Informatics 1A
Functional Programming Lectures 12–13

Data Representation and Data Abstraction

Don Sannella
University of Edinburgh
Part I

2023 Inf1A FP Competition
2023 Inf1A FP Competition

- Prizes: Amazon vouchers. And glory!
- Number of prizes depend on number and quality of entries.
- Write a Haskell program with interesting graphics. Be creative!
- Some entries from a previous year are online:
- Sponsored by Galois (galois.com)
- Submit code and image(s), list everyone who contributed, explain how to run. (Using process similar to tutorial submission — details to come.)
- Submission deadline: noon, Monday 20 November
- Prizes awarded: 2pm Tuesday 28 November
Part II

Efficiency and O-notation
Premature optimization is the root of all evil.

— Donald Knuth —
Premature optimization is the root of all evil in programming.

— Tony Hoare —
Left vs. Right

Let $xss = [x_{s1}, \ldots, x_{sm}]$ consist of $m$ lists each of length $n$.

Associated to the left, $\text{foldl} \ (++) \ [\ ] \ xss$.

$$(((\[ \] ++ x_{s1}) ++ x_{s2}) ++ x_{s3}) \cdots ++ x_{sm}$$

Number of steps

$0 + n + 2n + 3n + \ldots + (m - 1)n = O(m^2n)$

$m$ times

Associated to the right, $\text{foldr} \ (++) \ [\ ] \ xss$.

$$x_{s1} ++ \cdots (x_{sm-2} ++ (x_{sm-1} ++ (x_{sm} ++ [])))$$

Number of steps

$$n + n + n + \cdots + n = O(mn)$$

$m$ times

steps. When $m = 1000$, the first takes a thousand times as long as the second!
$t = n$ vs $t = n^2$
\( t = 2n \) vs \( t = 0.5n^2 \)
Big-O notation

**Definition** We say $f$ is $O(g)$ when $g$ is an upper bound for $f$, for big enough inputs. To be precise, $f$ is $O(g)$ if there are constants $c$ and $m$ such that $f(n) \leq cg(n)$ for all $n \geq m$.

For instance: $2n + 10$ is $O(n)$ because $2n + 10 \leq 4n$ for all $n \geq 5$. 
Big-O notation

**Definition** We say $f$ is $O(g)$ when $g$ is an upper bound for $f$, for big enough inputs. To be precise, $f$ is $O(g)$ if there are constants $c$ and $m$ such that $f(n) \leq cg(n)$ for all $n \geq m$.

For instance: $2n + 10$ is $O(n)$ because $2n + 10 \leq 4n$ for all $n \geq 5$.

**Constant factors don’t matter**

$O(n) = O(an + b)$, for any $a$ and $b$

$O(n^2) = O(an^2 + bn + c)$, for any $a$, $b$, and $c$

$O(n^3) = O(an^3 + bn^2 + cn + d)$, for any $a$, $b$, $c$, and $d$

$O(\log_2(n)) = O(\log_{10}(n))$
$O(n), O(n^2), O(n^3), O(n^4)$
$O(\log n), O(n), O(n \log n), O(2^n)$
\[ O(\log n), O(n \log n), O(2^n) \]

\[ O(\log n) \] “logarithmic”: divide and conquer search algorithms

\[ O(n) \] “linear”: normal list search algorithms

\[ O(n \log n) \]: sorting algorithms

\[ O(2^n) \] “exponential”: tautology checking
Part III

Sets as lists
module List
  (Set, empty, insert, set, element, equal) where

import Test.QuickCheck

type Set a = [a]

empty :: Set a
empty = []

insert :: a -> Set a -> Set a
insert x xs = x:xs

set :: [a] -> Set a
set xs = xs
List.hs (2)

\[
\text{element} :: \text{Eq } a \Rightarrow a \rightarrow \text{Set } a \rightarrow \text{Bool} \\
x \ '\text{element} ' \ xs \ = \ x \ '\text{elem} ' \ xs
\]

\[
\text{equal} :: \text{Eq } a \Rightarrow \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Bool} \\
x s \ '\text{equal} ' \ ys \ = \ x s \ '\text{subset} ' \ ys \ \&\& \ ys \ '\text{subset} ' \ x s \\
\text{where} \\
x s \ '\text{subset} ' \ ys \ = \ \text{and} \ [ \ x \ '\text{elem} ' \ ys \ | \ x \leftarrow x s ]
\]
prop_element :: [Int] -> Bool
prop_element ys = and [ x `element` s == odd x | x <- ys ]
where
  s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_element

