

Informatics 1

Functional Programming Lectures 13 and 14

Monday 7 and Tuesday 8 November 2011

# Abstract Types

Philip Wadler

University of Edinburgh

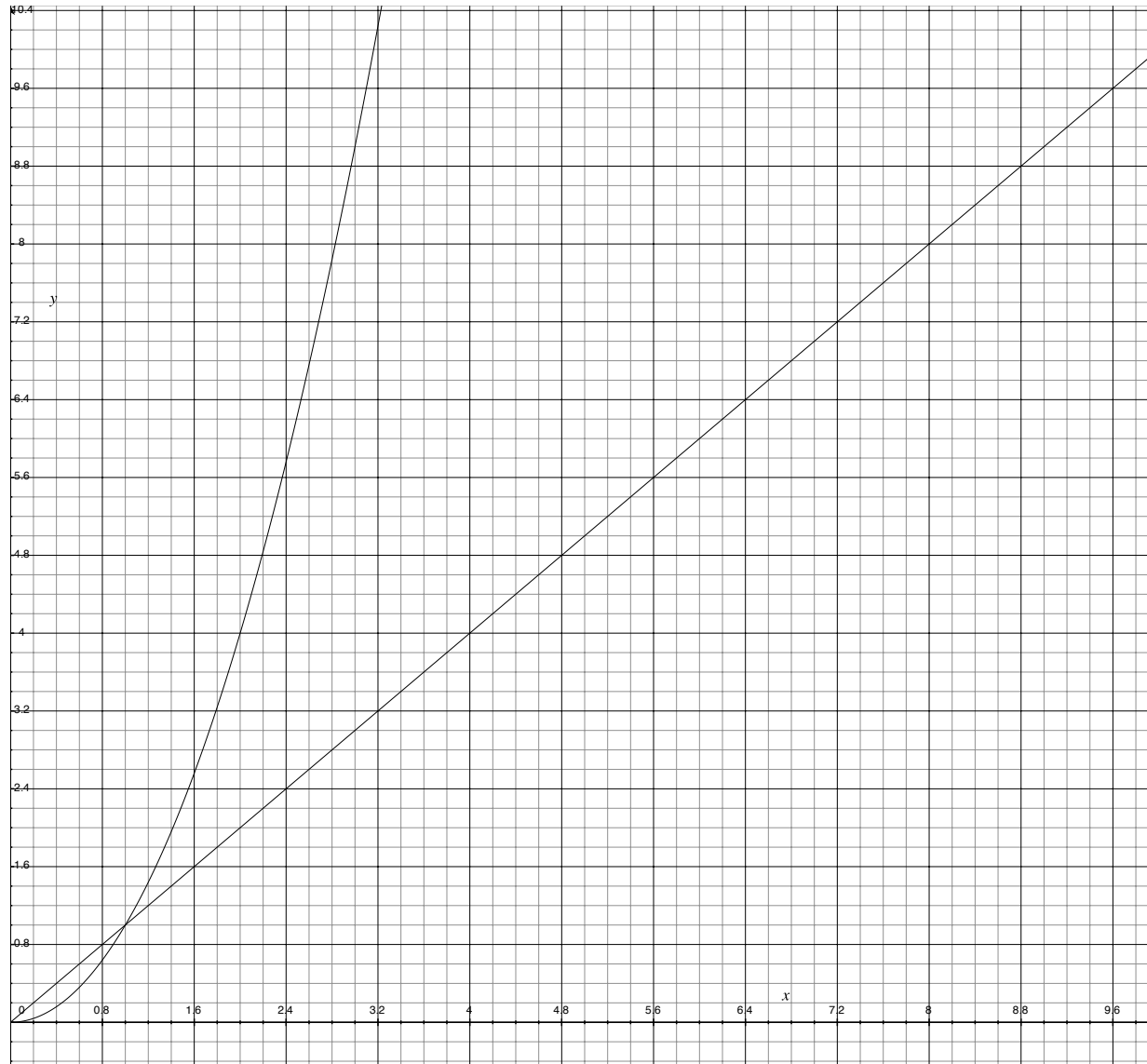
## Class exam results

- Minimum mark: 0  
Maximum mark: 100  
Average mark: 54
- You can do better!  
Last year, 60% of students received over 90% on final exam
- Extra tutorial:  
4–5pm Wednesday 9 November 2011  
Appleton Tower, room 4.12  
Do a previous class or final exam and bring your answer

