Tour de Prelude

Before we start with any bigger libraries, it's useful to have a basic overview of what the "default" libraries do.

- `Prelude`
- `System.IO`
- `Control.Monad`

Prelude

`Prelude` is imported to all modules by default.

Contains:

- standard types, typeclasses, and related utility functions and operators
- basic IO
- basic monadic tooling

Selection from Prelude — IO

getLine, *getContents* :: IO String
getChar :: IO Char
readFile :: FilePath → IO String

putChar :: Char → IO ()
putStr, *putStrLn* :: String → IO ()
writeFile, *appendFile* :: FilePath → String → IO ()

readLn :: Read *a* ⇒ IO *a*
print :: Show *a* ⇒ *a* → IO ()

interact :: (String → String) → IO ()

Selection from Prelude — lists

$head, last :: [a] \to a$
$tail, init :: [a] \to [a]$
$(+\!\!+) :: [a] \to [a] \to [a]$
$(!!) :: [a] \to \text{Int} \to a$
$elem :: \text{Eq } a \Rightarrow [a] \to a \to \text{Bool}$
$length :: [a] \to \text{Int}$
$take, drop :: \text{Int} \to [a] \to [a]$
$takeWhile, dropWhile :: (a \to \text{Bool}) \to [a] \to [a]$
$lines, words :: \text{String} \to [\text{String}]$
$unlines, unwords :: [\text{String}] \to \text{String}$

Selection from Prelude — lists

$map :: (a \to b) \to [a] \to [b]$

$filter :: (a \to \text{Bool}) \to [a] \to [a]$

$foldl :: (a \to x \to a) \to a \to [x] \to a$

$foldr :: (x \to a \to a) \to a \to [x] \to a$

$scanl :: (a \to x \to a) \to a \to [x] \to [a]$

$scanr :: (x \to a \to a) \to a \to [x] \to [a]$

$lookup :: \text{Eq } a \Rightarrow a \to [(a, b)] \to \text{Maybe } b$

$reverse :: [a] \to [a]$

$replicate :: \text{Int} \to a \to [a]$

$repeat :: a \to [a]$

$cycle :: [a] \to [a]$

$iterate :: (a \to a) \to a \to [a]$

Selection from Prelude — lists

import Data.List

intersperse :: $a \to [a] \to [a]$ – inserts an element everywhere

intercalate :: $[a] \to [[a]] \to [a]$ – inserts a list

nub :: Eq $a \Rightarrow [a] \to [a]$ – ”unique” without Ord

sort :: Ord $a \Rightarrow [a] \to [a]$ – optimized mergesort

Selection from Prelude — lists

$$nubBy :: (a \to a \to \text{Bool}) \to [a] \to [a]$$
$$sortOn :: \text{Ord } b \Rightarrow (a \to b) \to [a] \to [a]$$
$$sortBy :: (a \to a \to \text{Ordering}) \to [a] \to [a]$$
$$groupBy :: (a \to a \to \text{Bool}) \to [a] \to [[a]]$$

Sorting ‘from the end’:

$$sortOn\ reverse\ [\texttt{"foo"}, \texttt{"bar"}, \texttt{"baz"}, \texttt{"oni"}] \quad \leadsto [\texttt{"oni"}, \texttt{"foo"}, \texttt{"bar"}, \texttt{"baz"}]$$

Nice splitting of sentence parts:

$$groupBy\ ((\equiv)\ `on`\ isLetter)\ \texttt{"Hello, Haskell!"}$$
$$[\texttt{"Hello"}, \texttt{", "}, \texttt{"Haskell"}, \texttt{"!"}]$$

$$sortBy\ (comparing\ length \diamond compare)\ [\texttt{"bb"}, \texttt{"aaaa"}, \texttt{"bbbb"}, \texttt{"aa"}]$$
$$\leadsto [\texttt{"aa"}, \texttt{"bb"}, \texttt{"aaaa"}, \texttt{"bbbb"}]$$

Selection from Prelude — functions

```
import Data.Function
const :: a → b → a    - useful for ignoring inputs
fix :: (a → a) → a    - infinite self-application
(&) :: a → (a → b) → b    - "shell pipe"
on :: (b → b → c) → (a → b) → a → a → c

import Data.Functor
(⟨&⟩) Functor f :: f a → (a → b) → f b    - "actual shell pipe"
```

Typical use of *on*:

```
((+) `on` f) x y = f x + f y
sortBySecond = sortBy (compare `on` snd)
```

...c.f.: *liftA2*

Selection from Prelude — errors

$\bot :: a$
error :: [Char] → *a*
fail :: MonadFail *m* ⇒ String → *m a*
ioError :: IOError → IO *a*
userError :: String → IOError

fail is a leftover from the original definiton of monads; in some cases it behaves quite rationally (e.g., *fail* in Maybe is Nothing). *ioError* is a nice *fail* for IO.

error is like ⊥ with an extra error message.

