Finger trees: a simple general-purpose data structure

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

ROSS PATERSON
Department of Computing, City University
London EC1V OHB, UK
(*e-mail:* `ross@soi.city.ac.uk`)

Abstract

We introduce 2-3 finger trees, a functional representation of persistent sequences supporting access to the ends in amortized constant time, and concatenation and splitting in time logarithmic in the size of the smaller piece. Representations achieving these bounds have appeared previously, but 2-3 finger trees are much simpler, as are the operations on them. Further, by defining the split operation in a general form, we obtain a general purpose data structure that can serve as a sequence, priority queue, search tree, priority search queue and more.

1 Introduction

Lists are the functional programmer's favourite data type. They are well supported by most if not all functional programming languages. However, one could argue that this support sometimes leads programmers to use lists when a more general sequence type would be more appropriate (Okasaki, 2000). The operations one might expect from a sequence abstraction include adding and removing elements at both ends (the deque operations), concatenation, insertion and deletion at arbitrary points, finding an element satisfying some criterion, and splitting the sequence into subsequences based on some property. Many efficient functional implementations of subsets of these operations are known, but supporting more operations efficiently is difficult. The best known general implementations are very complex, and little used.

This paper introduces functional 2-3 finger trees, a general implementation that performs well, but is much simpler than previous implementations with similar bounds. The data structure and its many variations are simple enough that we are able to give a concise yet complete executable description using the functional programming language Haskell (Peyton Jones, 2003). The paper should be accessible to anyone with a basic knowledge of Haskell and its widely used extension to multiple-parameter type classes (Peyton Jones *et al.*, 1997). Although the structure makes essential use of laziness, it is also suitable for strict languages that provide a lazy evaluation primitive.

The rest of the paper is organized as follows. The next section contains some preliminary material used throughout the paper. In Section 3 we describe the finger tree type, an elegant example of a first-order nested type, and show how it supports deque operations and concatenation. In Section 4 we define a variation of the finger tree type that supports a splitting operation, abstracting over the annotation used to guide splitting. In this way we obtain a general purpose data structure that can serve as a sequence, priority queue, search tree, priority search queue and more. Moreover, these trees perform well in each role, though specialized implementations are usually faster. Section 5 concludes, including a discussion of related work.

We have included several exercises to support the digestion of the material and to discuss further variations and extensions.

2 Preliminaries

This section briefly reviews two fundamental concepts, monoids and reductions, that are at the heart of the later development.

2.1 Monoids

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

```
class Monoid a where
  ∅   :: a
  (⊕) :: a → a → a
```

We shall introduce a variety of monoids later on.

A single type is sometimes the carrier of several different monoids. In this case, we will introduce an isomorphic type via a **newtype** declaration for each monoidal structure of interest.

2.2 Right and left reductions

Monoids are often used with reductions. A *reduction* is a function that collapses a structure of type $f\ a$ into a single value of type a (Meertens, 1996): empty subtrees are replaced by a constant, such as $\emptyset$; intermediate results are combined using a binary operation, such as '$\oplus$'. A reduction with a monoid yields the same result however we nest the applications of '$\oplus$', but for a reduction with an arbitrary constant and binary operation we must specify a particular nesting. If we arrange that applications of the combining operation are only nested to the right, or to the left, then we obtain *skewed reductions*, which we single out as a type class.

```
class Reduce f where
  reducer :: (a → b → b) → (f a → b → b)
  reducel :: (b → a → b) → (b → f a → b)
```

Skewed reductions are far more general than reductions: they not only take an arbitrary binary operation and an arbitrary element, the arguments of the binary operation may even have different types.

The order of *reducer*'s parameters is arranged as to suggest an alternative reading: *reducer* $(\prec)$ lifts the operation '$\prec$' to f-structures. As an example, if '$\prec$' adds an element to a container, then *reducer* $(\prec)$ adds a complete f-structure.

Skewed reductions need not be written by hand; they can be defined generically for arbitrary f-structures (Hinze & Jeuring, 2003). Nonetheless, we shall provide all the necessary instances in order to keep the paper self-contained.

On lists, reductions specialize to *fold*s.

$$\begin{aligned}
&\mathbf{instance}\ \mathit{Reduce}\ [\,]\ \mathbf{where}\\
&\quad \mathit{reducer}\ (\prec)\ x\ z = \mathit{foldr}\ (\prec)\ z\ x \quad \text{-- NB. } x \text{ and } z \text{ are swapped}\\
&\quad \mathit{reducel}\ (\succ)\ x\ z = \mathit{foldl}\ (\succ)\ x\ z
\end{aligned}$$

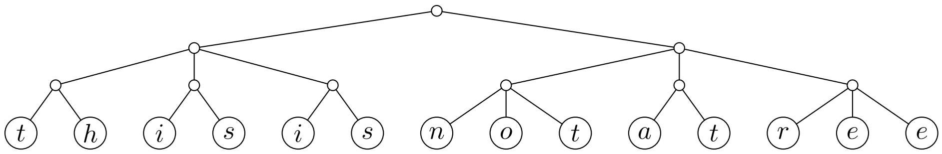
Reductions are particularly useful for converting between structures.

$$\begin{aligned}
&\mathit{toList} \quad :: (\mathit{Reduce}\ f) \Rightarrow f\ a \to [\,a\,]\\
&\mathit{toList}\ s = s\ :'\ [\,]\ \mathbf{where}\ (:') = \mathit{reducer}\ (:)
\end{aligned}$$

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

3 Simple sequences

We first develop the definition of 2-3 finger trees, and then show how they efficiently implement sequences. As a starting point, consider ordinary 2-3 trees, like the one below:



The structural constraint that all leaves occur at the same level may be expressed by defining a *non-regular* or *nested type* (Bird & Meertens, 1998), as follows:

$$\begin{aligned}
&\mathbf{data}\ \mathit{Tree}\ a\ = \mathit{Zero}\ a \mid \mathit{Succ}\ (\mathit{Tree}\ (\mathit{Node}\ a))\\
&\mathbf{data}\ \mathit{Node}\ a = \mathit{Node2}\ a\ a \mid \mathit{Node3}\ a\ a\ a
\end{aligned}$$

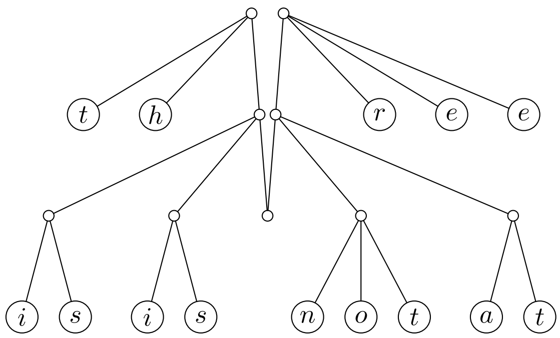
Values of type *Tree* a have the form $\mathit{Succ}^n\ (\mathit{Zero}\ t)$ for some n, where t has type $\mathit{Node}^n\ a$, the type of well-formed 2-3 trees of depth n. Note that the internal nodes of these trees contain no keys; in this section we shall store all data in the leaves of our trees.

Operations on such trees typically take time logarithmic in the size of the tree, but for a sequence implementation we would like to add and remove elements from either end in constant time.

3.1 Finger trees

A structure providing efficient access to nodes of a tree near a distinguished location is called a *finger* (Guibas *et al.*, 1977). In an imperative setting, this would be done by reversing pointers. In a functional setting, we can transform the structure in a manner reminiscent of Huet's "zipper" structure (1997).

To provide efficient access to the ends of a sequence, we wish to place fingers at the left and right ends of this tree. Imagine taking hold of the end nodes of the example tree above and lifting them up together. We would obtain a tree that looks like this:



Because all leaves of the original 2-3 tree were at the same depth, the left and right spines have the same length, and we can pair corresponding nodes on these spines to make a single central spine. Hanging off the sides of these nodes are 2-3 trees, whose depth increases as we descend the central spine. The first level contains two or three elements on each side, while the others have one or two subtrees. At the bottom of the spine, we have either a single 2-3 tree or none, depending on whether the old root had a degree of 3 or 2. We can describe this structure as follows:

$$\begin{aligned}\textbf{data}\ \mathit{FingerTree}\ a &= \mathit{Empty}\\ &\mid \mathit{Single}\ a\\ &\mid \mathit{Deep}\ (\mathit{Digit}\ a)\ (\mathit{FingerTree}\ (\mathit{Node}\ a))\ (\mathit{Digit}\ a)\end{aligned}$$

where a digit is a buffer of elements stored left to right (in the picture digits are depicted by filled circles), here represented as a list to simplify the presentation:

$$\textbf{type}\ \mathit{Digit}\ a = [a]$$

As noted above, in transformed 2-3 trees these lists have length 2 or 3 at the top level and 1 or 2 at lower levels. We shall relax this, allowing between one and four subtrees at each level. As we shall see in the next subsection, this relaxation provides just enough slack to buffer deque operations efficiently. By the way, the buffer type is called *Digit* because finger trees are a so-called *numerical representation* (Okasaki, 1998), a data structure that is modelled after a number system.

