Numerical Representations as Higher-Order Nested Datatypes

December 1998

RALF HINZE

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

Abstract

Number systems serve admirably as templates for container types: a container object of size n is modelled after the representation of the number n and operations on container objects are modelled after their number-theoretic counterparts. Binomial queues are probably the first data structure that was designed with this analogy in mind. In this paper we show how to express these so-called numerical representations as higher-order nested datatypes. A nested datatype allows to capture the structural invariants of a numerical representation, so that the violation of an invariant can be detected at compile-time. We develop a programming method which allows to adapt algorithms to the new representation in a mostly straightforward manner. The framework is employed to implement three different container types: binary random-access lists, binomial queues, and 2-3 finger search trees. The latter data structure, which is treated in some depth, can be seen as the main innovation from a data-structural point of view. It appears that 2-3 finger search trees are the best known purely functional implementation of ordered sequences. In detail, 2-3 finger search trees support the operations *findMin*, *findMax*, *deleteMin*, and *deleteMax* in $\Theta(1)$ amortized time and *member*, *insert*, and *delete* in $\Theta(\log(\min\{d, n-d\}))$ amortized time where d is the distance from the smallest element. In addition, concatenation is supported in $\Theta(\log(\min\{n_1, n_2\}))$, splitting in $\Theta(\log(\min\{d, n-d\}))$, and merge in $\Theta(n_s \log(n_\ell/n_s))$ amortized time where n_s is the size of the shorter and n_ℓ the size of the longer sequence. These bounds remain valid even if the data structure is used in a persistent setting.

'I wish life was not so short,' he thought. 'Languages take such a time, and so do all the things one wants to know about.'"
— J.R.R. Tolkien, *The Lost Road*

1 Introduction

Consider the following definition of 2-3 trees, taken from the introductory textbook 'Data Structures and Algorithms' by Aho, Hopcroft and Ullman (1983). "A 2-3 tree is a tree with the following two properties.

1. Each interior node has two or three children.
2. Each path from the root to a leaf has the same length."

When implementing 2-3 trees in our favourite programming language we are faced with the problem that we must enforce these properties using the linguistic concepts of the language. If the programming language offers sum types, the formulation of the first property is straightforward. Here is how a suitable definition might look like in the functional programming language Haskell (Peterson & Hammond, 1997).

```
data Tree23 a  =  Empty
               |  Node2 (Tree23 a) a (Tree23 a)
               |  Node3 (Tree23 a) a (Tree23 a) a (Tree23 a)
```

The second condition, however, appears to be problematic. Note that the constraint is not reflected in the definition above: *Node2 Empty* 1 (*Node2 Empty* 2 *Empty*) is an element of *Tree23 Int* but is not a legal 2-3 tree. It is probably unclear how to express this condition in most of today's programming languages. Usually, the programmer is responsible to check that the condition is preserved throughout the program. For that reason the second property is sometimes referred to as a *datatype invariant*. It goes without saying that it would be preferable if the programmer could delegate the proof obligation to the compiler — or rather, its type checker. Hence, the question naturally arises as to whether it is possible to devise a practical type system which allows to formulate structural constraints like the one above. Unsurprisingly, the answer is yes. But perhaps surprisingly, we do not need to be inventive since Haskell's type system is already sufficient for this purpose! In order to express structural constraints one has to combine two of its more advanced features, namely nested datatypes (Bird & Meertens, 1998) and higher-order polymorphism (Jones, 1995a).[1] Nested datatypes are also known as non-uniform datatypes (Kubiak *et al.*, 1992) or as non-regular datatypes (Paterson, 1994).

A nested datatype is a recursive, parameterised datatype whose definition involves a recursive call which does not have the same arguments as on the left-hand side. The simplest example of a nested datatype is the following.

```
data Perfect a  =  Zero a | Succ (Perfect (a, a))
```

Note that the recursive call on the right-hand side, *Perfect* (a, a), is a substitution instance of the left-hand side.

The history of nested datatypes is quite revealing. The Hindley-Milner type system (Milner, 1978) permitted nested datatypes from the very beginning. They have, however, been largely ignored in practice because the same type system rejects the definition of recursive functions on such types. The following definition illustrates the reason for this refusal.

```
height           ::  Perfect a → Int
height (Zero a)  =   0
height (Succ p)  =   1 + height p
```

[1] Strictly speaking, higher-order polymorphism is not necessary for expressing structural invariants. It simplifies, however, programming with nested datatypes.

The function *height* is unusual in that the recursive call has type *Perfect* $(a, a) \rightarrow$ *Int*, which is a substitution instance of the declared type. The Hindley-Milner system, however, requires both types to be equal. This restriction was already noticed by A. Mycroft who proposed a suitable extension of the type system (1984). Unfortunately, it was later shown that typability in this system is undecidable (Henglein, 1993). The designers of Haskell 1.4 (Peterson & Hammond, 1997) took a rather pragmatic approach to the problem of undecidability: *polymorphic recursion* is allowed if the programmer explicitly provides a type signature. Thus, the definition of *height* is perfectly acceptable to Haskell 1.4.

To see why nested datatypes are relevant to our goal of enforcing structural constraints consider the following sample elements of *Perfect Int*:

```
Zero 1
Succ (Zero (1,2))
Succ (Succ (Zero ((1,2),(3,4))))
Succ (Succ (Succ (Zero (((1,2),(3,4)),((5,6),(7,8)))))) .
```

If we interpret pairs as nodes, we see that elements of type *Perfect a* correspond to perfect binary leaf trees (Dielissen & Kaldewaij, 1995). Recall that in a perfect leaf tree each path from the root to a leaf has the same length. The definition of *height* furthermore shows that the prefix of *Succ* and *Zero* constructors encodes the height of the leaf tree.

Binary leaf trees belong to the class of *leaf-oriented* trees where elements are stored in the leaves. Now, 2-3 trees are typically *node-oriented*, that is elements are stored in the inner nodes. How do we adjust the definition of perfect leaf trees to node-oriented trees? A possible solution is to supply an additional type parameter for the type of search keys.

```
data Tree23 a k   =  Zero a | Succ (Tree23 (Node23 a k) k)
data Node23 a k   =  Node2 a k a | Node3 a k a k a
```

Now, *Tree23* () *k* could be used to represent 2-3 trees over the base type *k*. A potential disadvantage of this representation is that the type definitions do not explicate that the first parameter is a container type over the second parameter. The type parameters of *Node23*, for instance, are completely unrelated. An alternative representation, which models the situation more faithfully, is based on so-called *higher-order nested datatypes*. A nested datatype is termed higher-order if the type parameter which is instantiated in a recursive call ranges over type constructors rather than types. Here is the higher-order variant of the datatype *Perfect*.

```
data Perfect bush a  =  Zero (bush a) | Succ (Perfect (Fork bush) a)
data Fork bush a     =  Fork (bush a) (bush a)
```

The higher-order variant makes explicit that the argument of *Zero* is a container type over the base type *a*. One advantage of the higher-order approach is that the above definition is easily adapted to other types of trees. For 2-3 trees we simply replace *Fork* by *Node23*.

```
data Tree23 tree k  =  Zero (tree k) | Succ (Tree23 (Node23 tree) k)
data Node23 tree k  =  Node2 (tree k) k (tree k)
                    |  Node3 (tree k) k (tree k) k (tree k)
```

We will discuss the relationship between first-order and higher-order nests exhaustively in Sec. 5.3. For the moment we merely remark that higher-order nests are preferable for a particular technical reason.

The complete the definition of 2-3 trees we must define a polymorphic datatype which can be used as an initial parameter to *Tree23*.

```
data Empty k  =  Empty
```

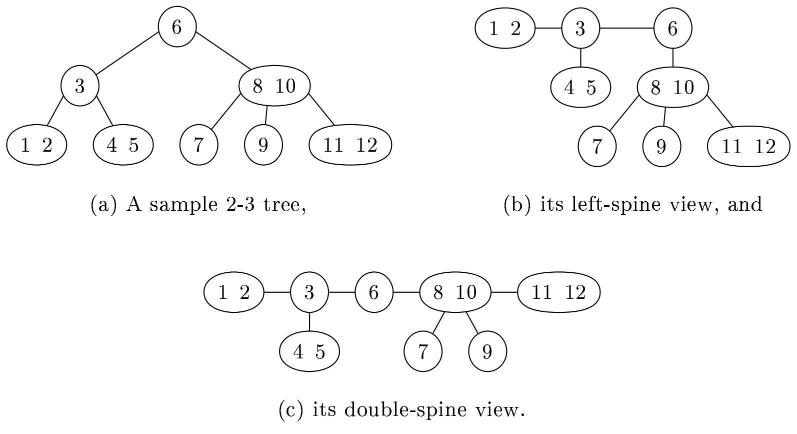
The type *Tree23 Empty k* is the desired representation of 2-3 trees over the base type *k*. Let us consider some examples. The empty 2-3 tree is denoted by *Zero Empty*; the term *Succ* (*Zero* (*Node2 Empty* 1 *Empty*)) denotes a 2-3 tree which consists of a single 2-node labelled with 1. The 2-3 tree depicted in Fig. 1(a) is represented by

```
Succ (Succ (Succ (Zero (Node2 (Node2 (Node3 Empty 1 Empty 2 Empty)
                                      3 (Node3 Empty 4 Empty 5 Empty))
                               6 (Node3 (Node2 Empty 7 Empty)
                                  8 (Node2 Empty 9 Empty)
                                  10 (Node3 Empty 11 Empty 12 Empty)))))) .
```

Note that *Tree23 Empty k* encodes all structural invariants of 2-3 trees. The unbalanced tree *Node2 Empty* 1 (*Node2 Empty* 2 *Empty*), for instance, cannot be made an element of *Tree23 Empty Int*. The type system enforces that the subtrees are of the same height.