Exceptions

Catching system errors:

```
import System.IO.Error

catchIOError :: IO a → (IOError → IO a) → IO a
tryIOError :: IO a → IO (Either IOError a)
```

More generic:

```
import Control.Exception

throw :: Exception e ⇒ e → a
catch :: Exception e ⇒ IO a → (e → IO a) → IO a
try :: Exception e ⇒ IO a → IO (Either e a)
```

Generic exceptions

```
import Control.Monad.Catch

class Monad m ⇒ MonadThrow m where
  throwM :: Exception e ⇒ e → m a

class MonadThrow m ⇒ MonadCatch m where
  catch :: Exception e ⇒ m a → (e → m a) → m a
```

Instances should satisfy:

- $throwM\ e \gg f = throwM\ e$
- $catch\ (throwM\ e)\ f = f\ e$

Bracket pattern

Bracket captures the common semantics of creating and destroying a resource:

$$bracket :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$$
$$bracket\_ :: IO\ a \to IO\ b \to IO\ c \to IO\ c$$

Usually: *bracket openTheFile closeTheFile processTheFile*

Similar:

$$bracketOnError :: IO\ a \to (a \to IO\ b) \to (a \to IO\ c) \to IO\ c$$

$$finally :: IO\ a \to IO\ b \to IO\ a$$
$$onException :: IO\ a \to IO\ b \to IO\ a$$

Errors vs. exceptions

N.B. There is a semantic difference between an error (a failure of the program) and an exception (a situation that the programmer expected, but decided to manage it like with a failure).

Probability	Problem kind	Implementation
Unrealistic	Error	occurrence explained in documentation
Negligible	Error	⊥, *error* + documentation
Low, external origin	Exception	*error*, IOError
Low, internal origin	Exception	Exception, MonadFail, MonadCatch
Expectable	Data with exception semantics	MonadFail, MonadCatch, Alternative
Common	Normal program flow	Alternative, Either, Maybe, *bool*

Files

We hold the files by a Handle.

```
import System.IO

openFile :: FilePath → IOMode → IO Handle
hClose :: Handle → IO ()
withFile :: FilePath → IOMode → (Handle → IO r) → IO r
```

Other IO functions have their own Handle versions:

hFileSize, *hIsEOF*, *hFlush*, *hSeek*, *hTell*, *hReady*, *hGetChar*, *hGetContents*, *hGetLine*, *hPutChar*, *hPutStr*, *hPutStrLn*, *hPrint*, ...

import Control.Monad

mapM :: (Monad *m*, Traversable *t*) ⇒ (*a* → *m b*) → *t a* → *m* (*t b*)
*mapM_* :: (Monad *m*, Foldable *t*) ⇒ (*a* → *m b*) → *t a* → *m* ()

sequence :: (Monad *m*, Traversable *t*) ⇒ *t* (*m a*) → *m* (*t a*)
forever :: Applicative *f* ⇒ *f a* → *f b*

Extras:

- similar: *zipWithM*, *replicateM*, *foldM*, *sequence_*
- *forM* and *forM_* are flipped *mapM*
- *when* and *unless* instead of **if**
- Applicative iteration is often sufficient — use *traverse* and *traverse_*.

Containers

Overview

- Foldable & Traversable & common Monoids
- Maybe, Either, List
- Tree
- Sequence
- Array & Vector
- Graph
- Set, Map
- IntSet, IntMap

Not today: efficient text containers.

