Informatics 1A

Functional Programming Lecture 12

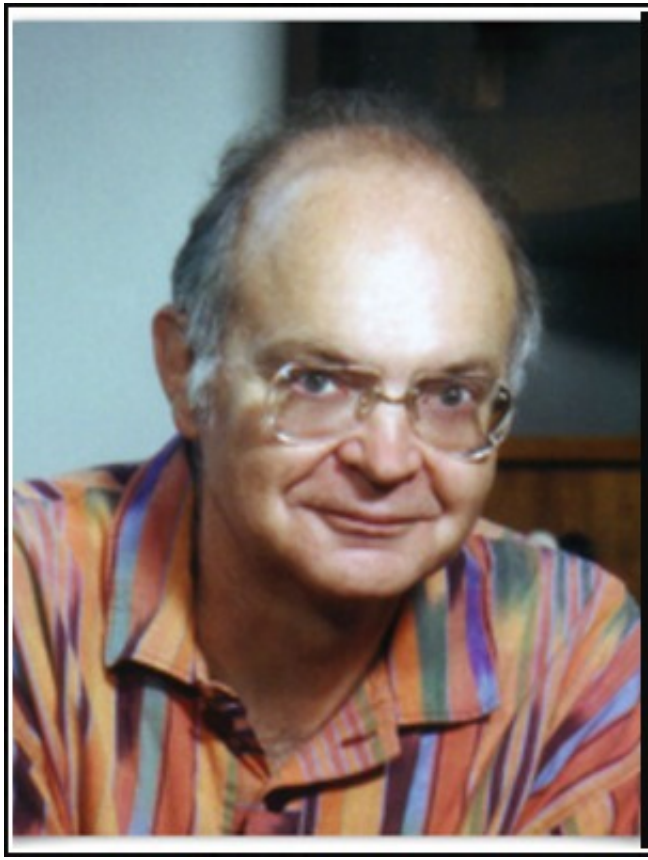
Data Representation and Data Abstraction