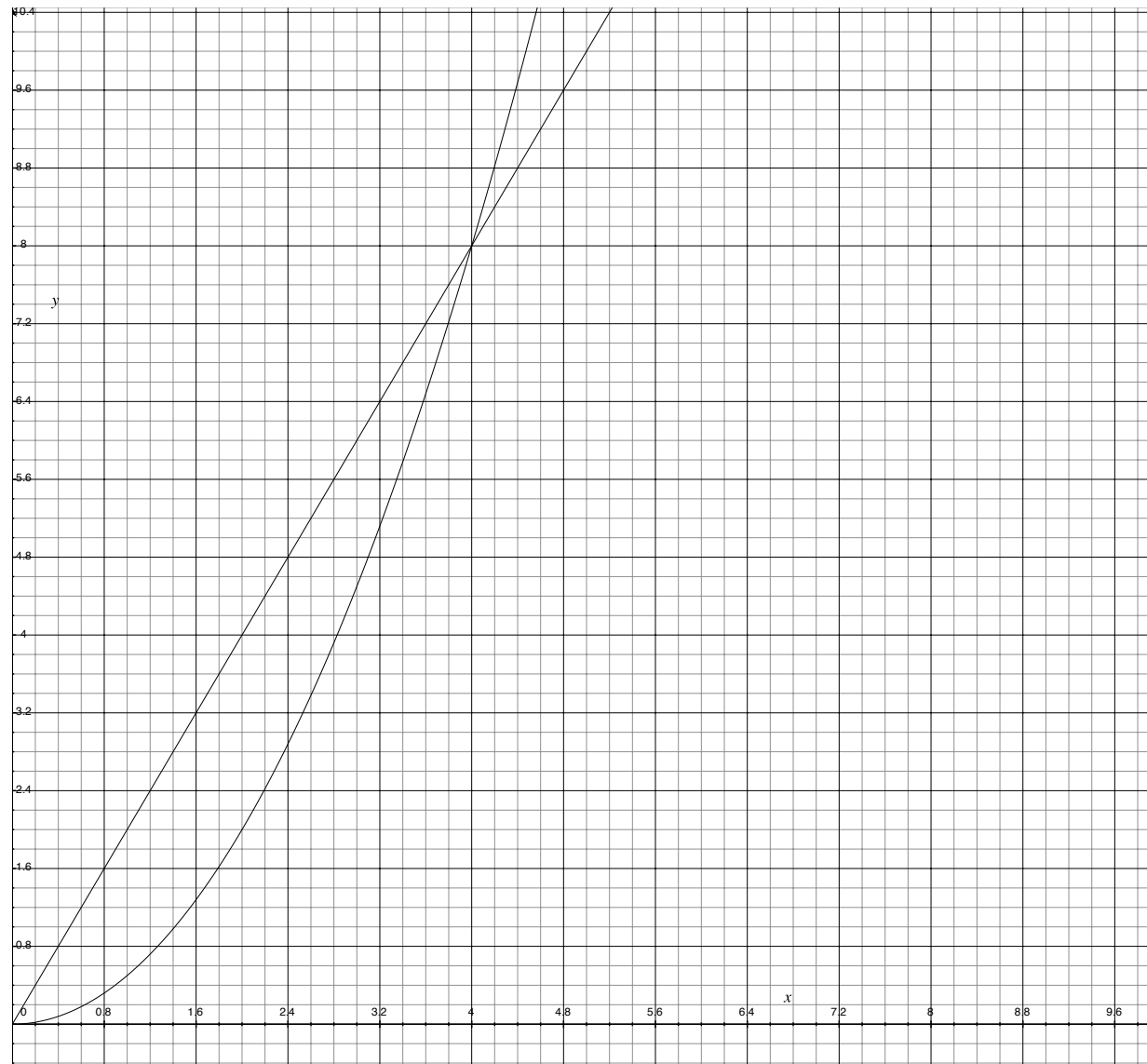
Part I

Complexity

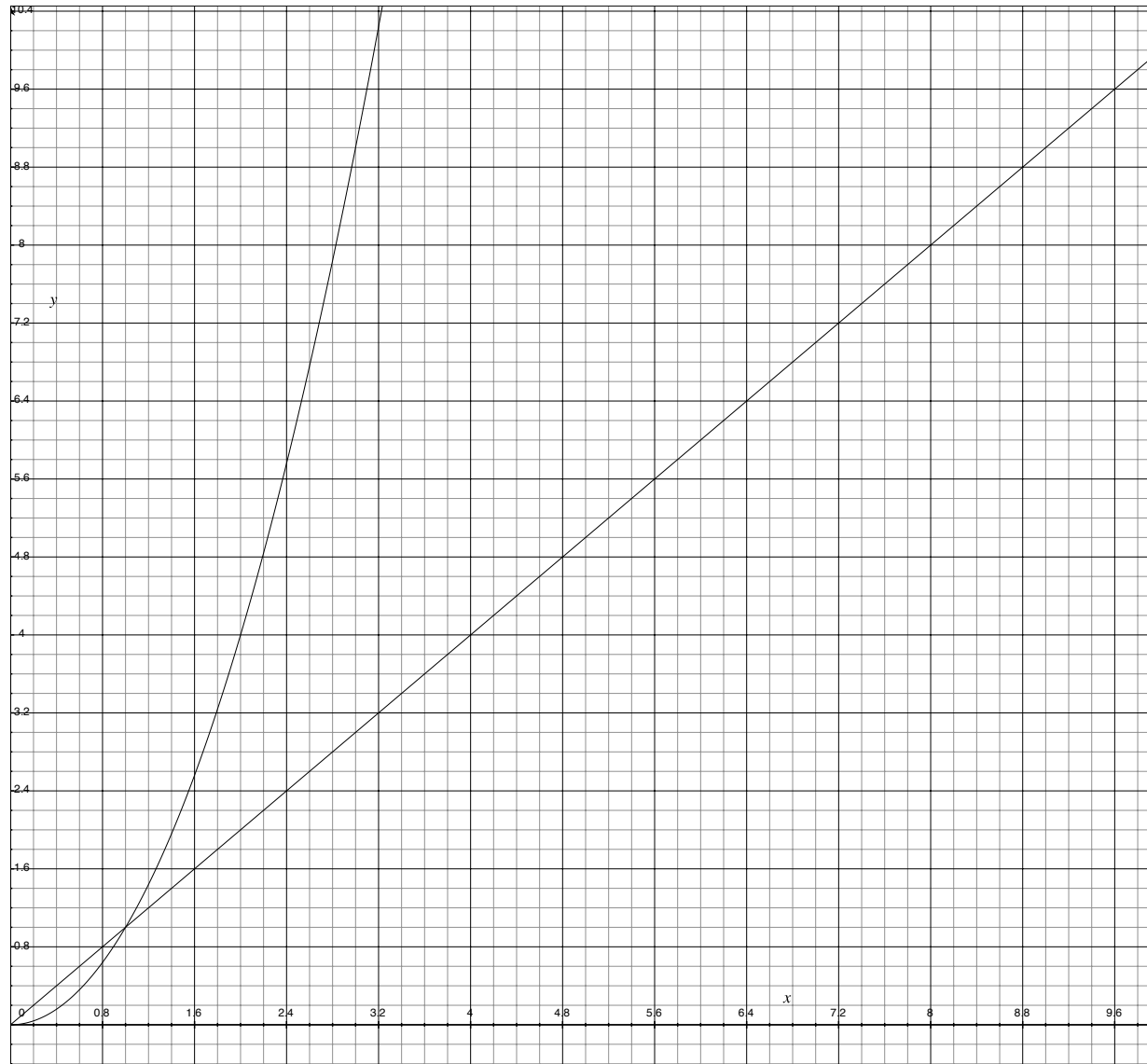
$t = n$  vs  $t = n^2$



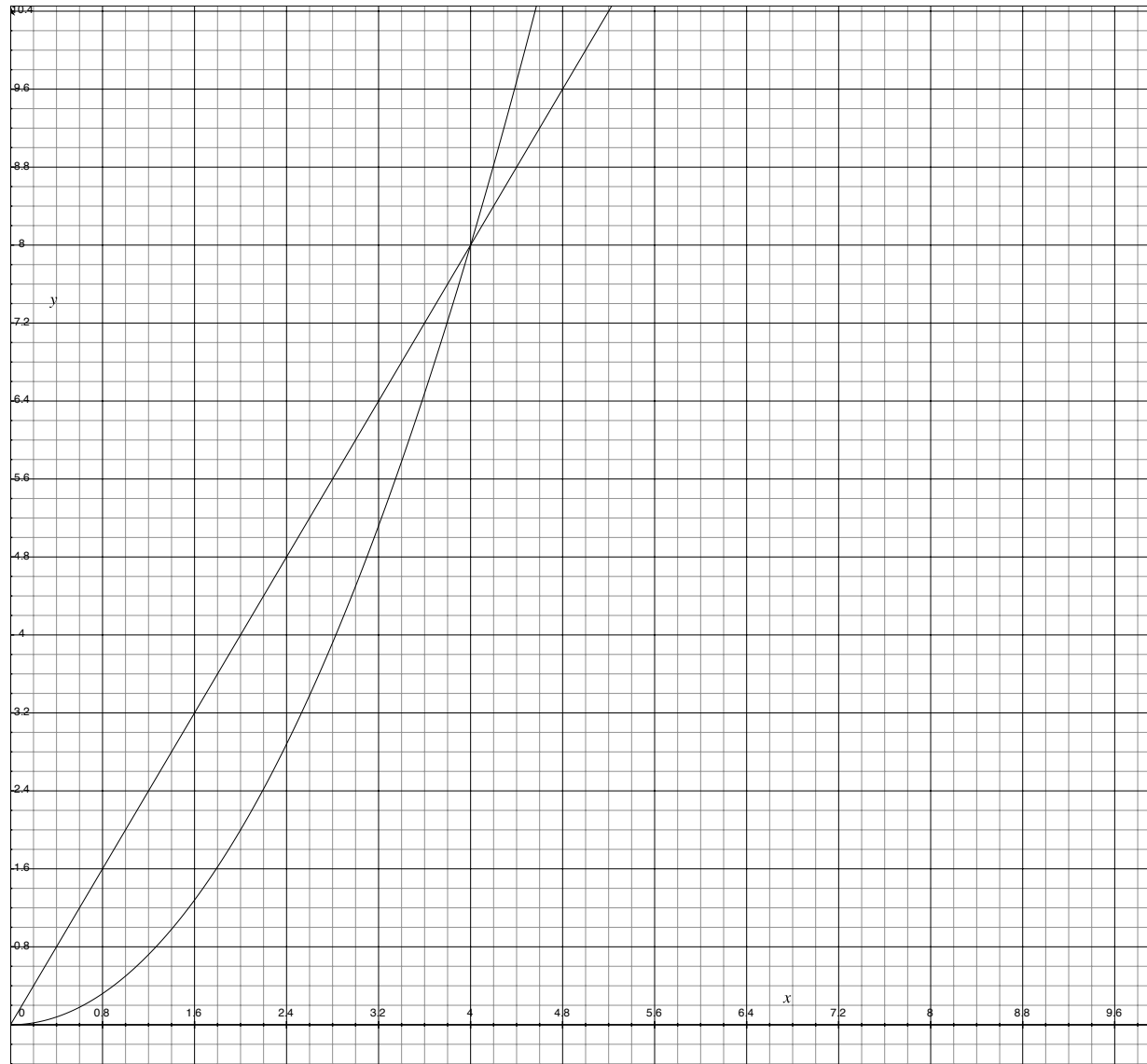
$$t = 2n \text{ vs } t = 0.5n^2$$



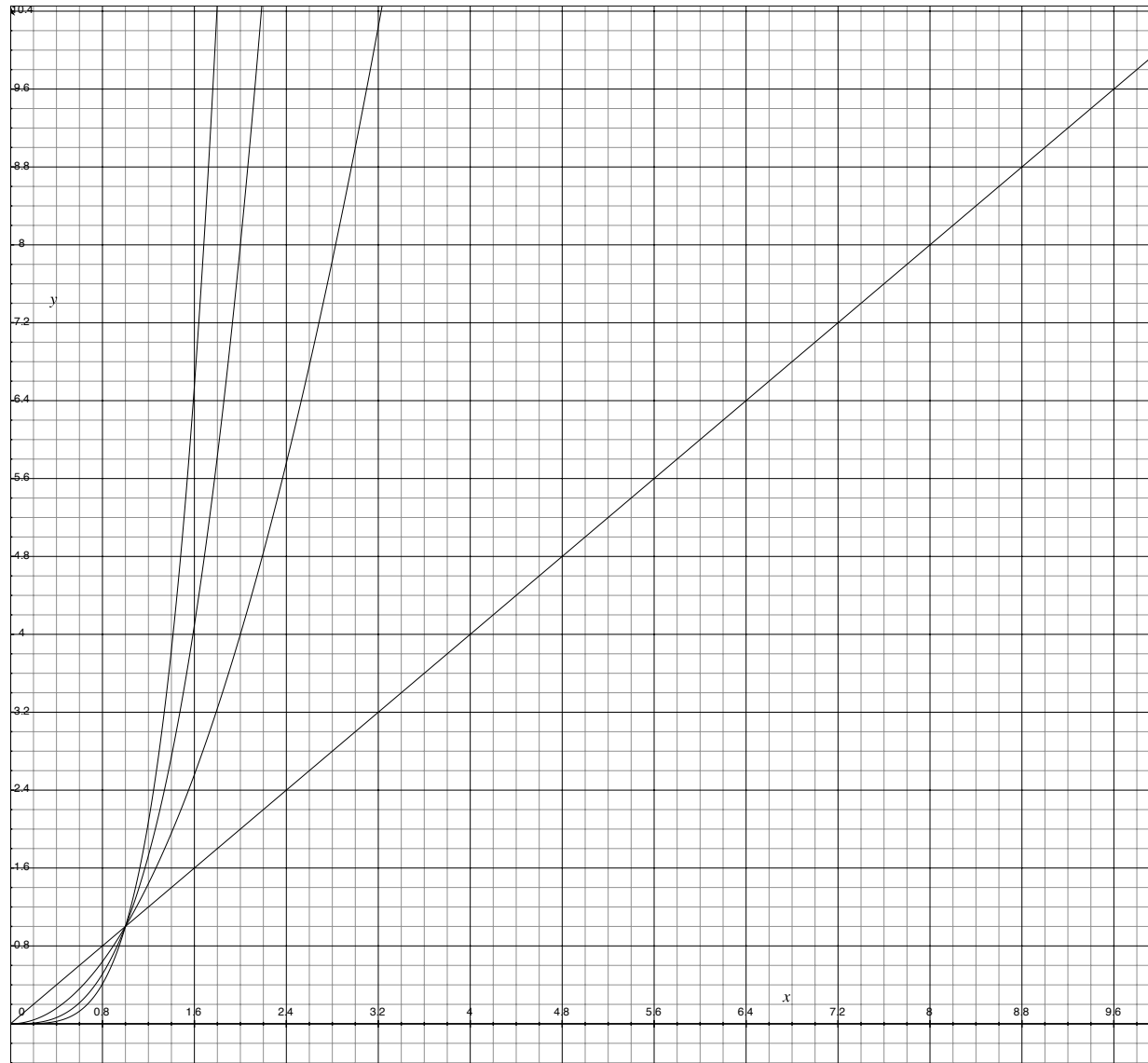
# $O(n)$ vs $O(n^2)$



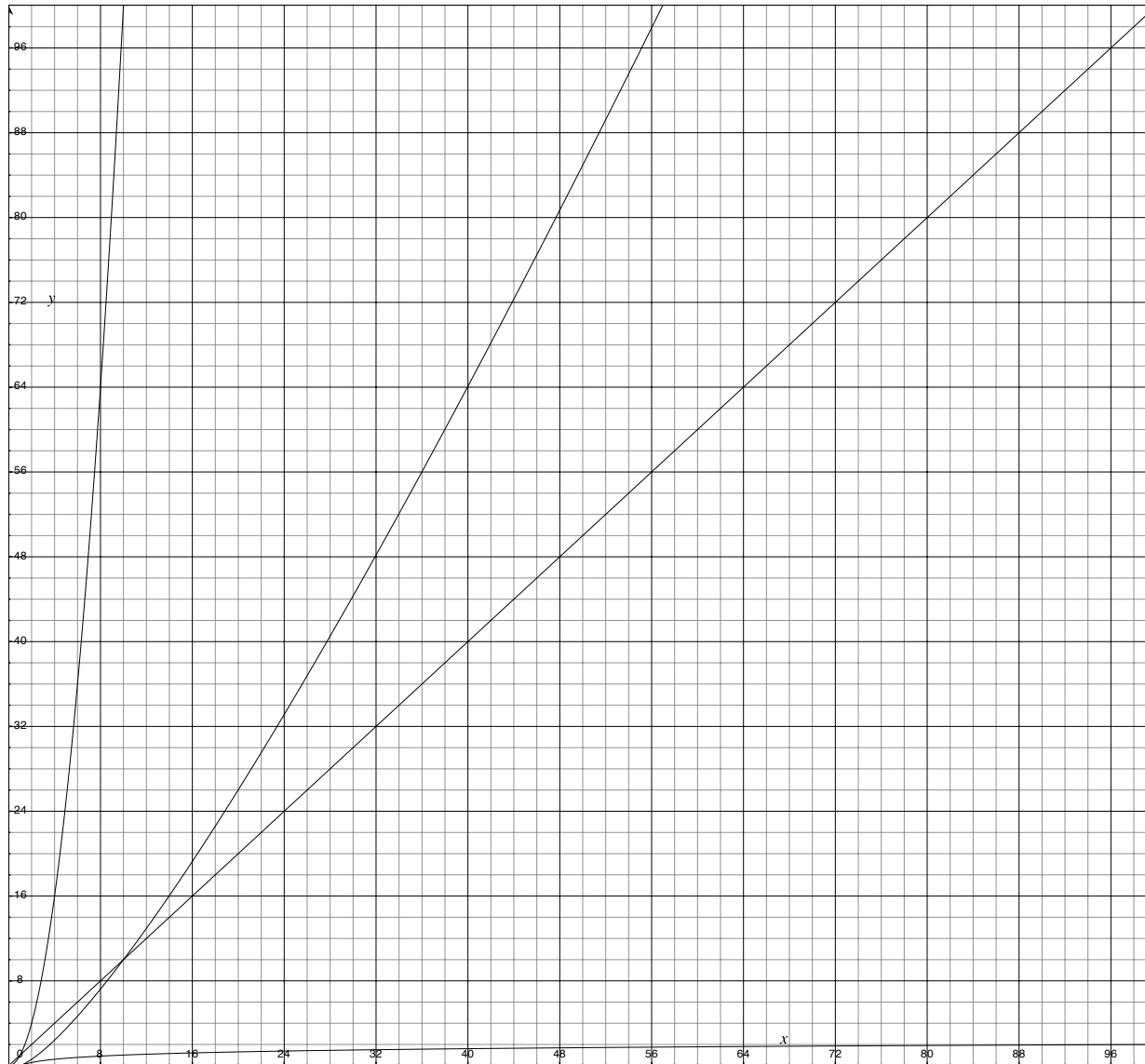
# $O(n)$ vs $O(n^2)$



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



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



## Part II

# Sets as lists without abstraction

# ListUnabs.hs (1)

```
module ListUnabs
  (Set, nil, insert, set, element, equal, check) where
import Test.QuickCheck

type Set a = [a]

nil :: Set a
nil = []

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

set :: [a] -> Set a
set xs = xs
```

## ListUnabs.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 ]
```

## ListUnabs.hs (3)

```
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 ListUnabs> check
-- +++ OK, passed 100 tests.
```

# ListUnabsTest.hs

```
module ListUnabsTest where  
import ListUnabs
```

```
test :: Int -> Bool
```

```
test n =
```

```
  s `equal` t
```

```
  where
```

```
    s = set [1,2..n]
```

```
    t = set [n,n-1..1]
```

```
-- Prelude ListUnabsTest> test 100000
```

```
-- True
```

```
-- (81.40 secs, 20932356 bytes)
```

```
breakAbstraction :: Set a -> a
```

```
breakAbstraction = head
```

```
-- not a function!
```

```
-- head (set [1,2,3]) == 1 /= 3 == head (set [3,2,1])
```

## Part III

Sets as *ordered* lists  
without abstraction

# OrderedListUnabs.hs (1)