The non-regular definition of the *FingerTree* type determines the unusual shape of these trees, which is the key to their performance. The top level contains elements of type a, the next of type $\mathit{Node}\ a$, and so on: the nth level contains elements of type $\mathit{Node}^n\ a$, namely 2-3 trees of depth n. Thus a sequence of n elements is represented by a tree of depth $\Theta(\log n)$. Moreover, an element d positions from the nearest end is stored at a depth of $\Theta(\log d)$ in the tree.

Before moving on, we instantiate the generic definition of reduction to these types. Reduction of nodes is trivial:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Reduce\ Node}\ \textbf{where} \\
\quad \mathit{reducer}\ (\prec)\ (\mathit{Node2}\ a\ b)\quad z = a \prec (b \prec z) \\
\quad \mathit{reducer}\ (\prec)\ (\mathit{Node3}\ a\ b\ c)\ z = a \prec (b \prec (c \prec z)) \\
\quad \mathit{reducel}\ (\succ)\ z\ (\mathit{Node2}\ b\ a)\quad = (z \succ b) \succ a \\
\quad \mathit{reducel}\ (\succ)\ z\ (\mathit{Node3}\ c\ b\ a) = ((z \succ c) \succ b) \succ a
\end{array}
$$

Reduction of finger trees uses single and double liftings of the binary operation:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Reduce\ FingerTree}\ \textbf{where} \\
\quad \mathit{reducer}\ (\prec)\ \mathit{Empty}\qquad\qquad z = z \\
\quad \mathit{reducer}\ (\prec)\ (\mathit{Single}\ x)\qquad z = x \prec z \\
\quad \mathit{reducer}\ (\prec)\ (\mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf})\ z = \mathit{pr} \prec' (m \prec'' (\mathit{sf} \prec' z)) \\
\qquad \textbf{where}\ (\prec')\ = \mathit{reducer}\ (\prec) \\
\qquad\qquad\quad (\prec'') = \mathit{reducer}\ (\mathit{reducer}\ (\prec)) \\
\quad \mathit{reducel}\ (\succ)\ z\ \mathit{Empty}\qquad\qquad = z \\
\quad \mathit{reducel}\ (\succ)\ z\ (\mathit{Single}\ x)\qquad = z \succ x \\
\quad \mathit{reducel}\ (\succ)\ z\ (\mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf}) = ((z \succ' \mathit{pr}) \succ'' m) \succ' \mathit{sf} \\
\qquad \textbf{where}\ (\succ')\ = \mathit{reducel}\ (\succ) \\
\qquad\qquad\quad (\succ'') = \mathit{reducel}\ (\mathit{reducel}\ (\succ))
\end{array}
$$

3.2 Deque operations

We shall now proceed to show that finger trees make efficient deques, with all operations taking $\Theta(1)$ amortized time. This holds even if the structure is used in a *persistent* manner, where several versions of a tree may coexist (Driscoll *et al.*, 1989). The analysis is essentially the same as that for Okasaki's implicit deques (1998). Indeed, the *FingerTree* type differs from implicit deques primarily in storing *Node*s instead of pairs, with a corresponding increase in the size of digits. This change provides the necessary flexibility for efficient implementation of concatenation and splitting, as we shall see later.

Adding a new element to the left of the sequence is trivial, except when the initial buffer already contains four elements. In that case, we push three of the elements as a *Node*, leaving two behind:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{infixr}\ 5 \triangleleft} \\
(\triangleleft) & :: a \rightarrow \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a \\
a \triangleleft \mathit{Empty} & = \mathit{Single}\ a \\
a \triangleleft \mathit{Single}\ b & = \mathit{Deep}\ [a]\ \mathit{Empty}\ [b] \\
a \triangleleft \mathit{Deep}\ [b, c, d, e]\ m\ \mathit{sf} & = \mathit{Deep}\ [a, b]\ (\mathit{Node3}\ c\ d\ e \triangleleft m)\ \mathit{sf} \\
a \triangleleft \mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf} & = \mathit{Deep}\ ([a] +\!\!+\ \mathit{pr})\ m\ \mathit{sf}
\end{array}
$$

The function '$\triangleleft$' requires a non-schematic form of recursion, called *polymorphic* *recursion* (Mycroft, 1984): the recursive call adds a node (not an element) to a finger tree that contains *Node*s.

Adding to the right end of the sequence is the mirror image of the above:

$$
\begin{array}{l}
\textbf{infixl}\ 5 \triangleright \\
(\triangleright) \qquad\qquad\qquad\qquad :: \mathit{FingerTree}\ a \rightarrow a \rightarrow \mathit{FingerTree}\ a \\
\mathit{Empty} \qquad\qquad\quad\ \triangleright a = \mathit{Single}\ a \\
\mathit{Single}\ b \qquad\qquad\quad \triangleright a = \mathit{Deep}\ [b]\ \mathit{Empty}\ [a] \\
\mathit{Deep}\ \mathit{pr}\ m\ [e, d, c, b] \triangleright a = \mathit{Deep}\ \mathit{pr}\ (m \triangleright \mathit{Node3}\ e\ d\ c)\ [b, a] \\
\mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf} \qquad\quad \triangleright a = \mathit{Deep}\ \mathit{pr}\ m\ (\mathit{sf} +\!\!+ [a])
\end{array}
$$

We shall also require the liftings of '$\triangleleft$' and '$\triangleright$':

$$
\begin{array}{l}
(\triangleleft') :: (\mathit{Reduce}\ f) \Rightarrow f\ a \rightarrow \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a \\
(\triangleleft') = \mathit{reducer}\ (\triangleleft) \\
(\triangleright') :: (\mathit{Reduce}\ f) \Rightarrow \mathit{FingerTree}\ a \rightarrow f\ a \rightarrow \mathit{FingerTree}\ a \\
(\triangleright') = \mathit{reducel}\ (\triangleright)
\end{array}
$$

Conversion to a finger tree may be defined using either '$\triangleleft'$' or '$\triangleright'$'; we use the former:

$$
\begin{array}{l}
\mathit{toTree} \quad :: (\mathit{Reduce}\ f) \Rightarrow f\ a \rightarrow \mathit{FingerTree}\ a \\
\mathit{toTree}\ s = s \triangleleft' \mathit{Empty}
\end{array}
$$

To deconstruct a sequence, we define a type that expresses a view of the left end of a sequence as either empty or an element together with the rest of the sequence:

$$\textbf{data}\ \mathit{View}_L\ s\ a = \mathit{Nil}_L \mid \mathit{Cons}_L\ a\ (s\ a)$$

The left view of a sequence maps a sequence to this type:

$$
\begin{array}{l}
\mathit{view}_L \qquad\qquad\qquad :: \mathit{FingerTree}\ a \rightarrow \mathit{View}_L\ \mathit{FingerTree}\ a \\
\mathit{view}_L\ \mathit{Empty} \qquad\quad\ = \mathit{Nil}_L \\
\mathit{view}_L\ (\mathit{Single}\ x) \qquad = \mathit{Cons}_L\ x\ \mathit{Empty} \\
\mathit{view}_L\ (\mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf}) = \mathit{Cons}_L\ (\mathit{head}\ \mathit{pr})\ (\mathit{deep}_L\ (\mathit{tail}\ \mathit{pr})\ m\ \mathit{sf})
\end{array}
$$

Since the prefix *pr* of a *Deep* tree contains at least one element, we can get its head in constant time. However, the tail of the prefix may be empty, and thus unsuitable as a first argument to the *Deep* constructor. Hence we define a smart constructor that differs from *Deep* by allowing the prefix to contain zero to four elements, and in the empty case uses a left view of the middle tree to construct a tree of the correct shape:

$$
\begin{array}{l}
\mathit{deep}_L \qquad\quad :: [a] \rightarrow \mathit{FingerTree}\ (\mathit{Node}\ a) \rightarrow \mathit{Digit}\ a \rightarrow \mathit{FingerTree}\ a \\
\mathit{deep}_L\ [\,]\ m\ \mathit{sf}\ \ = \textbf{case}\ \mathit{view}_L\ m\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Nil}_L \qquad\ \ \rightarrow \mathit{toTree}\ \mathit{sf} \\
\qquad\qquad\qquad\qquad \mathit{Cons}_L\ a\ m' \rightarrow \mathit{Deep}\ (\mathit{toList}\ a)\ m'\ \mathit{sf} \\
\mathit{deep}_L\ \mathit{pr}\ m\ \mathit{sf} = \mathit{Deep}\ \mathit{pr}\ m\ \mathit{sf}
\end{array}
$$

