Data Structures II

Advanced Functional Programming

Andres Löh (andres@cs.uu.nl)

Universiteit Utrecht

24 May 2005

Overview

Introduction (library overview)

Views

Binary search trees

Discussion

Overview

Introduction (library overview)

Views

Binary search trees

Discussion

Excerpt from the Haskell hierarchical libraries I

Data.Array	standard immutable arrays
Data.Array.*	mutable and unboxed arrays
Data.Bits	class for bit operations
Data.Bool	standard bool type and logical operations
Data.Char	standard characters and character classes
Data.Complex	standard complex numbers
Data.Dynamic	dynamic types
Data.Either	standard binary sum type
Data.FiniteMap	deprecated, see Data.Map
Data.FunctorM	monadic functor class
Data.Generics	"scrap your boilerplate" combinators
Data.Graph	easy-to-use graph library
Data.Graph.Inductive	"functional graph library"

Excerpt from the Haskell hierarchical libraries II

Data.HashTable	ephemeral hash table implementation (IO)
Data.Int	standard integers
Data.IntMap	efficient finite maps with integers keys
Data.IntSet	efficient sets of integers
Data.IORef	mutable variables (IO)
Data.Ix	class of array index types
Data.Lists	standard lists
Data.Map	"generic" finite maps
Data.Maybe	standard option/exception type
Data.Monoid	monoid class
Data.PackedString	packed (space-efficient) strings

Excerpt from the Haskell hierarchical libraries III

Data.Queue	single-ended queues
Data.Ratio	standard rational numbers
Data.Set	“generic” sets
Data.STRef	mutable variables (ST)
Data.Tree	(limited) tree operations
Data.Tuple	standard tuples
Data.Typeable	class for dynamic type information
Data.Unique	unique identities (IO)
Data.Version	very simplistic version numbers
Data.Word	words of different bit-length

Overview

Introduction (library overview)

Views

Binary search trees

Discussion

Using data types

- High-level recursion operators.
- Special syntax.
- Pattern matching.

How to use pattern matching on deques?

Deque implementation:

$$\textbf{data}\ \mathsf{Deque}\ a = D\ !\,\mathsf{Int}\ [a]\ !\,\mathsf{Int}\ [a]$$

Other possible implementation:

$$\textbf{data}\ \mathsf{Deque}\ a = D\ [a]\ [a]\ [a]$$

Pattern matching on the implementation type is bad, because

- it breaks the abstraction
- the implementation might respect invariants that are not obvious from the representation type

Using functions to destruct deques is tedious

Example: remove the first and the last element of a deque, return their sum and the resulting deque.

$$\begin{aligned}&\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int})\\&\mathit{removefl}\ q = (\mathit{head}\ q + \mathit{last}\ q, \mathit{init}\ (\mathit{tail}\ q))\end{aligned}$$

Imagine we could write

$$\begin{aligned}&\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int})\\&\mathit{removefl}\ (f \triangleleft q \triangleright l) = (f + l, q)\end{aligned}$$

instead.

Using functions to destruct deques is tedious

Example: remove the first and the last element of a deque, return their sum and the resulting deque.

$$\begin{aligned} &\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int}) \\ &\mathit{removefl}\ q = (\mathit{head}\ q + \mathit{last}\ q, \mathit{init}\ (\mathit{tail}\ q)) \end{aligned}$$

Imagine we could write

$$\begin{aligned} &\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int}) \\ &\mathit{removefl}\ (f \triangleleft q \triangleright l) = (f + l, q) \end{aligned}$$

instead.

Datatypes and functions

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Front}\ a = \mathit{Nil} \mid a \triangleleft \mathsf{Deque}\ a\\
&\textbf{data}\ \mathsf{Back}\ a\ = \mathit{Lin} \mid \mathsf{Deque}\ a \triangleright a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{front} :: \mathsf{Deque}\ a \to \mathsf{Front}\ a\\
&\mathit{front}\ q = \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Nil}\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{else}\ \ \mathit{head}\ q \triangleleft \mathit{tail}\ \ q\\
&\mathit{back}\ \ :: \mathsf{Deque}\ a \to \mathsf{Back}\ a\\
&\mathit{back}\ \ q = \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Lin}\\
&\qquad\qquad\qquad\qquad\quad\ \textbf{else}\ \ \mathit{init}\ \ q \triangleright \mathit{last}\ q
\end{aligned}
$$