Ohad Kammar

The University of Edinburgh

Part I

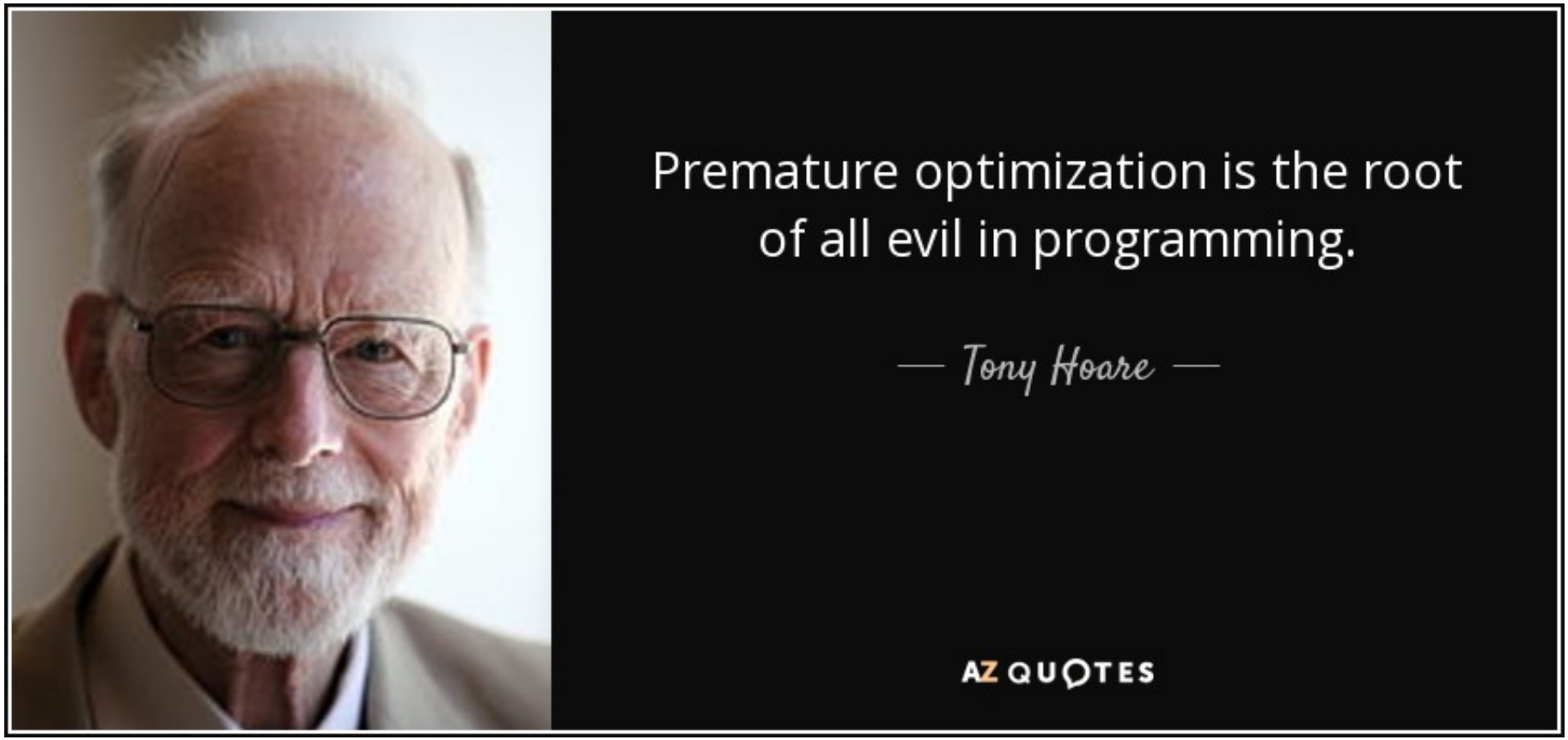
Efficiency and O-notation



Premature optimization is the root of all evil.

— Donald Knuth —

AZ QUOTES


Premature optimization is the root of all evil in programming.
— Tony Hoare —
AZ QUOTES

Left vs. Right

Let $xss = [xs_1, \ldots, xs_m]$ consist of m lists each of length n.

Associated to the left, `foldl (++) [] xss`.

$$((([] \mathbin{++} xs_1) \mathbin{++} xs_2) \mathbin{++} xs_3) \cdots \mathbin{++} xs_m$$

Number of steps

$$\underbrace{0 + n + 2n + 3n + \ldots + (m-1)n}_{m \text{ times}} = O(m^2 n)$$

Associated to the right, `foldr (++) [] xss`.

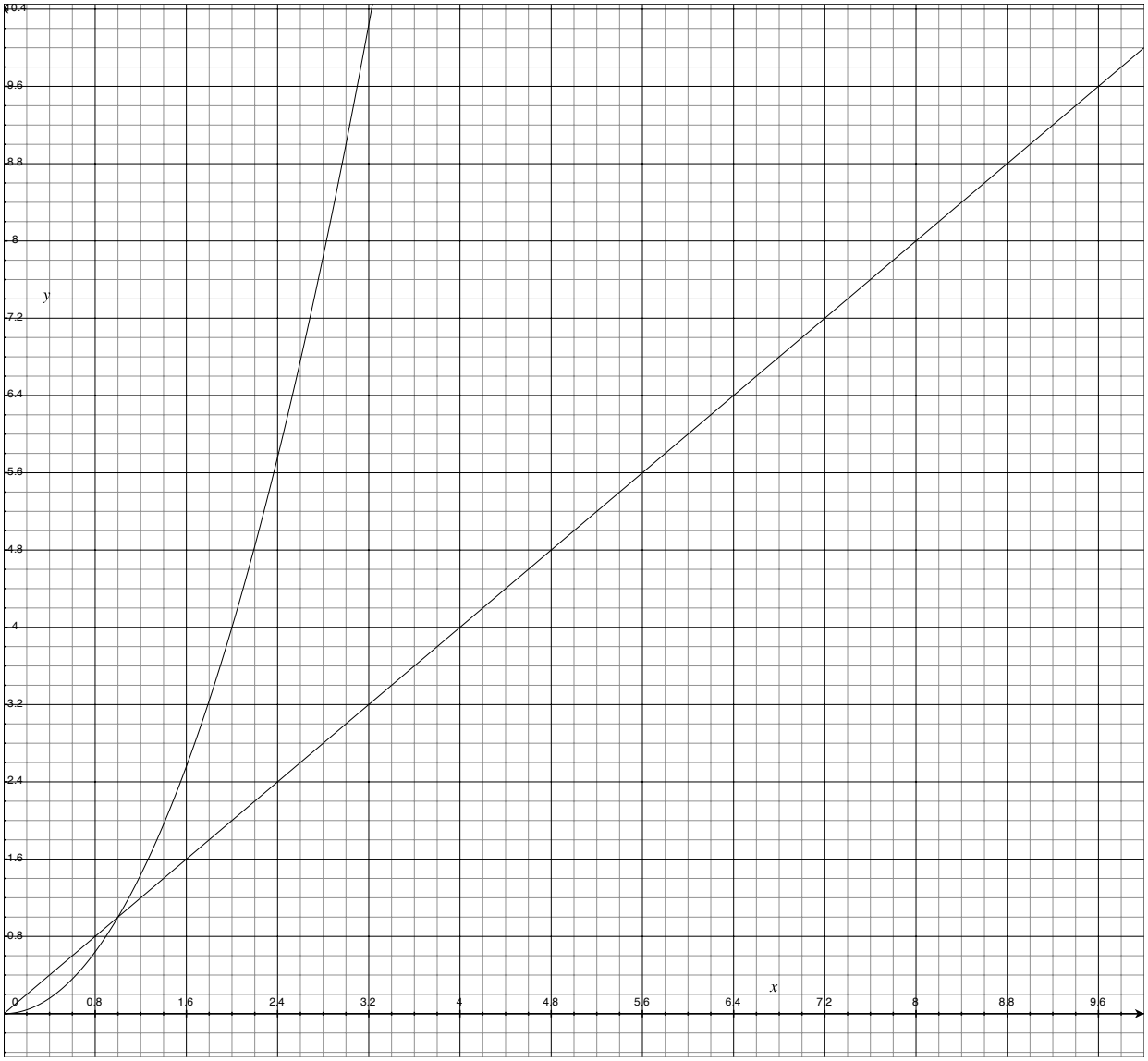
$$xs_1 \mathbin{++} \cdots (xs_{m-2} \mathbin{++} (xs_{m-1} \mathbin{++} (xs_m \mathbin{++} [])))$$