```
module OrderedListUnabs
  (Set, nil, insert, set, element, equal, check) where

import 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) ]
```

## OrderedListUnabs.hs (2)

```
nil :: Set a
nil = []
```

```
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)
```

## OrderedListUnabs.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
```

## OrderedListUnabs.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` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]
```

```
check =
  quickCheck prop_invariant >>
  quickCheck prop_element
```

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

# OrderedListUnabsTest.hs

```
module OrderedListUnabsTest where
import OrderedListUnabs

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

-- Prelude OrderedListUnabsTest> test 100000
-- True
-- (83.98 secs, 107886888 bytes)

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!
```

## Part IV

# Sets as trees without abstraction

# TreeUnabs.hs (1)

```
module TreeUnabs
  (Set (Nil,Node), nil, insert, set, element, equal, check) 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 ]
```

## TreeUnabs.hs (2)

```
nil :: Set a
nil  = 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 nil
```

## TreeUnabs.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
```

## TreeUnabs.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` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]
```

```
check =
  quickCheck prop_invariant >>
  quickCheck prop_element
```

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

# TreeUnabsTest.hs

```
module TreeUnabsTest where
import TreeUnabs

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

-- Prelude TreeUnabsTest> test 100000
-- True
-- (4623.23 secs, 957412683436 bytes)

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



## Part V

Sets as *balanced* trees  
without abstraction

# BalancedTreeUnabs.hs (1)

```
module BalancedTreeUnabs
  (Set (Nil,Node), nil, insert, set, element, equal, check) 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
```

## BalancedTreeUnabs.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)
```

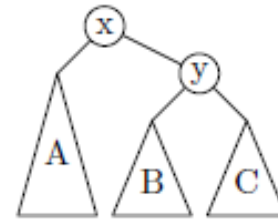
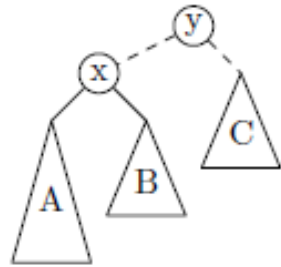
## BalancedTreeUnabs.hs (3)

```
nil :: Set a
nil  = Nil
```

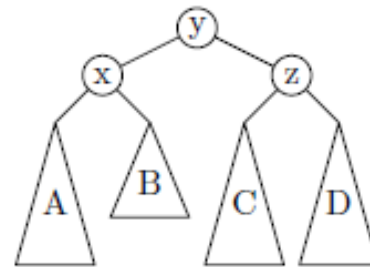
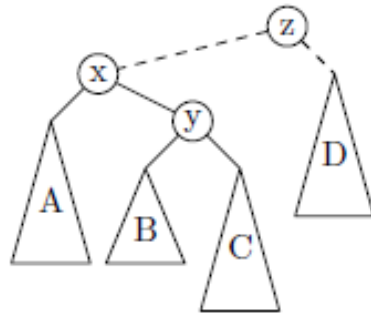
```
insert :: Ord a => a -> Set a -> Set a
insert x Nil    = node nil x nil
insert x (Node l y r _)
  | x == y      = node l y r
  | x < y       = rebalance (node (insert x l) y r)
  | x > y       = rebalance (node l y (insert x r))
```

```
set :: Ord a => [a] -> Set a
set  = foldr insert nil
```

# Reblancing



Node (Node a x b) y c  $\rightarrow$  Node a x (Node b y c)



Node (Node a x (Node b y c) z d)  
 $\rightarrow$  Node (Node a x b) y (Node c z d)

## BalancedTreeUnabs.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
```

## BalancedTreeUnabs.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
```

## BalancedTreeUnabs.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` s == odd x | x <- ys ]
  where
    s = set [ x | x <- ys, odd x ]
```

```
check =
  quickCheck prop_invariant >>
  quickCheck prop_element
```

