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
Data.Array.*
Data.Bits
Data.Bool
Data.Char
Data.Complex
Data.Dynamic
Data.Either
Data.FiniteMap
Data.FunctorM
Data.Generics
Data.Graph
Data.Graph.Inductive

standard immutable arrays
mutable and unboxed arrays
class for bit operations
standard bool type and logical operations
standard characters and character classes
standard complex numbers
dynamic types
standard binary sum type
deprecated, see Data.Map
monadic functor class
“scrap your boilerplate” combinators
easy-to-use graph library
“functional graph library”
<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data.HashTable</td>
<td>ephemeral hash table implementation (IO)</td>
</tr>
<tr>
<td>Data.Int</td>
<td>standard integers</td>
</tr>
<tr>
<td>Data.IntMap</td>
<td>efficient finite maps with integers keys</td>
</tr>
<tr>
<td>Data.IntSet</td>
<td>efficient sets of integers</td>
</tr>
<tr>
<td>Data.IORef</td>
<td>mutable variables (IO)</td>
</tr>
<tr>
<td>Data.Ix</td>
<td>class of array index types</td>
</tr>
<tr>
<td>Data.Lists</td>
<td>standard lists</td>
</tr>
<tr>
<td>Data.Map</td>
<td>“generic” finite maps</td>
</tr>
<tr>
<td>Data.Maybe</td>
<td>standard option/exception type</td>
</tr>
<tr>
<td>Data.Monoid</td>
<td>monoid class</td>
</tr>
<tr>
<td>Data.PackedString</td>
<td>packed (space-efficient) strings</td>
</tr>
</tbody>
</table>
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:

```hs
data Deque a = D ! Int [a] ! Int [a]
```

Other possible implementation:

```hs
data 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.

\[
\text{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int})
\]

\[
\text{removefl} \ q = (\text{head} \ q + \text{last} \ q, \text{init} \ (\text{tail} \ q))
\]

Imagine we could write

\[
\text{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int})
\]

\[
\text{removefl} \ (f \left< q \right> l) = (f + l, q)
\]

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.

```haskell
removefl :: Deque Int → (Int, Deque Int)
removefl q = (head q + last q, init (tail q))
```

Imagine we could write

```haskell
removefl :: Deque Int → (Int, Deque Int)
removefl (f ≪ q ≫ l) = (f + l, q)
```

instead.
**Datatypes and functions**

```plaintext
data Front a = Nil | a ◁ Deque a
data Back a = Lin | Deque a ▶ a
```

```plaintext
front :: Deque a → Front a
front q = if isEmpty q then Nil
         else head q ◁ tail q

back :: Deque a → Back a
back q = if isEmpty q then Lin
         else init q ▶ last q
```
Datatypes and functions

```haskell
data Front a = Nil | a ▷ Deque a
data Back a = Lin | Deque a ▷ a

front :: Deque a → Front a
front q = if isEmpty q then Nil
          else head q ▷ tail q

