

# FUNCTIONAL DATA STRUCTURES

matthew@brecknell.net

# Functional data structures

2

- **May contain functions!**

```
newtype Parser a = P { parse :: String -> Maybe (a, String) }
```

...but not in this talk

# Functional data structures

3

## □ Are immutable

```
push :: a -> Stack a -> Stack a
```

```
type Stack a = [a]  
push = (:)
```



# Benefits of immutability

4

- Code may be easier to understand, more modular, more composable, etc.
- Everything is persistent.
- Sharing is always safe.

```
queens n = iterate build [[]] !! n where
  build solns = [ q:cs | cs <- solns, q <- [1..n], safe q cs ]

safe q = and . zipWith (\r c -> q /= c && abs (q-c) /= r) [1..]
```

# Costs of immutability

5

- At worst,  $O(f(n))$  becomes  $O(f(n \log n))$ ,
  - ▣ but usually, there are alternatives with performance within a constant factor.
- Some familiar data structures are available only in a restricted form (e.g. arrays, hash maps).
- You have to change the way you think.

# Review: big-O notation

6

- In mathematics, big-O is used to describe the behaviour of functions as their arguments get large, in the simplest possible terms.
- $O(f(n))$  is the set of functions which are eventually bounded above by a constant multiple of  $f(n)$
- Examples, in subset-inclusion order:
  - ▣  $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(2^n), O(n!)$
  - ▣ If  $f(n) = (n + 1)(\log 3n + 1)$ , then  $f(n)$  is in  $O(n \log n)$

# Review: big-O notation

7

- For data structures, big-O is used to describe the execution time of operations.
- For example, to determine whether an item is contained in a collection of  $n$  items takes:
  - ▣  $O(n)$  time, if the collection is an unordered list.
  - ▣  $O(\log n)$  time, if it is a balanced search tree.

# Review: big-O notation

8

- Amortised execution time:
  - ▣ average per-operation cost of a *sequence* of operations
  - ▣ useful for specifying data structures, provided:
    1. there are no hard real-time requirements
    2. we are concerned with serial execution

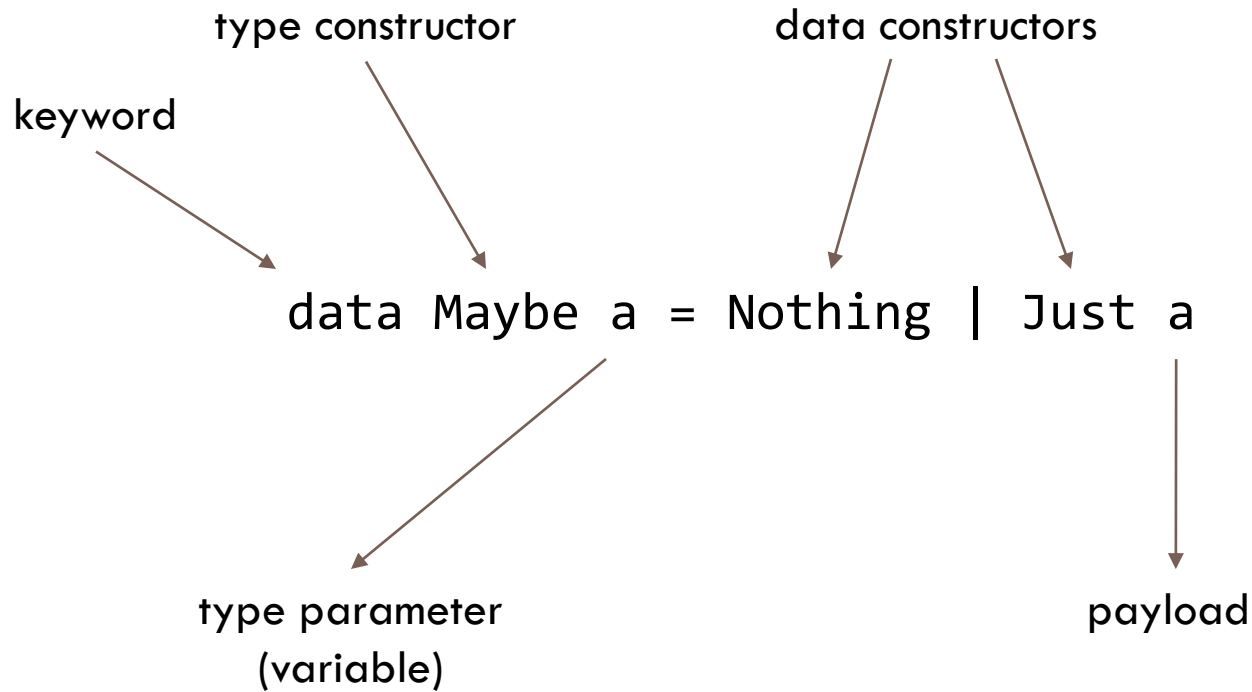
# Review: big-O notation

9

- Example: incrementing a counter (current value  $n$ ):
  - worst-case execution time is the number of bits which might need to be toggled:  $O(\log n)$
  - amortised cost of a sequence of increments:  
 $1 + 1/2 + 1/4 + \dots = 2$