Number of steps

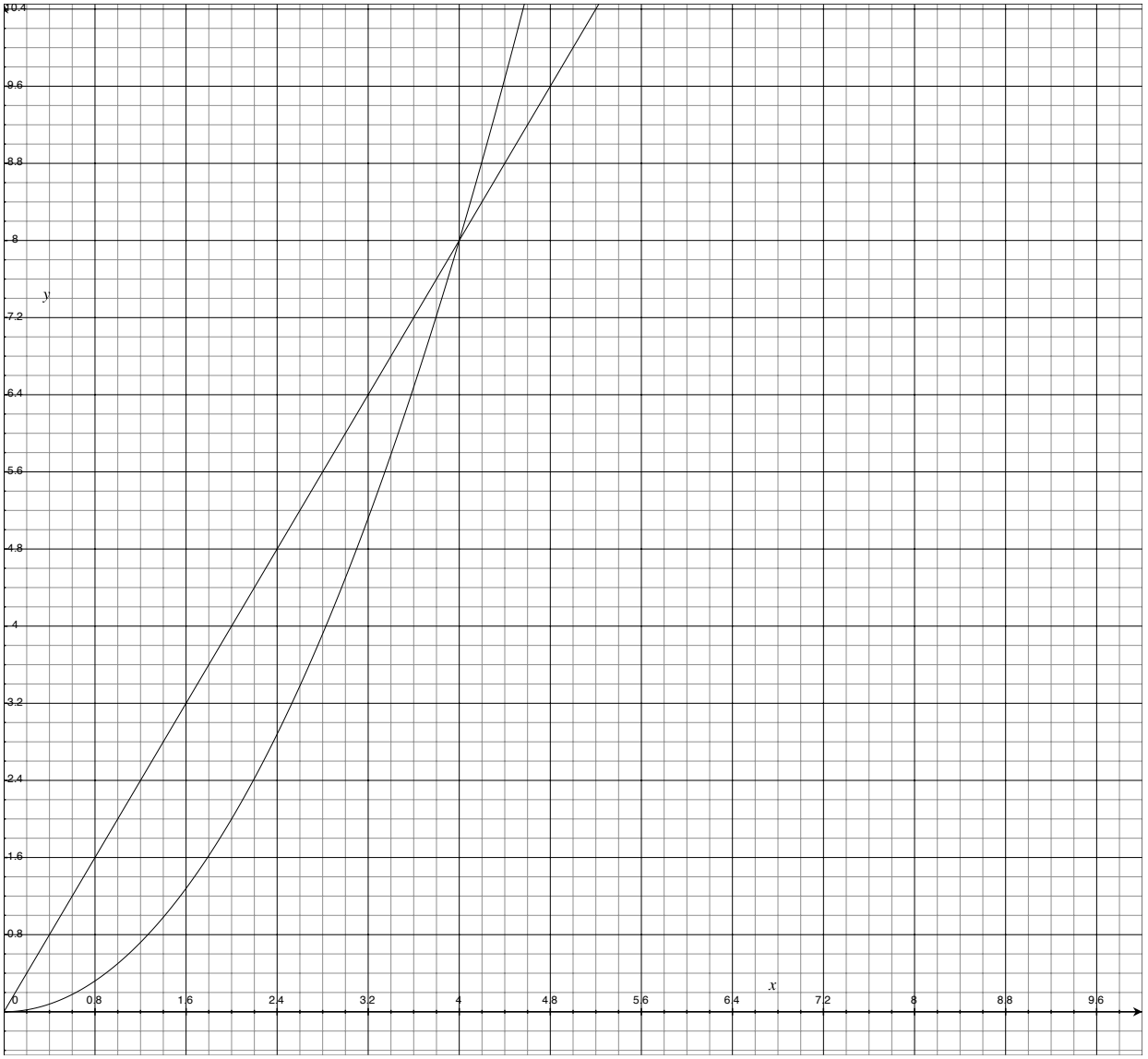
$$\underbrace{n + n + n + \cdots + n}_{m \text{ times}} = O(mn)$$

steps. When $m = 1000$, the first takes a thousand times as long as the second!

