Informatics 1

Functional Programming Lecture 12

Data Abstraction

Don Sannella

University of Edinburgh

Part I

Sets as lists
without abstraction

We will look again at our four ways of implementing sets.

ListUnabs.hs (1)

```
module ListUnabs
  (Set,empty,insert,set,element,equal,check) 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
```

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]

breakAbstraction :: Set a -> a
breakAbstraction =  head

-- not a function!
-- head (set [1,2,3]) == 1 /= 3 == head (set [3,2,1])
```

How would I use this module? I might make sets out of [1,2,...,n] and [n,n-1,...,1] and check to see if they are equal.

Because set = list, all list function can be applied to sets!

But head isn't a function on sets: set [1,2,3] `equal` set [3,2,1] but head(set [1,2,3]) /= head(set [3,2,1])
It isn't enough to write documentation saying "please don't apply head to sets". We need a better solution.

This is called "breaking the abstraction".

Part II

Sets as *ordered* lists without abstraction

OrderedListUnabs.hs (1)

```
module OrderedListUnabs
  (Set,empty,insert,set,element,equal,check) 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) ]
```

OrderedListUnabs.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)
```

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]

breakAbstraction :: Set a -> a
breakAbstraction =  head
-- now it's a function
-- head (set [1,2,3]) == 1 == head (set [3,2,1])

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!
```

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]

breakAbstraction :: Set a -> a
breakAbstraction =  head
-- now it's a function
-- head (set [1,2,3]) == 1 == head (set [3,2,1])
```

Head is a function now because it always returns the smallest element.

```
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!
```

But now I can break the abstraction by using equal on lists that don't satisfy the invariant - nothing to stop me.

Could also check membership in t - will get the wrong answer

Membership and equality rely on the invariant.

Part III

Sets as ordered trees without abstraction

TreeUnabs.hs (1)

```
module TreeUnabs
  (Set(Nil,Node),empty,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)

```
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
```

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]

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!
```

Works very well for balanced trees. But trees may not be balanced.
set [1,2,...,n] will be very unbalanced, for instance.
x `element` set [1,2,...,n] is O(n), not O(log n).

Again, we can break the abstraction by building a tree that doesn't respect the invariant - gets equal to give the wrong answer.

Part IV

Sets as *balanced* trees without abstraction

BalancedTreeUnabs.hs (1)

```
module BalancedTreeUnabs
  (Set(Nil,Node),empty,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)

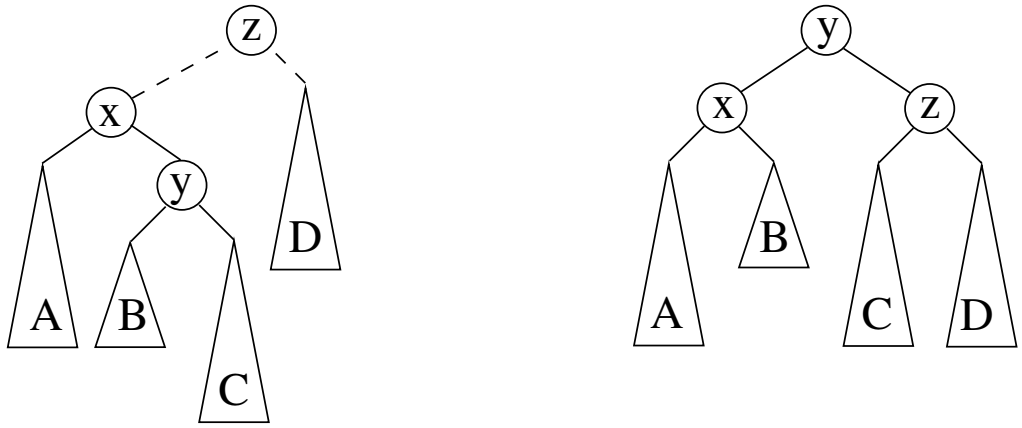
```
empty :: Set a
empty  =  Nil

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)
```

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 BalancedTreeUnabs> 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]

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!
```

We can still break the invariant.

Part VII

Data Abstraction

How do we keep people from breaking our abstraction?
It's easy - we use data constructors, and are very careful about who gets to use them.

ListAbs.hs (1)

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

We need to include a constructor: MkSet

empty uses the constructor

insert needs to extract the list, add an element, then turn the result back into a set.

set is just MkSet

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 ]
```

Just as before, once the list has been extracted from the set

Ditto for equal

It seems a little tedious and pointless, all this unpacking and re-packing using MkSet.
But wait a minute.

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]

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

Now we can't break the abstraction: head works on lists, not on sets!

But wait a minute: somebody could just extract the list from a set, using pattern matching with MkSet.

We prevent that by not exporting MkSet - it's only available inside the module.

It's mine, you can't have it.

Hiding—the secret of abstraction

```
module ListAbs(Set,empty,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),empty,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
```

In the module heading, I exported Set but not MkSet. If I do export MkSet, then it can be used to break the abstraction.

By not exporting MkSet, you can guarantee that nobody can break your abstraction.
The only way that people can get access to the representation is via the functions provided by the module.

Hiding—the secret of abstraction

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

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

For trees, it's exactly the same: I don't export Nil and Node, so I can't build a tree that violates the invariant.
This makes the constructors accessible only inside the module, making the abstraction unbreakable.
That's the secret to protecting the abstraction and having control over the representation.

Preserving the invariant

```
module TreeAbsInvariantTest where
import TreeAbs
```

You can ensure that the invariant holds by checking that it holds for all functions in the module that produce values of type Set.

```
prop_invariant_empty = invariant empty

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

A function like insert, which takes a Set as argument, needs to PRESERVE the invariant.

```
prop_invariant_set xs = invariant (set xs)
```

In this case, set will satisfy the invariant since it just combines empty and insert.

```
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!

The constructors are mine. You can't have them. That's the secret of data abstraction.

Then you can separate getting things right from making them fast. How? Create data abstractions in modules, and protect the abstraction. If you pick an inefficient representation, find a better one that provides the same "interface" (functions/types it exports). You can then replace the bad representation by the efficient one, without changing anything else!
Protection of the abstraction means that the rest of the program CAN'T depend on details that might change.