Datatypes and functions

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Front}\ a = \mathit{Nil} \mid a \lhd \mathsf{Deque}\ a \\
&\textbf{data}\ \mathsf{Back}\ a\ \ = \mathit{Lin} \mid \mathsf{Deque}\ a \rhd a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{front} :: \mathsf{Deque}\ a \rightarrow \mathsf{Front}\ a \\
&\mathit{front}\ q = \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Nil} \\
&\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ \ \mathit{head}\ q \lhd \mathit{tail}\ \ q \\
&\mathit{back}\ \ :: \mathsf{Deque}\ a \rightarrow \mathsf{Back}\ a \\
&\mathit{back}\ \ q = \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Lin} \\
&\qquad\qquad\qquad\qquad\qquad\ \textbf{else}\ \ \mathit{init}\ \ \ q \rhd \mathit{last}\ q
\end{aligned}
$$

But then?

$$
\begin{aligned}
&\mathit{removefl} :: \mathsf{Deque\ Int} \to (\mathsf{Int}, \mathsf{Deque\ Int})\\
&\mathit{removefl}\ q = \mathbf{case}\ \mathit{back}\ q\ \mathbf{of}\\
&\qquad (q' \triangleright l) \to \mathbf{case}\ \mathit{front}\ q'\ \mathbf{of}\\
&\qquad\qquad (f \triangleleft q'') \to (f + l, q'')
\end{aligned}
$$

This gets even worse if we want *removefl* to return $(0, \mathit{empty})$ on queues with less than two elements.

But then?

$$
\begin{aligned}
&\mathit{removefl} :: \mathsf{Deque\ Int} \to (\mathsf{Int}, \mathsf{Deque\ Int}) \\
&\mathit{removefl}\ q = \mathbf{case}\ \mathit{back}\ q\ \mathbf{of} \\
&\qquad\qquad\qquad (q' \triangleright l) \to \mathbf{case}\ \mathit{front}\ q'\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad (f \triangleleft q'') \to (f + l, q'')
\end{aligned}
$$

This gets even worse if we want *removefl* to return $(0, \mathit{empty})$ on queues with less than two elements.

But then?

$$
\begin{array}{l}
\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int}) \\
\mathit{removefl}\ q = \textbf{case}\ \mathit{back}\ q\ \textbf{of} \\
\qquad (q' \triangleright l) \rightarrow \textbf{case}\ \mathit{front}\ q'\ \textbf{of} \\
\qquad\qquad (f \triangleleft q'') \rightarrow (f + l, q'') \\
\qquad\qquad \_ \qquad\quad\ \rightarrow (0, q) \\
\qquad \_ \rightarrow \qquad (0, q)
\end{array}
$$

This gets even worse if we want *removefl* to return $(0, \mathit{empty})$ on queues with less than two elements.

Pattern guards

$$
\begin{array}{l}
\mathit{removefl} :: \mathsf{Deque\ Int} \to (\mathsf{Int}, \mathsf{Deque\ Int}) \\
\mathit{removefl}\ q \\
\quad \mid q' \triangleright l \leftarrow \mathit{back}\ q, f \triangleleft q'' \leftarrow \mathit{front}\ q' = (f + l, q'')
\end{array}
$$

Pattern guards

- are implemented in GHC;
- allow "list comprehension"-syntax in guards
- are a conservative extension of normal guards

Pattern guards

$$
\begin{aligned}
&\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int})\\
&\mathit{removefl}\ q\\
&\qquad \mid q' \triangleright l \leftarrow \mathit{back}\ q, f \triangleleft q'' \leftarrow \mathit{front}\ q = (f + l, q'')\\
&\qquad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\qquad\quad\;\; = (0, q)
\end{aligned}
$$

Pattern guards

- are implemented in GHC;
- allow “list comprehension”-syntax in guards
- are a conservative extension of normal guards

Pattern guards

$$
\begin{array}{l}
\mathit{removefl} :: \mathsf{Deque\ Int} \rightarrow (\mathsf{Int}, \mathsf{Deque\ Int}) \\
\mathit{removefl}\ q \\
\quad \mid q' \triangleright l \leftarrow \mathit{back}\ q, f \triangleleft q'' \leftarrow \mathit{front}\ q = (f + l, q'') \\
\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\qquad\quad = (0, q)
\end{array}
$$

Pattern guards

- are implemented in GHC;
- allow "list comprehension"-syntax in guards
- are a conservative extension of normal guards

Views

```
view Front a of Deque a = Nil | a ▷ Deque a
  where
    front q = if isEmpty q then Nil
                           else head q ▷ tail q
view Back a  of Deque a = Lin | Deque a ◁ a
  where
    back  q = if isEmpty q then Lin
                           else init q ◁ last q
```

```
removefl :: Deque Int → (Int, Deque Int)
removefl (f ▷ q ◁ l) = (f + l, q)
```