-- Prelude List> check
-- +++ OK, passed 100 tests.
Part IV

Sets as *ordered* lists
module OrderedList
    (Set, empty, insert, set, element, equal) where

import Data.List (nub, sort)
import Test.QuickCheck

type Set a = [a]

invariant :: Ord a => Set a -> Bool
invariant xs =
    and [ x < y | (x,y) <- zip xs (tail xs) ]
empty :: Set a
empty = []

insert :: Ord a => a -> Set a -> Set a
insert x [] = [x]
insert x (y:ys) | x < y = x : y : ys
| x == y = y : ys
| x > y = y : insert x ys

set :: Ord a => [a] -> Set a
set xs = nub (sort xs)
element :: Ord a => a -> Set a -> Bool
x 'element' [] = False
x 'element' (y:ys) | x < y = False
|     | x == y = True
|     | x > y = x 'element' ys

equal :: Eq a => Set a -> Set a -> Bool
xs 'equal' ys = xs == ys
prop_invariant :: [Int] -> Bool
prop_invariant xs = invariant s
    where
        s = set xs

prop_element :: [Int] -> Bool
prop_element ys =
    and [ x `element` s == odd x | x <- ys ]
    where
        s = set [ x | x <- ys, odd x ]

check =
    quickCheck prop_invariant >>
    quickCheck prop_element

Prelude OrderedList> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
Part V

Sets as ordered trees
module Tree
    (Set (Nil, Node), empty, insert, set, element, equal) where

import Test.QuickCheck

data Set a = Nil | Node (Set a) a (Set a)

list :: Set a -> [a]
list Nil = []
list (Node l x r) = list l ++ [x] ++ list r

invariant :: Ord a => Set a -> Bool
invariant Nil = True
invariant (Node l x r) =
    invariant l && invariant r &&
    and [ y < x | y <- list l ] &&
    and [ y > x | y <- list r ]
empty :: Set a
empty = Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil = Node Nil x Nil
insert x (Node l y r)
  | x == y = Node l y r
  | x < y = Node (insert x l) y r
  | x > y = Node l y (insert x r)

set :: Ord a => [a] -> Set a
set = foldr insert empty
element :: Ord a => a -> Set a -> Bool
x `element` Nil = False
x `element` (Node l y r)
  | x == y    = True
  | x < y     = x `element` l
  | x > y     = x `element` r

equal :: Ord a => Set a -> Set a -> Bool
s `equal` t = list s == list t
prop_invariant :: [Int] -> Bool
prop_invariant xs = invariant s
  where
    s = set xs

prop_element :: [Int] -> Bool
prop_element ys =
  and [ x 'element' s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude Tree> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
Part VI

Sets as *balanced* trees
A balanced binary tree in real life
A balanced binary tree, Computer Science version
module BalancedTree
    (Set (Nil, Node), empty, insert, set, element, equal) where

import Test.QuickCheck

type Depth = Int

data Set a = Nil | Node (Set a) a (Set a) Depth

node :: Set a -> a -> Set a -> Set a
    node l x r = Node l x r (1 + (depth l 'max' depth r))

depth :: Set a -> Int
    depth Nil = 0
    depth (Node _ _ _ d) = d
list :: Set a -> [a]
list Nil = []
list (Node l x r _) = list l ++ [x] ++ list r

invariant :: Ord a => Set a -> Bool
invariant Nil = True
invariant (Node l x r d) =
  invariant l && invariant r &&
  and [ y < x | y <- list l ] &&
  and [ y > x | y <- list r ] &&
  abs (depth l - depth r) <= 1 &&
  d == 1 + (depth l `max` depth r)
empty :: Set a
empty = Nil

insert :: Ord a => a -> Set a -> Set a
insert x Nil = node empty x empty
insert x (Node l y r _) 
  | x == y = node l y r
  | x < y = rebalance (node (insert x l) y r)
  | x > y = rebalance (node l y (insert x r))

set :: Ord a => [a] -> Set a
set = foldr insert empty
Rebalancing

Node (Node a x b) y c  -->  Node a x (Node b y c)