Data.Foldable

```
class Foldable (t :: * → *) where
  Data.Foldable.toList :: t a → [a]
  null :: t a → Bool
  length :: t a → Int
  elem :: Eq a ⇒ a → t a → Bool
  maximum, minimum :: Ord a ⇒ t a → a
  sum, product :: Num a ⇒ t a → a
  Data.Foldable.fold :: Monoid m ⇒ t m → m
  foldMap :: Monoid m ⇒ (a → m) → t a → m
  foldr, foldr' :: (a → b → b) → b → t a → b
  foldl, foldl' :: (b → a → b) → b → t a → b
  foldr1, foldl1 :: (a → a → a) → t a → a
```

Utils: *and*, *or*, *all*, *any*, *foldlM*, *foldrM*, *Data.Foldable1.fold1*

Use either *foldl'* or *foldr*!

foldl sometimes (quite often) consumes surprising amounts of memory.

Useful monoid adaptors for folding things

- Sum, Product
- Any, All
- Endo (endofunctors aka functions closed on a set, $\diamond$ becomes $(\bullet)$)
- $(,)$ (point-wise)
- Ap ($\diamond$ becomes *liftA2* $(\diamond)$)
- Alt ($\diamond$ becomes $\langle|\rangle$)
- Arg (“contains” first argument but compares on second one)

Semigroups only (empty element does not exist):

- First, Last
- Min, Max

Most of these are **newtype** wrappers with no runtime overhead.

Side note: many applicative instances use monoids

$$(\mathsf{Sum}\ 1, (+\!+)) \langle * \rangle (\mathsf{Sum}\ 5, \texttt{"hello"}) \langle * \rangle (\mathsf{Sum}\ 100, \texttt{"haskell"})$$
$$\rightsquigarrow (\mathsf{Sum}\ 106, \texttt{"hellohaskell"})$$


Vorsicht
Funktor
2 m Abstand halten

```
class (Functor t, Foldable t) ⇒ Traversable t where
  traverse :: Applicative f ⇒ (a → f b) → t a → f (t b)
  sequenceA :: Applicative f ⇒ t (f a) → f (t a)
  mapM :: Monad m ⇒ (a → m b) → t a → m (t b)
  sequence :: Monad m ⇒ t (m a) → m (t a)
```

Similar: *traverse_*, *sequence_*, *mapM_*, *forM*, *mapAccumL*, *mapAccumR*

Data.Maybe

$$
\begin{aligned}
&\mathit{maybe} && :: b \to (a \to b) \to \text{Maybe } a \to b \\
&\mathit{fromMaybe} && :: a \to \text{Maybe } a \to a \\
&\mathit{fromJust} && :: \text{Maybe } a \to a \\
&\mathit{isJust} && :: \text{Maybe } a \to \text{Bool} \\
&\mathit{isNothing} && :: \text{Maybe } a \to \text{Bool} \\
&\mathit{listToMaybe} && :: [a] \to \text{Maybe } a \\
&\mathit{maybeToList} && :: \text{Maybe } a \to [a] \\
&\mathit{mapMaybe} && :: (a \to \text{Maybe } b) \to [a] \to [b] \\
&\mathit{catMaybes} && :: [\text{Maybe } a] \to [a]
\end{aligned}
$$

Data.Either

$$
\begin{aligned}
&\mathit{either} && :: (a \to c) \to (b \to c) \to \text{Either } a\ b \to c \\
&\mathit{fromLeft} && :: a \to \text{Either } a\ b \to a \\
&\mathit{fromRight} && :: b \to \text{Either } a\ b \to b \\
&\mathit{isLeft} && :: \text{Either } a\ b \to \text{Bool} \\
&\mathit{isRight} && :: \text{Either } a\ b \to \text{Bool} \\
&\mathit{lefts} && :: [\text{Either } a\ b] \to [a] \\
&\mathit{rights} && :: [\text{Either } a\ b] \to [b] \\
&\mathit{partitionEithers} && :: [\text{Either } a\ b] \to ([a], [b])
\end{aligned}
$$

```
cycle        :: [a] → [a]
repeat       :: a → [a]
replicate    :: Int → a → [a]

intersperse  :: a → [a] → [a]
intercalate  :: [a] → [[a]] → [a]

group        :: Eq a ⇒ [a] → [[a]]
nub          :: Eq a ⇒ [a] → [a]

inits        :: [a] → [[a]]
tails        :: [a] → [[a]]

sort         :: Ord a ⇒ [a] → [a]
sortBy       :: (a → a → Ordering) → [a] → [a]
sortOn       :: Ord b ⇒ (a → b) → [a] → [a]
```

data Tree a = Node {*rootLabel* :: a, *subForest* :: Forest a}
type Forest a = [Tree a]

drawTree :: Tree String → String
drawForest :: Forest String → String

levels :: Tree a → [[a]]
flatten :: Tree a → [a]

foldTree :: (a → [b] → b) → Tree a → b
unfoldTree :: (b → (a, [b])) → b → Tree a

Principial use of Data.Tree

```
ghci> putStrLn $ drawTree
               $ fmap show
               $ Node 123 [Node 12 [Node 1 [], Node 2 []], Node 3 []]
123
|
+- 12
|  |
|  +- 1
|  |
|  `- 2
|
`- 3
```

