

# Finger trees:

## The swiss army knife of data structures

RALF HINZE

Institut für Informatik III, Universität Bonn

Römerstraße 164, 53117 Bonn, Germany

Email: [ralf@informatik.uni-bonn.de](mailto:ralf@informatik.uni-bonn.de)

Homepage: <http://www.informatik.uni-bonn.de/~ralf>

October, 2004

Joint work with Ross Paterson, City University, London

(Pick up the slides at [.../~ralf/talks.html#T38](http://www.informatik.uni-bonn.de/~ralf/talks.html#T38).)

# Salient features of finger trees

- ▶ 2-3 finger trees: a purely functional sequence data type
- ▶ a persistent data type
- ▶ deque operations in amortised constant time
- ▶ concatenation and splitting in time logarithmic in the size of the smaller piece
- ▶ general purpose data structure: can serve as a sequence, priority queue, search tree, priority search queue and more
- ▶ performs well in practice
- ▶ much simpler than previous implementations with similar bounds
  - H. Kaplan and R. E. Tarjan, *Purely functional representations of catenable sorted lists*, ACM Symposium on the Theory of Computing, 1996.
  - C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

# Outline of the talk

- ✕ Preliminaries (4–6)
- ✕ Finger trees and catenation (8–16)
- ✕ Annotated finger trees and splitting (18–25)
- ✕ Applications (27–30)

# Preliminaries—monoids and reductions

A type with an associative operation and an identity forms a **monoid**.

```
class Monoid  $\alpha$  where
```

```
   $\emptyset$   ::  $\alpha$ 
```

```
   $(\oplus)$  ::  $\alpha \rightarrow \alpha \rightarrow \alpha$ 
```

A **reduction** is a function that collapses a structure of type  $\varphi \alpha$  into a single value of type  $\alpha$ :

- ▶ empty subtrees are replaced by  $\emptyset$ ;
- ▶ intermediate results are combined using ' $\oplus$ '.

# Preliminaries—skewed reductions

If we arrange that applications of ' $\oplus$ ' are only nested to the right, or to the left, we obtain **skewed reductions**:

**class** *Reduce*  $\varphi$  **where**

*reducer*  $:: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow (\varphi \alpha \rightarrow \beta \rightarrow \beta)$

*reducel*  $:: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \varphi \alpha \rightarrow \beta)$

☞ Skewed reductions need not be written by hand; they can be defined generically for arbitrary  $\varphi$ -structures.

# Reductions—continued

**Alternative reading:** *reducer* ( $\prec$ ) lifts the operation ' $\prec$ ' to  $\varphi$ -structures.

$$\begin{array}{l} toList \quad :: (Reduce \varphi) \Rightarrow \varphi \alpha \rightarrow [\alpha] \\ toList \ s = s :' [] \textbf{ where } (:') = reducer \ (:) \end{array}$$

Think of  $s$  as being cons'ed to the empty list.

# Outline of the talk

- ✓ Preliminaries (4–6)
- ✗ Finger trees and catenation (8–16)
- ✗ Annotated finger trees and splitting (18–25)
- ✗ Applications (27–30)

## 2-3 nodes

Finger trees are constructed from 2-3 nodes:

```
data Node  $\alpha$  = Node2  $\alpha$   $\alpha$  | Node3  $\alpha$   $\alpha$   $\alpha$ 
```

☞ 2-3 nodes contain two or three subtrees, but no keys; all data is stored in the leaves of the tree.