Views

$$
\begin{aligned}
&\textbf{view } \mathsf{Front}\ a\ \textbf{of } \mathsf{Deque}\ a = \mathit{Nil} \mid a \triangleright \mathsf{Deque}\ a\\
&\quad \textbf{where}\\
&\qquad \mathit{front}\ q = \textbf{if } \mathit{isEmpty}\ q\ \textbf{then } \mathit{Nil}\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{head}\ q \triangleright \mathit{tail}\ q\\
&\textbf{view } \mathsf{Back}\ a\ \textbf{of } \mathsf{Deque}\ a = \mathit{Lin} \mid \mathsf{Deque}\ a \triangleleft a\\
&\quad \textbf{where}\\
&\qquad \mathit{back}\ q = \textbf{if } \mathit{isEmpty}\ q\ \textbf{then } \mathit{Lin}\\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{init}\ q \triangleleft \mathit{last}\ q
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{removefl} :: \mathsf{Deque\ Int} \to (\mathsf{Int}, \mathsf{Deque\ Int})\\
&\mathit{removefl}\ (f \triangleright q \triangleleft l) = (f + l, q)
\end{aligned}
$$

Uni-directional views

```
view Front a of Deque a = Nil | a ▷ Deque a
  where
    front q = if isEmpty q then Nil
                           else  head q ▷ tail q
```

- View constructors such as *Nil* and (▷) must not appear on the right hand side of functions, except in the view transformation. Why?
- The view type must not be recursive. Why?
- However, the view transformation may use the view recursively …

Uni-directional views

$$\begin{array}{l}\textbf{view}\ \mathsf{Front}\ a\ \textbf{of}\ \mathsf{Deque}\ a = \mathit{Nil} \mid a \triangleright \mathsf{Deque}\ a\\ \quad \textbf{where}\\ \qquad \mathit{front}\ q = \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Nil}\\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{head}\ q \triangleright \mathit{tail}\ q\end{array}$$

- View constructors such as *Nil* and $(\triangleright)$ must not appear on the right hand side of functions, except in the view transformation. Why?
- The view type must not be recursive. Why?
- However, the view transformation may use the view recursively …

Recursive view definitions

$$
\begin{array}{l}
\textbf{view}\ \mathsf{Ord}\ a \Rightarrow \mathsf{Minimum}\ a\ \textbf{of}\ [a] \quad = \mathit{Empty} \mid \mathit{Min}\ a\ [a] \\
\quad \textbf{where} \\
\qquad \mathit{min}\ [\,] \qquad\qquad\quad = \mathit{Empty} \\
\qquad \mathit{min}\ (x : \mathit{Empty}) = \mathit{Min}\ x\ [\,] \\
\qquad \mathit{min}\ (x : \mathit{Min}\ y\ \mathit{ys}) = \textbf{if}\ x \leqslant y\ \textbf{then}\ \mathit{Min}\ x\ (y : \mathit{ys}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \ \mathit{Min}\ y\ (x : \mathit{xs}) \\
\mathit{sort} :: \mathsf{Ord}\ a \Rightarrow [a] \rightarrow [a] \\
\mathit{sort}\ \mathit{Empty} \qquad\ = [\,] \\
\mathit{sort}\ (\mathit{Min}\ x\ \mathit{xs}) = x : \mathit{sort}\ \mathit{xs}
\end{array}
$$

Views on classes

```
class ListLike l where
  null :: l a → Bool
  nil  :: l a
  cons :: a → l a → l a
  head :: l a → a
  tail :: l a → l a
view (ListLike l) ⇒ List l a of l a = Nil | Cons a l
  where
    list xs = if null xs then Nil
                         else Cons (head xs) (tail xs)
```

Can views be instances of classes?

Wadler's Views

$$
\begin{array}{l}
\textbf{view}\ \mathsf{Front}\ a\ \textbf{of}\ \mathsf{Deque}\ a = \mathit{Nil} \mid a \triangleright \mathsf{Deque}\ a \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
\mathit{in}\ q & = & \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Nil} \\
 & & \phantom{\textbf{if}\ \mathit{isEmpty}\ q\ }\textbf{else}\ \ \mathit{head}\ q \triangleright \mathit{tail}\ q \\
\mathit{out}\ \mathit{Nil} & = & \mathit{empty} \\
\mathit{out}\ (a \triangleright q) & = & \mathit{cons}\ a\ q
\end{array} \\
\textbf{view}\ \mathsf{Back}\ a\ \ \textbf{of}\ \mathsf{Deque}\ a = \mathit{Lin} \mid \mathsf{Deque}\ a \triangleleft a \\
\quad \textbf{where} \\
\qquad \begin{array}{lcl}
\mathit{in}\ q & = & \textbf{if}\ \mathit{isEmpty}\ q\ \textbf{then}\ \mathit{Lin} \\
 & & \phantom{\textbf{if}\ \mathit{isEmpty}\ q\ }\textbf{else}\ \ \mathit{init}\ \ q \triangleleft \mathit{last}\ q \\
\mathit{out}\ \mathit{Lin} & = & \mathit{empty} \\
\mathit{out}\ (q \triangleleft a) & = & \mathit{snoc}\ a\ q
\end{array}
\end{array}
$$

Views in Haskell?