The question remains how to adapt the usual functions for search, insertion and deletion to this representation of 2-3 trees. At first sight it seems that a complete redesign of the usual algorithmic solutions is necessary. As one of the main contributions of this paper we develop a method which renders programming with higher-order nested datatypes surprisingly simple. In most cases a straightforward and rather mechanical transformation suffices to adapt functions on regular datatypes to their non-regular colleagues.

We will use this framework to implement three different container types: binary random access lists, binomial queues, and 2-3 finger search trees. A common characteristic of these datatypes is that they are numerical representations in the sense of C. Okasaki (1998, p. 115). A container type qualifies as a numerical representation if it is modelled after some representation of the natural numbers. Haskell's predefined lists, for instance, can be seen as being modelled after the unary representation of the natural numbers. The data structures listed above are modelled after the binary number system. A binary random-access list, for example, is a sequence of perfect binary leaf trees increasing by height. The length of a random-access list, say, n dictates its structure: it contains a leaf tree of height i if the i-th bit in the binary representation of n is 1. The best-known example for a numerical representation are



(a) A sample 2-3 tree, (b) its left-spine view, and (c) its double-spine view.

Fig. 1. Different views of 2-3 trees.

probably binomial queues: a binomial queue is a sequence of heap-ordered binomial trees increasing by height. Since numerical representations typically contain very regular data structures as components, higher-order nested datatypes are ideally suited for their translation into a programming language.

To see how 2-3 trees fit into this setting consider the 2-3 tree shown in Fig. 1(a). The offsprings of the nodes in the left spine form a sequence of 2-3 trees of successive heights. Fig. 1(b) depicts the same tree under the so-called *left-spine view*. While both representations are isomorphic to each other, they exhibit different access characteristics. Accessing the leftmost, ie smallest element takes $\Theta(\log n)$ steps for 2-3 search trees and only constant time under the left-spine view. Insertion into a 2-3 tree requires $\Theta(\log n)$ time whereas insertion into a 2-3 tree under the left-spine view is independent of the tree size: it costs $\Theta(\log d)$ steps where d is the distance from the smallest element. If we symmetrize the design we obtain what we call the *double-spine view* of 2-3 trees, see Fig. 1(c). In essence we have a double-strand of 2-3 trees increasing by height which allows us to access the front and the rear end simultaneously. Consequently, insertion requires only $\Theta(\log(\min\{d, n-d\}))$ steps.

2-3 trees under the left-spine or the double-spine view are by no means a new data structure. They correspond to *finger search trees* (Guibas *et al.*, 1977) with a static finger at the front end and/or at the rear end of the sequence. A finger search tree is a data structure which represents an ordered sequence in such a way that searches are fast in the vicinity of a finger, where a finger points to an arbitrary position within the sequence. In an imperative setting finger search trees are usually implemented using level-linked trees. In a level-linked tree each node has links to the neighboring nodes on the same level. In addition there are upward links from children to parents. In a functional setting level-linked trees are prohibitive since an update necessarily takes $\Theta(n)$ steps. To see why this is the case note that a level-linked tree is strongly connected when interpreted as a graph. The absence of side-effects renders it necessary to copy the complete tree even if only a single node is changed. In other words, level-linked trees cannot be adapted to a persistent setting where an implementation must support updates and queries to any previous version of the data structure. It seems that in order to retain some of the good properties of finger search trees one must confine oneself to a static finger at the front and/or at the rear end of the sequence.

For functional programmers 2-3 finger search trees are of interest because they constitute the best known purely functional data structure for *ordered sequences*. Ordered sequences generalize priority queues by adding functions for searching and deleting elements and for concatenation and splitting. The implementation based on the double-spine view of 2-3 trees has the following *amortized* time bounds:[2]

member, *insert* and *delete*	take	$\Theta(\log(\min\{d, n-d\}))$ steps,
findMin and *findMax*	take	$\Theta(1)$ steps,
deleteMin and *deleteMax*	take	$\Theta(1)$ steps,
concatenation	takes	$\Theta(\log(\min\{n_1, n_2\}))$ steps,
partition	takes	$\Theta(\log(\min\{d, n-d\}))$ steps, and
merge	takes	$\Theta(n_s \log(n_\ell / n_s))$ steps,

where d is the distance from the smallest element, n_s is the size of the shorter and n_ℓ the size of the longer sequence. These bounds remain valid even if the data structure is used in a persistent setting.

To see the importance of this container abstraction note that both Dijkstra's algorithm for single-source shortest paths (Dijkstra, 1959) and Prim's algorithm for minimum-weight spanning trees (Prim, 1957) require a *decreaseKey* operation which can be easily defined in terms of *insert* and *delete*. Furthermore, note that purely functional implementations of priority queues (King, 1994; Paulson, 1996; Okasaki, 1996a; Brodal & Okasaki, 1996; Okasaki, 1998) do not support *decreaseKey* efficiently. In an imperative setting *delete* and *decreaseKey* require the maintenance of additional pointers into the data structure, a technique which cannot be reproduced in a functional language. The only exception to the above statement are priority queue implementations based on search tree schemes, for instance, splay heaps (Okasaki, 1998, Sec. 5.4). Search trees, however, do not reach the good time bounds of 2-3 fingers search trees.

The rest of this paper is organized as follows. Sec. 2 motivates the use of higher-order nested datatypes for expressing structural constraints. Sec. 3 reviews binary numerical representations and Sec. 4 reviews tree structures which are typically used in binary numerical representations. Sec. 5 develops the framework using binary random-access lists as a running example. Sec. 6 applies the framework to binomial heaps. 2-3 trees under the left-spine and under the double-spine view are discussed at length in Sec. 7 and 8. Finally, Sec. 9 reviews related work and Sec. 10 concludes.

[2] We assume, within the big-O notation, that $\log x = \log_2(\max\{2, x\})$.

2 Higher-order nested datatypes

The purpose of this section is to show that *higher-order* nested datatypes naturally arise if one tries to encode structural constraints into a datatype. Let us begin with a simple, but seemingly uninteresting example. Our first aim is to devise a list type which has the enigmatic property that the i-th element of a list of this type is equal to i. Of course, Haskell's predefined numeric types are unsuited for this purpose. Instead, we take the unary representation of the natural numbers, also known as Peano numerals, as a starting point.

$$\textbf{data}\ \mathit{Nat} \quad = \quad \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

The type *Nat* contains all natural numbers. To gain better control we decompose *Nat* into the base and the recursion case. It is tradition in Haskell to name types and values alike. We will stick to this tradition whenever possible.

$$\begin{aligned} \textbf{data}\ \mathit{Zero} \quad &= \quad \mathit{Zero} \\ \textbf{data}\ \mathit{Succ}\ \mathit{nat} \quad &= \quad \mathit{Succ}\ \mathit{nat} \end{aligned}$$

The type constructor *Succ* takes the 'former recursive component' as a type parameter. The type $\mathit{Succ}^n\ \mathit{Zero}$ contains only a single element, namely $\mathit{Succ}^n\ \mathit{Zero}$, the representation of n. Using these prerequisites the definition of the envisaged list type is straightforward.

$$\textbf{data}\ \mathit{Nats}\ \mathit{nat} \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ \mathit{nat}\ (\mathit{Nats}\ (\mathit{Succ}\ \mathit{nat}))$$

Since the recursive call $\mathit{Nats}\ (\mathit{Succ}\ \mathit{nat})$ is a substitution instance of the left-hand side, *Nats* is an example of a nested datatype. A list of type $\mathit{Nats}\ \mathit{nat}$ contains a sequence of elements of type nat, $\mathit{Succ}\ \mathit{nat}$, $\mathit{Succ}^2\ \mathit{nat}$, and so forth. The type definition is reminiscent of the function *nats* given by

$$\mathit{nats}\ n \quad = \quad n : \mathit{nats}\ (n+1),$$

which generates the infinite list of successive integers beginning with n. This correspondence shows that we are in a sense computing with types when defining a nested datatype.