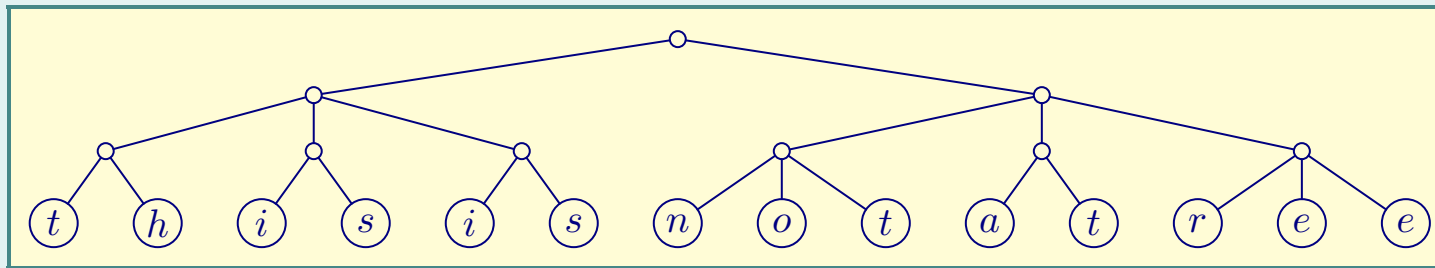
# 2-3 trees

Conventional 2-3 trees can be defined as follows:

```
data Tree  $\alpha$  = Zero  $\alpha$  | Succ (Tree (Node  $\alpha$ ))
```

☞ The *Tree* type is an example of a **non-regular** or **nested type**.

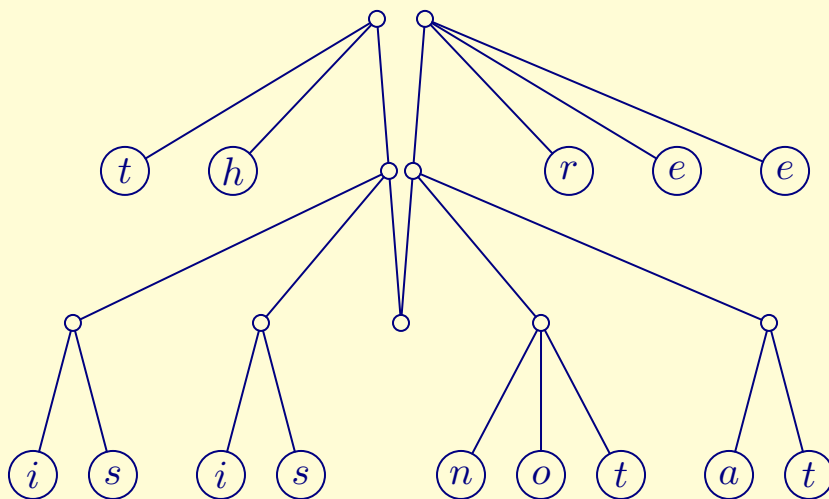
A typical 2-3 tree:



All leaves of a 2-3 tree are at the same depth, the left and right spines have the same length.

## 2-3 finger trees

If we take hold of the end nodes of a 2-3 tree and lift them up together, we obtain a tree that looks like this:



Each pair of nodes on the central spine is merged into a single node (called *Deep*).

☞ A sequence of  $n$  elements is represented by a tree of depth  $\Theta(\log n)$ ; an element  $d$  positions from the nearest end is stored at a depth of  $\Theta(\log d)$ .

## 2-3 finger trees—continued

Finger trees provide efficient access to the ends of a sequence:

```
data FingerTree  $\alpha$  = Empty
                  | Single  $\alpha$ 
                  | Deep (Digit  $\alpha$ ) (FingerTree (Node  $\alpha$ )) (Digit  $\alpha$ )
```

The top level contains elements of type  $\alpha$ , the next of type *Node*  $\alpha$ , and so on: the  $n$ th level contains elements of type *Node* <sup>$n$</sup>   $\alpha$ , namely 2-3 trees of depth  $n$ .

☞ The *FingerTree* type is a second example of a **non-regular** or **nested type**.

A digit is a buffer of **one** to **four** elements, represented as a list to simplify the presentation.

```
type Digit  $\alpha$  = [ $\alpha$ ]
```

# Finger trees—reductions

**instance** *Reduce* *Node* **where**

*reducer* ( $\prec$ ) (*Node2* *a b*)  $z = a \prec (b \prec z)$

*reducer* ( $\prec$ ) (*Node3* *a b c*)  $z = a \prec (b \prec (c \prec z))$

**instance** *Reduce* *FingerTree* **where**

*reducer* ( $\prec$ ) *Empty*  $z = z$

*reducer* ( $\prec$ ) (*Single* *x*)  $z = x \prec z$

*reducer* ( $\prec$ ) (*Deep* *pr m sf*)  $z = pr \prec' (m \prec'' (sf \prec' z))$