Then we can define separate functions *isEmpty*, head_L, and tail_L using the view:

$$
\begin{array}{lll}
\mathit{isEmpty} & :: \mathit{FingerTree}\ a \rightarrow \mathit{Bool} & \\
\mathit{isEmpty}\ x & = \textbf{case}\ \mathit{view}_L\ x\ \textbf{of}\ \mathit{Nil}_L & \rightarrow \mathit{True} \\
 & \qquad\qquad\qquad\qquad\quad \mathit{Cons}_L\ \_\ \_ & \rightarrow \mathit{False}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{head}_L & :: \mathit{FingerTree}\ a \rightarrow a \\
\mathit{head}_L\ x & = \textbf{case}\ \mathit{view}_L\ x\ \textbf{of}\ \mathit{Cons}_L\ a\ \_ \rightarrow a
\end{array}
$$

$$
\begin{array}{ll}
\mathit{tail}_L & :: \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a \\
\mathit{tail}_L\ x & = \textbf{case}\ \mathit{view}_L\ x\ \textbf{of}\ \mathit{Cons}_L\ \_\ x' \rightarrow x'
\end{array}
$$

In a lazy language like Haskell, the tail part of the view is not computed unless it is used. In a strict language, it might be more useful to provide the three separate functions as primitives.

We omit the code for the right view, which is a mirror image of the left view, including a function deep_R corresponding to deep_L.

Each deque operation may recurse down the spine of the finger tree, and thus take $\Theta(\log n)$ time in the worst case. However, it can be shown that these operations take only $\Theta(1)$ amortized time, even in a persistent setting. The analysis is essentially the same as Okasaki's for implicit deques (Okasaki, 1998). We outline the main points here.

We classify digits of two or three elements (which are isomorphic to elements of type *Node a*) 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. Thus, at most half of the operations descend one level, at most a quarter two levels, and so on. Consequently, in a sequence of operations, the average cost is constant.

The same bounds hold in a persistent setting if subtrees are suspended using lazy evaluation. This ensures that transformations deep in the spine do not take place until a subsequent operation needs to go down that far. Because of the above properties of safe and dangerous digits, by that time enough cheap shallow operations will have been performed to pay for this expensive evaluation.

This argument may be formalized using Okasaki's debit analysis (1998) by assigning the suspension of the middle tree in each *Deep* node as many debits as the node has safe digits. (It may have 0, 1 or 2.) A deque operation that descends k levels turns k dangerous digits into safe ones, and thus creates k debits to pay for the work involved. Applying Okasaki's technique of *debit passing* to any debits already attached to these k nodes, one can show that each operation must discharge at most one debit, so the deque operations run in $\Theta(1)$ amortized time.

Lazy evaluation is the norm in a language such as Haskell. In a strict language that provides a lazy evaluation primitive, we need only suspend the middle subtree of each *Deep* node, so only $\Theta(\log n)$ suspensions are required in a tree of size n. Even in a lazy language, some space could be saved in practice by ensuring that these were the only suspensions in the tree, for example by using Haskell's strictness annotations. Indeed we might wish to make the operations even more strict, for example by forcing the middle subtree m in the recursive case of '$\triangleleft$' to avoid building a chain of suspensions. According to the above debit analysis, this suspension has been paid for by this time, so the amortized bounds would be unaffected.

While we chose to push a *Node3* in '$\triangleleft$' and '$\triangleright$', leaving two elements behind, pushing a *Node2* instead also yields the same bounds. Our choice means that the tree is mostly populated with nodes of degree 3, and therefore slightly shallower. It is also faster in queue-like applications, where elements are mostly added at one end and mostly removed from the other, because it means that only every third operation propagates to the next level.

Exercise 1. In the above presentation, we represented digits as lists for simplicity. A more accurate and efficient implementation would use the type

$$
\begin{aligned}
\textbf{data}\ \mathit{Digit}\ a = {} & \mathit{One}\ \ \ a \\
\mid {} & \mathit{Two}\ \ \ a\ a \\
\mid {} & \mathit{Three}\ a\ a\ a \\
\mid {} & \mathit{Four}\ \ a\ a\ a\ a
\end{aligned}
$$

Rework the above definitions to use this definition of *Digit*. □

3.3 Concatenation

We now consider concatenation of sequences represented as finger trees. The tree *Empty* will be an identity for concatenation, and concatenation with *Single*s reduces to '$\triangleleft$' or '$\triangleright$', so the only difficult case is concatenation of two *Deep* trees. 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

$$
\begin{aligned}
& \mathit{FingerTree}\ (\mathit{Node}\ a) \to \mathit{Digit}\ a \to \mathit{Digit}\ a \to \mathit{FingerTree}\ (\mathit{Node}\ a) \to \\
& \quad \mathit{FingerTree}\ (\mathit{Node}\ a)
\end{aligned}
$$

Since it is easy to combine the two digit arguments into a list of *Node*s, we can obtain a recursive function by generalizing the concatenation function to take an additional list of elements:

$$
\begin{aligned}
& \mathit{app3} && :: \mathit{FingerTree}\ a \to [a] \to \mathit{FingerTree}\ a \to \mathit{FingerTree}\ a \\
& \mathit{app3}\ \mathit{Empty}\ ts\ xs && = ts \triangleleft'\ xs \\
& \mathit{app3}\ xs\ ts\ \mathit{Empty} && = xs \triangleright'\ ts \\
& \mathit{app3}\ (\mathit{Single}\ x)\ ts\ xs && = x \triangleleft (ts \triangleleft'\ xs) \\
& \mathit{app3}\ xs\ ts\ (\mathit{Single}\ x) && = (xs \triangleright'\ ts) \triangleright x \\
& \mathit{app3}\ (\mathit{Deep}\ pr_1\ m_1\ sf_1)\ ts\ (\mathit{Deep}\ pr_2\ m_2\ sf_2) && \\
& && = \mathit{Deep}\ pr_1\ (\mathit{app3}\ m_1\ (\mathit{nodes}\ (sf_1 +\!\!+\ ts +\!\!+\ pr_2))\ m_2)\ sf_2
\end{aligned}
$$

where '$\triangleleft'$' and '$\triangleright'$' are the previously defined liftings of '$\triangleleft$' and '$\triangleright$', and *nodes* groups a list of elements into *Node*s:

$$
\begin{aligned}
& \mathit{nodes} && :: [a] \to [\mathit{Node}\ a] \\
& \mathit{nodes}\ [a, b] && = [\mathit{Node2}\ a\ b] \\
& \mathit{nodes}\ [a, b, c] && = [\mathit{Node3}\ a\ b\ c] \\
& \mathit{nodes}\ [a, b, c, d] && = [\mathit{Node2}\ a\ b, \mathit{Node2}\ c\ d] \\
& \mathit{nodes}\ (a : b : c : xs) && = \mathit{Node3}\ a\ b\ c : \mathit{nodes}\ xs
\end{aligned}
$$

Now we can concatenate two finger trees by starting with an empty list between them:

$$(\bowtie) \quad :: \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a \rightarrow \mathit{FingerTree}\ a$$
$$xs \bowtie ys = \mathit{app3}\ xs\ [\,]\ ys$$

It is easy to check that in each of the invocations of *app3* arising from '$\bowtie$', the list argument has length at most 4, so each of these invocations takes $\Theta(1)$ time. The recursion terminates when we reach the bottom of the shallower tree, with up to 4 insertions, so the total time taken is $\Theta(\log(\min\{n_1, n_2\}))$, where n_1 and n_2 are the numbers of elements in the argument trees.

Note that the debit analysis of Section 3.2 remains valid, because although '$\bowtie$' must discharge any debits on the nodes on the spines that are merged, there will be at most $\Theta(\log(\min\{n_1, n_2\}))$ of these.

Exercise 2. The definition of *nodes* prefers to use *Node3* where possible, but any other choices would still yield the same asymptotic bounds. If *nodes* were redefined to prefer *Node2*, what would be the bound on the length of the list arguments in the invocations of *app3* arising from '$\bowtie$'? □

Exercise 3. Consider replacing the *Digit* type as discussed in Exercise 1, and also using a variant of *app3* for each possible size of the middle argument. This leads to a faster but much larger definition. *Hint:* it is both simpler and less error-prone to write a program that *generates* the Haskell source for this definition. □

4 Annotated sequences

Finger trees serve remarkably well as implementations of catenable deques. In this section we shall present a modification of these trees, which also provides positional and set-theoretic operations, such as taking or dropping the first n elements of a sequence or partitioning an ordered sequence around a pivot element.