- Views are not implemented in any Haskell implementation.
- If they apply in both directions, isomorphism has to be checked manually.
- It is difficult to estimate the efficiency of pattern matching in the presence of views.
- It is said that pattern guards are enough.
- Nevertheless, I think that views would make a useful addition to Haskell.

Views in Haskell?

- Views are not implemented in any Haskell implementation.
- If they apply in both directions, isomorphism has to be checked manually.
- It is difficult to estimate the efficiency of pattern matching in the presence of views.
- It is said that pattern guards are enough.
- Nevertheless, I think that views would make a useful addition to Haskell.

Overview

Introduction (library overview)

Views

Binary search trees

Discussion

Binary search trees

$$\textbf{data}\ \mathsf{BinTree}\ a = \mathit{Tip} \mid \mathit{Node}\ (\mathsf{BinTree}\ a)\ a\ (\mathsf{BinTree}\ a)$$

Binary search tree (BST) property: Invariant for *Node l x r*:

$$\mathit{all}\ (\leqslant x)\ (\mathit{toList}\ l) \wedge \mathit{all}\ (x \leqslant)\ (\mathit{toList}\ r)$$

Binary search trees

$$\textbf{data}\ \mathsf{BinTree}\ a = \mathit{Tip} \mid \mathit{Node}\ (\mathsf{BinTree}\ a)\ a\ (\mathsf{BinTree}\ a)$$

$$\begin{aligned}
&\mathit{toList} :: \mathsf{BinTree}\ a \to [a] \\
&\mathit{toList}\ \mathit{Tip} &&= [\,] \\
&\mathit{toList}\ \mathit{Node}\ l\ x\ r &&= \mathit{toList}\ l +\!\!+ [x] +\!\!+ \mathit{toList}\ r
\end{aligned}$$

Binary search tree (BST) property: Invariant for $\mathit{Node}\ l\ x\ r$:

$$\mathit{all}\ (\leqslant x)\ (\mathit{toList}\ l) \land \mathit{all}\ (x \leqslant)\ (\mathit{toList}\ r)$$

Binary search trees

$$\textbf{data}\ \mathsf{BinTree}\ a = \mathit{Tip} \mid \mathit{Node}\ (\mathsf{BinTree}\ a)\ a\ (\mathsf{BinTree}\ a)$$

$$\begin{aligned}
&\mathit{toList} :: \mathsf{BinTree}\ a \to [a] \\
&\mathit{toList} \qquad\qquad\qquad = \mathit{toList}'\ [\,] \\
&\mathit{toList}' :: \mathsf{BinTree}\ a \to [a] \to [a] \\
&\mathit{toList}'\ \mathit{Tip} \qquad\quad\; = \mathit{id} \\
&\mathit{toList}'\ \mathit{Node}\ l\ x\ r = (\mathit{toList}'\ l \mathbin{+\!\!+}) \cdot (x{:}) \cdot (\mathit{toList}'\ r \mathbin{+\!\!+})
\end{aligned}$$

Binary search tree (BST) property: Invariant for *Node l x r*:

$$\mathit{all}\ ({\leqslant}\ x)\ (\mathit{toList}\ l) \wedge \mathit{all}\ (x\ {\leqslant})\ (\mathit{toList}\ r)$$

Binary search trees

$$\textbf{data}\ \mathsf{BinTree}\ a = \mathit{Tip} \mid \mathit{Node}\ (\mathsf{BinTree}\ a)\ a\ (\mathsf{BinTree}\ a)$$

$$\begin{aligned}
&\mathit{toList} :: \mathsf{BinTree}\ a \to [a] \\
&\mathit{toList} = \mathit{toList}'\ [\,] \\
&\mathit{toList}' :: \mathsf{BinTree}\ a \to [a] \to [a] \\
&\mathit{toList}'\ \mathit{Tip} = \mathit{id} \\
&\mathit{toList}'\ \mathit{Node}\ l\ x\ r = (\mathit{toList}'\ l{+\!\!+}) \cdot (x{:}) \cdot (\mathit{toList}'\ r{+\!\!+})
\end{aligned}$$