**where** ( $\prec'$ ) = *reducer* ( $\prec$ )

( $\prec''$ ) = *reducer* (*reducer* ( $\prec$ ))

## Finger trees—deque operations

## Adding a new element to the left of the sequence:

$(\triangleleft)$	$:: \alpha \rightarrow \text{FingerTree } \alpha \rightarrow \text{FingerTree } \alpha$
$a \triangleleft \text{Empty}$	$= \text{Single } a$
$a \triangleleft \text{Single } b$	$= \text{Deep } [a] \text{ Empty } [b]$
$a \triangleleft \text{Deep } [b, c, d, e] \ m \ sf$	$= \text{Deep } [a, b] (\text{Node3 } c \ d \ e \triangleleft m) \ sf$
$a \triangleleft \text{Deep } pr \ m \ sf$	$= \text{Deep } ([a] \uplus pr) \ m \ sf$

We classify digits of two or three elements (which correspond to nodes) as **safe**, and those of one or four elements as **dangerous**.

☞ A deque operation may only propagate to the next level from a dangerous digit, but in doing so it makes that digit safe, so that the next operation to reach that digit will not propagate.

👉 **Lazy evaluation** guarantees that the amortised bound of  $\Theta(1)$  also holds in a persistent setting.

# Deque operations—continued

In the sequel we shall also require the lifting of ' $\triangleleft$ ':

$$\begin{aligned}(\triangleleft') &:: (\text{Reduce } \varphi) \Rightarrow \varphi \alpha \rightarrow \text{FingerTree } \alpha \rightarrow \text{FingerTree } \alpha \\(\triangleleft') &= \text{reducer } (\triangleleft)\end{aligned}$$

Conversion to a finger tree may be defined using ' $\triangleleft'$ ':

$$\begin{aligned}\text{toTree} &:: (\text{Reduce } \varphi) \Rightarrow \varphi \alpha \rightarrow \text{FingerTree } \alpha \\ \text{toTree } s &= s \triangleleft' \text{ Empty}\end{aligned}$$

# Finger trees—concatenation

Concatenation of two *Deep* trees:

$$\text{Deep } pr_1 \ m_1 \ sf_1 \bowtie \text{Deep } pr_2 \ m_2 \ sf_2 = \text{Deep } pr_1 \ \dots \ sf_2$$

We can use the prefix of the first tree as the prefix of the result, and the suffix of the second tree as the suffix of the result.


To combine the rest to make the new middle subtree, we require a function of type

$$\text{FingerTree } (\text{Node } \alpha) \rightarrow \text{Digit } \alpha \rightarrow \text{Digit } \alpha \rightarrow \text{FingerTree } (\text{Node } \alpha) \rightarrow \text{FingerTree } (\text{Node } \alpha)$$


For simplicity, we combine the two digit arguments into a list of *Nodes*.

# Concatenation—continued

$$\begin{aligned} \text{app3} &:: \text{FingerTree } \alpha \rightarrow [\alpha] \rightarrow \text{FingerTree } \alpha \rightarrow \text{FingerTree } \alpha \\ \text{app3 } \text{Empty } ts \ xs &= ts \triangleleft' xs \\ \text{app3 } xs \ ts \ \text{Empty} &= xs \triangleright' ts \\ \text{app3 } (\text{Single } x) \ ts \ xs &= x \triangleleft (ts \triangleleft' xs) \\ \text{app3 } xs \ ts \ (\text{Single } x) &= (xs \triangleright' ts) \triangleright x \\ \text{app3 } (\text{Deep } pr_1 \ m_1 \ sf_1) \ ts \ (\text{Deep } pr_2 \ m_2 \ sf_2) \\ &= \text{Deep } pr_1 \ (\text{app3 } m_1 \ (\text{nodes } (sf_1 \uplus ts \uplus pr_2)) \ m_2) \ sf_2 \end{aligned}$$

 *nodes* groups a list of at most 12 elements into a list of 2-3 nodes.

$$\begin{aligned} (\boxtimes) &:: \text{FingerTree } \alpha \rightarrow \text{FingerTree } \alpha \rightarrow \text{FingerTree } \alpha \\ xs \boxtimes ys &= \text{app3 } xs \ [] \ ys \end{aligned}$$

 The recursion terminates when we reach the bottom of the shallower tree, so the total time taken is  $\Theta(\log(\min\{n_1, n_2\}))$ .