Construction nomenclature

Type	Constructors (patterns)	Universal destructor (interpreter)
Maybe	Nothing,Just	$maybe :: b \to (a \to b) \to \text{Maybe } a \to b$
Either	Left,Right	$either :: (a \to c) \to (b \to c) \to \text{Either } a\ b \to c$
(,)	(,)	$uncurry :: (a \to b \to c) \to (a, b) \to c$
[]	[],(:)	$foldr :: (a \to b \to b) \to b \to [a] \to b$
(→)		$(\&) :: a \to (a \to b) \to b$

Can you make `foldl'` from `foldr`?

Data.Sequence

Sequence is a better list:

- indexing in $O(\log n)$
- access to both ends in $O(1)$
- most non-global operations run in between $O(1)$ and $O(\log n)$

Extra operations:

```
(⟨|)   :: a → Seq a → Seq a    - add to the beginning
(|⟩)   :: Seq a → a → Seq a    - add to the end

data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a → ViewL a
viewr :: Seq a → ViewR a

(⟩⟨)   :: Seq a → Seq a → Seq a    - O(log n) concat
```

Arrays have indexes:

```
class Ord a ⇒ Ix a where
  range     :: (a, a) → [a]
  index     :: (a, a) → a → Int
  inRange   :: (a, a) → a → Bool
  rangeSize :: (a, a) → Int
```

Array type: `Array idx a`

Array access by index is guaranteed $O(1)$. The implementation is built into GHC, modeled like a function with continual bounded integer domain.

$$
\begin{aligned}
&\mathit{array} && :: \mathsf{Ix}\ i \Rightarrow (i, i) \to [(i, e)] \to \mathsf{Array}\ i\ e \\
&\mathit{listArray} && :: \mathsf{Ix}\ i \Rightarrow (i, i) \to [e] \to \mathsf{Array}\ i\ e \\
&\mathit{assocs} && :: \mathsf{Ix}\ i \Rightarrow \mathsf{Array}\ i\ e \to [(i, e)] \\
&\mathit{bounds} && :: \mathsf{Array}\ i\ e \to (i, i) \\
&\mathit{elems} && :: \mathsf{Array}\ i\ e \to [e] \\
&(!) && :: \mathsf{Ix}\ i \Rightarrow \mathsf{Array}\ i\ e \to i \to e \\
&(//) && :: \mathsf{Ix}\ i \Rightarrow \mathsf{Array}\ i\ e \to [(i, e)] \to \mathsf{Array}\ i\ e \\
&\mathit{ixmap} && :: (\mathsf{Ix}\ j, \mathsf{Ix}\ i) \Rightarrow \\
& && \qquad (i, i) \to (i \to j) \to \mathsf{Array}\ j\ e \to \mathsf{Array}\ i\ e
\end{aligned}
$$

Data.Vector

Vectors are arrays with a nicer API. Indexing is still $O(1)$, index is Int, indexes start at zero.

```
length      :: Vector a → Int
(!)         :: Vector a → Int → a
(!?)        :: Vector a → Int → Maybe a

unsafeIndex :: Vector a → Int → a    - faster but with UB
unsafeHead  :: Vector a → a
unsafeLast  :: Vector a → a

imap        :: (Int → a → b) → Vector a → Vector b

(//)        :: Vector a → [(Int, a)] → Vector a
update      :: Vector a → Vector (Int, a) → Vector a
update_     :: Vector a → Vector Int → Vector a → Vector a
```

Data.Vector.Mutable

Sometimes the algorithms enjoy the possibility to chaotically write to array indexes. With immutable vectors, each of the writes is essentially $O(n)$.