Binary search tree (BST) property: Invariant for $\mathit{Node}\ l\ x\ r$:

$$\mathit{all}\ ({\leqslant}\ x)\ (\mathit{toList}\ l) \wedge \mathit{all}\ (x\ {\leqslant})\ (\mathit{toList}\ r)$$

Searching an element in a BST

$$
\begin{aligned}
&elem :: \mathsf{Ord}\ a \Rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{Bool}\\
&elem\ x\ Tip = False\\
&elem\ x\ (Node\ l\ y\ r)\\
&\quad \mid x == y \ = True\\
&\quad \mid x < y \ = elem\ x\ l\\
&\quad \mid x > y \ = elem\ x\ r
\end{aligned}
$$

Inserting an element in a BST

```haskell
insert :: Ord a => a -> BinTree a -> BinTree a
insert x  Tip = Node Tip x Tip
insert x  (Node l y r)
    | x <= y = Node (insert x l) y r
    | x > y  = Node l y (insert x r)
```

Observations:

- Biased insertion.
- It would be easy to disallow duplicates.
- Can lead to unbalanced trees.

Inserting an element in a BST

$$
\begin{aligned}
&\mathit{insert} :: \mathsf{Ord}\ a \Rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a\\
&\mathit{insert}\ x\ \ \mathit{Tip} = \mathit{Node}\ \mathit{Tip}\ x\ \mathit{Tip}\\
&\mathit{insert}\ x\ \ (\mathit{Node}\ l\ y\ r)\\
&\qquad \mid x \leqslant y = \mathit{Node}\ (\mathit{insert}\ x\ l)\ y\ r\\
&\qquad \mid x > y = \mathit{Node}\ l\ y\ (\mathit{insert}\ x\ r)
\end{aligned}
$$

Observations:

- Biased insertion.
- It would be easy to disallow duplicates.
- Can lead to unbalanced trees.

Sorting using a BST

$$
\begin{aligned}
&\mathit{sort} :: [a] \to [a] \\
&\mathit{sort} = \mathit{toList} \cdot \mathit{foldr}\ \mathit{insert}\ \mathit{Tip}
\end{aligned}
$$

Performance?

Quadratic in the worst-case, unless BST is balanced

Sorting using a BST

$$
\begin{aligned}
&sort :: [a] \rightarrow [a] \\
&sort = toList \cdot foldr\ insert\ Tip
\end{aligned}
$$

Performance?

Quadratic in the worst-case, unless BST is balanced

Sorting using a BST

$$
\begin{aligned}
&sort :: [a] \to [a] \\
&sort = toList \cdot foldr\ insert\ Tip
\end{aligned}
$$

Performance?

Quadratic in the worst-case, unless BST is balanced

Balancing schemes

There are multiple balancing schemes known:

- AVL trees
- Red-black trees
- ...

It turns out to be more efficient to balance relatively rarely, because when used with random elements, sufficient balancing is often achieved on its own.

Balancing schemes

There are multiple balancing schemes known:

- AVL trees
- Red-black trees
- …

It turns out to be more efficient to balance relatively rarely, because when used with random elements, sufficient balancing is often achieved on its own.

Data.Map and Data.Set

- From Daan Leijen's DData library.
- Since ghc-6.4 the standard finite map and set types.
- Implemented as balanced BSTs.
- Based on *Efficient sets: a balancing act* by Stephen Adams, JFP 3(4), pages 553–562, October 1993.

Rotations

Balancing is based on rotations (drawing).

$$
\begin{aligned}
&\mathit{singleL}, \mathit{singleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a \\
&\mathit{singleL}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)) = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r \\
&\mathit{singleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r) = \mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{doubleL}, \mathit{doubleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a \\
&\mathit{doubleL}\ \ (\mathit{Node}\ l\ x\ (\mathit{Node}\ (\mathit{Node}\ m\ y\ n)\ z\ r)) \\
&\qquad\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r) \\
&\mathit{doubleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ n))\ z\ r) \\
&\qquad\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r)
\end{aligned}
$$

Rotations

Balancing is based on rotations (drawing).

$$
\begin{aligned}
&\mathit{singleL}, \mathit{singleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a\\
&\mathit{singleL}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)) = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r\\
&\mathit{singleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r) = \mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{doubleL}, \mathit{doubleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a\\
&\mathit{doubleL}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ (\mathit{Node}\ m\ y\ n)\ z\ r))\\
&\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r)\\
&\mathit{doubleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ n))\ z\ r)\\
&\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r)
\end{aligned}
$$

Rotations

Balancing is based on rotations (drawing).