# Outline of the talk

- ✓ Preliminaries (4–6)
- ✓ Finger trees and catenation (8–16)
- ✗ Annotated finger trees and splitting (18–25)
- ✗ Applications (27–30)

# Annotations

An annotation represents a **cached** reduction with some monoid.

```
class (Monoid  $\nu$ )  $\Rightarrow$  Measured  $\alpha$   $\nu$  where  
   $\| \cdot \| :: \alpha \rightarrow \nu$ 
```

Think of  $\alpha$  as the type of some tree and  $\nu$  as the type of an associated measurement.

☞ The *Measured* class is an example of a **multiple parameter type class**.

# Annotated 2-3 nodes

Measurements are cached in 2-3 nodes:

```
data Node  $\nu$   $\alpha$  = Node2  $\nu$   $\alpha$   $\alpha$  | Node3  $\nu$   $\alpha$   $\alpha$   $\alpha$ 
```

The **smart constructor** *node2* automatically adds an explicit annotation

```
node2      :: (Measured  $\alpha$   $\nu$ )  $\Rightarrow$   $\alpha \rightarrow \alpha \rightarrow$  Node  $\nu$   $\alpha$   
node2 a b = Node2 ( $\|a\| \oplus \|b\|$ ) a b
```

which is then looked-up by the measure function:

```
instance (Monoid  $\nu$ )  $\Rightarrow$  Measured (Node  $\nu$   $\alpha$ )  $\nu$  where  
   $\|$ Node2 v _ _ $\|$     = v  
   $\|$ Node3 v _ _ _ $\|$  = v
```

# Annotated 2-3 finger trees

Measurements are cached in deep nodes:

```
data FingerTree  $\nu$   $\alpha$ 
  = Empty
  | Single  $\alpha$ 
  | Deep  $\nu$  (Digit  $\alpha$ ) (FingerTree  $\nu$  (Node  $\nu$   $\alpha$ )) (Digit  $\alpha$ )
```


# Annotated finger trees—splitting

The central operation that puts the annotations to good use is splitting a sequence on properties of an accumulated measurement.

**Example:** accumulating the size of subtrees yields the positions of elements.

To represent a container split around a distinguished element, with containers of elements to its left and right, we introduce the data type

**data** *Split*  $\tau \alpha = \text{Split } (\tau \alpha) \alpha (\tau \alpha)$

 Splitting a finger tree provides us with a third finger besides the ones at the left and right ends of the tree.

# Splitting—specification

The function *splitTree*, which splits a finger tree, expects three arguments: a predicate on measurements, an accumulator, and a finger tree.

Specification:

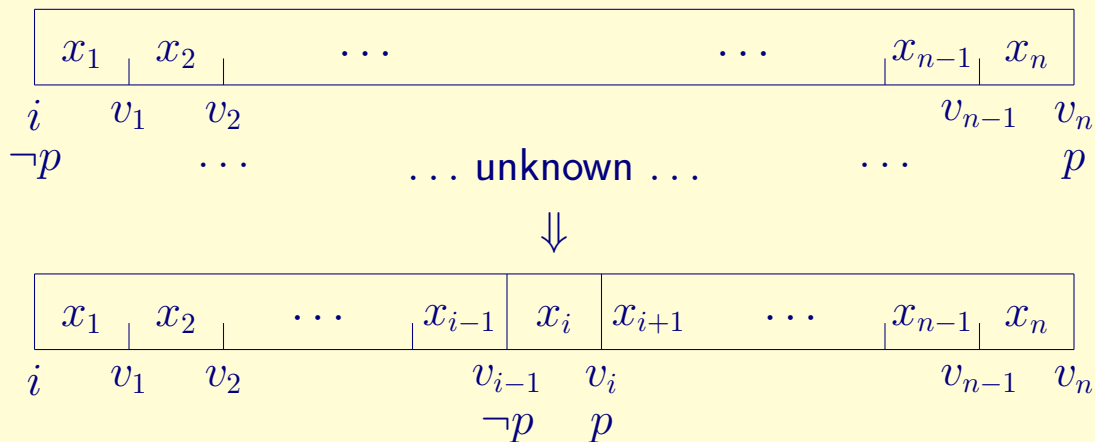
$$\neg p \ i \wedge p \ (i \oplus \|t\|) \implies$$
$$\text{let } \textit{Split} \ l \ x \ r = \textit{splitTree} \ p \ i \ t$$
$$\text{in } \textit{toList} \ l \ ++ \ [x] \ ++ \ \textit{toList} \ r = \textit{toList} \ t$$
$$\wedge \neg p \ (i \oplus \|l\|) \wedge p \ (i \oplus \|l\| \oplus \|x\|)$$