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

# BalancedTreeUnabsTest.hs

```
module BalancedTreeUnabsTest where
import BalancedTreeUnabs

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

-- Prelude BalancedTreeUnabsTest> test 100000
-- True
-- (6.03 secs, 1389813752 bytes)

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 VI

# Complexity, revisited

# Summary

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

\* average case / worst case

# Interpreted vs. Compiled

## Interpreted

```
[comrie]wadler: ls BalancedTreeUnabs*
```

```
BalancedTreeUnabs.hs          (source code file)
```

```
BalancedTreeUnabsTest.hs
```

```
[comrie]wadler: ghci BalancedTreeUnabsTest
```

```
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
```

```
[1 of 2] Compiling BalancedTreeUnabs
```

```
      ( BalancedTreeUnabs.hs, interpreted )
```

```
[2 of 2] Compiling BalancedTreeUnabsTest
```

```
      ( BalancedTreeUnabsTest.hs, interpreted )
```

```
Ok, modules loaded: BalancedTreeUnabsTest, BalancedTreeUnabs.
```

```
*BalancedTreeUnabsTest> :s +s (turn on statistics)
```

```
*BalancedTreeUnabsTest> test 100000
```

```
True
```

```
(49.12 secs, 5233074808 bytes)
```

# Interpreted vs. Compiled

## Compiled

```
[comrie]wadler: ghc --make BalancedTreeUnabsTest
[1 of 2] Compiling BalancedTreeUnabs
          ( BalancedTreeUnabs.hs, BalancedTreeUnabs.o )
[2 of 2] Compiling BalancedTreeUnabsTest
          ( BalancedTreeUnabsTest.hs, BalancedTreeUnabsTest.o )
[comrie]wadler: ls BalancedTreeUnabs*
BalancedTreeUnabs.hi      (interface file)
BalancedTreeUnabs.o      (object code file)
BalancedTreeUnabs.hs     (source code file)
BalancedTreeUnabsTest.hi
BalancedTreeUnabsTest.o
BalancedTreeUnabsTest.hs
[comrie]wadler: ghci BalancedTreeUnabsTest
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Ok, modules loaded: BalancedTreeUnabsTest, BalancedTreeUnabs.
Prelude BalancedTreeUnabsTest> :s +s turn on statistics
Prelude BalancedTreeUnabsTest> test 100000
True
(5.71 secs, 1391977524 bytes)
```

Part VII

Data Abstraction

# ListAbs.hs (1)

```
module ListAbs
  (Set, nil, insert, set, element, equal, check) where
import Test.QuickCheck

newtype Set a = MkSet [a]

nil :: Set a
nil = MkSet []

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

set :: [a] -> Set a
set xs = MkSet xs
```

## ListAbs.hs (2)

```
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 ]
```

## ListAbs.hs (3)

```
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.
```

# ListAbsTest.hs

```
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]
-- Prelude ListAbsTest> test 100000
-- True
-- (80.96 secs, 18886896 bytes)

-- Following no longer type checks!
-- breakAbstraction :: Set a -> a
-- breakAbstraction = head
```

# Hiding—the secret of abstraction

```
module ListAbs (Set, nil, insert, set, element, equal)
```

```
> ghci ListAbs.hs
```

```
Ok, modules loaded: SetList, MainList.
```

```
*ListAbs> let s0 = set [2,7,1,8,2,8]
```

```
*ListAbs> let MkSet xs = s0 in xs
```

```
Not in scope: data constructor `MkSet`
```

**VS.**

```
module ListUnhidden (Set (MkSet), nil, insert, element, equal)
```

```
> ghci ListUnhidden.hs
```

```
*ListUnhidden> let s0 = set [2,7,1,8,2,8]
```

```
*ListUnhidden> let MkSet xs = s0 in xs
```

```
[2,7,1,8,2,8]
```

```
*ListUnhidden> head xs
```

# Hiding—the secret of abstraction

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

```
> ghci TreeAbs.hs
```

```
Ok, modules loaded: SetList, MainList.
```

```
*TreeAbs> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
```

```
Not in scope: data constructor `Node`, `Nil`
```

VS.

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

```
> ghci TreeUnabs.hs
```

```
*SetList> let s0 = Node (Node Nil 3 Nil) 2 (Node Nil 1 Nil)
```

```
*SetList> invariant s0
```

```
False
```

It's mine!



*Страна Мам.ру*