As a common characteristic, all of these operations involve searching for an element with a certain property. For example, partitioning an ordered sequence involves searching for the minimum element that is larger than or equal to the pivot element. Thus, if we want to implement the additional operations with reasonable efficiency, we must be able to steer this search. To this end we augment the internal nodes of a finger tree by an additional field that contains positional or ordering information or both.

4.1 Measurements

We take the fairly liberal view that an annotation represents a *cached* reduction with some monoid. Reductions, possibly cached, are captured by the following class declaration (for clarity, we use mixfix notation, which is not possible in Haskell).

$$\textbf{class}\ (\mathit{Monoid}\ v) \Rightarrow \mathit{Measured}\ a\ v\ \textbf{where}$$
$$\quad \|\cdot\| :: a \rightarrow v$$

Think of a as the type of some tree and v as the type of an associated measurement. As customary, v must possess a monoidal structure so that we can easily combine measurements of subtrees independent of their nesting. For instance, the 'size' measure maps onto the monoid of natural numbers with addition. In this particular example, the binary operation is not only associative but also commutative, but this need not be the case in general.

The class *Measured* is parameterized by both a and v and represents a true many-to-many relation on types: the same measurement may be imposed on a variety of types; the same type may be measured in many different ways. Unfortunately, this means that the class declaration is not legal Haskell 98, which only admits single-parameter type classes. Multiple parameter classes (Peyton Jones *et al.*, 1997), however, are supported as an extension by the most widely used Haskell implementations.

4.2 Caching measurements

Obtaining a measure should be cheap. We shall ensure that finger trees support this operation with a bounded number of '$\oplus$' operations. To this end we cache measurements in 2-3 nodes.

$$\textbf{data}\ \mathit{Node}\ v\ a = \mathit{Node2}\ v\ a\ a \mid \mathit{Node3}\ v\ a\ a\ a$$

The *smart constructors* *node2* and *node3* automatically add an explicit annotation

$$\begin{array}{ll}
\mathit{node2} & :: (\mathit{Measured}\ a\ v) \Rightarrow a \to a \to \mathit{Node}\ v\ a \\
\mathit{node2}\ a\ b & = \mathit{Node2}\ (\|a\| \oplus \|b\|)\ a\ b \\
\mathit{node3} & :: (\mathit{Measured}\ a\ v) \Rightarrow a \to a \to a \to \mathit{Node}\ v\ a \\
\mathit{node3}\ a\ b\ c & = \mathit{Node3}\ (\|a\| \oplus \|b\| \oplus \|c\|)\ a\ b\ c
\end{array}$$

This annotation is then retrieved by the measure function:

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Monoid}\ v) \Rightarrow \mathit{Measured}\ (\mathit{Node}\ v\ a)\ v\ \textbf{where} \\
\quad \|\mathit{Node2}\ v\ _\ _\| \quad = v \\
\quad \|\mathit{Node3}\ v\ _\ _\ _\| = v
\end{array}$$

The smart constructors and the instance declaration are completely generic: they work for arbitrary annotations.

Digits are measured on the fly.

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Measured}\ a\ v) \Rightarrow \mathit{Measured}\ (\mathit{Digit}\ a)\ v\ \textbf{where} \\
\quad \|xs\| = \mathit{reducel}\ (\lambda i\ a \to i \oplus \|a\|)\ \emptyset\ xs
\end{array}$$

Recall that a digit is a buffer of one to four elements. Because the length of the buffer is bounded by a constant, the number of '$\oplus$' operations is also bounded. Another possibility is to cache the measure of a digit, adding to the cost of digit construction but yielding a saving when computing the measure. The choice between these strategies would depend on the expected balance of query and modification operations, but they would differ only by a constant factor.

Finger trees are modified in a similar manner to 2-3 nodes. Again there is a choice of caching strategies, but it suffices to cache deep nodes:

$$\begin{aligned}&\textbf{data}\ \mathit{FingerTree}\ v\ a = \mathit{Empty}\\ &\qquad\qquad\qquad\qquad\quad \mid \mathit{Single}\ a\\ &\qquad\qquad\qquad\qquad\quad \mid \mathit{Deep}\ v\ (\mathit{Digit}\ a)\ (\mathit{FingerTree}\ v\ (\mathit{Node}\ v\ a))\ (\mathit{Digit}\ a)\end{aligned}$$

The top level of an annotated finger tree contains elements of type a, the second level of type $\mathit{Node}\ v\ a$, the third of type $\mathit{Node}\ v\ (\mathit{Node}\ v\ a)$, and so on. Whereas the tree type a changes from level to level, the annotation type v remains the same. In a nutshell, *FingerTree* is nested in a, but regular in v.

Again, a smart constructor takes care of filling in the missing annotation

$$\begin{aligned}&\mathit{deep} :: (\mathit{Measured}\ a\ v) \Rightarrow\\ &\qquad \mathit{Digit}\ a \to \mathit{FingerTree}\ v\ (\mathit{Node}\ v\ a) \to \mathit{Digit}\ a \to \mathit{FingerTree}\ v\ a\\ &\mathit{deep}\ pr\ m\ sf = \mathit{Deep}\ (\|pr\| \oplus \|m\| \oplus \|sf\|)\ pr\ m\ sf\end{aligned}$$

This annotation is then available for subsequent measurements:

$$\begin{aligned}&\textbf{instance}\ (\mathit{Measured}\ a\ v) \Rightarrow \mathit{Measured}\ (\mathit{FingerTree}\ v\ a)\ v\ \textbf{where}\\ &\quad \|\mathit{Empty}\| &&= \emptyset\\ &\quad \|\mathit{Single}\ x\| &&= \|x\|\\ &\quad \|\mathit{Deep}\ v\ \_\ \_\ \_\| &&= v\end{aligned}$$

4.3 Construction, deconstruction and concatenation

None of the sequence operations introduced in Section 3 benefits from the annotations. Consequently, their re-implementations can simply ignore the additional field in pattern matching and recompute it when constructing nodes. The latter is most easily accomplished by replacing the original constructors by their smart counterparts (in expressions). The re-implementation of '$\triangleleft$' illustrates the necessary amendments:

$$\begin{aligned}&\textbf{infixr}\ 5 \triangleleft\\ &(\triangleleft) :: (\mathit{Measured}\ a\ v) \Rightarrow a \to \mathit{FingerTree}\ v\ a \to \mathit{FingerTree}\ v\ a\\ &a \triangleleft \mathit{Empty} &&= \mathit{Single}\ a\\ &a \triangleleft \mathit{Single}\ b &&= \mathit{deep}\ [a]\ \mathit{Empty}\ [b]\\ &a \triangleleft \mathit{Deep}\ \_\ [b, c, d, e]\ m\ sf &&= \mathit{deep}\ [a, b]\ (\mathit{node3}\ c\ d\ e \triangleleft m)\ sf\\ &a \triangleleft \mathit{Deep}\ \_\ pr\ m\ sf &&= \mathit{deep}\ ([a] +\!\!+\ pr)\ m\ sf\end{aligned}$$

The remaining operations are changed accordingly. Their time bounds remain as in Section 3, except that they now refer to the number of '$\oplus$' operations required to compute the new annotations.

4.4 Splitting

A *scan* is an iterated reduction that maps the sequence $x_1, \ldots, x_n$ of elements to the sequence $v_1, \ldots, v_n$ of accumulated measurements where $v_j = i \oplus \|x_1\| \oplus \cdots \oplus \|x_j\|$ for some start value i. The next operation we introduce will use the annotations to efficiently split a sequence based on properties of the scan with the monoid. For instance, a 'size' scan yields the positions of elements so that we can split a sequence at a given position. Since '$\oplus$' is associative, a scan can be calculated on the fly using the cached reductions in the nodes.

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

$$\mathbf{data}\ \mathit{Split}\ f\ a = \mathit{Split}\ (f\ a)\ a\ (f\ a)$$

The *Split* type singles out an element and its context, in a similar way to a finger in an imperative setting or a zipper in a functional setting. Splitting a finger tree provides us with a third finger besides the ones at the left and right ends of the tree.