First conjunct: *Split* *l x r* is actually a split of *t*.

Second conjunct: *splitTree* splits *t* at a point where *p* applied to the accumulated measurement changes from *False* to *True*.

☞ The split is unique for **monotonic** predicates satisfying  $p \ x \implies p \ (x \oplus y)$ .

# Specification of splitting—continued




where  $v_j = i \oplus \|x_1\| \oplus \dots \oplus \|x_j\|$  is the accumulated measurement.

☞  $x_i$  is the distinguished element where  $p$  changes from *False* to *True*.

# Splitting a tree

```
splitTree :: (Measured  $\alpha$   $\nu$ )  $\Rightarrow$   
            ( $\nu \rightarrow \text{Bool}$ )  $\rightarrow \nu \rightarrow \text{FingerTree } \nu \alpha \rightarrow \text{Split } (\text{FingerTree } \nu) \alpha$   
splitTree p i (Single x) = Split Empty x Empty  
splitTree p i (Deep _ pr m sf)  
  | p vpr = let Split l x r      = splitDigit p i pr  
            in Split (toTree l) x (deepL r m sf)  
  | p vm  = let Split ml xs mr   = splitTree p vpr m  
            Split l x r      = splitDigit p (vpr  $\oplus$  ||ml||) (toList xs)  
            in Split (deepR pr ml l) x (deepL r mr sf)  
  | else  = let Split l x r      = splitDigit p vm sf  
            in Split (deepR pr m l) x (toTree r)  
where vpr = i  $\oplus$  ||pr||  
      vm  = vpr  $\oplus$  ||m||
```

  $\text{deep}_L$  is a smart variant of *Deep* that allows the prefix to be empty;  
*splitDigit* splits a digit.



# Splitting—continued

Using *splitTree* we can define a function *split* that produces two sequences:

$$\begin{aligned} \textit{split} &:: (\textit{Measured } \alpha \ \nu) \Rightarrow \\ &(\nu \rightarrow \textit{Bool}) \rightarrow \textit{FingerTree } \nu \ \alpha \rightarrow (\textit{FingerTree } \nu \ \alpha, \textit{FingerTree } \nu \ \alpha) \\ \textit{split } p \ \textit{Empty} &= (\textit{Empty}, \textit{Empty}) \\ \textit{split } p \ xs & \\ \quad | \ p \parallel xs &= (l, x \triangleleft r) \\ \quad | \ \textit{else} &= (xs, \textit{Empty}) \\ \textbf{where } \textit{Split } l \ x \ r &= \textit{splitTree } p \ \emptyset \ xs \end{aligned}$$

☞ We require  $\neg p \ \emptyset$  to establish the invariant of *splitTree*.

☞ Interestingly, the specification of *splitTree* can be strengthened so that we don't need any preconditions on *p*!