The relevance of the introductory example becomes apparent if we switch from numbers to container types. Let us design a list type which constrains the i-th element to be a stack containing exactly i elements. First note that stacks correspond closely to Peano numerals, see (Okasaki, 1998, p. 115).

$$\begin{aligned} \textbf{data}\ \mathit{Nat} \quad &= \quad \mathit{Zero} & \qquad \textbf{data}\ \mathit{Stack}\ a \quad &= \quad \mathit{Empty} \\ &\mid \quad \mathit{Succ}\ \mathit{Nat} & &\mid \quad \mathit{Push}\ a\ (\mathit{Stack}\ a) \end{aligned}$$

The two type definitions have essentially the same structure. The only difference is that a stack contains elements while a natural number does not. In view of this correspondence the idea suggests itself to follow the road which led to the definition of the datatype *Nats*. The decomposition of *Stack* yields the following type definitions.

$$\begin{aligned}\mathbf{data}\ \mathit{Empty}\ a &= \mathit{Empty}\\ \mathbf{data}\ \mathit{Push}\ \mathit{stack}\ a &= \mathit{Push}\ a\ (\mathit{stack}\ a)\end{aligned}$$

Since *Stack* is a type constructor rather than a type, the type parameter *stack* consequently ranges over type constructors. Formally, we have that *Succ* has kind $* \to *$ while *Push* has kind $(* \to *) \to (* \to *)$. Recall that the kind system of Haskell specifies the 'type' of a type constructor, see (Jones, 1995b). The '$*$' kind represents nullary constructors like *Bool*, *Int*, or *Nat*. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The kinds of the decomposed types are easy to determine. Assume that the original type has kind κ. The decomposed types are of kind $\kappa \to \kappa$ if the corresponding constructor has a recursive component and of kind κ otherwise.

Turning back to our example we have that $\mathit{Push}^n\ \mathit{Empty}$ is the type of polymorphic stacks of size n. The type *Nats* is easy to adapt.

$$\mathbf{data}\ \mathit{Stacks}\ \mathit{stack}\ a = \mathit{Nil} \mid \mathit{Cons}\ (\mathit{stack}\ a)\ (\mathit{Stacks}\ (\mathit{Push}\ \mathit{stack})\ a)$$

The type parameter which is instantiated in the recursive call now ranges over type constructors. Hence, *Stacks* is an example of a *higher-order nested datatype*. A list of type *Stacks stack a* contains elements of types *stack a*, *Push stack a*, $\mathit{Push}^2\ \mathit{stack}\ a$, and so forth.

3 Binary numerical representations

This section briefly reviews the basics of numerical representations. For a more in-depth treatment of the topic we refer the interested reader to the recent textbook by C. Okasaki (1998).

Number systems serve admirably as templates for container types: a container object of size n is modelled after the representation of the number n and operations on container objects are modelled after their number-theoretic counterparts:

inserting an element	corresponds to	incrementing a number,
deleting an element	corresponds to	decrementing a number,
merging two container objects	corresponds to	adding two numbers, and
halving a container object	corresponds to	dividing a number by two.

The data structures presented in this paper are without exception based on the binary number system. Recall that the value of the binary number $(b_0 \ldots b_{n-1})_2$ is $\sum_{i=0}^{n-1} b_i 2^i$. Note that we write the least significant digit first. A binary representation is essentially a sequence of 'digits' such that the i-th 'digit' contains b_i data structures of size 2^i. The digits b_i are usually drawn from the set $\{0, 1\}$. This is, however, by no means compelling. Alternative choices include $\{1, 2\}$, and $\{1, 2, 3\}$. If the digit zero is not admissible, we have a so-called *zeroless* representation. If there is more than one way to represent some numbers, we have a so-called *redundant* number system. Thus, the 1-2 system is zeroless and the 1-2-3 system is zeroless and redundant.

The choice of an appropriate number system is, of course, dictated by the set of operations a particular container abstraction must support. Let us therefore examine the properties of the different systems more closely. In the 0-1 system binary increment and decrement are defined as follows (ϵ denotes the empty sequence).

$$\begin{array}{rcl@{\qquad}rcl} \mathit{inc}\,\epsilon & = & 1 & \mathit{dec}\,1 & = & \epsilon \\ \mathit{inc}\,(0a) & = & 1a & \mathit{dec}\,(0a) & = & 1(\mathit{dec}\,a) \\ \mathit{inc}\,(1a) & = & 0(\mathit{inc}\,a) & \mathit{dec}\,(1a) & = & 0a \end{array}$$

Both operations take $\Theta(k)$ steps in the worst case where k is the number of digits. It is well-known that averaging over a sequence of increment operations yields an amortized time bound of $\Theta(1)$ (Cormen *et al.*, 1991, Ch. 18). The same amortized bound holds for the binary decrement. It is interesting to contrast the 'standard' binary system with the 1-2 system.

$$\begin{array}{rcl@{\qquad}rcl} \mathit{inc}\,\epsilon & = & 1 & \mathit{dec}\,1 & = & \epsilon \\ \mathit{inc}\,(1a) & = & 2a & \mathit{dec}\,(1a) & = & 2(\mathit{dec}\,a) \\ \mathit{inc}\,(2a) & = & 1(\mathit{inc}\,a) & \mathit{dec}\,(2a) & = & 1a \end{array}$$

The digit 1 takes over the rôle of 0 and 2 takes the rôle of 1. The 1-2 system may be preferable for two reasons. Since the digit 0 is abandoned in favour of the digit 2, each natural number has a unique representation. In particular there is no need to disallow leading zeros. Furthermore, the representation is dense which improves the running time of access functions like *head*.

A potential disadvantage of both systems and of non-redundant number systems in general is that they do not support both increment *and* decrement efficiently. If alternating increment and decrement operations oscillate around $(11\ldots1)_2$, the worst case arises at every operation. A redundant number system avoids this situation as follows. Assume that we use the digits 1, 2, and 3. The worst-case for an increment is a sequence of threes while the worst-case for a decrement is a sequence of ones: $\mathit{inc}\,(333.333.3) = 222.222.21$ and $\mathit{dec}\,(111.111.1) = 222.222$. Now, a subsequent decrement or increment operation only takes constant time: $\mathit{dec}\,(222.222.21) = 122.222.21$ and $\mathit{inc}\,(222.222) = 322.222$. To see how this works consider the definitions of increment and decrement.

$$\begin{array}{rcl@{\qquad}rcl} \mathit{inc}\,\epsilon & = & 1 & \mathit{dec}\,1 & = & \epsilon \\ \mathit{inc}\,(1a) & = & 2a & \mathit{dec}\,(1a) & = & 2(\mathit{dec}\,a) \\ \mathit{inc}\,(2a) & = & 3a & \mathit{dec}\,(2a) & = & 1a \\ \mathit{inc}\,(3a) & = & 2(\mathit{inc}\,a) & \mathit{dec}\,(3a) & = & 2a \end{array}$$

For each operation we can identify expensive and cheap digits. The digit 3 is expensive for the increment while 1 is expensive for the decrement. The trick is that whenever an operation processes an expensive digit it leaves a cheap digit behind. Thus, both increment and decrement run in $\Theta(1)$ amortized time.

Unfortunately, the amortized time bounds break down in a *persistent* setting. To illustrate, assume that $a = 3 \cdot (2^n - 1)$ and consider 2^n independent threads which all increment a. The worst-case arises for every increment and these costs cannot be counterbalanced by preceding cheap operations. C. Okasaki has shown in a series of papers (1995a; 1995c; 1996b) that *lazy evaluation* can mediate the conflict between amortization and persistence. Under the regime of lazy evaluation *inc* and *dec* become constant time operations since the calls are essentially delayed. An operation which inspects a number is now potentially costly since it must evaluate the delayed calls to *inc* and *dec* beforehand. However, once a call is evaluated its value is memoized so that subsequent operations need not redo the work. Reconsider the example above. Incrementing is now a constant time operation. Furthermore, lazy evaluation guarantees that a is evaluated at most once. So if *inc* a is evaluated in one of the threads, the work done for a is shared by all threads.

Using a so-called *debit argument* one can prove that *inc*, *dec*, and operations which only inspect the least significant digit run in $\Theta(1)$ amortized time. The principle idea is that each delayed call is assigned a number of debits which must be paid off before its value can be used. Each debit represents a constant amount of delayed work. For the *lazy 1-2-3 number system* we show that the operations preserve the following debit invariant: expensive digits (ie 1 and 3) are allowed zero debits and cheap digits (ie 2) are allowed one debit. Now, assume that the number a begins with a sequence of 3s followed by a 1 or a 2. Incrementing a changes the 3s to 2s. For each step the newly created digit is assigned a debit of one. In the last step the digit 1 or 2 is changed to 2 or 3. At most two debits must now be discharged to restore the invariant. Since the unshared cost of *inc* is $\Theta(1)$, the operation takes $\Theta(1)$ amortized time. The proof for *dec* is completely analogous. Finally, an operation which inspects only the least significant digit runs in $\Theta(1)$ amortized time since at most one debit must be discharged.