$$
\begin{aligned}
&\mathit{singleL}, \mathit{singleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a \\
&\mathit{singleL}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)) = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r \\
&\mathit{singleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ r) = \mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{doubleL}, \mathit{doubleR} :: \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a \\
&\mathit{doubleL}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ (\mathit{Node}\ m\ y\ n)\ z\ r)) \\
&\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r) \\
&\mathit{doubleR}\ (\mathit{Node}\ (\mathit{Node}\ l\ x\ (\mathit{Node}\ m\ y\ n))\ z\ r) \\
&\qquad = \mathit{Node}\ (\mathit{Node}\ l\ x\ m)\ y\ (\mathit{Node}\ n\ z\ r)
\end{aligned}
$$

Smart constructor

A smart constructor is a function that takes the role of a constructor, but performs additional operations such as to establish invariants.

Recall *makeQ*.

Now $node :: \mathrm{Ord}\ a \Rightarrow \mathrm{BinTree}\ a \to a \to \mathrm{BinTree}\ a \to \mathrm{BinTree}\ a$.

Smart constructor

A smart constructor is a function that takes the role of a constructor, but performs additional operations such as to establish invariants.

Recall *makeQ*.

Now *node* :: Ord $a \Rightarrow$ BinTree $a \rightarrow a \rightarrow$ BinTree $a \rightarrow$ BinTree a.

Smart constructor

A smart constructor is a function that takes the role of a constructor, but performs additional operations such as to establish invariants.

Recall *makeQ*.

Now $node :: \mathsf{Ord}\ a \Rightarrow \mathsf{BinTree}\ a \rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a$.

Keeping the tree balanced

To be able to check the balance of a tree efficiently, we change the representation:

```
data BinTree a = Tip | Node !Int (BinTree a) a (BinTree)
```

New invariant for *Node s l x r*:

```
s == length (toList l) + 1 + length (toList r)
```

```
size :: BinTree a → Int
size Tip              = 0
size (Node s _ _ _) = s
```

Keeping the tree balanced

To be able to check the balance of a tree efficiently, we change the representation:

$$\mathbf{data}\ \mathsf{BinTree}\ a = \mathit{Tip} \mid \mathit{Node}\ !\,\mathsf{Int}\ (\mathsf{BinTree}\ a)\ a\ (\mathsf{BinTree})$$

New invariant for $\mathit{Node}\ s\ l\ x\ r$:

$$s == \mathit{length}\ (\mathit{toList}\ l) + 1 + \mathit{length}\ (\mathit{toList}\ r)$$

$$\begin{aligned}
&\mathit{size} :: \mathsf{BinTree}\ a \rightarrow \mathsf{Int}\\
&\mathit{size}\ \mathit{Tip} &&= 0\\
&\mathit{size}\ (\mathit{Node}\ s\ _\ _\ _) &&= s
\end{aligned}$$

Keeping the tree balanced

Smart constructor, assumes that the tree was originally balanced at that only one of the two trees has been changed by one element.

$$
\begin{array}{l}
\mathit{node} :: \mathsf{Ord}\ a \Rightarrow \mathsf{BinTree}\ a \rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a \\
\mathit{node}\ l\ x\ r \\
\quad \begin{array}{|lcl}
sl + sr \leqslant 1 & = & \mathit{Node}\ s\ l\ x\ r \\
sr \geqslant \delta * sl & = & \mathit{rotateL}\ l\ x\ r \\
sl \geqslant \delta * sr & = & \mathit{rotateR}\ l\ x\ r \\
\mathit{otherwise} & = & \mathit{Node}\ s\ l\ x\ r
\end{array} \\
\quad \mathbf{where}\ \begin{array}[t]{lcl}
sl & = & \mathit{size}\ l \\
sr & = & \mathit{size}\ r \\
s & = & \mathit{size}\ l + 1 + \mathit{size}\ r
\end{array}
\end{array}
$$

The constant δ can be chosen within certain parameters and is 5 in Data.Map.

Insertion and deletion

During insertion and deletion, the smart constructor is used to maintain the balance.

$$
\begin{array}{l}
\mathit{insert} :: \mathsf{Ord}\ a \Rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a \\
\mathit{insert}\ x\ \ \mathit{Tip} = \mathit{Node}\ \mathit{Tip}\ x\ \mathit{Tip} \\
\mathit{insert}\ x\ \ (\mathit{Node}\ s\ l\ y\ r) \\
\quad \begin{array}{lll}
\mid x < y & = & \mathit{node}\ (\mathit{insert}\ x\ l)\ y\ r \\
\mid x > y & = & \mathit{node}\ l\ y\ (\mathit{insert}\ x\ r) \\
\mid x == y & = & \mathit{Node}\ s\ l\ x\ r
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{delete} :: \mathsf{Ord}\ k \Rightarrow a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a \\
\mathit{delete}\ x\ \ \mathit{Tip} = \mathit{Tip} \\
\mathit{delete}\ x\ \ (\mathit{Node}\ s\ l\ y\ r) \\
\quad \begin{array}{lll}
\mid x < y & = & \mathit{node}\ (\mathit{delete}\ x\ l)\ y\ r \\
\mid x > y & = & \mathit{node}\ l\ y\ (\mathit{delete}\ x\ r) \\
\mid x == y & = & \mathit{glue}\ l\ r
\end{array}
\end{array}
$$