$t = n$ vs $t = n^2$


10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6
x

$t = 2n$ vs $t = 0.5n^2$


10.4
9.6
8.8
8
y
7.2
6.4
5.6
4.8
4
3.2
2.4
1.6
0.8
x
0
0.8
1.6
2.4
3.2
4
4.8
5.6
6.4
7.2
8
8.8
9.6

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 $n \geq 5$.

(We should write $(\lambda n.2n + 10)$ is $O(\lambda n.n)$, but the imprecise notation is common, so we use it.)

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

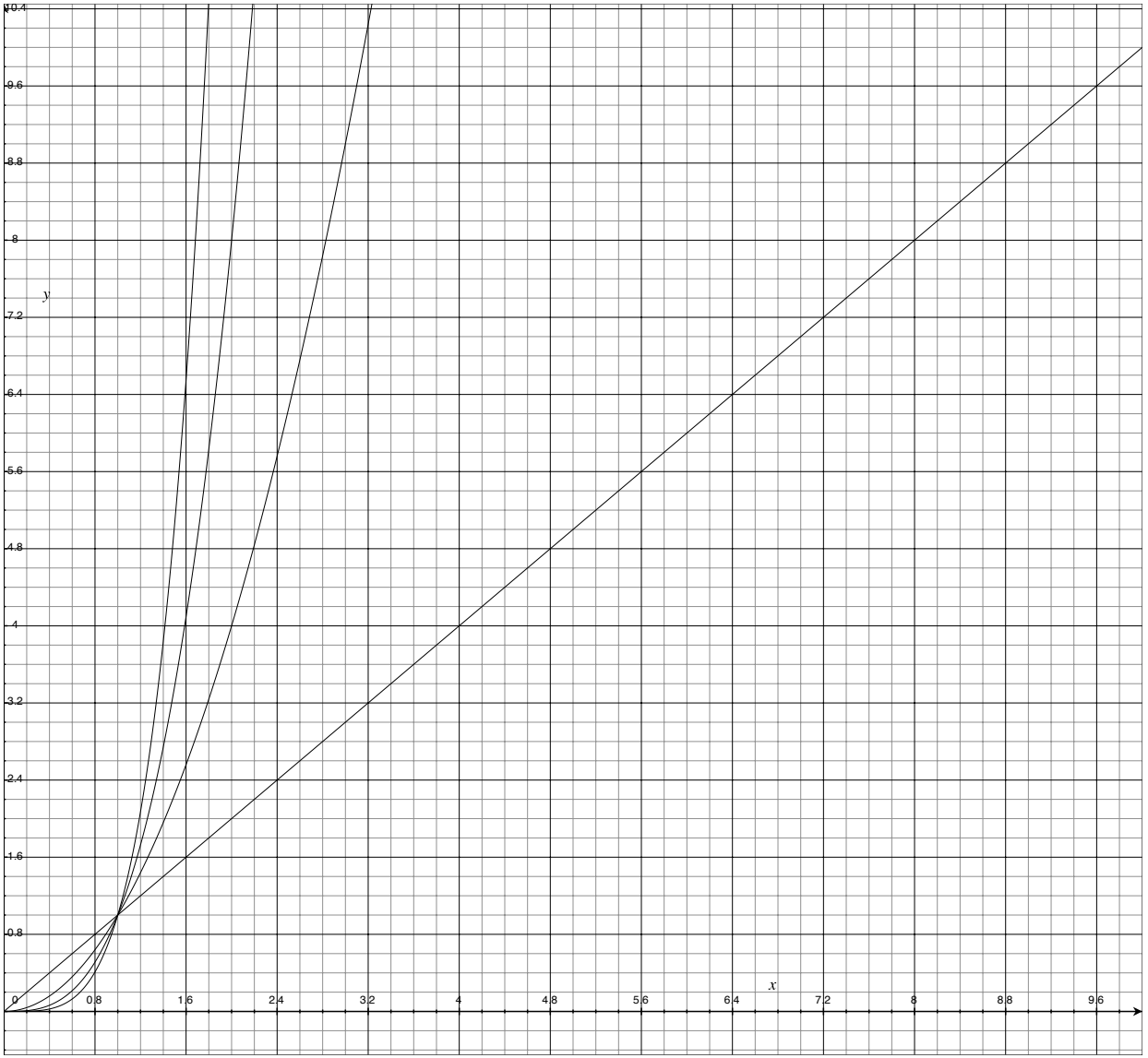
$O(n) = O(an + b)$, for any a and b

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

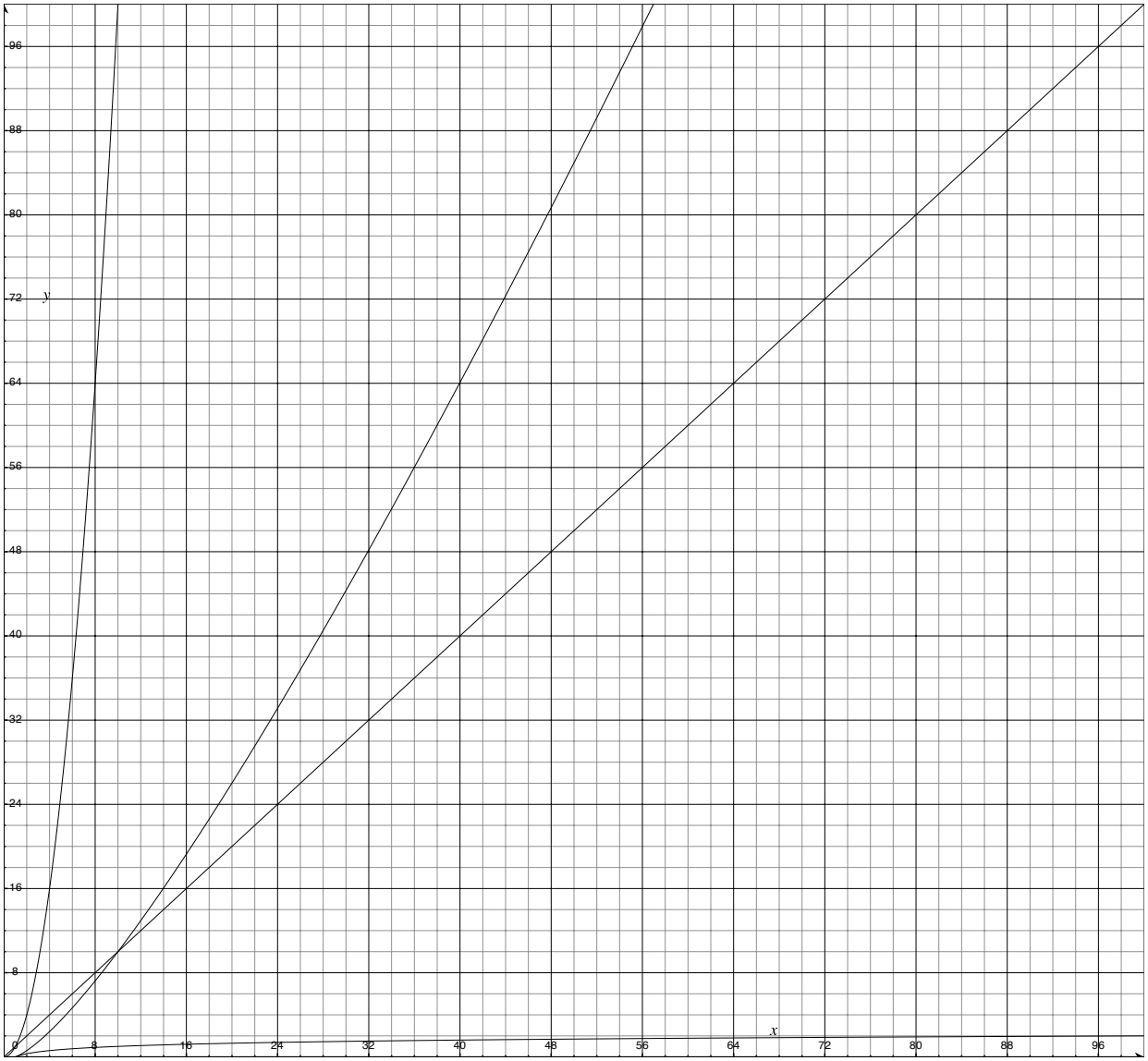
$O(n^3) = O(an^3 + bn^2 + cn + d)$, for any a, b, c, and d

$O(log_2(n)) = O(log_{10}(n))$

$$O(n), O(n^2), O(n^3), O(n^4)$$



$O(\log n), O(n), O(n \log n), O(2^n)$



$O(\log n), O(n), O(n \log n), O(2^n)$

The worst-case complexity of an algorithm is $O(f)$ when:

$$(\lambda n.\, \max \{\text{cost of running on input } x | \text{size} x \leq n\}) \text{ is } O(f)$$

$O(\log n)$ "logarithmic": divide and conquer search algorithms

$O(n)$ "linear": normal list search algorithms

$O(n \log n)$: sorting algorithms

$O(2^n)$ "exponential": tautology or satisfiability checking

We will sometimes talk about 'average-case' complexity by restricting x to be 'typical', when the worst case only happens for specific, 'a-typical', inputs.