# Outline of the talk

- ✓ Preliminaries (4–6)
- ✓ Finger trees and catenation (8–16)
- ✓ Annotated finger trees and splitting (18–25)
- ✗ Applications (27–30)

# Application—random-access sequences

**Random-access sequences** support fast positional operations. To this end we annotate finger trees with sizes:

```
newtype Size = Size { getSize :: Nat }  
instance Monoid Size where  
   $\emptyset$  = Size 0  
  Size m  $\oplus$  Size n = Size (m + n)
```

```
newtype Seq  $\alpha$  = Seq (FingerTree Size (Elem  $\alpha$ ))  
  
newtype Elem  $\alpha$  = Elem { getElem ::  $\alpha$  }  
instance Measured (Elem  $\alpha$ ) Size where  
   $\|$  Elem _  $\|$  = Size 1
```

☞ Think of *Elem* as a marker for elements.

## Random-access sequences—continued

The length of a sequence can be computed in constant time:

$$\begin{array}{lcl} length & :: & Seq\ \alpha \rightarrow Nat \\ length\ (Seq\ xs) & = & getSize\ ||xs|| \end{array}$$

Splitting a sequence at a certain position is simply a matter of calling *split* with the appropriate arguments:

$$\begin{array}{lcl} splitAt & :: & Nat \rightarrow Seq \alpha \rightarrow (Seq \alpha, Seq \alpha) \\ splitAt \ i \ (Seq \ xs) & = & (Seq \ l, Seq \ r) \\ \text{where } (l, r) & = & split \ (Size \ i <) \ xs \end{array}$$

# Application—ordered sequences

Another possibility is to annotate with **split** or **signpost keys**:

```
data Key  $\alpha$  = NoKey | Key { getKey ::  $\alpha$  }  
instance Monoid (Key  $\alpha$ ) where  
   $\emptyset$  = NoKey  
   $k \oplus \text{NoKey} = k$   
   $\_ \oplus k = k$ 
```

If we maintain the sequence in key order, we have an implementation of **ordered sequences**:

```
newtype OrdSeq  $\alpha$  = OrdSeq (FingerTree (Key  $\alpha$ ) (Elem  $\alpha$ ))  
  
instance Measured (Elem  $\alpha$ ) (Key  $\alpha$ ) where  
   $\| \text{Elem } x \| = \text{Key } x$ 
```

# Ordered sequences—continued

Partitioning before the first element  $\geq k$ :

$$\begin{aligned} \text{partition} &:: (\text{Ord } \alpha) \Rightarrow \alpha \rightarrow \text{OrdSeq } \alpha \rightarrow (\text{OrdSeq } \alpha, \text{OrdSeq } \alpha) \\ \text{partition } k &(\text{OrdSeq } xs) = (\text{OrdSeq } l, \text{OrdSeq } r) \\ &\textbf{where } (l, r) = \text{split } (\geq \text{Key } k) \text{ } xs \end{aligned}$$

Perhaps surprisingly, insertion can also be defined in terms of *split*:

$$\begin{aligned} \text{insert} &:: (\text{Ord } \alpha) \Rightarrow \alpha \rightarrow \text{OrdSeq } \alpha \rightarrow \text{OrdSeq } \alpha \\ \text{insert } x &(\text{OrdSeq } xs) = \text{OrdSeq } (l \bowtie (\text{Elem } x \triangleleft r)) \\ &\textbf{where } (l, r) = \text{split } (\geq \text{Key } x) \text{ } xs \end{aligned}$$

# Outline of the talk

- ✓ Preliminaries (4–6)
- ✓ Finger trees and catenation (8–16)
- ✓ Annotated finger trees and splitting (18–25)
- ✓ Applications (27–30)

# Conclusion

## Other applications:

- ▶ max-priority queues (annotation: priorities),
- ▶ order statistics (annotation: sizes and split keys),
- ▶ interval trees (annotation: split keys and priorities).

## Variations:

- ▶ a more accurate and efficient implementation of *Digit*:

```
data Digit  $\alpha$  = One    $\alpha$ 
              | Two    $\alpha$   $\alpha$ 
              | Three  $\alpha$   $\alpha$   $\alpha$ 
              | Four   $\alpha$   $\alpha$   $\alpha$   $\alpha$ 
```

- ▶ the reliance on a right identity ( $a \oplus \emptyset = a$ ) can be removed.