# Review: Algebraic data types

10



# Review: Algebraic data types

11

- Data constructors are (special) functions:

```
Just "cause" :: Maybe String
Just (Just "cause") :: Maybe (Maybe String)
Just Nothing :: Maybe (Maybe a)
```

- Functions on data are defined by case analysis:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

# Recursive types: lists

12

## □ Lists

```
data [a] = [] | (:) a [a]
```

## □ List functions: $O(n)$

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x:r else r
                  where r = filter p xs
```

# Recursive types: lists

13

## □ Common recursive structure

```
data [a] = [] | (:) a [a]
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

```
foldr f z [] = z
```

```
foldr f z (x:xs) = f x (foldr f z xs)
```

## □ List functions revisited:

```
map :: (a -> b) -> [a] -> [b]
```

```
map f = foldr (\x r -> f x : r) []
```

```
filter :: (a -> Bool) -> [a] -> [a]
```

```
filter p = foldr (\x r -> if p x then x:r else r) []
```

# Recursive types: lists

14

## □ Concatenation: $O(n_{xs})$

```
(++) :: [a] -> [a] -> [a]  
xs ++ ys = foldr (:) ys xs
```



# Recursive types: lists

15

## □ Sharing

```
*Demo> view $ let (xs,ys,zs) = ([1,2],[3,4],[5,6])  
           in (ys, ys ++ xs, xs, zs ++ xs, zs)
```

# Recursive types: lists

16

- What's wrong with this?

```
reverse :: [a] -> [a]
reverse = foldr (\x r -> r ++ [x]) []
```

- Ouch! Quadratic complexity:  $O(n^2)$
- Solution: use a *left* fold:  $O(n)$

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs

reverse = foldl (\a x -> x:a) []
```

# Recursive types: lists

17

- Concatenating lists of lists:  $O(n)$

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

- Why not this?

```
concat :: [[a]] -> [a]
concat = foldl (++) []
```

# Aside: foldr vs foldl

18

- `foldr` is the *real* fold on lists: it is an instance of the *generic fold*, or *catamorphism*, which can be defined generically for all regular data types.
- `foldr` has the same power (but not necessarily the same efficiency) as the list data type itself.
- `foldl` is just a particular traversal which is specific to the list data type. It can be written as a `foldr`:

```
foldl f z xs = foldr (\x r z -> r (f z x)) id xs z
```

# Recursive types: trees

19

## □ Trees

```
data Tree a = Leaf | Branch a (Tree a) (Tree a)
```

## □ Common recursive structure:

```
foldTree :: (a -> b -> b -> b) -> b -> Tree a -> b  
foldTree f z = fold where  
  fold Leaf = z  
  fold (Branch x l r) = f x (fold l) (fold r)
```

# Recursive types: trees

20

- Depth-first in-order traversal:  $O(n^2)$

```
inorder :: Tree a -> [a]
inorder = foldTree (\x l r -> l ++ [x] ++ r) []
```

- Better:  $O(n)$

```
inorder :: Tree a -> [a]
inorder t = foldTree (\x l r k -> l (x : r k)) id t []
```

# FIFO Queues

21

## □ Specification: $O(1)$

```
enqueue :: Queue a -> a -> Queue a
dequeue :: Queue a -> Maybe (a, Queue a)
```

## □ First implementation:

```
data Queue a = Queue [a] [a]

enqueue (Queue f r) x = Queue f (x:r)

dequeue (Queue (x:f) r) = Just (x, Queue f r)
dequeue (Queue [] r) = case reverse r of
  [] -> Nothing
  x:f -> Just (x, Queue f [])
```

# FIFO Queues

22

- Problem:
  - ▣ If used ephemerally, costs are  $O(1)$  amortized.
  - ▣ If used persistently, costs degrade to  $O(n)$  amortised.
- Example:

```
degraded :: Int -> [Int]
degraded n = map (fst . fromJust . dequeue . enqueue nq) [1..n]
  where nq = foldl enqueue empty_queue [1..n]
```

# FIFO Queues: amortised

23

- Solution: use lazy evaluation to ensure that we amortise the cost of reverse *before* it is required!
- Invariant:  $|r| \leq |f|$