back :: Deque a → Back a
back q = if isEmpty q then Lin
        else init q ▷ last q
```
But then?

\[ \text{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int}) \]

\[ \text{removefl } q = \text{case back } q \text{ of} \]
\[ (q' \triangleright l) \rightarrow \text{case front } q' \text{ of} \]
\[ (f \triangleleft q'') \rightarrow (f + l, q'') \]

This gets even worse if we want \text{removefl} to return \((0, \text{empty})\) on queues with less than two elements.
But then?

\[
removefl :: \text{Deque } \text{Int} \rightarrow (\text{Int}, \text{Deque } \text{Int})
\]

\[
removefl \ q = \text{case } \text{back } \ q \ \text{of}
\]

\[
(q' \bowtie l) \rightarrow \text{case } \text{front } q' \ \text{of}
\]

\[
(f \lefttriangle q'') \rightarrow (f + l, q'')
\]

This gets even worse if we want \textit{removefl} to return \((0, \text{empty})\) on queues with less than two elements.
But then?

\[
\text{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int}) \\
\text{removefl} \ q = \text{case back} \ q \ \text{of} \\
\quad (q' \triangleright l) \rightarrow \text{case front} \ q' \ \text{of} \\
\quad \quad (f \triangleleft q'') \rightarrow (f + l, q'') \\
\quad \quad \quad \rightarrow (0, q) \\
\quad \quad _ \rightarrow (0, q)
\]

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

\[\text{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int}) \]
\[\text{removefl } q \quad | \quad q' \triangleright l \leftarrow \text{back } q, f \leftarrow q'' \leftarrow \text{front } q' = (f + l, q'')\]

Pattern guards

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

\[
\text{removefl} :: \text{Deque \: Int} \rightarrow (\text{Int}, \text{Deque \: Int})
\]
\[
\text{removefl} \: q
\]
\[
\begin{cases}
q' \triangleright l \gets \text{back} \: q, f \leftarrow q'' \gets \text{front} \: q = (f + l, q'') \\
\text{otherwise} & (0, q)
\end{cases}
\]

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

removefl :: Deque Int → (Int, Deque Int)
removefl q
| q' ▷ l ← back q, f ◁ q'' ← front q = (f + l, q'')
| otherwise = (0, q)

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

\textbf{view} Front \textit{a} of Deque \textit{a} = Nil \mid a \triangleright Deque \textit{a}
where
\begin{align*}
front q & = \textbf{if} \ \text{isEmpty} \ q \ \textbf{then} \ Nil \\
& \quad \textbf{else} \ \text{head} \ q \triangleright \text{tail} \ q
\end{align*}

\textbf{view} Back \textit{a} of Deque \textit{a} = Lin \mid \text{Deque} \textit{a} \triangleleft \textit{a}
where
\begin{align*}
back q & = \textbf{if} \ \text{isEmpty} \ q \ \textbf{then} \ Lin \\
& \quad \textbf{else} \ \text{init} \ q \triangleleft \text{last} \ q
\end{align*}

\textit{removefl} :: \text{Deque Int} \rightarrow (\text{Int}, \text{Deque Int})
\textit{removefl} (f \triangleright q \triangleleft l) = (f + l, q)
Views

\[
\text{view Front } a \text{ of Deque } a = \text{Nil } \mid a \triangleright \text{Deque } a \\
\text{where} \\
\quad \text{front } q = \text{if isEmpty } q \text{ then Nil} \\
\quad \quad \text{ else head } q \triangleright \text{tail } q
\]

\[
\text{view Back } a \text{ of Deque } a = \text{Lin } \mid \text{Deque } a \triangleleft a \\
\text{where} \\
\quad \text{back } q = \text{if isEmpty } q \text{ then Lin} \\
\quad \quad \text{ else init } q \triangleleft \text{last } q
\]

\[
\text{removefl :: Deque Int } \to (\text{Int, Deque Int}) \\
\text{removefl } (f \triangleright q \triangleleft l) = (f + l, q)
\]
Uni-directional views

```haskell
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

\[
\text{view Front } a \text{ of Deque } a = \text{Nil} \mid a \triangleright \text{Deque } a
\]
\[
\text{where }
front q = \begin{cases} 
\text{Nil} & \text{if isEmpty } q \\
\text{head } q \triangleright \text{tail } q & \text{else}
\end{cases}
\]

▷ View constructors such as \text{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

\textbf{view} \textit{Ord} \ a \ \Rightarrow \ \text{Minimum} \ a \ \textbf{of} \ [a] \ = \ \text{Empty} \ | \ \text{Min} \ a \ [a] \\

\textbf{where}

\textit{min} \ [\ ] \ = \ \text{Empty} \\
\textit{min} \ \ (x: \text{Empty}) \ = \ \text{Min} \ x \ [\ ] \\
\textit{min} \ \ (x: \text{Min} \ y \ ys) \ = \ \textbf{if} \ x \ \leq \ y \ \textbf{then} \ \text{Min} \ x \ (y: \textbf{ys}) \\
\textbf{else} \ \text{Min} \ y \ (x: \textbf{xs})

\textit{sort} \ :: \ \textit{Ord} \ a \ \Rightarrow \ [a] \ \rightarrow \ [a] \\
\textit{sort} \ \text{Empty} \ = \ [\ ] \\
\textit{sort} \ \ (\text{Min} \ x \ \textbf{xs}) \ = \ x: \textbf{sort} \ \textbf{xs}
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

\[
\text{view Front } a \text{ of Deque } a = \text{Nil} \mid a \triangleright \text{Deque } a
\]

where

\[
\begin{align*}
\text{in } q &= \begin{cases} 
\text{Nil} & \text{if } \text{isEmpty } q \\
\text{head } q \triangleright \text{tail } q & \text{else}
\end{cases} \\
\text{out } \text{Nil} &= \text{empty} \\
\text{out } (a \triangleright q) &= \text{cons } a \ q
\end{align*}
\]

\[
\text{view Back } a \text{ of Deque } a = \text{Lin} \mid \text{Deque } a \leftarrow a
\]

where

\[
\begin{align*}
\text{in } q &= \begin{cases} 
\text{Lin} & \text{if } \text{isEmpty } q \\
\text{init } q \leftarrow \text{last } q & \text{else}
\end{cases} \\
\text{out } \text{Lin} &= \text{empty} \\
\text{out } (q \leftarrow a) &= \text{snoc } a \ q
\end{align*}
\]
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

