Informatics 1
Introduction to Computation
Lecture 14

Laziness, Higher-order, and Sorting

Don Sannella
University of Edinburgh
Part I

The importance of being lazy
Searching for the first odd number

```haskell
ho :: Int -> [Int]
ho n = (take 1 . filter odd) [0..n]

comp :: Int -> [Int]
comp n = take 1 [ x | x <- [0..n], odd x ]

rec :: Int -> [Int]
rec n = helper 0
  where
    helper :: Int -> [Int]
    helper i | i > n        = []
              | odd i         = [i]
              | otherwise     = helper (i+1)
```
Quickcheck

prop_odd :: Int -> Bool
prop_odd n = a == b && b == c
  where
    a = ho n
    b = comp n
    c = rec n

[1 of 1] Compiling Main
Ok, one module loaded.
> quickCheck prop_odd
+++ OK, passed 100 tests.
Timing

> :set +s
> ho 1000000
[1]
(0.00 secs, 64,776 bytes)
> comp 1000000
[1]
(0.00 secs, 64,984 bytes)
> rec 1000000
[1]
(0.00 secs, 65,168 bytes)
How it works: rec

```
rec :: Int -> [Int]
rec n = helper 0
  where
    helper :: Int -> [Int]
    helper i | i > n    = []
              | odd i     = [i]
              | otherwise = helper (i+1)
```

```
rec 1000000
=
  helper 0
=
  helper 1
=
  [1]
```
How it works: ho

ho :: Int -> [Int]
ho n = (take 1 . filter odd) [0..n]

ho 1000000
=
(take 1 . filter odd) [0..1000000]
=
take 1 (filter odd [0..1000000])
=
take 1 (filter odd (0 : [1..1000000]))
=
take 1 (filter odd (1 : [2..1000000]))
=
take 1 (1 : filter odd [2..1000000])
=
1 : take 0 (filter odd [2..1000000])
=
1 : []
Part II

Sum of odd squares
three ways
Sum of odd squares

```haskell
ho :: Int -> Int
ho n = (foldl (+) 0 . map (^2) . filter odd) [0..n]

comp :: Int -> Int
comp n = sum [ x^2 | x <- [0..n], odd x ]

rec :: Int -> Int
rec n = helper 0 0
  where
    helper :: Int -> Int -> Int
    helper i a | i > n = a
                | odd i = helper (i+1) (a + i^2)
                | otherwise = helper (i+1) a
```
Quickcheck

prop_sqr :: Int -> Bool
prop_sqr n = a == b && b == c
  where
    a = ho n
    b = comp n
    c = rec n

Ok, one module loaded.
> quickCheck prop_sqr
+++ OK, passed 100 tests.
Runtimes in ghci

> :set +s
> ho 1000000
166666666666500000
(0.43 secs, 596,687,792 bytes)
> comp 1000000
166666666666500000
(0.67 secs, 628,685,832 bytes)
> rec 1000000
166666666666500000
(1.02 secs, 692,881,968 bytes)
The Moral

Usually coding involves tradeoffs: 

*simple* and *slow*

vs.

*complex* and *fast*.

The big win is when you can find a way to be both *simple* and *fast*.
Part III

Sorting
three ways
Insertion sort

\[
\text{foldr} :: (a \to b \to b) \to b \to [a] \to b
\]
\[
\text{foldr} \ f \ e \ [] \ = \ e
\]
\[
\text{foldr} \ f \ e \ (x:xs) \ = \ x \ \text{\textquotesingle}f\text{\textquotesingle} \ \text{foldr} \ f \ e \ xs
\]
\[
\text{foldr} \ f \ e \ [x,y,z] \ = \ (x \ \text{\textquotesingle}f\text{\textquotesingle} \ (y \ \text{\textquotesingle}f\text{\textquotesingle} \ (z \ \text{\textquotesingle}f\text{\textquotesingle} \ e)))
\]

\[
\text{isort} :: \text{Ord} \ a \Rightarrow [a] \to [a]
\]
\[
isort = \text{foldr} \ \text{insert} \ []
\]
\[
\text{where}
\]
\[
\text{insert} :: \text{Ord} \ a \Rightarrow a \to [a] \to [a]
\]
\[
\text{insert} \ x \ [] \ = \ [x]
\]
\[
\text{insert} \ x \ (y : ys) \ | \ x \ \text{\textless=} \ y \ = \ x : y : ys
\]
\[
\text{\textbar} \ \text{otherwise} \ = \ y : \text{insert} \ x \ ys
\]
Quicksort

```haskell
qsort :: Ord a => Int -> [a] -> [a]
qsort k xs | length xs <= k = isort xs
qsort k (y:xs) =
  qsort k [ x | x <- xs, x < y ]
  ++ [ y ] ++
  qsort k [ x | x <- xs, x >= y ]
```
Merge sort

```haskell
msort :: Ord a => Int -> [a] -> [a]
msort k xs | length xs <= k = isort xs
            | otherwise        = merge (msort k (take m xs))
                              (msort k (drop m xs))

    where
m = length xs `div` 2
merge :: Ord a => [a] -> [a] -> [a]
merge xs []      = xs
merge [] ys      = ys
merge (x:xs) (y:ys) | x <= y   = x : merge xs (y:ys)
                      | otherwise = y : merge (x:xs) ys
```
Why quicksort and mergesort are $O(n \log n)$

- $n$  \hspace{1em} \text{number of elements to be sorted}
- $k$  \hspace{1em} \text{cutoff size}
Part IV

A few graphs
Time in seconds vs size of list (random list)

Quicksort
Merge sort
Time in seconds vs size of list (random list)

- Quicksort
- Merge sort
- Insertion sort
Time is seconds vs size of list (sorted list)

Quicksort Merge sort Insertion sort
Time in seconds vs Cutoff Size (random list of length 1024)

Quicksort
Merge Sort