The data structures introduced in the subsequent sections are all based on variations of the *lazy* binary number system. Since Haskell is a non-strict language, the implementation of lazy number systems comes actually for free.

4 Regular tree structures

There are essentially three types of trees that are used in binary numerical representations: perfect binary leaf trees, pennants, and binomial trees (Okasaki, 1998). We will consider each in turn. A leaf tree is a full binary tree with elements stored in the leaves. Formally, it is an element of the following datatype.

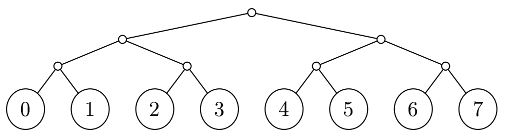
$$\textbf{data}\ \mathit{Bush}\ a \quad = \quad \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Bush}\ a)\ (\mathit{Bush}\ a)$$

Since we are only interested in perfect leaf trees we decompose *Bush* into the base and the recursion case.

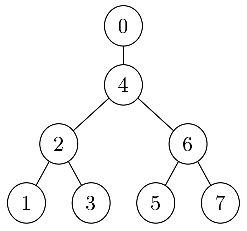
$$\begin{aligned}\textbf{data}\ \mathit{Leaf}\ a \quad &= \quad \mathit{Leaf}\ a\\ \textbf{data}\ \mathit{Fork}\ \mathit{bush}\ a \quad &= \quad \mathit{Fork}\ (\mathit{bush}\ a)\ (\mathit{bush}\ a)\end{aligned}$$

In the remainder of this paper only the decomposed types will be given. The original type definitions can be easily reconstructed in each case. A perfect binary leaf tree of rank n is an element of $\mathit{Fork}^n\ \mathit{Leaf}\ a$.

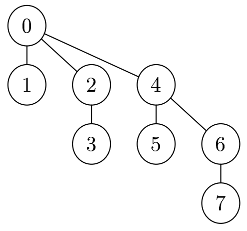
A pennant is a perfect binary search tree with an additional node on top. Here



(a) A leaf tree,



(b) a pennant,



(c) and a binomial tree.

Fig. 2. Regular tree structures of rank 3.

are the necessary type definitions.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Empty}\ a & = & \mathit{Empty} \\
\textbf{data}\ \mathit{Bin}\ \mathit{tree}\ a & = & \mathit{Bin}\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a) \\
\textbf{data}\ \mathit{Pennant}\ \mathit{tree}\ a & = & \mathit{Pennant}\ a\ (\mathit{tree}\ a)
\end{array}
$$

A pennant of rank n is an element of type $\mathit{Pennant}\ (\mathit{Bin}^n\ \mathit{Empty})\ a$.

Finally, a binomial tree is a multiway branching tree which is defined as follows: a binomial tree of rank 0 consists of a single node, a tree of rank n has $n-1$ descendants which are binomial trees of ranks 0 upto $n-1$. This structure is captured by the following definitions.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Lin}\ a & = & \mathit{Lin} \\
\textbf{data}\ \mathit{Snoc}\ \mathit{subtrees}\ a & = & \mathit{Snoc}\ (\mathit{subtrees}\ a)\ (\mathit{Node}\ \mathit{subtrees}\ a) \\
\textbf{data}\ \mathit{Node}\ \mathit{subtrees}\ a & = & \mathit{Node}\ a\ (\mathit{subtrees}\ a)
\end{array}
$$

The subtrees are organized using so-called snoc lists[3] which are build from left to right. A binomial tree of rank n is an element of $\mathit{Node}\ (\mathit{Snoc}^n\ \mathit{Lin})\ a$. Fig. 2 depicts the three types of trees.

Since leaf trees, pennants, and binomial trees of rank n each contain 2^n elements, it probably comes as no surprise that the three structures are isomorphic to each other. The correspondence between leaf trees and pennants is best explained by viewing a leaf tree as the initial configuration of a tournament and the corresponding pennant as the corresponding final configuration. An inner node of a leaf tree

[3] Note that *Lin* and *Snoc* are *Nil* and *Cons* spelled backward.

represents a match; a leaf stands for a participant. The top node of the corresponding pennant contains the winner of the tournament; the inner nodes are decorated with the loser of the matches. The relation between pennants and binomial trees is even more direct. First note that the decomposed types are essentially tuple types: *Empty* and *Lin* are nullary tuples, *Bin* is a triple, and *Pennant*, *Snoc*, and *Node* are pairs. Looking more carefully, we see that *Snoc* is a nested pair of the form $(a, (b, c))$ which is trivially isomorphic to the triple (a, b, c) alias *Bin a b c*. Thus *Snoc*, the type of binomial forests, is really a 'binarized' variant of *Bin*, the type of perfect binary search trees. The types *Empty* and *Lin* and the types *Pennant* and *Node* (and *Push* of Sec. 2) are even identical! To eliminate this unnecessary diversity we will henceforth use *Empty* for nullary nodes and use the latter two types as synonyms for the following datatype.

$$\textbf{data}\ \mathit{Front}\ \mathit{tree}\ a \quad = \quad a \triangleleft \mathit{tree}\ a$$

Now, using a straightforward induction on the rank one can show that $\mathit{Fork}^k\ \mathit{Leaf} \cong \mathit{Pennant}\ (\mathit{Bin}^k\ \mathit{Empty}) \cong \mathit{Node}\ (\mathit{Snoc}^k\ \mathit{Empty})$. The isomorphisms remain valid if we drop the size constraints. The second isomorphism, for instance, is called the *natural correspondence* between binary trees and forests in (Knuth, 1997).

While leaf trees and pennants are isomorphic to each other, it is important to note that they exhibit different access characteristics. Accessing the leftmost element in a leaf tree of rank k requires k steps. In a pennant this can be done in constant time since the leftmost element of a leaf tree appears as the top element of the corresponding pennant. The catenable sequences of C. Okasaki (1995a), for instance, make use of this fact. The differences between pennants and binomial trees are really minor: pennants are more space economical while linking and unlinking is slightly faster with binomial trees (Hinze, 1998a).

5 Random-access lists

By now we have all the necessary prerequisites at hand to tackle the implementation of binary numerical representations on the basis of higher-order nested datatypes. As the first example we will translate binary random-access lists into this framework. Random-access lists implement one-sided *indexable sequences*. This container abstraction combines typical list operations with operations for accessing and updating list elements by position. Random-access lists are one-sided in that they only support operations at the front end efficiently.

Fig. 3 displays a suitable signature for indexable sequences. Note that the usual list operations *null*, *head*, and *tail* have been combined into a single function, called *front*. The type $\langle a, b \rangle$ which occurs in its type signature is termed an *optional pair*: an element of that type is either *Null* or a pair of the form $\langle a, b \rangle$. The call *front s* has consequently two possible outcomes: if the sequence s is empty, *front* returns *Null*, otherwise it yields $\langle a, s' \rangle$ where a is the head of s and s' its tail.

Random-access lists were originally presented by V.J. Dielissen and A. Kaldewaij (1995) as leftist left-complete leaf trees. The observation that this data structure can be recast as a numerical representation is due to C. Okasaki (1998). A binary

```
data ⟨a, b⟩   =   Null | ⟨a, b⟩
empty         ::  IxSequence a
cons          ::  a → IxSequence a → IxSequence a
front         ::  IxSequence a → ⟨a, IxSequence a⟩
snoc          ::  IxSequence a → a → IxSequence a
rear          ::  IxSequence a → ⟨IxSequence a, a⟩
access        ::  IxSequence a → Int → a
update        ::  IxSequence a → Int → a → IxSequence a
fromList      ::  [a] → IxSequence a
toList        ::  IxSequence a → [a]
```

Fig. 3. A signature for indexable sequences.

random-access list is a sequence of perfect binary leaf trees increasing by rank. This structure is captured by the following data type definition.

```
data RandomAccessList bush a
  =  Nil
  |  Zero (RandomAccessList (Fork bush) a)
  |  One (bush a) (RandomAccessList (Fork bush) a)
type IxSequence  =  RandomAccessList Leaf
```

The representation of a sequence is uniquely determined by the binary decomposition of its size. A sequence of size $(1011)_2$, for instance, is represented by the term *One* b_0 (*Zero* (*One* b_2 (*One* b_3 *Nil*))) where b_i is of type Fork^i *Leaf* a.

