

The Structure of Typed
Programming Languages

Brief Intro to Haskell

CS386L- Programming Languages

Dr. Greg Lavender
Department of Computer Sciences
The University of Texas at Austin

CS386L: Lecture-09 - A Brief Intro to Haskell

Haskell

- Named after Haskell Curry
- Haskell is a non-strict (i.e., lazily evaluated) pure (non side-effecting) functional programming language
 - referentially transparent
 - lots of usage of recursive functions defined on lists
 - similar syntax to ML, but some would say an improvement
 - ML allows side effects
- both interpreters and compilers are available
 - see www.haskell.org
 - Hugs interpreter
 - GHC - Glasgow Haskell Compiler

4/5/04 2

Lazy evaluation

- Allows some interesting functions and expressions to be written
 - `:` is the list "cons" operator, which is right associative

```
ones = 1:ones
ones => 1:1:... => [1,1,1,1,1,1,1,1,1,1,...
```

- lazy lists - partial results are printed asap

```
[m .. n] | n >= m = m:[m+1..n]
           | otherwise = []
[1..n] => 1:[1+1..n] => 1:2:[2+1..n]=>...=> 1:2:...:n:[]
```

- this allows "infinite" lists

```
[0..] => [0,1,2,3,4,5,6,7,8,9,10,...]
```

A Lazy Fibonacci

- Two fibonacci functions: one consuming time & space, the other space
 - standard recursive fibonacci function

```
fibonacci :: Int -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-2) + fibonacci (n-1)
```

- infinite fibonacci sequence

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
fibn :: Int -> [Integer]
fibn n = take n fibs
```

- fibonacci function that selects nth element of the sequence

```
fib :: Int -> Integer
fib n = fibs !! n
```

Lazy evaluation

- pass arguments by name
- do not evaluate function arguments until they are needed

```
square :: Int->Int
square x = x * x
```

```
square 5 => 5 * 5
square (2+4) => (2+4) * (2+4)
```

List comprehensions

- Ways to generate lists in a set-like notation
 - list of pairs obtained from two lists

```
[(x,y) | x <- xs, y<-ys]
```

- The higher-order map function can be trivially defined as a list comprehension

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

- compare to the recursive and folded versions

```
map f (x:xs) = (f x):map f xs
map f xs = foldr (\x ys -> (f x):ys) [] xs
```

Lazy Prime Sieve

- using an infinite list and a list comprehension

```
primes :: [Integer]
primes = sieve [2 .. ] -- Sieve of Eratosthenes
sieve (x:xs) = x : sieve [y | y <- xs, (y `rem` x) /= 0]

nthprime n = take n primes
```

Folding over a list

- Fold is a higher-order function that applies a function argument successively to the elements of a list, carrying forward its previous result
 - requires three arguments
 - the function, an initial starting value and the list
 - it returns a value of the type of the initial argument
 - folding from the left
 - $\text{foldl} (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$
 - $\text{foldl} f st [a_1, a_2, \dots, a_n, a_{n-1}]$
 $\Rightarrow ((\dots(st 'f' a_1) 'f' a_2) 'f' \dots 'f' a_{n-1}) 'f' a_n$
 - folding from the right
 - $\text{foldr} f st [a_1, a_2, \dots, a_n, a_{n-1}]$
 $\Rightarrow a_1 'f' (a_2 'f' \dots 'f' (a_{n-1} 'f' (a_n 'f' st)) \dots)$

Foldl vs Foldr

- what are the differences in computing $n!$
 - $\text{foldr } (*) 1 [1..n] \Rightarrow (1*(2*(3*(4*(\dots n*1)\dots))))$
 - $\text{foldl } (*) 1 [1..n] \Rightarrow ((\dots (((1*1)*2)*3)\dots)*n)$
- note that `foldr` requires the whole expression to be formed before it can do its computation, required $O(n)$ space
- what about `foldl`?
 - `foldl` is not strict in its second argument, but we can force this using the function
 - `strict :: (a->b) -> a -> b`
 - `strict f x = seq x (f x)`
 - where `seq :: a -> b -> b` is a predefined Haskell function such that `seq x y` evaluates `x` before returning `y`

4/5/04

11

A Strict Foldl

- A revised factorial that used $O(1)$ space
 - define a version of fold that uses the strict function:


```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f st []      = st
foldl' f st (x:xs) = strict (foldl' f) (f st x) xs
```
 - then fact becomes


```
fact = foldl' (*) 1 [1..n]
      => foldl' (*) 1 [2..n]
      => foldl' (*) 2 [3..n]
      => foldl' (*) 6 [4..n]
      => ...
```
 - note that `foldl'` consumes an entire list before giving a result, so it is useless if applied to infinite lists as it will never produce a result.

4/5/04

12

Examples using Foldl

- summing a list of integers
 - note that the starting value is the additive identity

```
sum :: [Int] -> Int
sum xs = foldl (+) 0 xs
```

- computing the length of a list
 - we use an anonymous lambda with a "don't care" 2nd argument since we don't care what is in the list

```
length :: [a] -> Int
length xs = foldl (\n _ -> n+1) 0 xs
```

- Note that both functions can be curried

```
sum = foldl (+) 0
length = foldl (\n _ -> n+1) 0
```

4/5/04

13

Fold is like iteration

- a factorial function
 - note that the starting value is the multiplicative identity
 - also note that computing factorial using a fold is not recursive, but iterative

```
fact 0 = 1
fact n = foldl (*) 1 [2..n]
```

4/5/04

14

Using fold efficiently

- how would you (naively) compute the mean of a list of elements?

```
mean :: (Num a) => [a] -> Int
mean xs = sum xs / length xs
```

- this approach has the drawback of traversing the list twice
 - how can we improve on this so that we only traverse the list once?
 - note that the length of the list can be computed while we are computing the sum

A More Efficient Mean Function

- Consider a function mean that computes the mean value of a list using foldl
 - foldl takes a function, an initial value, and a list as arguments
 - note that suml is a curried function that returns a pair (s,l) where s is the sum and l is the length

```
suml :: (Num a) => [a] -> (a, Int)
suml = foldl (\(s,l) x -> (s+x,l+1)) (0,0)
```

```
mean xs = (fst p) / fromIntegral (snd p)
          where p = suml xs
```

An Even More Efficient Mean Function

- A strict version of the mean function
 - using a strict version of foldl

```
suml' :: (Num a) => [a] -> (a, Int)
suml' = foldl' (\(s,l) x -> (s+x,l+1)) (0,0)
```

```
mean' xs = (fst p) / fromIntegral (snd p)
  where p = suml' xs
```

Homework Assignment

- Implement Horner's method for evaluating a polynomial using an appropriate version of the fold function
- Horner's rule for evaluating a polynomial using recursion

$$p_n(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n$$

where $p_n(x) = h_n(x)$
 $h_0(x) = a_0$
 $h_i(x) = h_{i-1}(x) * x + a_i, i = 1, 2, \dots, n$

- The horner function type is:
 - `horner :: (Num a) => a -> [a] -> a`
 - i.e., takes a numeric value of type `a` (the argument `x`) and a list of coefficients, each of of type `a`, and computes the value of the polynomial.