Example:
quicksort is $O(n^2)$ if the list is nearly sorted (a-typical case)
quicksort is $O(n \log n)$ in other cases

Part II

Sets as lists

List.hs (1)

```
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)

```
element :: Eq a => a -> Set a -> Bool
x `element` xs  =  x `elem` xs

equal :: Eq a => Set a -> Set a -> Bool
xs `equal` ys  =  xs `subset` ys && ys `subset` xs
  where
  xs `subset` ys  =  and [ x `elem` ys | x <- xs ]
```

List.hs (3)

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

check =
  quickCheck prop_element

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

For complexity purposes, the size of a list is its length.

Informatics 1A

Functional Programming Lecture 13

Data Representation and Data Abstraction

Ohad Kammar

The University of Edinburgh

Part III

2025 Inf1A FP Competition

2025 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:
  `https://homepages.inf.ed.ac.uk/wadler/fp-competition-2019/`
- 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 17 November**
- Prizes awarded: **2pm Tuesday 25 November**

Recap

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 $n \geq m$.

The worst-case complexity of an algorithm is $O(f)$ when:

$$(\lambda n.\, \max \{\text{cost of running on input } x | \text{size} x \leq n\}) \text{ is } O(f)$$

The `Set` API

- `List.hs` implementation

Part IV

Sets as *ordered* lists

OrderedList.hs (1)

```
module OrderedList
  (Set,empty,insert,set,element,equal) where

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

type Set a = [a]

-- invariant [1,2,3] == True
-- invariant [3,1,2] == False
-- invariant [3,2,1] == False

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

OrderedList.hs (2)

```
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)
```

OrderedList.hs (3)

```
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
```

OrderedList.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

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

check =
  quickCheck prop_invariant >>
  quickCheck prop_element
```

```
Prelude OrderedList> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

NB: For complexity purposes, the size of a list is its length.

Part V

Sets as ordered trees

Tree.hs (1)

```
module Tree
  (Set(Nil,Node),empty,insert,set,element,equal) where
import Test.QuickCheck

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

ex1 :: Set Int
ex1 = Node (Node Nil 1 Nil) 2 (Node (Node Nil 3 Nil) 4 Nil)

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 ]
```

Tree.hs (2)

```
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

ex2 :: Set Int
ex2 = set [3,1,4,2]
```

Tree.hs (3)

```
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
```

Tree.hs (4)