Now, since random-access lists based on the 0-1 system are discussed at length in (Okasaki, 1998, Sec. 9.2.1 and 10.1.2) we will implement a zeroless representation based on the 1-2 system instead. The lessons we are going to learn will prove useful when we tackle the implementation of 2-3 finger search trees in Sec. 7 and 8. Recall that binary numeric representations are based on the idea that the digit d corresponds to a container object comprising d trees of equal rank. The 1-2 system offers an interesting alternative: a one of weight n can be represented by a tree of rank n and a two by a *single* tree of rank $n + 1$. This representation is superior in practice since it eliminates some of the redundancy which is present in the standard design. The type of random-access lists consequently takes the following form.

```
data RandomAccessList bush a
  =  Nil
  |  One (bush a) (RandomAccessList (Fork bush) a)
  |  Two (Fork bush a) (RandomAccessList (Fork bush) a)
```

In what follows we will often abbreviate type and constructor names by their first letter. The two-level type structure of *IS* and *RAL* — *IS* calls the recursively defined *RAL* with b set to L — is characteristic for higher-order nested datatypes. Functions on higher-order nests typically follow this type structure. The function *cons* which extends a sequence at the front end may serve as an example. The element is first converted into a singleton tree.

$$\begin{array}{lcl} \mathit{cons} & :: & a \to \mathit{IxSequence}\ a \to \mathit{IxSequence}\ a \\ \mathit{cons}\ a\ s & = & \mathit{incr}\ (\mathit{Leaf}\ a)\ s \end{array}$$

The helper function *incr* then adds a bush to the list of bushes following the recursion pattern of the binary increment (see Sec. 3).

$$\begin{array}{lcl} \mathit{incr} & :: & \mathit{bush}\ a \to \mathit{RAL}\ \mathit{bush}\ a \to \mathit{RAL}\ \mathit{bush}\ a \\ \mathit{incr}\ b\ \mathit{Nil} & = & \mathit{One}\ b\ \mathit{Nil} \\ \mathit{incr}\ b_1\ (\mathit{One}\ b_2\ \mathit{ds}) & = & \mathit{Two}\ (\mathit{Fork}\ b_1\ b_2)\ \mathit{ds} \\ \mathit{incr}\ b_1\ (\mathit{Two}\ b_2\ \mathit{ds}) & = & \mathit{One}\ b_1\ (\mathit{incr}\ b_2\ \mathit{ds}) \end{array}$$

Note that *incr* is doubly polymorphic: it works on arbitrary element types and on arbitrary 'base type' constructors. Furthermore note that *incr* uses *higher-order polymorphic recursion*. The recursive call in the third equation has type $F\ b\ a \to \mathit{RAL}\ (F\ b)\ a \to \mathit{RAL}\ (F\ b)\ a$ which is a substitution instance of the declared type.

To transform an ordinary list into a random-access list we simple iterate *cons* starting with the empty sequence *Nil*.

$$\begin{array}{lcl} \mathit{fromList} & :: & [a] \to \mathit{IxSequence}\ a \\ \mathit{fromList} & = & \mathit{foldr}\ \mathit{cons}\ \mathit{Nil} \end{array}$$

The expression *fromList* $[1\,..\,5]$, for example, evaluates to

$$\mathit{One}\ (L\ 1)\ (\mathit{Two}\ (F\ (F\ (L\ 2)\ (L\ 3))\ (F\ (L\ 4)\ (L\ 5)))\ \mathit{Nil})\ .$$

The 'classical' algorithm for shrinking the sequence at the front is modelled after the binary decrement: $\mathit{dec}\,1 = \epsilon$, $\mathit{dec}\,(1a) = 2(\mathit{dec}\,a)$, and $\mathit{dec}\,(2a) = 1a$. Striving for elegance we use a slightly different approach. First note that the first two cases can be simplified if we temporarily allow a leading zero: $\mathit{dec}\,(1a) = 0a$. We then eliminate the zero by repeatedly applying the following rules: $0\epsilon = \epsilon$, $01a = 20a$, and $02a = 21a$.

$$\begin{array}{lcl} \mathit{zero} & :: & \mathit{RAL}\ (\mathit{Fork}\ \mathit{bush})\ a \to \mathit{RAL}\ \mathit{bush}\ a \\ \mathit{zero}\ \mathit{Nil} & = & \mathit{Nil} \\ \mathit{zero}\ (\mathit{One}\ b\ \mathit{ds}) & = & \mathit{Two}\ b\ (\mathit{zero}\ \mathit{ds}) \\ \mathit{zero}\ (\mathit{Two}\ (\mathit{Fork}\ b_1\ b_2)\ \mathit{ds}) & = & \mathit{Two}\ b_1\ (\mathit{One}\ b_2\ \mathit{ds}) \end{array}$$

Building upon *zero* the implementation of *front* is straightforward.

$$\begin{array}{lcl} \mathit{front} & :: & \mathit{IxSequence}\ a \to \langle a, \mathit{IxSequence}\ a\rangle \\ \mathit{front}\ \mathit{Nil} & = & \mathit{Null} \\ \mathit{front}\ (\mathit{One}\ (\mathit{Leaf}\ a)\ \mathit{ds}) & = & \langle a, \mathit{zero}\ \mathit{ds}\rangle \\ \mathit{front}\ (\mathit{Two}\ (\mathit{Fork}\ (\mathit{Leaf}\ a)\ b)\ \mathit{ts}) & = & \langle a, \mathit{One}\ b\ \mathit{ts}\rangle \end{array}$$

5.1 Traversing a random-access list

Things become interesting when we write functions that need to inspect the leaf trees of a random-access list. The function *toList*, which transforms a random-access list into an ordinary list, serves as a motivating example.

To begin with here is the function *flatten* which listifies an ordinary leaf tree of type *Bush a*.

```
flatten               ::  Bush a → [a]
flatten (Leaf a)      =   [a]
flatten (Fork ℓ r)    =   flatten ℓ ++ flatten r
```

Note that this implementation is not the most efficient one: it takes quadratic instead of linear time in the worst case. This is, however, easily corrected using standard techniques (Hughes, 1986).

Now, we let 'function follow type'. To obtain 'flatteners' for perfect leaf trees we simply decompose *flatten* into the base and the recursion case.

```
unleaf                       ::  Leaf a → [a]
unleaf (Leaf a)              =   [a]
unfork                       ::  (bush a → [a]) → (Fork bush a → [a])
unfork flatten (Fork ℓ r)    =   flatten ℓ ++ flatten r
```

The function *unfork* takes the 'former recursive call' as a parameter imitating the transformations at the type level. The polymorphic function $unfork^n$ *unleaf* consequently flattens a perfect leaf tree of type $Fork^n$ *Leaf a*. The function *listify* which flattens a random-access list naturally follows the recursion scheme of *RAL*.

```
listify                       ::  (bush a → [a]) → (RAL bush a → [a])
listify flatten Nil           =   []
listify flatten (One b ds)    =   flatten b ++ listify (unfork flatten) ds
listify flatten (Two b ds)    =   unfork flatten b ++ listify (unfork flatten) ds
```

While recursing down the list *listify* builds the appropriate flatteners $unfork^n$ *flatten* of type $Fork^n$ *bush a* → [*a*]. The recursion starts with *flatten* set to *unleaf*.

```
toList      ::  IxSequence a → [a]
toList s    =   listify unleaf s
```

Note that *toList* takes only $\Theta(n \log n)$ time in the worst case.

While the construction of *toList* is entirely straightforward, the higher-order approach becomes unwieldy if more than a single function is involved, say, a group of mutually recursive functions. Fortunately, there is a simple alternative at hand. Instead of passing the flatteners explicitly we employ Haskell's type classes to pass them implicitly. Here are the necessary definitions.

```
class Flatten bush where
    flatten               ::  bush a → [a]
instance Flatten Leaf where
    flatten (Leaf a)      =   [a]
instance (Flatten bush) ⇒ Flatten (Fork bush) where
    flatten (Fork ℓ r)    =   flatten ℓ ++ flatten r
```

We can actually re-use the original definition of *flatten* without any change. All we have to do is to supply the necessary class and instance definitions. It is worth noting that the switch from higher-order functions to type classes does not incur a run-time penalty. A good compiler should essentially generate the same code in both cases.