```
data Queue a = Q Int [a] Int [a]
```

```
check (Q f1 f r1 r)  
  | f1 < r1 = Q (f1 + r1) (f ++ reverse r) 0 []  
check q = q
```

```
enqueue (Q f1 f r1 r) x = check (Q f1 f (r1+1) (x:r))
```

```
dequeue (Q f1 (x:f) r1 r) = Just (x, check (Q (f1-1) f r1 r))  
dequeue (Q _ [] _ _) = Nothing
```

# FIFO Queues: real-time

24

- For *worst-case*  $O(1)$  behaviour:
  - ▣ perform reverse incrementally, and
  - ▣ systematically schedule suspended computations.

```
data Queue a = Q [a] [a] [a]
```

```
rotate [] [y] a = y:a
```

```
rotate (x:xs) (y:ys) a = x : rotate xs ys (y:a)
```

```
exec f r [] = let { fr = rotate f r [] } in Q fr [] fr
```

```
exec f r (x:s) = Q f r s
```

```
enqueue (Q f r s) x = exec f (x:r) s
```

```
dequeue (Q [] r s) = Nothing
```

```
dequeue (Q (x:f) r s) = Just (x, exec f r s)
```

# Search trees

25

- Order invariant ensures items can be found without backtracking.
- Balance (shape) invariant:
  - ▣ strong enough to ensure search paths are  $O(\log n)$ ,
  - ▣ relaxed enough to ensure rebalancing operations are also  $O(\log n)$ .
- Sharing unmodified sub-trees ensures efficiency for persistent implementations.

# Search trees

26

## □ Data.Map

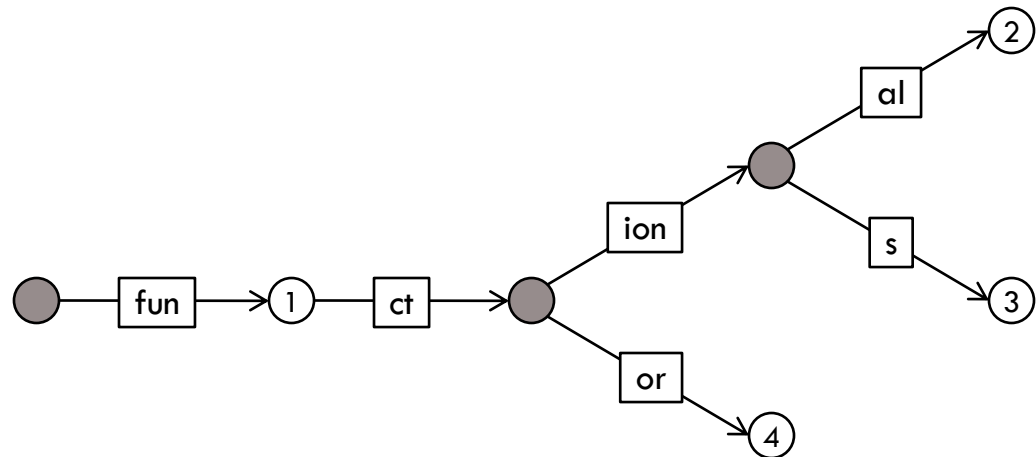
```
*Demo> let m = fromList [ (i,i) | i <- [0..9] ]  
*Demo> view m  
*Demo> view (m, delete 7 m)  
*Demo> view (m, delete 3 m)  
*Demo> view (m, split 5 m)
```

# Patricia tries

27

- Efficient alternative to hash maps, when keys can be interpreted as sequences of alternatives.

fun : 1  
functional : 2  
functions : 3  
functor : 4



# Patricia tries

28

- `Data.IntMap` is a Patricia trie on integers as sequences of bits, with worst-case  $O(\min(n, W))$  operations on individual keys.

```
*Demo> let m = fromList [ (i,i) | i <- [0..15] ]
*Demo> view m
*Demo> view (m, delete 7 m, insert 17 17 m)
*Demo> let u = fromList [ (e,e) | i <- [0..7], let e = 2^i ]
*Demo> view u
*Demo> view (m, u, union m u)
```

# Generalised tries

- Actually, trie structures can be generated for essentially any algebraic data type!
  - ▣ Tries of product types become nested tries of tries,
  - ▣ Tries of sum types become tuples of tries,
  - ▣ And so on, recursively.
- Using generic programming techniques, this can even be automated.

# Zippers

30

- Functional analogues of iterators.
- Allow efficient local edits and navigation within structures.
- The zipper for a structure is found by turning the structure inside out.

# Zipper

31

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
data Con a = T | L a (Tree a) (Con a) | R a (Tree a) (Con a)
```

```
data Zip a = Z { sub :: Tree a, con :: Con a }
```

```
toZip :: Tree a -> Zip a
```

```
toZip tree = Z tree T
```

```
left :: Zip a -> Maybe (Zip a)
```

```
left (Z (Node e l r) c) = Just (Z l (L e r c))
```

```
left (Z Leaf c) = Nothing
```

```
right :: Zip a -> Maybe (Zip a)
```

```
right (Z (Node e l r) c) = Just (Z r (R e l c))
```

```
right (Z Leaf c) = Nothing
```