**data** BinTree $a = \text{Tip} \ | \ Node (\text{BinTree} \ a) \ a (\text{BinTree} \ a)$

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

$\forall x (\leq x) (\text{toList} \ l) \land \forall x (x \leq) (\text{toList} \ r)$
Binary search trees

```haskell
data BinTree a = Tip | Node (BinTree a) a (BinTree a)

toList :: BinTree a -> [a]
toList Tip = []
toList (Node l x r) = toList l ++ [x] ++ toList r
```

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

\[ \text{all} \ (\leq x) \ (\text{toList} \ l) \land \text{all} \ (x \leq) \ (\text{toList} \ r) \]
**Binary search trees**

```haskell
data BinTree a = Tip | Node (BinTree a) a (BinTree a)

toList :: BinTree a → [a]
toList = toList' []
toList' :: BinTree a → [a] → [a]
toList' Tip = id
toList' (Node l x r) = (toList' l ++) · (x:) · (toList' r ++)
```

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

\[ \forall x \in \text{toList} l \quad \forall y \in \text{toList} r \quad (x \leq y) \]

\[ \forall x \in \text{toList} r \quad \forall y \in \text{toList} l \quad (y \leq x) \]
Binary search trees

\textbf{data} BinTree \( a = \text{Tip} \mid \text{Node} \ (\text{BinTree} \ a) \ a \ (\text{BinTree} \ a) \)

\begin{align*}
toList & :: \text{BinTree} \ a \ \to \ [a] \\
toList & = \text{toList}' \ [ ] \\
toList' & :: \text{BinTree} \ a \ \to \ [a] \ \to \ [a] \\
toList' \ \text{Tip} & = \text{id} \\
toList' \ \text{Node} \ l \ x \ r & = (\text{toList}' \ l++) \cdot (x:) \cdot (\text{toList}' \ r++)
\end{align*}

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

\( \forall \leq x \ (\text{toList} \ l) \land \forall \ (x \leq) \ (\text{toList} \ r) \)
Searching an element in a BST

\[
\text{elem} :: \text{Ord } a \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{Bool}
\]
\[
\text{elem } x \text{ Tip } = \text{False}
\]
\[
\text{elem } x \ (\text{Node } l \ y \ r)
\]
\[
| \ x =\ y = \text{True} \\
| \ x <\ y = \text{elem } x \ l \\
| \ x >\ y = \text{elem } x \ r
\]
Inserting an element in a BST

`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

\[
\text{insert} :: \text{Ord } a \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]

\[
\text{insert } x \ \text{Tip} = \text{Node } \text{Tip} \ x \ \text{Tip}
\]

\[
\text{insert } x \ (\text{Node } l \ y \ r)
\]

\[
| \ x \leq \ y = \text{Node } (\text{insert } x \ l) \ y \ r
\]

\[
| \ x > y = \text{Node } l \ y \ (\text{insert } x \ r)
\]

Observations:

- Biased insertion.
- It would be easy to disallow duplicates.
- Can lead to unbalanced trees.
Sorting using a BST

\[ \text{sort} :: [a] \rightarrow [a] \]
\[ \text{sort} = \text{toList} \cdot \text{foldr insert Tip} \]

Performance?

Quadratic in the worst-case, unless BST is balanced
Sorting using a BST

\[ \text{sort} :: [a] \rightarrow [a] \]
\[ \text{sort} = \text{toList} \cdot \text{foldr insert Tip} \]

Performance?