Insertion and deletion

During insertion and deletion, the smart constructor is used to maintain the balance.

$$
\begin{aligned}
&\mathit{insert} :: \mathsf{Ord}\ a \Rightarrow a \to \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a\\
&\mathit{insert}\ x\ \ \mathit{Tip} = \mathit{Node}\ \mathit{Tip}\ x\ \mathit{Tip}\\
&\mathit{insert}\ x\ \ (\mathit{Node}\ s\ l\ y\ r)\\
&\qquad \mid x < y \quad = \mathit{node}\ (\mathit{insert}\ x\ l)\ y\ r\\
&\qquad \mid x > y \quad = \mathit{node}\ l\ y\ (\mathit{insert}\ x\ r)\\
&\qquad \mid x == y \quad = \mathit{Node}\ s\ l\ x\ r
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{delete} :: \mathsf{Ord}\ k \Rightarrow a \to \mathsf{BinTree}\ a \to \mathsf{BinTree}\ a\\
&\mathit{delete}\ x\ \ \mathit{Tip} = \mathit{Tip}\\
&\mathit{delete}\ x\ \ (\mathit{Node}\ s\ l\ y\ r)\\
&\qquad \mid x < y \quad = \mathit{node}\ (\mathit{delete}\ x\ l)\ y\ r\\
&\qquad \mid x > y \quad = \mathit{node}\ l\ y\ (\mathit{delete}\ x\ r)\\
&\qquad \mid x == y \quad = \mathit{glue}\ l\ r
\end{aligned}
$$

Rotating once or twice

$$
\begin{array}{l}
\mathit{rotateL}\ l\ x\ r@(\mathit{Node}\ \_\ \mathit{lr}\ \_\ \mathit{rr}) \\
\quad \mid \alpha * \mathit{size}\ \mathit{lr} < \mathit{size}\ \mathit{rr} = \mathit{singleL}\ l\ x\ r \\
\quad \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{doubleL}\ l\ x\ r \\
\mathit{rotateR}\ l@(\mathit{Node}\ \_\ \mathit{ll}\ \_\ \mathit{rl})\ x\ r \\
\quad \mid \alpha * \mathit{size}\ \mathit{rl} < \mathit{size}\ \mathit{ll} \ = \mathit{singleR}\ l\ x\ r \\
\quad \mid \mathit{otherwise} \qquad\qquad\quad = \mathit{doubleR}\ l\ x\ r
\end{array}
$$

Again, α is a constant that can be chosen, and is 0.5 in Data.Map.
The functions *singleL*, *doubleL*, *singleR*, *doubleR* need to be adapted to the correct type, but are otherwise the same as mentioned before.

Glueing two balanced trees together

```haskell
glue :: BinTree a -> BinTree a -> BinTree a
glue Tip r = r
glue l Tip = l
glue l r
    | size l > size r = let (x, l') = deleteFindMax l in node l' x r
    | otherwise       = let (x, r') = deleteFindMin r in node l  x r'
```

```haskell
deleteFindMax :: BinTree a -> (a, BinTree a)
deleteFindMax (Node l x Tip) = (x, l)
deleteFindMax (Node l x r)   =
    let (y, r') = deleteFindMax r in (y, node l x r')
```

Glueing two balanced trees together

$$
\begin{aligned}
&\mathit{glue} :: \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a \rightarrow \mathsf{BinTree}\ a\\
&\mathit{glue}\ \mathit{Tip}\ r = r\\
&\mathit{glue}\ l\ \mathit{Tip}\ = l\\
&\mathit{glue}\ l\ r\\
&\quad \mid \mathit{size}\ l > \mathit{size}\ r = \mathbf{let}\ (x, l')\ = \mathit{deleteFindMax}\ l\ \mathbf{in}\ \mathit{node}\ l'\ x\ r\\
&\quad \mid \mathit{otherwise} \qquad = \mathbf{let}\ (x, r') = \mathit{deleteFindMin}\ r\ \mathbf{in}\ \mathit{node}\ l\ \ x\ r'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{deleteFindMax} :: \mathsf{BinTree}\ a \rightarrow (a, \mathsf{BinTree}\ a)\\
&\mathit{deleteFindMax}\ (\mathit{Node}\ l\ x\ \mathit{Tip}) = (x, l)\\
&\mathit{deleteFindMax}\ (\mathit{Node}\ l\ x\ r) \quad\ =\\
&\quad \mathbf{let}\ (y, r') = \mathit{deleteFindMax}\ r\ \mathbf{in}\ (y, \mathit{node}\ l\ x\ r')
\end{aligned}
$$