The use of type classes has the further advantage that the flatteners are constructed automatically. Consider the revised definition of *listify*.

```
listify              ::  (Flatten bush) ⇒ RAL bush a → [a]
listify Nil          =   []
listify (One b ds)   =   flatten b ++ listify ds
listify (Two b ds)   =   flatten b ++ listify ds
```

Note that the right-hand sides of the last two equations are identical! This is, of course, due to the fact that *flatten* is overloaded. In the second equation *flatten* is of type *bush a* → [*a*] while the occurrence in the third equation has type *Fork bush a* → [*a*]. However, the uniform treatment of both cases is one of the reasons for using the 1-2 system and for representing the digit two by a single tree. The function *toList* boils down to a type specialized variant of *listify*.

```
toList     ::  IxSequence a → [a]
toList s   =   listify s
```

Remark. The beautiful thing about nested datatypes is that they sometimes open new perspectives of formulating algorithms. It is well-known that there are basically two methods of building a tree (Bird, 1997): recursive, or top down, and iterative, or bottom up. For both methods there is a corresponding inverse method which takes a tree to a list. The recursive method implemented by *listify* works by flattening the subtrees and appending the results. The iterative method repeatedly splits the trees into subtrees until only singleton trees remain which are then converted into a list of elements. The second method is equally straightforward to implement due to the rigid structure of the datatype *RAL*. The function *listify* now returns a list of trees.

```
listify              ::  RAL bush a → [bush a]
listify Nil          =   []
listify (One b ds)   =   b : unforks (listify ds)
listify (Two b ds)   =   unforks (b : listify ds)
```

Note that each of the recursive calls yields a list of trees of type *F b a* which *unforks* splits into their components.

```
unforks                     ::  [Fork bush a] → [bush a]
unforks []                  =   []
unforks (Fork b1 b2 : ts)   =   b1 : b2 : unforks ts
```

The rest is easy.

```
toList     ::  IxSequence a → [a]
toList s   =   [a | Leaf a ← listify s]
```

It should be noted that this variant of *toList* takes only linear time since each node is processed in a constant number of steps. □

The implementation of *access*, which accesses the i-th element of a sequence, and of *update*, which updates the i-th element of a sequence, is left as an exercise to the reader. As before there are basically two approaches. The first works by searching the correct tree, and then searching the correct element (Okasaki, 1998, p.122). The second, which is more elegant, employs the structure of random-access lists (Okasaki, 1998, p.145).

5.2 Variations

Many different variations of random-access lists are conceivable: one could use a different number system, one could employ a different tree type, or one could symmetrize the design to support two-sided indexable sequences. Let us consider each dimension of the design space in turn.

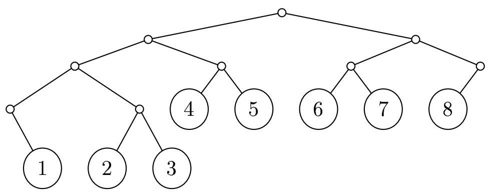
The pros and cons of different number systems and the choice of the right system are explained in great detail in (Okasaki, 1998, Ch. 9 & Sec. 11.1). So we content ourselves with a few cursory remarks. First of all, note that the 1-2 number system is preferable to the 0-1 system because it supports *head* in $\Theta(1)$ and *access s i* and *update s i a* in $\Theta(\log i)$ as opposed to $\Theta(\log n)$ in the 0-1 system. The number system of choice, however, is the 1-2-3 system since it supports both increment and decrement to run in $\Theta(1)$ amortized time. The type *RandomAccessList* is easily adapted by adding a fourth data constructor for the digit 3.

We have seen in Sec. 4 that perfect binary leaf trees, pennants, and binomial trees are isomorphic to each other. The question naturally arises as to whether there is a gain in using, say, pennants instead of perfect binary leaf trees? It turns out that pennants are preferable if one wants to implement *ordered sequences* which support an efficient membership test. The search for an element would then take $\Theta(\log d)$ time where d is the distance of the query element to the front of the ordered sequence. The type of random-access lists is easily adopted to the new situation.

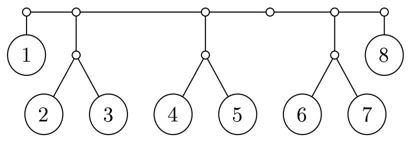
```
data SearchableList tree a
  =  Nil
  |  One (Pennant tree a) (SearchableList (Bin tree) a)
  |  Two (Pennant (Bin tree) a) (SearchableList (Bin tree) a)
```

Unfortunately, insertion and deletion are expensive since the structure of pennants is too rigid. One can prove a lower bound of $\Omega(\sqrt{n})$ for insertion and deletion if the data structure is uniquely determined by the number of elements it contains (Snyder, 1977). Sec. 7 explains how to solve this problem by replacing perfect binary search trees with 2-3 trees, ie by using a redundant representation.

The original article by V.J. Dielissen and A. Kaldewaij (1995) also contains a description of random-access deques which can grow and shrink at both ends. To support a symmetric set of operations we must essentially symmetrize the design of random-access lists. Speaking in terms of tree transformations we must switch from the left-spine view to the *double-spine view*. Fig. 4 displays a binary leaf tree



(a) A sample leaf tree and



(b) its double-spine view.

Fig. 4. Different views of leaf trees.

and the same tree under the double-spine view. The succession of trees in Fig. 4(b) corresponds to the number $12\epsilon 11$ where ϵ marks the root of the tree. It should be clear that the representation is no longer unique even if we confine ourselves to the digits 1 and 2. This redundancy is intended because the deque operations should operate largely independent on both ends. However, the front and the rear part should not be totally independent in order to avoid unbalanced situations like, for instance, $\epsilon 11 \ldots 11$. In this case the first element of the queue is the last element of the right spine which takes $\Theta(\log n)$ time to access. For this reason we require that both parts are of equal length. This allows to represent $1(2\epsilon 1)1$ using a ternary and a nullary constructor.

```
data RandomAccessDeque bush a
  = Nil
  | Simple (bush a)
  | Composite (Digit bush a) (RandomAccessDeque (Fork bush) a)
              (Digit bush a)
data Digit bush a  =  One (bush a) | Two (Fork bush a)
type IxSequence    =  RandomAccessDeque Leaf
```

Singleton deques require special treatment since the smallest digit is 1. This explains why the datatype *RAD* comprises a third constructor which accommodates a single tree. It is important to note that *RAD* is a bottom-up representation of a leaf tree: in $C\ d\ x\ d'$ the digit d contains the leftmost tree on the left spine while d' contains the rightmost tree on the right spine. The leaf tree depicted in Fig. 4(b), for instance, is represented by

```
C (One (L 1))
  (C (Two (F (F (L 2) (L 3)) (F (L 4) (L 5))))
     Nil
     (One (F (L 6) (L 7))))
  (One (L 8)) .
```

The code for extending a sequence at the front end is very similar to the one for random-access lists and is left as an exercise to the reader. Instead we implement the function for extending a sequence at the rear end which is defined symmetrically.

```
snoc                              :: IxSequence a → a → IxSequence a
snoc s a                          =  rcni s (Leaf a)
rcni                              :: RAD bush a → bush a → RAD bush a
rcni Nil b                        =  Simple b
rcni (Simple b₁) b₂               =  Composite (One b₁) Nil (One b₂)
rcni (Composite d x (One b₁)) b₂  =  Composite d x (Two (Fork b₁ b₂))
rcni (Composite d x (Two b₁)) b₂  =  Composite d (rcni x b₁) (One b₂)
```

5.3 First-order versus higher-order nested datatypes

Random-access lists as well as other sequence types are usually implemented using first-order nested datatypes (Okasaki, 1997; Okasaki, 1998). In this section we shed some light on the relationship between first-order and higher-order nests and discuss advantages and disadvantages of both representations. For ease of reference here is the first-order variant of *RAL* (Okasaki, 1998, p. 119) adapted to the naming conventions of the previous sections.

```
data Fork bush  =  Fork bush bush
data RandomAccessList bush
  =  Nil
  |  One bush (RandomAccessList (Fork bush))
  |  Two (Fork bush) (RandomAccessList (Fork bush))
```

From a theoretical point of view both approaches are, in fact, equivalent. The first-order variant can be easily defined in terms of its higher-order cousin, we have $RAL^{fo}\ a \cong RAL^{ho}\ Leaf\ a$. The other way round is equally easy, we have $RAL^{ho}\ bush\ a \cong RAL^{fo}\ (bush\ a)$. A similar relation holds between the first-order and the higher-order definition of 2-3 trees introduced in Sec. 1. Let *Const a* be the constant functor given by **data** *Const a b* = *Const a*. We have $Tree23^{fo}\ a\ k = Tree23^{ho}\ (Const\ a)\ k$ and $Tree23^{ho}\ tree\ k = Tree23^{fo}\ (tree\ k)\ k$. Which approach is superior is consequently more a question of practical matters.

One advantage of the first-order variant is obvious: there is only one type instead of two. This in turn simplifies the definition of operations on nests. To exemplify, consider the first-order variant of *cons*.

$$\begin{array}{lcl}
\mathit{cons} & :: & \mathit{bush} \to \mathit{RAL}\ \mathit{bush} \to \mathit{RAL}\ \mathit{bush} \\
\mathit{cons}\ b\ \mathit{Nil} & = & \mathit{One}\ b\ \mathit{Nil} \\
\mathit{cons}\ b_1\ (\mathit{One}\ b_2\ \mathit{ds}) & = & \mathit{Two}\ (\mathit{Fork}\ b_1\ b_2)\ \mathit{ds} \\
\mathit{cons}\ b_1\ (\mathit{Two}\ b_2\ \mathit{ds}) & = & \mathit{One}\ b_1\ (\mathit{cons}\ b_2\ \mathit{ds})
\end{array}$$

The time and space behaviour, however, is not affected — provided we define the types *Leaf* and *Const* via **newtype** declarations instead of **data** declarations.