Quadratic in the worst-case, unless BST is balanced
Sorting using a BST

\[
\text{sort} :: [a] \rightarrow [a] \\
\text{sort} = \text{toList} \cdot \text{foldr insert} \; \text{Tip}
\]

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.
Rotations

Balancing is based on rotations (drawing).

\[
\text{singleL, singleR :: BinTree } a \to \text{ BinTree } a \\
\text{singleL } (\text{Node } l \ x \ (\text{Node } m \ y \ r)) = \text{Node } (\text{Node } l \ x \ m) \ y \ r \\
\text{singleR } (\text{Node } (\text{Node } l \ x \ m) \ y \ r) = \text{Node } l \ x \ (\text{Node } m \ y \ r)
\]

\[
\text{doubleL, doubleR :: BinTree } a \to \text{ BinTree } a \\
\text{doubleL } (\text{Node } l \ x \ (\text{Node } (\text{Node } m \ y \ n) \ z \ r)) \\
\quad = \text{Node } (\text{Node } l \ x \ m) \ y \ (\text{Node } n \ z \ r) \\
\text{doubleR } (\text{Node } (\text{Node } l \ x \ (\text{Node } m \ y \ n)) \ z \ r) \\
\quad = \text{Node } (\text{Node } l \ x \ m) \ y \ (\text{Node } n \ z \ r)
\]
Rotations

Balancing is based on rotations (drawing).

\[\text{singleL, singleR} :: \text{BinTree } a \rightarrow \text{BinTree } a\]
\[\text{singleL} (\text{Node } l \ x \ (\text{Node } m \ y \ r)) = \text{Node} (\text{Node } l \ x \ m) \ y \ r\]
\[\text{singleR} (\text{Node} \ (\text{Node } l \ x \ m) \ y \ r) = \text{Node} \ l \ x \ (\text{Node } m \ y \ r)\]

\[\text{doubleL, doubleR} :: \text{BinTree } a \rightarrow \text{BinTree } a\]
\[\text{doubleL} (\text{Node } l \ x \ (\text{Node} \ (\text{Node } m \ y \ n) \ z \ r))\]
\[= \text{Node} (\text{Node } l \ x \ m) \ y \ (\text{Node } n \ z \ r)\]
\[\text{doubleR} (\text{Node} \ (\text{Node } l \ x \ (\text{Node } m \ y \ n)) \ z \ r)\]
\[= \text{Node} \ (\text{Node } l \ x \ m) \ y \ (\text{Node } n \ z \ r)\]
Rotations

Balancing is based on rotations (drawing).

singleL, singleR :: BinTree a → BinTree a
singleL (Node l x (Node m y r)) = Node (Node l x m) y r
singleR (Node (Node l x m) y r) = Node l x (Node m y r)

doubleL, doubleR :: BinTree a → BinTree a
doubleL (Node l x (Node (Node m y n) z r))
  = Node (Node l x m) y (Node n z r)
doubleR (Node (Node l x (Node m y n)) z r)
  = Node (Node l x m) y (Node n z r)
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 \(\text{node} :: \text{Ord} \ a \Rightarrow \text{BinTree} \ a \rightarrow a \rightarrow \text{BinTree} \ a \rightarrow \text{BinTree} \ a\).
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 \( \text{node} :: \text{Ord } a \to \text{BinTree } a \to a \to \text{BinTree } a \to \text{BinTree } a \).
A **smart constructor** is a function that takes the role of a constructor, but performs additional operations such as to establish invariants.

Recall $\textit{makeQ}$.

Now $\textit{node} :: \text{Ord } a \Rightarrow \text{BinTree } a \rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a$. 
Keeping the tree balanced

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

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

New invariant for `Node s l x r`:

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

```haskell
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:

```
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

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

\[
\text{node} :: \text{Ord } a \Rightarrow \text{BinTree } a \rightarrow a \rightarrow \text{BinTree } a \\
\text{node } l \ x \ r \\
\left| \begin{array}{l}
\text{sl} + \text{sr} \leq 1 = \text{Node } s \ l \ x \ r \\
\text{sr} \geq \delta \times \text{sl} = \text{rotateL } l \ x \ r \\
\text{sl} \geq \delta \times \text{sr} = \text{rotateR } l \ x \ r \\
\text{otherwise} = \text{Node } s \ l \ x \ r \\
\end{array} \right.
\]

where \( sl = \text{size } l \)

\( sr = \text{size } r \)

\( s = \text{size } l + 1 + \text{size } r \)

The constant \( \delta \) 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.