# Zippers

32

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

```
data Con a = T | L a (Tree a) (Con a) | R a (Tree a) (Con a)
```

```
data Zip a = Z { sub :: Tree a, con :: Con a }
```

```
up :: Zip a -> Maybe (Zip a)
```

```
up (Z l (L e r c)) = Just (Z (Node e l r) c)
```

```
up (Z r (R e l c)) = Just (Z (Node e l r) c)
```

```
up (Z t T) = Nothing
```

```
edit :: (Tree a -> Tree a) -> Zip a -> Zip a
```

```
edit f (Z t c) = Z (f t) c
```

```
fromZip :: Zip a -> Tree a
```

```
fromZip = sub . reach up
```

```
reach :: (a -> Maybe a) -> a -> a
```

```
reach f x = maybe x (reach f) (f x)
```

# Arrays

33

- Functional arrays are write-once, read-many data structures, with  $O(1)$  access to elements.
- Producing an new array is always  $O(n)$ , so
  - ▣ Updating individual elements is inefficient, but
  - ▣ Whole-of-array updates are fine.
- Thanks to smart constructors and lazy evaluation, this restriction is not as severe as you might think.

# Arrays

34

## □ Simple examples

```
squares :: (Int,Int) -> Array Int Int
squares bounds = listArray bounds [ i*i | i <- range bounds ]
```

```
frequencies :: (Int,Int) -> [Int] -> Array Int Int
frequencies bounds items =
    accumArray (+) 0 bounds [ (i,1) | i <- items ]
```

# Arrays for memoisation

35

## □ Background: recursion using fixed point operator

```
fibonacci :: Int -> Int
fibonacci n | n <= 1 = n
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```

```
fibr :: (Int -> Int) -> Int -> Int
fibr rec n | n <= 1 = n
fibr rec n = rec (n-1) + rec (n-2)
```

```
fix :: (a -> a) -> a
fix f = let { rec = f rec } in rec
```

```
fibonacci = fix fibr
```

# Arrays for memoisation

36

## □ Memoising fixed point operator

```
fix :: (a -> a) -> a
fix f = let { rec = f rec } in rec
```

```
memofix :: forall i e. Ix i
=> (i,i) -> ((i -> e) -> i -> e) -> i -> e
memofix bounds f = rec where
  memo :: Array i e
  memo = listArray bounds [ f rec i | i <- range bounds ]
  rec i
    | inRange bounds i = memo ! i
    | otherwise = f rec i
```

```
fibonacci = memofix (2,100) fibr
```

# Arrays for dynamic programming

37

## □ Knapsack problem

```
data Item = Item { cost :: Int, value :: Int }
```

```
knapsack_rec :: [Item] -> (Int -> Int) -> Int -> Int
```

```
knapsack_rec menu rec budget = maximum (map select menu) where  
  select i = if cost i <= budget  
            then value i + rec (budget - cost i)  
            else 0
```

```
knapsack :: [Item] -> Int -> Int
```

```
knapsack menu budget = memofix (0,budget) (knapsack_rec menu) budget
```

```
knapsack_naive = fix . knapsack_rec
```

# Numerical representation

38

## □ Data.Sequence

- Amortised  $O(1)$  access to both ends, and
- $O(\log n)$  indexing, concatenation, truncation.

```
data FingerTree a
  = Empty
  | Single a
  | Deep Int (Digit a) (FingerTree (Node a)) (Digit a)

data Digit a = One a | Two a a | Three a a a | Four a a a a

data Node a = Node2 Int a a | Node3 Int a a a
```

# Bibliography, further reading

39

- Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998. (Also see his PhD thesis of the same name).
- Ralf Hinze, Ross Paterson, *Finger trees: a simple general-purpose data structure*, *Journal of Functional Programming*, 2006.
- Graham Hutton, *A tutorial on the universality and expressiveness of fold*, *Journal of Functional Programming*, 1999.
- Ralf Hinze, *Generalizing generalized tries*, *Journal of Functional Programming*, 2000.
- Gerard Huet, *Functional pearl: the zipper*, *Journal of Functional Programming*, 1997.