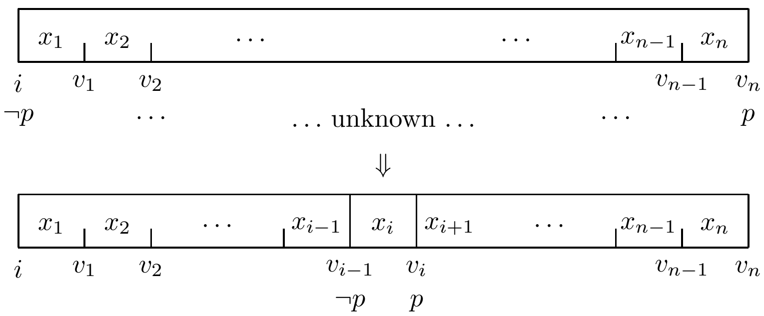
The function *splitTree*, which splits a finger tree, expects three arguments: a predicate p on measurements, an accumulator i, and a finger tree t. It is specified

$$\begin{array}{l} \neg p\ i\ \wedge\ p\ (i \oplus \|t\|) \Longrightarrow \\ \qquad \mathbf{let}\ \mathit{Split}\ l\ x\ r = \mathit{splitTree}\ p\ i\ t\ \mathbf{in}\ \ \mathit{toList}\ l \mathbin{+\!\!+} [x] \mathbin{+\!\!+} \mathit{toList}\ r = \mathit{toList}\ t \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \neg p\ (i \oplus \|l\|)\ \wedge\ p\ (i \oplus \|l\| \oplus \|x\|) \end{array}$$

The first conjunct states that $\mathit{Split}\ l\ x\ r$ is actually a split of t, that is, the order and multiplicity of elements is preserved. As usual for this type of condition, it will be fairly obvious from the program text that the property is satisfied. Indeed, we take great care that the relative order of elements is preserved in the syntax of our code.

The other conjunct states that *splitTree* splits t at a point where p applied to the accumulated measurement changes from *False* to *True*. The precondition expresses that this measurement is *False* at the left end and *True* at the right end, so there is at least one such point. The precondition implies, in particular, that t is non-empty. Splitting is most useful if there is *exactly* one such point, but we need not assume this. Uniqueness of the split is guaranteed for *monotonic* predicates satisfying $p\ x \Longrightarrow p\ (x \oplus y)$ or equivalently $\neg p\ (x \oplus y) \Longrightarrow \neg p\ x$ for all x and y. Alternatively, an application might maintain an invariant on sequences that guarantees a unique split for predicates of interest; Section 4.7 provides an example of this.

The picture below illustrates the functioning of *splitTree*.



where $v_j = i \oplus \|x_1\| \oplus \cdots \oplus \|x_j\|$. In the diagram, x_i is the distinguished element where p changes from *False* to *True*.

The helper function *splitDigit*, which satisfies an analogous specification, nicely illustrates splitting along a scan.

$$
\begin{array}{l}
\mathit{splitDigit} :: (\mathit{Measured}\ a\ v) \Rightarrow (v \to \mathit{Bool}) \to v \to \mathit{Digit}\ a \to \mathit{Split}\ [\,]\ a \\
\mathit{splitDigit}\ p\ i\ [a] = \mathit{Split}\ [\,]\ a\ [\,] \\
\mathit{splitDigit}\ p\ i\ (a : \mathit{as}) \\
\quad \mid p\ i' \qquad\quad = \mathit{Split}\ [\,]\ a\ \mathit{as} \\
\quad \mid \mathit{otherwise} \quad = \mathbf{let}\ \mathit{Split}\ l\ x\ r = \mathit{splitDigit}\ p\ i'\ \mathit{as}\ \mathbf{in}\ \mathit{Split}\ (a : l)\ x\ r \\
\quad \mathbf{where}\ i' = i \oplus \|a\|
\end{array}
$$

The type signature makes precise that splitting a digit may produce empty left and right buffers ('$[\,]$' is the list type constructor). The reader is invited to check that the invariant expressed in the specification is indeed preserved through the recursive calls. Since *splitDigit* scans the list from left to right, it actually splits the list at the *first* point where p flips. The function *splitTree* proceeds in a similar fashion, but it may skip whole subtrees (which become deeper as we descend the tree). This behaviour is necessary to make *splitTree* run in logarithmic time, but it also means that there is no guarantee that the calculated split is the first one. The definition of *splitTree* uses the auxiliary function deep_L from Section 3.2 and its mirror image deep_R.

$$
\begin{array}{l}
\mathit{splitTree} :: (\mathit{Measured}\ a\ v) \Rightarrow \\
\qquad\qquad (v \to \mathit{Bool}) \to v \to \mathit{FingerTree}\ v\ a \to \mathit{Split}\ (\mathit{FingerTree}\ v)\ a \\
\mathit{splitTree}\ p\ i\ (\mathit{Single}\ x) = \mathit{Split}\ \mathit{Empty}\ x\ \mathit{Empty} \\
\mathit{splitTree}\ p\ i\ (\mathit{Deep}\ \_\ \mathit{pr}\ m\ \mathit{sf}) \\
\quad \mid p\ \mathit{vpr} \qquad = \mathbf{let}\ \mathit{Split}\ l\ x\ r \qquad = \mathit{splitDigit}\ p\ i\ \mathit{pr} \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \ \mathit{Split}\ (\mathit{toTree}\ l)\ x\ (\mathit{deep}_L\ r\ m\ \mathit{sf}) \\
\quad \mid p\ \mathit{vm} \qquad = \mathbf{let}\ \mathit{Split}\ \mathit{ml}\ \mathit{xs}\ \mathit{mr} = \mathit{splitTree}\ p\ \mathit{vpr}\ m \\
\qquad\qquad\qquad\qquad\quad\ \mathit{Split}\ l\ x\ r \qquad = \mathit{splitDigit}\ p\ (\mathit{vpr} \oplus \|\mathit{ml}\|)\ (\mathit{toList}\ \mathit{xs}) \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \ \mathit{Split}\ (\mathit{deep}_R\ \mathit{pr}\ \mathit{ml}\ l)\ x\ (\mathit{deep}_L\ r\ \mathit{mr}\ \mathit{sf}) \\
\quad \mid \mathit{otherwise} = \mathbf{let}\ \mathit{Split}\ l\ x\ r \qquad = \mathit{splitDigit}\ p\ \mathit{vm}\ \mathit{sf} \\
\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \ \mathit{Split}\ (\mathit{deep}_R\ \mathit{pr}\ m\ l)\ x\ (\mathit{toTree}\ r) \\
\quad \mathbf{where}\ \mathit{vpr} = i \quad\ \oplus \|\mathit{pr}\| \\
\qquad\qquad\ \mathit{vm}\ = \mathit{vpr} \oplus \|m\|
\end{array}
$$

The cost of *splitTree* is proportional to the depth of the split point in the input finger tree, which as we observed in Section 3 is $\Theta(\log d)$, where d is the distance to the nearest end. Thus the cost of *splitTree* is $\Theta(\log(\min\{n_l, n_r\}))$ '$\oplus$' operations, where n_l and n_r are the sizes of the subtrees returned. As with '$\bowtie$', this bound is also sufficient to discharge any debits on spine nodes that are split, so that the debit analysis of Section 3.2 remains valid.

The above specification of *splitTree* is pleasingly simple, but we can strengthen it slightly. Plainly, *splitTree* requires a non-empty tree. An empty middle subtree m of a *Deep* node is not passed to a recursive call of *splitTree*, because in that case $\mathit{vm} = \mathit{vpr}$. Note also that the preconditions $\neg p\ i$ and $p\ (i \oplus \|t\|)$ are used only if the split occurs at one or other end of the sequence. Consequently, if we treat those cases specially it suffices to require that the initial tree t is non-empty:

$$
\begin{array}{l}
t \neq \mathit{Empty} \Longrightarrow \\
\qquad \textbf{let } \mathit{Split}\ l\ x\ r = \mathit{splitTree}\ p\ i\ t \textbf{ in } \begin{array}[t]{l} \mathit{toList}\ l +\!\!+ [x] +\!\!+ \mathit{toList}\ r = \mathit{toList}\ t \\ \wedge\ (l = \mathit{Empty} \vee \neg p\ (i \oplus \|l\|)) \\ \wedge\ (r = \mathit{Empty} \vee p\ (i \oplus \|l\| \oplus \|x\|)) \end{array}
\end{array}
$$

Thus the trees to the left and to the right are either empty, in which case we can't say anything about p, or they satisfy the previous conditions. This specification allows us to define a function *split* that produces two sequences, specified by

$$
\textbf{let } (l, r) = \mathit{split}\ p\ \mathit{xs} \textbf{ in } \begin{array}[t]{l} \mathit{toList}\ l +\!\!+ \mathit{toList}\ r = \mathit{toList}\ \mathit{xs} \\ \wedge\ (l = \mathit{Empty} \vee \neg p\ \|l\|) \\ \wedge\ (r = \mathit{Empty} \vee p\ (\|l\| \oplus \|\mathit{head}_L\ r\|)) \end{array}
$$

and implemented as follows:

$$
\begin{array}{l}
\mathit{split} :: (\mathit{Measured}\ a\ v) \Rightarrow \\
\qquad (v \to \mathit{Bool}) \to \mathit{FingerTree}\ v\ a \to (\mathit{FingerTree}\ v\ a, \mathit{FingerTree}\ v\ a) \\
\mathit{split}\ p\ \mathit{Empty} = (\mathit{Empty}, \mathit{Empty}) \\
\mathit{split}\ p\ \mathit{xs} \\
\quad \mid p\ \|\mathit{xs}\| \qquad = (l, x \triangleleft r) \\
\quad \mid \mathit{otherwise} = (\mathit{xs}, \mathit{Empty}) \\
\quad \textbf{where } \mathit{Split}\ l\ x\ r = \mathit{splitTree}\ p\ \emptyset\ \mathit{xs}
\end{array}
$$

Note that we don't need any preconditions on p. If we relied on the weaker specification, we would require $\neg p\ \emptyset$ even though p is never applied to $\emptyset$ in the code. As an aside, the only monotonic predicate that holds for $\emptyset$ is the one that is always true: when p is monotonic, $p\ \emptyset$ implies $p\ x$ for all x.