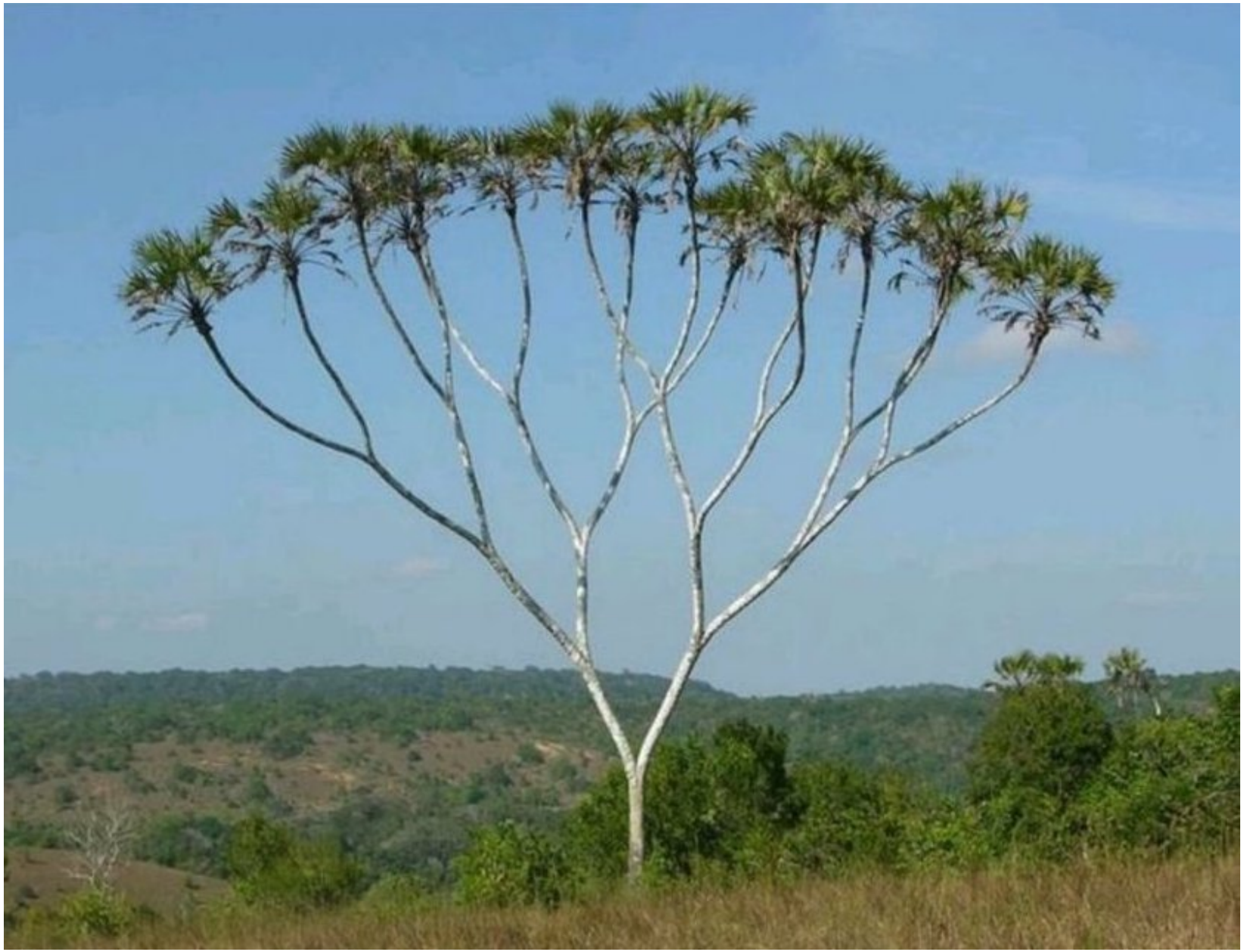
```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

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

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

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

The size of a tree is its number of nodes. It will be useful to also define its depth.

Part VI

Sets as *balanced* trees

A balanced binary tree in real life



A balanced binary tree, Computer Science version

BalancedTree.hs (1)

```
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
```

BalancedTree.hs (2)

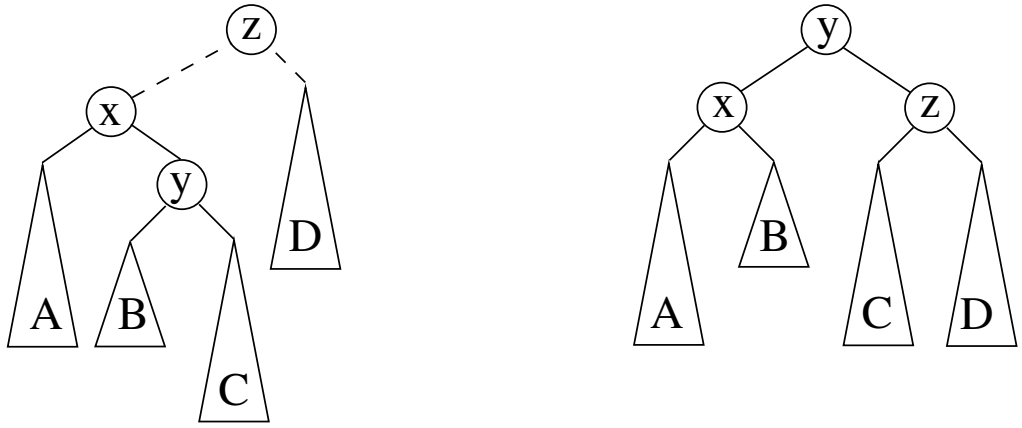
```
list :: Set a -> [a]
list Nil              =  []
list (Node l x r _)   =  list l ++ [x] ++ list r

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)
```

BalancedTree.hs (3)

```
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)
```

BalancedTree.hs (4)

```
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
```

BalancedTree.hs (5)

```
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
```

BalancedTree.hs (6)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

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

check =
  quickCheck prop_invariant >>
  quickCheck prop_element

-- Prelude BalancedTree> check
-- +++ OK, passed 100 tests.
-- +++ OK, passed 100 tests.
```