Node (Node a x (Node b y c) z d)  -->  Node (Node a x b) y (Node c z d)
rebalance :: Set a -> Set a
rebalance (Node (Node a x b _) y c _) |
  depth a >= depth b && depth a > depth c
  = node a x (node b y c)
rebalance (Node a x (Node b y c _) _) |
  depth c >= depth b && depth c > depth a
  = node (node a x b) y c
rebalance (Node (Node a x (Node b y c _) _) z d _)
  |
  depth (node b y c) > depth d
  = node (node a x b) y (node c z d)
rebalance (Node a x (Node (Node b y c _) z d _) _)
  |
  depth (node b y c) > depth a
  = node (node a x b) y (node c z d)
rebalance a = a
element :: Ord a => a -> Set a -> Bool
x `element` Nil  =  False
x `element` (Node l y r _)    
  | x == y    =  True
  | x < y     =  x `element` l
  | x > y     =  x `element` r

equal :: Ord a => Set a -> Set a -> Bool
s `equal` t  =  list s == list t
prop_invariant :: [Int] -> Bool
prop_invariant xs = invariant s
  where
    s = set xs

prop_element :: [Int] -> Bool
prop_element ys =
  and [ x `element` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude BalancedTree> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
module BalancedTreeTest where

import BalancedTree

test :: Int -> Bool
test n =
    s `equal` t
    where
        s = set [1,2..n]
        t = set [n,n-1..1]

badtest :: Bool
badtest =
    s `equal` t
    where
        s = set [1,2,3]
        t = (Node Nil 1 (Node Nil 2 (Node Nil 3 Nil 1) 2) 3)
        -- breaks the invariant!
Part VII

Complexity, revisited
## Summary

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>set</th>
<th>element</th>
<th>equal</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>OrderedList</td>
<td>$O(n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Tree</td>
<td>$O(\log n)^*$</td>
<td>$O(n \log n)^*$</td>
<td>$O(\log n)^*$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)^\dagger$</td>
<td>$O(n^2)^\dagger$</td>
<td>$O(n)^\dagger$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BalancedTree</td>
<td>$O(\log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

* average case / † worst case
Part VIII

Data Abstraction
module ListAbs
  (Set, empty, insert, set, element, equal) where
import Test.QuickCheck

data Set a = MkSet [a]

empty :: Set a
empty = MkSet []

insert :: a -> Set a -> Set a
insert x (MkSet xs) = MkSet (x:xs)

set :: [a] -> Set a
set xs = MkSet xs
element :: Eq a => a -> Set a -> Bool
  x `element` (MkSet xs) = x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
MkSet xs `equal` MkSet ys =
  xs `subset` ys && ys `subset` xs

  where
  xs `subset` ys = and [ x `elem` ys | x <- xs ]
prop_element :: [Int] -> Bool
prop_element ys =
    and [ x 'element' s == odd x | x <- ys ]
where
    s = set [ x | x <- ys, odd x ]

check =
    quickCheck prop_element

-- Prelude ListAbs> check
-- +++ OK, passed 100 tests.
module ListAbsTest where
import ListAbs

test :: Int -> Bool
test n =
  s 'equal' t
where
  s = set [1,2..n]
t = set [n,n-1..1]

-- Following no longer type checks!
-- breakAbstraction :: Set a -> a
-- breakAbstraction = head
Hiding—the secret of abstraction

```haskell
module ListAbs(Set,empty,insert,set,element,equal)

> ghci ListAbs.hs
Ok, modules loaded: SetList, MainList.
> let s0 = set [2,7,1,8,2,8]
> let MkSet xs = s0 in xs
Not in scope: data constructor ‘MkSet’

VS.

module ListUnhidden(Set(MkSet),empty,insert,element,equal)

> ghci ListUnhidden.hs
> let s0 = set [2,7,1,8,2,8]
> let MkSet xs = s0 in xs
[2,7,1,8,2,8]
> head xs
2
```
Hiding—the secret of abstraction

```haskell
module TreeAbs(Set,empty,insert,set,element,equal)

ghci TreeAbs.hs
Ok, modules loaded: SetList, MainList.
ghci TreeAbs.hs
> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
Not in scope: data constructor 'Node', 'Nil'
```

**VS.**

```haskell
module TreeUnabs(Set (Node,Nil),empty,insert,element,equal)

ghci TreeUnabs.hs
> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
> invariant s0
False
```
Preserving the invariant

```
module TreeAbsInvariantTest where
import TreeAbs

prop_invariant_empty = invariant empty

prop_invariant_insert x s =
    invariant s ==> invariant (insert x s)

prop_invariant_set xs = invariant (set xs)

check =
    quickCheck prop_invariant_empty >>
    quickCheck prop_invariant_insert >>
    quickCheck prop_invariant_set

-- Prelude TreeAbsInvariantTest> check
-- +++ OK, passed 1 tests.
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```
It's mine!