On the negative side, the class-based programming technique introduced in Sec. 5.1 cannot be adapted to first-order nests. Consider the definition of *RAL* above and note that the type parameter *bush* ranges over tree types: *bush* equals $\mathit{Fork}^i\ a$ in the i-th level of recursion when the initial call was *RAL a*. That said it is actually surprising that *RAL a* represents a viable sequence type at all. The definition only works because we can identify a singleton leaf tree with its only label. Now, how can we adapt the class-based programming technique? Consider the second definition of *toList* which employs the type class *Flatten*. To begin with we require *multi-parameter type classes* (Peyton Jones *et al.*, 1997).

$$\begin{array}{l}
\mathbf{class}\ \mathit{Flatten}\ \mathit{tree}\ a\ \mathbf{where} \\
\quad \mathit{flatten} \quad :: \quad \mathit{tree} \to [\,a\,] \\
\mathbf{instance}\ (\mathit{Flatten}\ \mathit{bush}\ a) \Rightarrow \mathit{Flatten}\ (\mathit{Fork}\ \mathit{bush})\ a\ \mathbf{where} \\
\quad \mathit{flatten}\ (\mathit{Fork}\ \ell\ r) \quad = \quad \mathit{flatten}\ \ell +\!\!+ \mathit{flatten}\ r
\end{array}$$

The auxiliary function *listify* now has the following signature.

$$\mathit{listify} \quad :: \quad (\mathit{Flatten}\ \mathit{bush}\ a) \Rightarrow \mathit{RAL}\ \mathit{bush} \to [\,a\,]$$

Unfortunately, *listify* is not nearly as useful as before because of the type context *Flatten bush a*. Ideally, we would like to have an instance declaration along the following lines which expresses our intent of identifying singleton trees with elements.

$$\begin{array}{l}
\mathbf{instance}\ \mathit{Flatten}\ a\ a\ \mathbf{where} \\
\quad \mathit{flatten}\ a \quad = \quad [\,a\,]
\end{array}$$

This causes, however, ambiguity problems since the definition non-trivially overlaps with the first instance declaration. Consider, for instance, the expression *toList s* where *s* has type *RAL* (*Fork a*). Should *toList s* deliver a list of type [*Fork a*] or a list of type [*a*]?

To summarize: first-order nests are the right choice for implementing sequence types since the sequence container abstraction is one of the few abstractions which is truely parametric with respect to the element type. For container abstractions which require to inspect the elements of a container higher-order nests appear to be preferable.

$$\begin{array}{lcl}
\textbf{data}\ \langle a, b\rangle & = & \mathit{Null} \mid \langle a, b\rangle \\
\mathit{empty} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \\
\mathit{splitMin} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \to \langle a, \mathit{OrdSequence}\ a\rangle \\
\mathit{splitMax} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \to \langle \mathit{OrdSequence}\ a, a\rangle \\
\mathit{member} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{OrdSequence}\ a \to \mathit{Bool} \\
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \\
\mathit{delete} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \\
(\bowtie) & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \\
\mathit{partition} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{OrdSequence}\ a \to (\mathit{OrdSequence}\ a, \mathit{OrdSequence}\ a) \\
\mathit{merge} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \\
\mathit{fromList} & :: & (\mathit{Ord}\ a) \Rightarrow [a] \to \mathit{OrdSequence}\ a \\
\mathit{toList} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{OrdSequence}\ a \to [a]
\end{array}$$

Fig. 5. A signature for ordered sequences.

6 Binomial queues

Binomial queues (Vuillemin, 1978) are probably the first data structure that was developed in close analogy to the binary number system. Binomial queues have been implemented several times in Haskell (King, 1994; Okasaki, 1996b; Hinze, 1998a). However, as we shall see it is not entirely straightforward to adapt the 'standard implementation' to the new setting. We have, in fact, included this application because it exemplifies one of the problems which arise in adapting algorithms to non-regular types. On the other hand, this application also shows some of the beauty and elegance of non-regular types.

Binomial queues implement one-sided *ordered sequences*. Fig. 5 displays a suitable signature for ordered sequences. This container abstraction generalizes priority queues by adding functions for searching and deleting elements and for concatenation and splitting. Binomial queues, however, do not support the latter operations efficiently. These operations have been included in view of the implementations based on finger search trees. Note that the usual operations on priority queues, *isEmpty*, *getMin*, and *deleteMin*, have been combined into a single function, called *splitMin*. The concatenation of two ordered sequences, $s_1 \bowtie s_2$, is only well-defined if $\mathit{max}\ s_1 \leqslant \mathit{min}\ s_2$. The operation *partition a s* splits s into two ordered sequences s_1 and s_2 with $\mathit{max}\ s_1 \leqslant a < \mathit{min}\ s_2$.

A binomial queue is a list of heap-ordered, binomial trees of increasing rank. Recall that a tree is heap-ordered if the elements along each root-to-frontier path are in non-decreasing order. Binomial queues are based on the 'standard' binary number system. There is no point in using a zeroless or a redundant representation.

```
data BinomialQueue subtrees a
  =  Nil
  |  Zero (BinomialQueue (Snoc subtrees) a)
  |  One (Node subtrees a) (BinomialQueue (Snoc subtrees) a)
type OrdSequence  =  BinomialQueue Empty
```

When linking two binomial trees we must be careful to preserve the heap property.

$$\begin{array}{lcl}
\mathit{link} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{Node}\ \mathit{ts}\ a \to \mathit{Node}\ \mathit{ts}\ a \to \mathit{Node}\ (\mathit{Snoc}\ \mathit{ts})\ a \\
\multicolumn{3}{l}{\mathit{link}\ t@(a \triangleleft \mathit{ts})\ u@(b \triangleleft \mathit{us})} \\
\quad \mid a \leqslant b & = & a \triangleleft \mathit{Snoc}\ \mathit{ts}\ u \\
\quad \mid \mathit{otherwise} & = & b \triangleleft \mathit{Snoc}\ \mathit{us}\ t
\end{array}$$

The type of *link* states that two trees of rank n are combined to form a tree of rank $n + 1$. Actually, the type of *link* is more general but we will never use *link* in its full generality — in a sense the type is too general.

Inserting an element into a binomial queue corresponds to incrementing a binary number,

$$\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \to \mathit{OrdSequence}\ a \to \mathit{OrdSequence}\ a \\
\mathit{insert}\ a\ q & = & \mathit{incr}\ (a \triangleleft \mathit{Empty})\ q \\
\mathit{incr} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{Node}\ \mathit{ts}\ a \to \mathit{BQ}\ \mathit{ts}\ a \to \mathit{BQ}\ \mathit{ts}\ a \\
\mathit{incr}\ t\ \mathit{Nil} & = & \mathit{One}\ t\ \mathit{Nil} \\
\mathit{incr}\ t\ (\mathit{Zero}\ \mathit{ds}) & = & \mathit{One}\ t\ \mathit{ds} \\
\mathit{incr}\ t\ (\mathit{One}\ t'\ \mathit{ds}) & = & \mathit{Zero}\ (\mathit{incr}\ (\mathit{link}\ t\ t')\ \mathit{ds})
\end{array}$$

and merging two queues is analogous to adding two binary numbers.

$$\begin{array}{lcl}
\mathit{merge} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{BQ}\ \mathit{ts}\ a \to \mathit{BQ}\ \mathit{ts}\ a \to \mathit{BQ}\ \mathit{ts}\ a \\
\mathit{merge}\ \mathit{Nil}\ \mathit{ds}_2 & = & \mathit{ds}_2 \\
\mathit{merge}\ \mathit{ds}_1\ \mathit{Nil} & = & \mathit{ds}_1 \\
\mathit{merge}\ (\mathit{Zero}\ \mathit{ds}_1)\ (\mathit{Zero}\ \mathit{ds}_2) & = & \mathit{Zero}\ (\mathit{merge}\ \mathit{ds}_1\ \mathit{ds}_2) \\
\mathit{merge}\ (\mathit{Zero}\ \mathit{ds}_1)\ (\mathit{One}\ t_2\ \mathit{ds}_2) & = & \mathit{One}\ t_2\ (\mathit{merge}\ \mathit{ds}_1\ \mathit{ds}_2) \\
\mathit{merge}\ (\mathit{One}\ t_1\ \mathit{ds}_1)\ (\mathit{Zero}\ \mathit{ds}_2) & = & \mathit{One}\ t_1\ (\mathit{merge}\ \mathit{ds}_1\ \mathit{ds}_2) \\
\mathit{merge}\ (\mathit{One}\ t_1\ \mathit{ds}_1)\ (\mathit{One}\ t_2\ \mathit{ds}_2) & = & \mathit{Zero}\ (\mathit{incr}\ (\mathit{link}\ t_1\ t_2)\ (\mathit{merge}\ \mathit{ds}_1\ \mathit{ds}_2))
\end{array}$$

Both *incr* and *merge* use polymorphic recursion. This explains why the type of *merge* is more general than actually needed. If we restricted its type to $(\mathit{Ord}\ a) \Rightarrow \mathit{OS}\ a \to \mathit{OS}\ a \to \mathit{OS}\ a$, the definition would no longer typecheck. It is worth noting that the running time of *merge* is $\Theta(\log n)$ since the amortized cost of the *incr* operations is proportional to the number of binomial trees (Okasaki, 1996b).