Part VII

Complexity, revisited

Summary

	`insert`	`set`	`element`	`equal`
`List`	$O(1)$	$O(1)$	$O(n)$	$O(n^2)$
`OrderedList`	$O(n)$	$O(n \log n)$	$O(n)$	$O(n)$
`Tree`	$O(\log n)^*$ $O(n)^\dagger$	$O(n \log n)^*$ $O(n^2)^\dagger$	$O(\log n)^*$ $O(n)^\dagger$	$O(n)$
`BalancedTree`	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n)$

* average case / † worst case

Part VIII

Data Abstraction

Ordered lists: remember the invariant?

```
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) ]
```

Ordered lists: breaking the invariant!

```
module OrderedListTest where
import OrderedList

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

badtest :: Int -> Bool
badtest n =
  s `equal` t
  where
  s = [1,2..n]      -- no call to set!
  t = [n,n-1..1]    -- no call to set! breaks the invariant!
```

Ordered trees: remember the invariant?

```
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 ]
```

Ordered trees: breaking the invariant!

```
module TreeTest where
import Tree

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 3 (Node Nil 2 Nil))
  -- breaks the invariant!
```

Ordered lists: add a hidden constructor!

```
module OrderedListAbs
  (Set,empty,insert,set,element,equal) where

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

data Set a = MkSet [a]

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

OrderedListAbs.hs (2)

```
empty :: Set a
empty =  MkSet []

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

set :: Ord a => [a] -> Set a
set xs  =  MkSet (nub (sort xs))
```

OrderedListAbs.hs (3)

```
element :: Ord a => a -> Set a -> Bool
x `element` MkSet ys  =  x `elt` ys
  where
  x `elt` []                  =  False
  x `elt` (y:ys) | x < y      =  False
                 | x == y     =  True
                 | x > y      =  x `elt` ys

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

OrderedListAbs.hs (4)

```
prop_invariant :: [Int] -> Bool
prop_invariant xs  =  invariant s
  where
  s = set xs

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

check =
  quickCheck prop_invariant >>
  quickCheck prop_element
```

```
Prelude OrderedListAbs> check
+++ OK, passed 100 tests.
+++ OK, passed 100 tests.
```

Ordered lists: can't break the invariant now!

```
module OrderedListAbsTest where
import OrderedListAbs

badtest :: Int -> Bool
badtest n =
  s `equal` t
  where
  s = [1,2..n]      -- no call to set!
  t = [n,n-1..1]    -- no call to set! breaks the invariant!
```

```
OrderedListAbsTest:7:3: error:
    • Couldn't match expected type 'Set a0' with actual type '[
    • In the first argument of 'equal', namely 's'
      In the expression: s `equal` t

OrderedListAbsTesttest.hs:7:13: error:
    • Couldn't match expected type 'Set a0' with actual type '[
    • In the second argument of 'equal', namely 't'
      In the expression: s `equal` t
```

Ordered lists: can't break the invariant now! (2)

```
module OrderedListAbsTest where
import OrderedListAbs

badtest :: Int -> Bool
badtest n =
  s `equal` t
  where
  s = MkSet [1,2..n]
  t = MkSet [n,n-1..1]   -- breaks the invariant!
```

```
OrderedListAbsTest.hs:8:7-11: error:
    Data constructor not in scope: MkSet :: [Int] -> Set t0
OrderedListAbsTest.hs:9:7-11: error:
    Data constructor not in scope: MkSet :: [Int] -> Set t0
```

Ordered trees: hide the constructor!

```
module TreeAbs
  (Set,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 ]
```

Ordered trees: can't break the invariant now!

```
module TreeAbsTest where
import TreeAbs

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

TreeAbsTest.hs:9:7-10: error:
    Data constructor not in scope: Node :: t0 -> Integer -> t3
TreeAbsTest.hs:9:13-16: error:
    Data constructor not in scope: Node :: t1 -> Integer -> t2
TreeAbsTest.hs:9:18-20: error:
    Data constructor not in scope: Nil
etc. etc.
```

Hiding—the secret of abstraction

```
module OrderedListAbs(Set,empty,insert,set,element,equal)

$ ghci OrderedListAbs.hs
> let s0 = MkSet [2,7,1,8,2,8]
Not in scope: data constructor `MkSet'
```

VS.

```
module OrderedList(Set(MkSet),empty,insert,element,equal)

$ ghci OrderedList.hs
> let s0 = MkSet [2,7,1,8,2,8]
> invariant s0
False
> 1 `element` s0
False
```

Hiding—the secret of abstraction

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

$ ghci TreeAbs.hs
> let s0 = Node Nil 1 (Node Nil 3 (Node Nil 2 Nil))
Not in scope: data constructor ‘Node’, ‘Nil’
```

VS.

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

$ ghci TreeUnabs.hs
> let s0 = Node Nil 1 (Node Nil 3 (Node Nil 2 Nil))
> invariant s0
False
> 2 `element` 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!