MVector *s a* stores an implicit state in monad *s* (such as IO, ST), writing is $O(1)$.

```
new :: PrimMonad m ⇒ Int → m (MVector (PrimState m) a)
  {-unsafeNew does not initialize the memory! :) -}

clone :: PrimMonad m ⇒
   MVector (PrimState m) a → m (MVector (PrimState m) a)

read :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → m a
write :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → a → m ()
modify :: PrimMonad m ⇒
   MVector (PrimState m) a → (a → a) → Int → m ()
swap :: PrimMonad m ⇒
   MVector (PrimState m) a → Int → Int → m ()
```

```
import Control.Monad
import qualified Data.Vector.Mutable as V

bubblesort mv = sequence_
   [do [left, right] ← traverse (V.read mv) [i, i + 1]
       when (left > right) $ V.swap mv i (i + 1)
    | let n = V.length mv, _ ← [0 .. n − 1], i ← [0 .. n − 2]]

demoVector = do v ← V.new 5 :: V.IOVector Int
                zipWithM (V.write v) [0 ..] [5, 3, 2, 5, 1]
                return v

printV v = traverse_ (V.read v ≫ print) [0 .. V.length v − 1]

main = do v ← demoVector
          bubblesort v
          printV v
```

```
type Vertex  = Int
type Edge    = (Vertex, Vertex)
type Bounds  = (Vertex, Vertex)
type Table a = Array Vertex a

buildG     :: Bounds → [Edge] → Graph
vertices   :: Graph → [Vertex]
edges      :: Graph → [Edge]

indegree   :: Graph → Table Int
outdegree  :: Graph → Table Int

path       :: Graph → Vertex → Vertex → Bool
reachable  :: Graph → Vertex → [Vertex]
components :: Graph → Forest Vertex
```

```
graphFromEdges :: Ord key ⇒ [(node, key, [key])] →
                             (Graph,
                              Vertex → (node, key, [key]),
                              key →     Maybe Vertex)
```

graphFromEdges' – does not return the key map

```
transposeG :: Graph → Graph

topSort    :: Graph → [Vertex]    - topological sort
dfs        :: Graph → [Vertex] → Forest Vertex    - rooted spanning trees
dff        :: Graph → Forest Vertex    - same for all roots
scc        :: Graph → Forest Vertex    - connected components
bcc        :: Graph → Forest [Vertex]    - 2-connected components
```

Sets and maps

Standard tree-ish containers: Map, IntMap, Set, IntSet

- Int-variants are more optimized
- There are Lazy variants too (occasionally useful)
- Common import method to prevent name collision:

```
import qualified Data.Map.Strict as M
```

$$
\begin{array}{lll}
\mathit{M.fromList} & :: \text{Ord } k \Rightarrow & [(k,a)] \to \text{M.Map } k\ a \\
\mathit{S.fromList} & :: \text{Ord } a \Rightarrow & [a] \to \text{S.Set } a \\
\mathit{M.assocs} & :: & \text{M.Map } k\ a \to [(k,a)] \\
\mathit{M.elems} & :: & \text{M.Map } k\ a \to [a] \\
\mathit{S.elems} & :: & \text{S.Set } a \to [a]
\end{array}
$$

$$
\mathit{S.insert}, \mathit{S.delete} :: \text{Ord } a \Rightarrow a \to \text{S.Set } a \to \text{S.Set } a
$$

$$
\begin{array}{lll}
\mathit{M.insert} & :: \text{Ord } k \Rightarrow & k \to a \to \text{M.Map } k\ a \to \text{M.Map } k\ a \\
\mathit{M.delete} & :: \text{Ord } k \Rightarrow & k \to \text{M.Map } k\ a \to \text{M.Map } k\ a \\
\mathit{M.adjust} & :: \text{Ord } k \Rightarrow & (a \to a) \to k \to \text{M.Map } k\ a \to \text{M.Map } k\ a \\
\text{(M.!)} & :: \text{Ord } k \Rightarrow & \text{M.Map } k\ a \to k \to a \\
\text{(M.!?)} & :: \text{Ord } k \Rightarrow & \text{M.Map } k\ a \to k \to \text{Maybe } a
\end{array}
$$

$$
\mathit{S.union}, \mathit{S.difference}, \mathit{S.intersection} \\
:: \text{Ord } a \Rightarrow \quad \text{S.Set } a \to \text{S.Set } a \to \text{S.Set } a
$$

Hashes vs. functional programming

Package `unordered-containers` additionally contains HashSet and HashMap

NB.: Hashing in containers without designing the hash function yourself is almost always a problem, in case of functional programming and immutable arrays this holds exponentially more. If you survived an advanced data structures course that proves the actual complexity of hashtable operations AND you can prove why there are 3 logarithms in the formula AND you still think the hash tables are a good idea, you can use them.