Sets

- The operations discussed correspond almost directly to the implementation of sets in Data.Set.
- All operations (lookup, insertion, deletion) are in $O(\log n)$.
- BSTs can be used persistently. When modified, a part of the tree must be copied.
- The Data.Set module supports more operations: update (logarithmic), union (linear), difference (linear), intersection (linear).
- What about a map on sets? It's $O(n \log n)$ in general, and linear only for monotonic functions.

Sets

- The operations discussed correspond almost directly to the implementation of sets in Data.Set.
- All operations (lookup, insertion, deletion) are in $O(\log n)$.
- BSTs can be used persistently. When modified, a part of the tree must be copied.
- The Data.Set module supports more operations: update (logarithmic), union (linear), difference (linear), intersection (linear).
- What about a map on sets? It's $O(n \log n)$ in general, and linear only for monotonic functions.

Sets

- The operations discussed correspond almost directly to the implementation of sets in Data.Set.
- All operations (lookup, insertion, deletion) are in $O(\log n)$.
- BSTs can be used persistently. When modified, a part of the tree must be copied.
- The Data.Set module supports more operations: update (logarithmic), union (linear), difference (linear), intersection (linear).
- What about a map on sets? It's $O(n \log n)$ in general, and linear only for monotonic functions.

Sets

- The operations discussed correspond almost directly to the implementation of sets in Data.Set.
- All operations (lookup, insertion, deletion) are in $O(\log n)$.
- BSTs can be used persistently. When modified, a part of the tree must be copied.
- The Data.Set module supports more operations: update (logarithmic), union (linear), difference (linear), intersection (linear).
- What about a map on sets? It's $O(n \log n)$ in general, and linear only for monotonic functions.

Finite maps

The step from sets to finite maps is very small:
We use set elements of the form $(key, value)$, where the order is determined only by the keys.

In practice, we use a specialized datatype

$$\textbf{data}\ \mathsf{Map}\ k\ a = \mathit{Tip} \mid \mathit{Node}\ !\,\mathsf{Int}\ (\mathsf{Map}\ k\ a)\ k\ a\ (\mathsf{Map}\ k\ a)$$

and adapt all the operations.

Finite maps

The step from sets to finite maps is very small:
We use set elements of the form $(key, value)$, where the order is determined only by the keys.

In practice, we use a specialized datatype

$$\mathbf{data}\ \mathsf{Map}\ k\ a = \mathit{Tip} \mid \mathit{Node}\ !\ \mathsf{Int}\ (\mathsf{Map}\ k\ a)\ k\ a\ (\mathsf{Map}\ k\ a)$$

and adapt all the operations.

What if operations on the key types are expensive?

…for example, if we use strings as keys.

Use a trie (aka digital search tree).

What if operations on the key types are expensive?

…for example, if we use strings as keys.

Use a trie (aka digital search tree).

What if operations on the key types are expensive?

…for example, if we use strings as keys.

Use a trie (aka digital search tree).

Tries

Trie representation for keys of type $[k]$:

$$\textbf{data}\ \mathsf{Trie}\ k\ a = \mathit{Node}\ (\mathsf{Maybe}\ a)\ (\mathsf{Map}\ k\ (\mathsf{Trie}\ k\ a))$$

- Can be generalized to other structures of keys than lists.
- Can be implemented as a type-indexed typs.
- Are currently not available as a standard GHC library.

Overview

Other data tructures

- Heaps/Priority queues.
- Hybrid structures: priority search queues, finger trees.
- Functional graphs.

Papers to read

- Okasaki
- Hinze

Practical advice

Be bold enough to use a non-list data structure once in a while.

At the very least, use finite maps when random-access is desired, and avoid arrays when multiple updates occur.

Observations

- Functional languages are suitable to express complex data structures clearly.
- Persistence is not always expensive.
- Laziness can sometimes be helpful in the context of persistence.
- There are still few, but nevertheless usable libraries for data structures available in Haskell.
- Views are useful.

Haskell as a language for data structures

Clear advantages, but also problems:

- lazy evaluation
- not a real module system
- no views
- at least multi-parameter type classes with fundeps needed