x2)

-- another way to generate the fibs
fibs = iterate2 (+) 0 1

iteratei :: (Integer -> a -> a) -> Integer -> a -> [a]
iteratei f i x = x : iteratei f (i + 1) (f i x)

-- yet another way to generate the facts
facts = iteratei (*) 1 1
```

Cyclic Definitions of Infinite Lists

```
ones = 1 : ones

-- cyclic definition of nats
nats = 0 : (map (1 +) nats

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = (f x y) : (zipWith f xs ys)

-- another cyclic definition of nats
nats = 0 : (zipWith (+) ones nats)

-- cyclic definition of facts
facts = 1 : (zipWith (*) poss facts)

-- cyclic definition of fibs
fibs = 0 : 1 : (zipWith (+) fibs (tail fibs)
```

Generating Primes

Idea: use the Sieve of Eratosthenes

```
sieve (x:xs) =
  x : (sieve (filter (\ y -> (rem  y x) /= 0) xs))

primes = sieve (tail (tail nats))
  -- start sieving at 2
```

Not only does this give an infinite list of primes, it does so efficiently by avoiding unnecessary divisions .

For more examples of lazy lists in Haskell , see Chapter 17 of Simon Thompson book *Haskell : The Craft of Functional Programming.*

Lazy Trees

Can use laziness to perform a two-pass tree walk in a single pass:

```
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving Show

addMax tr = tr'
  where (tr', m) = walk tr
        walk Leaf = (Leaf, 0)
        walk (Node l n r) = (Node l' (n + m) r', max3 n ml mr)
          where (l',ml) = walk l
                (r',mr) = walk r

max3 a b c = max a (max b c)

t = (Node (Node Leaf 1 (Node Leaf 7 Leaf)) 5 (Node Leaf 4 Leaf))

-- AddMax> addMax t
-- Node (Node Leaf 8 (Node Leaf 14 Leaf)) 12 (Node Leaf 11 Leaf)
```

See Hughes paper *Functional Programming Matters* for compelling lazy game tree example.

Streams: Lazy Lists for Scheme and Hoilec

Scheme Lists	Scheme Streams	Hoilec Lists	Hoilec Streams
`cons`	`cons-stream`	`prep`	`sprep`
`car`	`head`	`head`	`shead`
`cdr`	`tail`	`tail`	`stail`
`'()`	`the-empty-stream`	`#e`	`(sempty)`
`null?`	`null-stream`	`empty?`	`sempty?`

Note: Scheme and Hoilec streams are lazy only in their tails, *not* in their heads!

Hoilec Streams

`(sprep` E_{head} E_{tail}`)` returns a (potentially infinite) stream whose head is the value of E_{head} and whose tail is the value of E_{tail}. The evaluation of E_{tail} is delayed until it is needed.

`(shead` E_{stream}`)` returns the head element of the stream value of E_{stream}.

`(stail` E_{stream}`)` returns the tail element of the stream value of E_{stream}. This forces the computation of the delayed tail expression.

`sempty` returns the empty stream.

`(sempty?` E_{stream}`)` returns `#t` if E_{stream} is the empty stream and `#f` otherwise.

Hoilec Stream Examples I

```
;; Generate stream of integers starting with n
(def ints-from
  (fun (n)
    (sprep (ints-from (+ n 1))))) ; No base case!

;; Converts first n elements of infinite stream to a list
(def (sprefix n stream)
  (if (= n 0)
      #e
      (prep (shead stream)
            (sprefix (- n 1) (stail stream)))))

(def ones (sprep 1 ones))

(def (smap f stream)
  (if (sempty? stream)
      stream
      (sprep (f (shead stream)) (smap f (stail stream)))))

(def nats (sprep 0 (smap (fun (x) (+ x 1)) nats)))
```

Scheme Stream Examples II

```
(def smap2
  (fun (f str1 str2)
    (sprep (f (shead str1) (shead str2))
           (smap2 f (stail str1) (stail str2)))))

(def fibs
  (sprep 0
    (sprep 1
      (smap2 (fun (x y) (+ x y))
             fibs
             (stail fibs)))))
```

Can similarly translate other lazy list examples from Haskell to Hoilec and Scheme

See Section 3.5 of *Structure and Interpretation of Computer Programs (SICP)* for more Scheme stream examples.

Implementing Lazy Data in Strict Languages

- Use memoizing promises to implement lazy lists in Hoilec:

  `(delay` E`)` *desugars to* `(make-promise (fun ()` E`))`

  ```
  (def (force promise) (promise))
  ```

  `(sprep` E_1 E_2`)` *desugars to* `(list` E_1 `(delay` E_2`))`

  ```
  (define (shead stream) (nth 1 stream))

  (define (stail stream) (force (nth 2 stream)))

  (define (sempty) #e)

  (define (sempty? stream) (empty? stream))
  ```

- Can generalize this idea to handle infi nite trees.
- Can similarly implement lazy lists in OCaml.
- Lazy data is very helpful, but sometimes need even more laziness (e.g. translating `addMax` example to Scheme or OCaml).

Java Iterators

Like streams, Java's iterators can be conceptually infi nite. For example:

```
public class FibIterator implements Iterator<Integer> {

  private int a, b;

  public FibIterator () { a = 0; b = 1; }

  public boolean hasNext () { return true; }

  public Integer next () {
    int old_a = a; a = b; b = old_a + b;
    return new Integer(old_a);} // wrap int to satisfy next() spec

  public void remove () { ... ignore this ... }
}
```

- Unlike streams, enumerations are not *persistent*; can't hold on to a snapshot of the enumeration at a given point in time without copying it.
- While lazy lists are easy to adapt to trees, enumerations are inherently linear.