1.3 Programming with Lazy Evaluation

Dienstag, 3. Mai 2016   14:15

Haskell uses a non-strict evaluation strategy:

- in general, one uses a leftmost-outermost strategy
- pre-defined arithmetic operators (+, -, ...) and
  — " — comparison operators (>, <, ...) require strict evaluation of their arguments
- for pattern matching, arguments are evaluated a few steps until one can decide whether pattern matches

```haskell
infinity :: Int
infinity = infinity + 1
```

```haskell
mult :: Int -> Int -> Int
mult 0 y = 0
mult x y = x * y
```

`mult 0 infinity` terminates and results in 0

`0 * infinity` does not terminate

Haskell's evaluation strategy can be used for programs that operate on infinite data objects.

```haskell
from :: Num a => a -> [a]
from x = x : from (x+1)
```

```haskell
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs) = if n <= 0 then []
                else x : take (n-1) xs
```

`from x` computes $[x, x+1, x+2, \ldots]$

Can also be written as `[x ..]`

take n $[x_1, \ldots, x_n, x_{n+1}, \ldots] =$

$[x_1, \ldots, x_n]$.

`take 2 (from 5)`

We start evaluating from 5 to find

take 2 (from 5)

= take 2 (5 : from 6)

= 5 : take 1 (from 6)

= 5 : take 1 (6 : from 7)

= 5 : 6 : take 0 (from 7)

= 5 : 6 : [ ]

= [5, 6]

We start evaluating from 5 to find out whether pattern [ ] matches

One can compute with infinite lists!

Practical example for programming with infinite lists:

Computing prime numbers (by a variant of the "Sieve of Eratosthenes")

General strategy for programming with infinite lists:

- generate a (potentially infinite) list with approximations to the solution
- then repeatedly filter out the real solutions from this list.

Example: Compute the infinite list of all prime numbers

1. Compute the list of all natural numbers starting with 2.
2. Mark the first unmarked number in the list.
3. Remove all multiples of the just-marked number.
4. Go back to step 2.

from 2 or [2 ..]

[2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ...]

```
drop_mult :: Int -> [Int] -> [Int]
drop_mult x xs = [y | y <- xs, y `mod` x /= 0]
```

drop_mult x xs removes all multiples of x from the list xs

Thus: drop_mult 2 [3 ..] = [3, 5, 7, ...]

<u>dropall</u> : keeps the first element, but removes all multiples of it

Then: it keeps the first element of the remaining list, but removes all multiples of it

etc.

```
dropall :: [Int] -> [Int]
dropall (x:xs) = x : dropall (drop_mult x xs)
```

```
primes :: [Int]
primes = dropall [2 ..]
```

take 10 primes : results in the first 10 prime numbers

takeWhile (<100) primes : results in all primes < 100

[x | x <- primes, x < 100] : does not terminate

Circular Data Objects

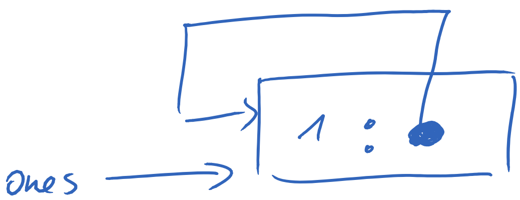
Some infinite objects can be represented in a "circular" way:

→ very efficient representation in terms of space

→ very efficient algorithms

```
ones :: [Int]
ones = 1 : ones
```

If a constant appears in the right-hand side of its defining equation, then it is represented as a circular object:



<u>Example</u> for efficient computation with circular objects:

Hamming-Problem

Goal: Compute a list with the following properties:

1. The list is sorted in ascending order and it has no duplicates.
2. The list starts with 1.
3. If the list contains a number x, then it also contains 2*x, 3*x, 5*x.
4. Apart from that, the list does not contain any other numbers.

[1, 2, 3, 4, 5, 6, 8, 9, 10, ...]

We need an auxiliary function to merge sorted (possibly infinite) lists:

```
mer :: Ord a => [a] -> [a] -> [a]
mer (x:xs) (y:ys) | x<y       = x : mer xs (y:ys)
                  | x==y      = x : mer xs ys
                  | otherwise = y : mer (x:xs) ys
```

```
hamming :: [Int]
hamming = 1 : mer ( map (2*) hamming )
                  ( mer (map (3*) hamming)
                        (map (5*) hamming))
```

Computing take n hamming takes O(n) steps.