Implementation:

```
class Hashable a where
  hashWithSalt :: Int → a → Int
member :: (Eq a, Hashable a) ⇒ a → HashSet a → Bool
```

...

Set is not a functor!

Extras: Functional data structures

Immutability of data often breaks the efficiency of traditional data structures.

Functional data structures minimize the amount of reallocation required to work with the structure.

Example: Functional queue

We want a potentially infinite queue with $O(1)$ enqueue a dequeue.

Problems:

- double-linked list can't be done immutably
- change of a single-linked list at the end costs $O(n)$ *allocations*

Example: Functional queue

Solution: let's specialize a write queue and read queue:

```
data FQueue a = FQueue {fqReadEnd :: [a], fqWriteEnd :: [a]}

fqEnqueue a q = q {fqWriteEnd = a : fqWriteEnd q}

fqRefill (FQueue [] x) = FQueue (reverse x) []
fqRefill q = q

fqDequeue q = case fqRefill q of
  r@(FQueue [] _) = Nothing
  FQueue (r : rs) w = Just (r, FQueue rs w)
```

Functional queue operation work in amortized O(1)

Enqueue is visibly $O(1)$.

Dequeue is amortized $O(1)$ — for each element that goes throught the queue, the total work is limited by:

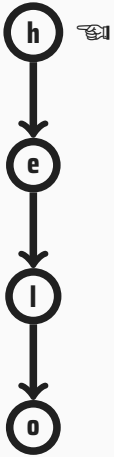
- adding to a beginning of the write-queue list
- 1 step of the reverse algorithm
- removal from the beginning of the read-queue list

Zippers and zipper trees

Zipper is a generic construction for converting data structures to functional ones. Let's demonstrate on trees:

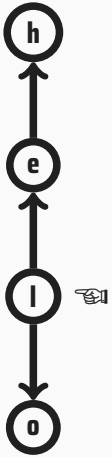
- Usual trees are held by root, the depth of operation is always taken from the root.
- Zipper trees are held by any element, from which the pointers go both up (to the root) and down (to other elements).

