Modern Programing Languages

Introduction to Haskell (3)

Wamberto Vasconcelos
w.w.vasconcelos@abdn.ac.uk

2016–2017

Plan of Lecture

1. Function Composition
2. Hamming Numbers
3. Prime Number Generation
4. List Comprehensions

Function Composition

- Partial application can be combined with the function composition operator to create expressions denoting nameless functions.
- This is the essence of functional programming:
  - put together a lot of functions to compose a program
  - then apply it to arguments (often data structures)
  - and evaluate it.
- Example: `ff = (foldr (+) 0).(map sqrt)`

  ```
  ff [1,4,16,121]
        ⇝ (foldr (+) 0) (map sqrt [1,4,16,121])
        ⇝ (foldr (+) 0 [1,2,4,11])
        ⇝ (1+2+4+11+0)
        ⇝ 18
  ```

Hamming Numbers (1)

- As a larger example, consider the lazy evaluation of the infinite Hamming series, named after Dr. Hamming of Bell Labs:
    - a series of integers in ascending order;
    - the first number in the series is 1;
    - if x is a member of the series, then so is $2 \times x$, $3 \times x$ and $5 \times x$
- We do it by merging 3 lazily evaluated ordered streams!

Hamming Numbers (2)

- Solution: (base cases of `mergeh` are missing!!)

```
mergeh (x:xs) (y:ys)
   | x < y      = x:(mergeh xs (y:ys))
   | x == y     = x:(mergeh xs ys)
   | otherwise = y:(mergeh (x:xs) ys)

ham = 1:mergeh (map (*2) ham)
               (mergeh (map (*3) ham)
                       (map (*5) ham))
```

- That is,

```
ham = 1:(foldr mergeh [] [ (map (*2) ham),
                           (map (*3) ham),
                           (map (*5) ham)])
```

Prime Number Generation

- The early members of a list of primes up to p are used to divide and test potential primes up to $p * p$ (same dodge as used in all_fib):

```
primes = 2:filter (isprime primes) [3..]
   where  isprime [] n = True
          isprime (p:ps) n
             | (p * p) > n   = True
             | n mod p == 0 = False
             | otherwise     = isprime ps n
```

- `isprime ps n` assumes `ps` is a list of primes in ascending order from 2 up and tests that none of them divide `n` (returns true)
- E.g. `isprime [2,3,5,7] 37` $\rightsquigarrow$ `True`

List Comprehensions (1)

- Haskell provides a special notation for describing lists in terms of other lists, called list comprehensions.
- Here is an example, which describes the list containing the squares of all even numbers between 1 and 20:

  ```
  [square x | x <- [1..20], even x]
  ```

- A list comprehension consists of a pattern, on the left of the `|` symbol, and a series of generators and restrictions, on its right.
- The pattern is an arbitrary Haskell expression.
- A generator has the form "Variable `<-` List".
- A restriction is any Boolean-valued expression.
- Generators and restrictions are separated by "`,`" symbols.

List Comprehensions (2)

- Function to create a list with copies of a number:

```
repeat ::  Int -> a -> [a]
repeat n x = [ x | y <- [1..n]]
```

- Cross-product of two sets:

```
cross_product ::  [a] -> [b] -> [(a,b)]
cross_product xs ys = [ (x, y) | x <- xs, y <- ys]
```

- An alternative definition of the prime numbers:

```
primes ::  [Int]
isprime ::  [Int] -> Int -> Bool
primes = [p | p <- [2..], isprime [2..(p-1)] p]
isprime ps p = [d | d <- ps, p mod d = 0] = []
```

List Comprehensions (3)

- Another example: an alternative definition of primes.
- At each level of recursion another sieve is inserted to filter the stream of potential primes by the next prime!!

```
primes2 = sieve [2..]

sieve (p:ps) =
      p:(sieve [n | n <- ps, (mod n p) /= 0])
```

- Bear in mind that the definition generates all primes, so we have to use it with some form of "restraint".
- For instance, one way to use the definition above is:

```
take 10 primes2 ⇝ [2,3,5,7,11,13,17,19,23,29]
```

List Compr. & Higher-Order Functions (1)

- The list comprehensions can be defined in terms of `map`, `filter` and `foldr`:

```
[(f p) | p <- ps] ≡ map f ps

[exp p | p <- ps, pred p,...] ≡
      [exp p | p <- (filter pred ps),...]

[exp p y | p <- ps, y <- ys,...] ≡
      concat [[exp p y| y <- ys,...]  | p <- ps]
          where concat = foldr ++ []
```

List Compr. & Higher-Order Functions (2)

- N.B.: `concat` flattens a list of lists, that is,

```
concat ::  [[a]] -> [a]
concat xs = foldr ++ [] xs
```

- For instance,

```
foldr ++ [] [[1,2], [4,5], [7,8,10]]
   ⇝ [1,2] ++ [4,5] ++ [7,8,10] ++ []
   ⇝ [1,2,4,5,7,8,10]
```

- There are also some useful rules for how `concat` interacts with `map` and `filter`:

```
(filter p).concat  ≡  concat.(map (filter p))
   (map f).concat  ≡  concat.(map (map f))
```