Finally, we define convenient shortcuts for the left and right part of a split.

$$
\begin{array}{l}
\mathit{takeUntil}, \mathit{dropUntil} :: (\mathit{Measured}\ a\ v) \Rightarrow \\
\qquad\qquad\qquad\qquad (v \to \mathit{Bool}) \to \mathit{FingerTree}\ v\ a \to \mathit{FingerTree}\ v\ a \\
\mathit{takeUntil}\ p = \mathit{fst}\ \cdot \mathit{split}\ p \\
\mathit{dropUntil}\ p = \mathit{snd} \cdot \mathit{split}\ p
\end{array}
$$

Exercise 4. Give a more efficient version of the following function:

$$
\begin{array}{l}
\mathit{lookupTree} :: (\mathit{Measured}\ a\ v) \Rightarrow (v \to \mathit{Bool}) \to v \to \mathit{FingerTree}\ v\ a \to (v, a) \\
\mathit{lookupTree}\ p\ i\ t = \textbf{let } \mathit{Split}\ l\ x\ r = \mathit{splitTree}\ p\ i\ t \textbf{ in } (i \oplus \|l\|, x)
\end{array}
$$

assuming $\neg p\ i\ \wedge\ p\ (i \oplus \|t\|)$. □

4.5 Application: random-access sequences

Let's move on to our first application. An efficient implementation of sequences ought to support fast positional operations such as accessing the nth element or splitting a sequence at a certain position. To this end we annotate finger trees with sizes.

$$\begin{array}{l}\textbf{newtype}\ \mathit{Size} = \mathit{Size}\{\,\mathit{getSize} :: \mathbb{N}\,\}\\ \qquad\qquad\qquad\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord})\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Monoid}\ \mathit{Size}\ \textbf{where}\\ \quad \emptyset \qquad\qquad\qquad\quad = \mathit{Size}\ 0\\ \quad \mathit{Size}\ m \oplus \mathit{Size}\ n = \mathit{Size}\ (m+n)\end{array}$$

As announced in Section 2.1 we introduce a new type because the underlying type $\mathbb{N}$ of natural numbers is the carrier of several different monoids. A second new type is needed for the elements of the sequence.

$$\textbf{newtype}\ \mathit{Elem}\ a = \mathit{Elem}\{\,\mathit{getElem} :: a\,\}$$

Think of *Elem* as a marker for elements. Now, we can define

$$\textbf{newtype}\ \mathit{Seq}\ a = \mathit{Seq}\ (\mathit{FingerTree}\ \mathit{Size}\ (\mathit{Elem}\ a))$$

$$\begin{array}{l}\textbf{instance}\ \mathit{Measured}\ (\mathit{Elem}\ a)\ \mathit{Size}\ \textbf{where}\\ \quad \|\mathit{Elem}\ \_\| = \mathit{Size}\ 1\end{array}$$

The instance declaration serves as a base case for measuring sizes: the size of an element is one.

In a library, the types *Size* and *Elem* would be hidden from the user by defining suitable wrapper functions. Note in this respect that the Haskell Report guarantees that the use of **newtype**s does not incur any run-time penalty.

The length of a sequence can now be computed in constant time.

$$\begin{array}{ll}\mathit{length} & :: \mathit{Seq}\ a \to \mathbb{N}\\ \mathit{length}\ (\mathit{Seq}\ \mathit{xs}) & = \mathit{getSize}\ \|\mathit{xs}\|\end{array}$$

Given these prerequisites, splitting a sequence at a certain position is simply a matter of calling *split* with the appropriate arguments.

$$\begin{array}{ll}\mathit{splitAt} & :: \mathbb{N} \to \mathit{Seq}\ a \to (\mathit{Seq}\ a, \mathit{Seq}\ a)\\ \mathit{splitAt}\ i\ (\mathit{Seq}\ \mathit{xs}) & = (\mathit{Seq}\ l, \mathit{Seq}\ r)\\ \quad \textbf{where}\ (l, r) & = \mathit{split}\ (\mathit{Size}\ i <)\ \mathit{xs}\end{array}$$

As an aside, the predicate $(\mathit{Size}\ i <)$ is monotonic as $i < m$ implies $i < m + n$ for natural numbers m and n, so in this case the split is unique.

The implementation of indexing is equally straightforward.

$$\begin{array}{ll}(!) & :: \mathit{Seq}\ a \to \mathbb{N} \to a\\ \mathit{Seq}\ \mathit{xs}\ !\ i & = \mathit{getElem}\ x\\ \quad \textbf{where}\ \mathit{Split}\ \_\ x\ \_ & = \mathit{splitTree}\ (\mathit{Size}\ i <)\ (\mathit{Size}\ 0)\ \mathit{xs}\end{array}$$

Exercise 5. Use the definition of the *Seq* type to define a specialization of the deque operations, noting that it is sometimes faster to recompute sizes using subtraction instead of addition. □

4.6 Application: max-priority queues

A second interesting choice is to annotate with the maximum operation. Though this operation is associative, it might not have an identity. We can turn it into a monoid by adjoining an identity element, here *MInfty* (minus infinity).

$$\begin{array}{l}
\mathbf{data}\ \mathit{Prio}\ a = \mathit{MInfty} \mid \mathit{Prio}\ a \\
\qquad\qquad\qquad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) \\
\mathbf{instance}\ (\mathit{Ord}\ a) \Rightarrow \mathit{Monoid}\ (\mathit{Prio}\ a)\ \mathbf{where} \\
\quad \emptyset \qquad\qquad\qquad = \mathit{MInfty} \\
\quad \mathit{MInfty} \oplus p \qquad = p \\
\quad p \qquad \oplus \mathit{MInfty} = p \\
\quad \mathit{Prio}\ m \oplus \mathit{Prio}\ n\ = \mathit{Prio}\ (m\ \text{`}\mathit{max}\text{`}\ n)
\end{array}$$

Unsurprisingly, measuring with this monoid gives us max-priority queues.

$$\begin{array}{l}
\mathbf{newtype}\ \mathit{PQueue}\ a = \mathit{PQueue}\ (\mathit{FingerTree}\ (\mathit{Prio}\ a)\ (\mathit{Elem}\ a)) \\
\mathbf{instance}\ (\mathit{Ord}\ a) \Rightarrow \mathit{Measured}\ (\mathit{Elem}\ a)\ (\mathit{Prio}\ a)\ \mathbf{where} \\
\quad \|\mathit{Elem}\ x\| = \mathit{Prio}\ x
\end{array}$$

Interestingly, the type *PQueue* is quite close to 2-3-4 heaps (Cormen *et al.*, 2001), which are priority queues based on 2-3-4 trees: all the data is stored in the leaves; the inner nodes hold the largest keys of their subtrees. As is typical of heaps, there is no constraint on the order of the elements within the leaves. Consequently, to insert an element we simply add it to the queue using '$\lhd$' or '$\rhd$'. Extracting the element with the largest key is implemented in terms of *splitTree* (we assume that the queue is non-empty).

$$\begin{array}{l}
\mathit{extractMax} \qquad\qquad\quad :: (\mathit{Ord}\ a) \Rightarrow \mathit{PQueue}\ a \to (a, \mathit{PQueue}\ a) \\
\mathit{extractMax}\ (\mathit{PQueue}\ q) = (x, \mathit{PQueue}\ (l \bowtie r)) \\
\quad \mathbf{where}\ \mathit{Split}\ l\ (\mathit{Elem}\ x)\ r = \mathit{splitTree}\ (\|q\| \leqslant)\ \emptyset\ q
\end{array}$$

Exercise 6. Show that it is possible to find an element $\geqslant$ any k in $\Theta(\log(n))$ time, and all m of them in $\Theta(m \log(n/m))$ time. □

4.7 Application: ordered sequences

The binary operation that always selects its second argument is also associative. As with *max* in the last section, it can be made into a monoid by adjoining an identity element, here called *NoKey*.

$$\begin{array}{l}
\mathbf{data}\ \mathit{Key}\ a = \mathit{NoKey} \mid \mathit{Key}\ a \\
\qquad\qquad\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Ord}) \\
\mathbf{instance}\ \mathit{Monoid}\ (\mathit{Key}\ a)\ \mathbf{where} \\
\quad \emptyset \qquad\quad\ = \mathit{NoKey} \\
\quad k \oplus \mathit{NoKey} = k \\
\quad _ \oplus k \qquad = k
\end{array}$$

In a tree annotated using this monoid, each subtree is annotated with the last key it contains. If we maintain the sequence in key order, we have an implementation of *ordered sequences*, with these annotations serving as *split* or *signpost keys*:

$$\begin{array}{l}
\mathbf{newtype}\ \mathit{OrdSeq}\ a = \mathit{OrdSeq}\ (\mathit{FingerTree}\ (\mathit{Key}\ a)\ (\mathit{Elem}\ a)) \\
\mathbf{instance}\ \mathit{Measured}\ (\mathit{Elem}\ a)\ (\mathit{Key}\ a)\ \mathbf{where} \\
\quad \|\mathit{Elem}\ x\| = \mathit{Key}\ x
\end{array}$$

The *Key* monoid can be seen as an optimization of the *Prio* monoid: if the sequence is ordered, maximum simplifies to 'second' with *NoKey* playing the role of *MInfty*.