Zipper list

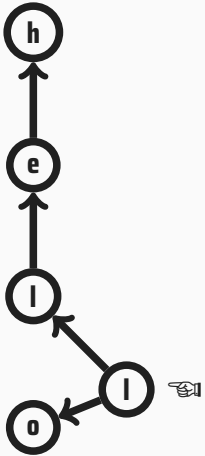


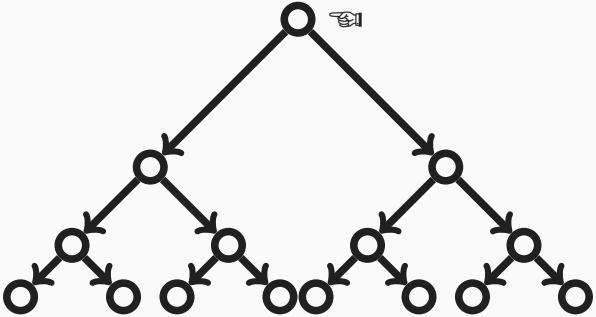
Zipper list

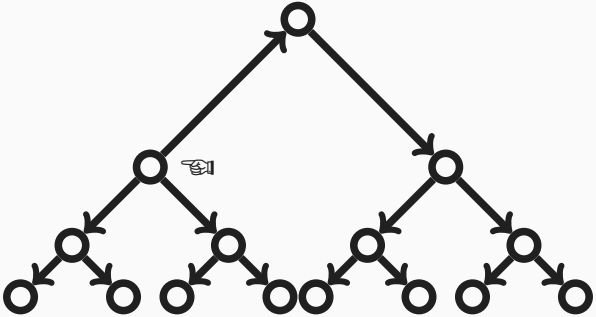
Zipper list

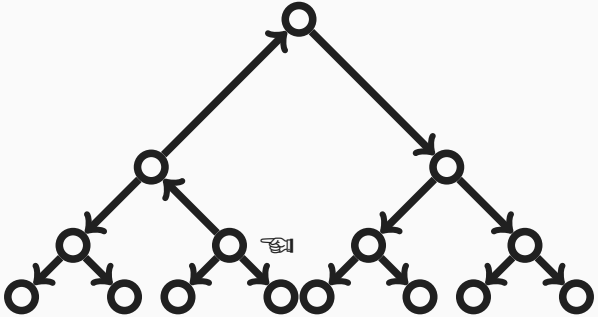


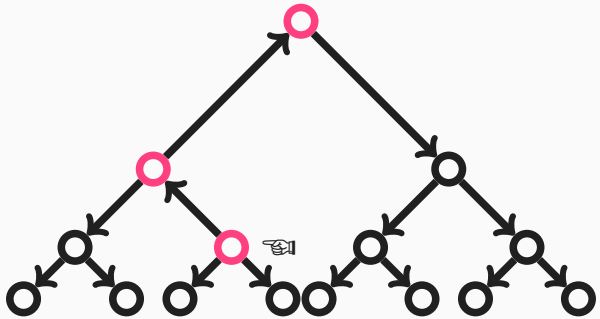
Zipper list











We need to specialize the spine.

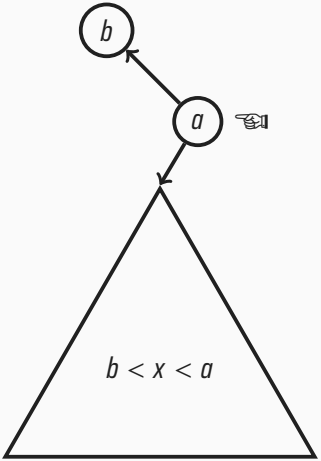
```
data BTree a = BNode (BTree a) a (BTree a) | Nil
data ZSpine a = RootR a (BTree a)
              | RootL (BTree a) a
              | SpineR (ZSpine a) a (BTree a)
              | SpineL (BTree a) a (ZSpine a)
data ZBTree a = Root (BTree a) | NonRoot (BTree a) (ZSpine a)

zUp, zLeft, zRight :: ZBTree a → ZBTree a
```

The implementation can be derived automatically from any data structure using *differentiation on terms*.

✎Gerard Huet, "Functional Pearl: The Zipper." *Journal of Functional Programming* 7 (5), pp. 549–554 (1997).

Avoiding unnecessary returns to root



Is it fast?

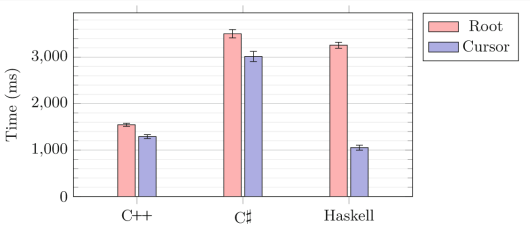


Fig. 6. 2-3 Tree insertion performance

✎Šefl, Vít. “Performance Analysis of Zippers.” In *Declarative Programming and Knowledge Management*, pp. 215-229. Springer, Cham, 2019. `https://arxiv.org/pdf/1908.10926.pdf`

Bonus: Memory allocation in the *nursery* in current GHC RTS is vastly faster than `malloc`.

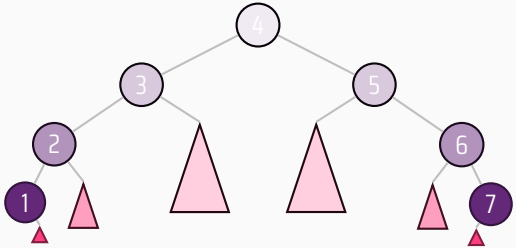
Holding points can be optimized to use-cases

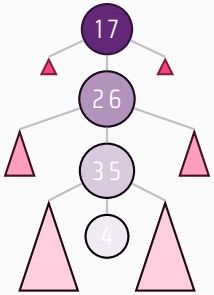
If we can assume “hot spots” in a data structure, we can optimize for that.

Example: Finger trees are held at minimum and maximum element. This allows quick access to the corner elements, and quick splits and merges.

✎Ralf Hinze and Ross Paterson, “Finger trees: a simple general-purpose data structure.” *Journal of Functional Programming* 16:2 (2006), pp. 197–217.

Finger tree (idea)





Finger trees are used in Sequences

Sequence (`Data.Sequence`):

- Requirements: fast access to both ends, log-*n* find of *k*-th element, fast concatenation
- Implementation: 2-3 finger tree with subtree size marks
- Trade-off: sequences can't be infinite (concatenation would not balance)