\[
\text{insert} :: \text{Ord } a \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]

\[
\text{insert } x \text{ Tip } = \text{Node } \text{Tip } x \text{ Tip}
\]

\[
\text{insert } x \ (\text{Node } s \ l \ y \ r)
\]

\[
\quad | x < y \quad = \text{node } (\text{insert } x \ l) \ y \ r
\]

\[
\quad | x > y \quad = \text{node } l \ y \ (\text{insert } x \ r)
\]

\[
\quad | x = y \quad = \text{Node } s \ l \ x \ r
\]

\[
\text{delete} :: \text{Ord } k \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]

\[
\text{delete } x \text{ Tip } = \text{Tip}
\]

\[
\text{delete } x \ (\text{Node } s \ l \ y \ r)
\]

\[
\quad | x < y \quad = \text{node } (\text{delete } x \ l) \ y \ r
\]

\[
\quad | x > y \quad = \text{node } l \ y \ (\text{delete } x \ r)
\]

\[
\quad | x = y \quad = \text{glue } l \ r
\]
Insertion and deletion

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

\[
\text{insert :: Ord } a \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]

\[
\begin{align*}
\text{insert } x \ \text{Tip} &= \text{Node Tip } x \ \text{Tip} \\
\text{insert } x \ (\text{Node } s \ l \ y \ r) &= \\
| x < y &= \text{node } (\text{insert } x \ l) \ y \ r \\
| x > y &= \text{node } l \ y \ (\text{insert } x \ r) \\
| x = y &= \text{Node } s \ l \ x \ r
\end{align*}
\]

\[
\text{delete :: Ord } k \Rightarrow a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]

\[
\begin{align*}
\text{delete } x \ \text{Tip} &= \text{Tip} \\
\text{delete } x \ (\text{Node } s \ l \ y \ r) &= \\
| x < y &= \text{node } (\text{delete } x \ l) \ y \ r \\
| x > y &= \text{node } l \ y \ (\text{delete } x \ r) \\
| x = y &= \text{glue } l \ r
\end{align*}
\]
Rotating once or twice

\[
\text{rotateL } l \times r@(Node \_ lr \_ rr) \\
| \alpha \ast \text{size } lr < \text{size } rr = \text{singleL } l \times r \\
| \text{otherwise} = \text{doubleL } l \times r
\]

\[
\text{rotateR } l@(Node \_ ll \_ rl) \times r \\
| \alpha \ast \text{size } rl < \text{size } ll = \text{singleR } l \times r \\
| \text{otherwise} = \text{doubleR } l \times r
\]

Again, \( \alpha \) is a constant that can be chosen, and is 0.5 in Data.Map.
The functions \text{singleL}, \text{doubleL}, \text{singleR}, \text{doubleR} need to be adapted to the correct type, but are otherwise the same as mentioned before.
Glueing two balanced trees together

```
trace glue :: BinTree a → BinTree a → BinTree a
trace glue Tip r = r
trace glue l Tip = l
trace 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'
```

```
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

\[
\text{glue} :: \text{BinTree } a \rightarrow \text{BinTree } a \rightarrow \text{BinTree } a
\]
\[
\text{glue } \text{Tip } r = r
\]
\[
\text{glue } l \text{ Tip } = l
\]
\[
\text{glue } l \ r
\]
\[
| \text{size } l > \text{size } r = \text{let } (x, l') = \text{deleteFindMax } l \ \text{in } \text{node } l' \ x \ r
\]
\[
| \text{otherwise} = \text{let } (x, r') = \text{deleteFindMin } r \ \text{in } \text{node } l \ x \ r'
\]

\[
\text{deleteFindMax} :: \text{BinTree } a \rightarrow (a, \text{BinTree } a)
\]
\[
\text{deleteFindMax } (\text{Node } l \ x \ \text{Tip}) = (x, l)
\]
\[
\text{deleteFindMax } (\text{Node } l \ x \ r) = \text{let } (y, r') = \text{deleteFindMax } r \ \text{in } (y, \text{node } l \ x \ r')
\]
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.
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.
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

```haskell
data Map k a = Tip | Node ! Int (Map k a) k a (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

\[
\textbf{data Map } k \ a = \text{Tip} \mid \text{Node} \ ! \ \text{Int} \ (\text{Map } k \ a) \ k \ a \ (\text{Map } k \ a)
\]

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 \([k]\):

```haskell
data Trie k a = Node (Maybe a) (Map k (Trie k a))
```

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

Introduction (library overview)

Views

Binary search trees

Discussion
Other data structures

- 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