Ordered sequences subsume priority queues, as we have immediate access to the smallest *and* the greatest element, and search trees, as we can partition ordered sequences in logarithmic time.

The first operation on ordered sequences partitions before the first element $\geqslant k$:

$$
\begin{array}{l}
\mathit{partition} :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{OrdSeq}\ a \rightarrow (\mathit{OrdSeq}\ a, \mathit{OrdSeq}\ a) \\
\mathit{partition}\ k\ (\mathit{OrdSeq}\ \mathit{xs}) = (\mathit{OrdSeq}\ l, \mathit{OrdSeq}\ r) \\
\quad \mathbf{where}\ (l, r) = \mathit{split}\ (\geqslant \mathit{Key}\ k)\ \mathit{xs}
\end{array}
$$

Perhaps surprisingly, insertion and deletion can also be defined in terms of *split*. The implementations below are both logarithmic, though special purpose versions would have smaller constant factors.

$$
\begin{array}{l}
\mathit{insert} :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{OrdSeq}\ a \rightarrow \mathit{OrdSeq}\ a \\
\mathit{insert}\ x\ (\mathit{OrdSeq}\ \mathit{xs}) = \mathit{OrdSeq}\ (l \bowtie (\mathit{Elem}\ x \triangleleft r)) \\
\quad \mathbf{where}\ (l, r) = \mathit{split}\ (\geqslant \mathit{Key}\ x)\ \mathit{xs} \\
\mathit{deleteAll} :: (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{OrdSeq}\ a \rightarrow \mathit{OrdSeq}\ a \\
\mathit{deleteAll}\ x\ (\mathit{OrdSeq}\ \mathit{xs}) = \mathit{OrdSeq}\ (l \bowtie r') \\
\quad \mathbf{where}\ (l,\ r)\ \ = \mathit{split}\ (\geqslant \mathit{Key}\ x)\ \mathit{xs} \\
\qquad\qquad\ (\_, r') = \mathit{split}\ (> \mathit{Key}\ x)\ r
\end{array}
$$

The non-strict split of *xs* yields a third finger pointing to the position where x must be inserted or deleted. This technique is reminiscent of splay trees (Sleator & Tarjan, 1985) except that the subsequent append ('$\bowtie$') does away with the additional finger. Deletion is defined to remove an arbitrary number of elements. The cost of the first *split* in *deleteAll* is logarithmic in the distance to the nearest end, while the cost of the second is logarithmic in the number of matches.

Building upon concatenation and split we can also define the *merge* of two ordered sequences.

$$
\begin{array}{l}
\mathit{merge} :: (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSeq}\ a \rightarrow \mathit{OrdSeq}\ a \rightarrow \mathit{OrdSeq}\ a \\
\mathit{merge}\ (\mathit{OrdSeq}\ \mathit{xs})\ (\mathit{OrdSeq}\ \mathit{ys}) = \mathit{OrdSeq}\ (\mathit{merge}'\ \mathit{xs}\ \mathit{ys}) \\
\quad \mathbf{where}\ \mathit{merge}'\ \mathit{as}\ \mathit{bs} = \mathbf{case}\ \mathit{view}_L\ \mathit{bs}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nil}_L \qquad\quad \rightarrow \mathit{as} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Cons}_L\ a\ \mathit{bs}' \rightarrow l \bowtie (a \triangleleft \mathit{merge}'\ \mathit{bs}'\ r) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{where}\ (l, r) = \mathit{split}\ (> \|a\|)\ \mathit{as}
\end{array}
$$

The function partitions the two input sequences into the minimum number of segments that must be reordered to obtain an ordered output sequence. The algorithm is similar to the merge of lists. It is, however, more efficient since it uses exponential and binary search rather than linear search. It can be shown that *merge* is asymptotically optimal: it takes $\Theta(m \log(n/m))$ amortized time where m and n are the lengths of the shorter and longer input sequences respectively (Moffat *et al.*, 1992). Furthermore, *merge* exhibits an *adaptive behaviour*: for sequences *xs* and *ys* with $\mathit{max}\ \mathit{xs} \leqslant \mathit{min}\ \mathit{ys}$ (or vice versa) merging degenerates to concatenation. So, in this favourable case we have a running time of $\Theta(\log m)$.

Exercise 7. In the above implementation, all the annotations stored in the tree have the form *Key* x, which suggests an optimization. Redefine *splitTree*, *deep* and the *Measured* instance for lists to avoid relying on $\emptyset$ being the *right* identity of '$\oplus$', so we can use the binary function $\lambda x\ y \rightarrow y$ instead. □

Exercise 8. Implement the intersection of two ordered sequences. □

Exercise 9. Combine sizes with split keys to get order statistics. The ith order statistic of a collection of n elements is the ith smallest element (Cormen *et al.*, 2001). □

4.8 Application: interval trees

Our final application augments finger trees to support query operations on sets of intervals: we show how to find an interval that overlaps with a given interval in $\Theta(\log(n))$, and all m of them in $\Theta(m \log(n/m))$ time. This application is particularly interesting as it makes use of two annotations that are maintained in parallel (a similar application is the subject of Exercise 9). For this we use the product of monoids, given by

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Monoid}\ a, \mathit{Monoid}\ b) \Rightarrow \mathit{Monoid}\ (a, b)\ \textbf{where} \\
\quad \emptyset \qquad\qquad\qquad\ = (\emptyset, \emptyset) \\
\quad (a, b) \oplus (a', b') = (a \oplus a', b \oplus b')
\end{array}
$$

A closed interval i is an ordered pair of real numbers

$$\textbf{data}\ \mathit{Interval} = \mathit{Interval}\{\mathit{low} :: \mathbb{R}, \mathit{high} :: \mathbb{R}\}$$

with $\mathit{low}\ i \leqslant \mathit{high}\ i$. Two intervals i and j *overlap* when

$$\mathit{low}\ i \leqslant \mathit{high}\ j \ \wedge\ \mathit{low}\ j \leqslant \mathit{high}\ i.$$

An interval tree is a finger tree augmented by split keys and priorities.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{IntervalTree} = \mathit{IntervalTree}\ (\mathit{FingerTree}\ (\mathit{Key}\ \mathbb{R}, \mathit{Prio}\ \mathbb{R})\ \mathit{Interval}) \\
\textbf{instance}\ \mathit{Measured}\ \mathit{Interval}\ (\mathit{Key}\ \mathbb{R}, \mathit{Prio}\ \mathbb{R})\ \textbf{where} \\
\quad \|i\| = (\mathit{Key}\ (\mathit{low}\ i), \mathit{Prio}\ (\mathit{high}\ i))
\end{array}
$$

We order the intervals by their low endpoints, and additionally annotate with the maximum of the high endpoints. Interval trees are an instance of *priority search queues* (McCreight, 1985), which simultaneously support dictionary operations on keys and priority queue operations on priorities.

The predicates below are useful for querying keys and priorities.

$$
\begin{array}{l}
\mathit{atleast}, \mathit{greater}\ :: \mathbb{R} \rightarrow (\mathit{Key}\ \mathbb{R}, \mathit{Prio}\ \mathbb{R}) \rightarrow \mathit{Bool} \\
\mathit{atleast}\ \ k\ (\_, n) = \mathit{Prio}\ k \leqslant n \\
\mathit{greater}\ k\ (n, \_) = n > \mathit{Key}\ k
\end{array}
$$

The function *intervalSearch* finds an interval overlapping with a given interval in the above times, if such an interval exists.

$$
\begin{array}{l}
\mathit{intervalSearch} \qquad\qquad :: \mathit{IntervalTree} \rightarrow \mathit{Interval} \rightarrow \mathit{Maybe\ Interval} \\
\mathit{intervalSearch}\ (\mathit{IntervalTree}\ t)\ i \\
\quad \mid \mathit{atleast}\ (\mathit{low}\ i)\ \|t\| \ \wedge\ \mathit{low}\ x \leqslant \mathit{high}\ i = \mathit{Just}\ x \\
\quad \mid \mathit{otherwise} \qquad\qquad\qquad\qquad\quad = \mathit{Nothing} \\
\quad \mathbf{where}\ \mathit{Split}\ \_\ x\ \_ = \mathit{splitTree}\ (\mathit{atleast}\ (\mathit{low}\ i))\ \emptyset\ t
\end{array}
$$

The *splitTree* gives us the interval x with the smallest low endpoint whose high endpoint is at least the low endpoint of the query interval—the initial conjunct $\mathit{atleast}\ (\mathit{low}\ i)\ \|t\|$ ensures that such an interval exists. It then remains to check that $\mathit{low}\ x \leqslant \mathit{high}\ i$. If this test fails, then we know that there can't be any intersecting interval: any x' to the left of x in the sequence doesn't satisfy $\mathit{low}\ i \leqslant \mathit{high}\ x'$; any x' to the right doesn't satisfy $\mathit{low}\ x' \leqslant \mathit{high}\ i$.

Finding all intervals is just as easy:

$$
\begin{array}{l}
\mathit{intervalMatch} \qquad\qquad\qquad\quad :: \mathit{IntervalTree} \rightarrow \mathit{Interval} \rightarrow [\mathit{Interval}] \\
\mathit{intervalMatch}\ (\mathit{IntervalTree}\ t)\ i = \mathit{matches}\ (\mathit{takeUntil}\ (\mathit{greater}\ (\mathit{high}\ i))\ t) \\
\quad \mathbf{where}\ \mathit{matches}\ xs = \mathbf{case}\ \mathit{view}_L\ (\mathit{dropUntil}\ (\mathit{atleast}\ (\mathit{low}\ i))\ xs)\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Nil}_L \qquad\quad \rightarrow [\,] \\
\qquad\qquad\qquad\qquad\qquad \mathit{Cons}_L\ x\ xs' \rightarrow x : \mathit{matches}\ xs'
\end{array}
$$

The application of *takeUntil* selects the intervals x with $\mathit{low}\ x \leqslant \mathit{high}\ i$, so it remains to select the intervals x from this sequence that also satisfy $\mathit{high}\ x \leqslant \mathit{low}\ i$. However, these intervals are not necessarily adjacent in the sequence, because it is ordered by the low endpoint of the intervals. For example, a long interval that extends to i might be followed in the sequence by a shorter one that does not. Hence we use *dropUntil* to find the next intersecting interval.

5 Related work and conclusion

Finger trees were originally defined by reversing certain pointers in a search tree to accelerate operations in the vicinity of distinguished positions in the tree, known as fingers (Guibas *et al.*, 1977). The original versions used various forms of B-trees, including 2-3 trees, variants using AVL trees have subsequently appeared (Tsakalidis, 1985).

Our functional 2-3 finger trees are an instance of a general design technique introduced by Okasaki (1998), called *implicit recursive slowdown*. We have already noted that these trees are an extension of his implicit deque structure, replacing pairs with 2-3 nodes to provide the flexibility required for efficient concatenation and splitting. Okasaki also presents a wealth of functional implementations of sequences and other data types. Implementations of functional deques with constant time concatenation are known (Kaplan & Tarjan, 1995; Okasaki, 1997), but these structures do not admit efficient insertion, deletion or splitting. Kaplan and Tarjan (1996) defined two functional data structures with the same time bounds as here, except that their bounds are worst case, and do not require laziness. They achieved this using structures based on segmented number systems, which are considerably more involved than the structure used here. Their second implementation is isomorphic to a restricted version of 2-3 finger trees, though their representation using

	Time (in ns) per operation randomly selected from			
	$\triangleleft$, $view_L$ (stack)	$\triangleright$, $view_L$ (queue)	$\triangleleft$, $view_L$, $\triangleright$, $view_R$ (deque)	*index*
Bankers queue	51	147	—	—
Bankers deque	56	229	75	—
Catenable deque	78	215	100	—
Skew binary random access list	44	—	—	295
Finger tree	67	106	89	—
Finger tree with sizes	74	128	94	482

Table 1. *Comparing persistent sequence implementations*

segmented spines is vastly more complex, as are the operations required to preserve their invariants. They also sketched a third implementation, claimed to provide a doubly logarithmic concatenation time, but many details of this variant are unclear.

The biggest advantage of 2-3 finger trees is the simplicity of the data structure and operations: the implementations of the deque operations follow naturally from the structure of finger trees, and so do the implementations of concatenation and splitting. In addition, first experiences with a prototypical implementation suggest that finger trees perform well in practice.

For example, Table 1 compares times for a number of persistent implementations of sequences. The times are averaged over 5,000,000 operations on a sequence of 10,000 elements, on a Pentium III 450 with GHC 6.4. Concatenation is not considered, as it is difficult to separate the cost of building a sequence using concatenations from that of accessing it. We conclude that finger trees offer a general purpose implementation of sequences that is competitive with existing persistent implementations of subsets of the operations, though ephemeral implementations will usually be faster. As search trees, finger trees seem to be 3 to 5 times slower than the best balanced binary tree implementations. Nevertheless, they may be competitive when a wider set of operations is required.

The treatment of annotations nicely demonstrates the benefits of abstraction: a single generic split function serves a multitude of purposes.

Acknowledgements

We are grateful to the anonymous referees for pointing out several typos and for valuable suggestions regarding presentation. Special thanks go to Ian Bayley for pointing out some infelicities in the introductory sections.

References

Bird, R. and Meertens, L. (1998) Nested datatypes. Jeuring, J. (ed), *Fourth International Conference on Mathematics of Program Construction, MPC'98, Marstrand, Sweden.* Lecture Notes in Computer Science 1422, pp. 52–67. Springer-Verlag.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001) *Introduction to Algorithms*. Second edn. The MIT Press.

Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E. (1989) Making data structures persistent. *Journal of Computer and System Sciences* **38**(1):86–124.

Guibas, L. J., McCreight, E. M., Plass, M. F. and Roberts, J. R. (1977) A new representation for linear lists. *Conference Record of the Ninth Annual ACM Symposium on Theory of Computing* pp. 49–60.

Hinze, R. and Jeuring, J. (2003) Generic Haskell: Practice and theory. Backhouse, R. and Gibbons, J. (eds), *Generic Programming: Advanced Lectures*. Lecture Notes in Computer Science 2793, pp. 1–56. Springer-Verlag.

Huet, G. (1997) Functional Pearl: The Zipper. *J. Functional Programming* **7**(5):549–554.

Kaplan, H. and Tarjan, R. E. (1995) Persistent lists with catenation via recursive slow-down. *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing* pp. 93–102.

Kaplan, H. and Tarjan, R. E. (1996) Purely functional representations of catenable sorted lists. *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing* pp. 202–211.

McCreight, E. M. (1985) Priority search trees. *SIAM Journal on Computing* **14**(2):257–276.

Meertens, L. (1996) Calculate polytypically! Kuchen, H. and Swierstra, S. (eds), *Proceedings 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, PLILP'96, Aachen, Germany.* Lecture Notes in Computer Science 1140, pp. 1–16. Springer-Verlag.

Moffat, A., Petersson, O. and Wormald, N. (1992) Sorting and/by merging finger trees. *ISAAC: 3rd International Symposium on Algorithms and Computation*, vol. 650, pp. 499–509.

Mycroft, A. (1984) Polymorphic type schemes and recursive definitions. Paul, M. and Robinet, B. (eds), *Proceedings of the International Symposium on Programming, 6th Colloquium, Toulouse, France.* Lecture Notes in Computer Science 167, pp. 217–228.

Okasaki, C. (1997) Catenable double-ended queues. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming* pp. 66–74. *ACM SIGPLAN Notices,* 32(8), August 1997.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Okasaki, C. (2000) Breadth-first numbering: lessons from a small exercise in algorithm design. *ACM SIGPLAN Notices* **35**(9):131–136.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries.* Cambridge University Press.

Peyton Jones, S., Jones, M. and Meijer, E. (1997) Type classes: Exploring the design space. *Proceedings of the Haskell Workshop, Amsterdam, The Netherlands.*

Sleator, D. D. and Tarjan, R. E. (1985) Self-adjusting binary search trees. *Journal of the ACM* **32**(3):652–686.

Tsakalidis, A. K. (1985) AVL-trees for localized search. *Information and Control* **67**:173–194.