6.1 Extracting the minimum element

The operation *splitMin*, which extracts the minimum element, essentially proceeds in two steps. First, the binomial heap with the minimum root is determined and replaced by *Zero*. The subtrees of this tree are then merged with the remaining trees of the queue. Since the lists are maintained in opposite order, we must reverse the former list beforehand. Let us consider how to realize the first step. A moment's reflection reveals that we are facing a problem because we cannot express the type of the function. Using mathematical notation it has type $\mathit{BQ}\ \mathit{ts}\ a \to \langle N\ (S^n\ E)\ a, \mathit{BQ}\ \mathit{ts}\ a\rangle$ where n is the rank of the extracted tree. How do we proceed? The answer is surprisingly simple. We must pull the list reversal forward. Reversing the binomial forest $S^n\ E\ a$ yields a list of type $\mathit{BQ}\ E\ a = \mathit{OS}\ a$, so that the first operation receives the type $BQ\ ts\ a \to \langle a, OS\ a, BQ\ ts\ a\rangle$. To reverse a binomial forest we use the folklore implementation which employs an auxiliary function *revcat* with *revcat as bs* = *reverse as* ++ *bs* (Bird, 1998, p. 289)

```
class Revcat subtrees where
    revcat                :: subtrees a → BQ subtrees a → OrdSequence a
instance Revcat Empty where
    revcat Empty q        =  q
instance (Revcat subtrees) ⇒ Revcat (Snoc subtrees) where
    revcat (Snoc ts t) q  =  revcat ts (One t q)
```

The function *revcat* is called while recursing down the list of binomial trees.

```
data ⟨a, b, c⟩                =  Null | ⟨a, b, c⟩
extract                       :: (Ord a, Revcat ts) ⇒ BQ ts a → ⟨a, OS a, BQ ts a⟩
extract Nil                   =  Null
extract (Zero ds)             =  case extract ds of
    Null                      →  Null
    ⟨m, q, ds′⟩               →  ⟨m, q, Zero ds′⟩
extract (One t@(a ◁ ts) ds)   =  case extract ds of
    Null                      →  ⟨a, revcat ts Nil, Nil⟩
    ⟨m, q, ds′⟩ | a ⩽ m       →  ⟨a, revcat ts Nil, Zero ds⟩
                | otherwise   →  ⟨m, q, One t ds′⟩
```

Note that *extract* relies in an essential way on lazy evaluation as *revcat* is called for every min-candidate. In a strict environment we must first determine the minimum and then selectively extract the minimum heap. The rest is easy. The function *splitMin* calls *extract* and merges the two binomial queues.

```
splitMin         :: (Ord a) ⇒ OrdSequence a → ⟨a, OrdSequence a⟩
splitMin q       =  case extract q of
    Null         →  Null
    ⟨m, q1, q2⟩  →  ⟨m, merge q1 q2⟩
```

6.2 Halving a binomial queue

Numerical representations support operations efficiently which can be interpreted as arithmetic functions. In this section we describe an algorithm for splitting a binomial queue of size n into queues of size $\lceil n\ /\ 2\rceil$ and $\lfloor n\ /\ 2\rfloor$. This operation corresponds roughly to a division by two — or rather, to an arithmetic shift to the left (assuming that the least significant bit comes first).

For splitting a single binomial tree we require *link*'s inverse.

```
unlink                 :: Node (Snoc ts) a → (Node ts a, Node ts a)
unlink (a ◁ Snoc ts u) =  (a ◁ ts, u)
```

It is worth noting that *unlink* is a total function. Its type states that a tree of rank

$n + 1$ is split into two trees of rank n. On regular types *unlink* is necessarily a partial function.

To split a binomial queue into halves we must essentially apply *unlink* to all binomial trees. For this purpose it is useful to lift function application into the realm of pairs.[4]

$$
\begin{array}{lcl}
(\diamond) & :: & (a \to b, c \to d) \to (a, c) \to (b, d) \\
(f_1, f_2) \diamond (a_1, a_2) & = & (f_1\ a_1, f_2\ a_2) \\
\mathit{halve}' & :: & \mathit{BQ}\ (\mathit{Snoc}\ \mathit{ts})\ a \to (\mathit{BQ}\ \mathit{ts}\ a, \mathit{BQ}\ \mathit{ts}\ a) \\
\mathit{halve}'\ \mathit{Nil} & = & (\mathit{Nil}, \mathit{Nil}) \\
\mathit{halve}'\ (\mathit{Zero}\ \mathit{ds}) & = & (\mathit{Zero}, \mathit{Zero}) \diamond \mathit{halve}'\ \mathit{ds} \\
\mathit{halve}'\ (\mathit{One}\ t\ \mathit{ds}) & = & (\mathit{One}, \mathit{One}) \diamond \mathit{unlink}\ t \diamond \mathit{halve}'\ \mathit{ds}
\end{array}
$$

The type of *halve′* reflects the fact that we can only halve binomial queues of even size. If the size is odd, we insert the remaining tree of rank 0 into the first half.

$$
\begin{array}{lcl}
\mathit{halve} & :: & (\mathit{Ord}\ a) \Rightarrow \mathit{BQ}\ \mathit{ts}\ a \to (\mathit{BQ}\ \mathit{ts}\ a, \mathit{BQ}\ \mathit{ts}\ a) \\
\mathit{halve}\ \mathit{Nil} & = & (\mathit{Nil}, \mathit{Nil}) \\
\mathit{halve}\ (\mathit{Zero}\ \mathit{ds}) & = & \mathit{halve}'\ \mathit{ds} \\
\mathit{halve}\ (\mathit{One}\ t\ \mathit{ds}) & = & (\mathit{incr}\ t, \mathit{id}) \diamond \mathit{halve}'\ \mathit{ds}
\end{array}
$$

7 2-3 trees under the left-spine view

Finger search trees are our major application of higher-order nested datatypes. This section studies 2-3 trees under the left-spine view. 2-3 trees under the double-spine view are dealt with in the next section. As we shall see both implementations make intensive use of the class-based programming technique introduced in Sec. 5.1. 2-3 trees under the left-spine view are the best known purely functional implementation of one-sided ordered sequences. The operations exhibit the following amortized time bounds:

member, *insert* and *delete*	take	$\Theta(\log d)$ steps, and
splitMin	takes	$\Theta(1)$ steps,

where d is the distance from the smallest element.

Recall from the introduction that a 2-3 tree under the left-spine view is essentially a list of 2-3 trees increasing by rank. Now, a 2-3 tree of rank n is an element of $\mathit{Node23}^n\ \mathit{Empty}\ a$ where *Node23* is given by

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Node23}\ \mathit{tree}\ a & = & \mathit{Node2}\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a) \\
& | & \mathit{Node3}\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a)\ .
\end{array}
$$

In contrast to the structures of Sec. 4 the size of a 2-3 tree is not determined by its rank: a 2-3 tree of rank n contains between $2^n - 1$ and $3^n - 1$ elements. The results of L. Snyder (1977) show that a redundant representation is necessary to be able to implement *insert* and *delete* efficiently.

[4] This function is termed *cross′* in (Bird, 1998, p. 43).

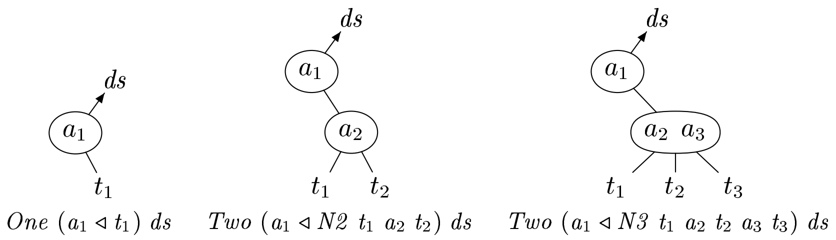

ds
a_1
t_1
$One\ (a_1 \triangleleft t_1)\ ds$
ds
a_1
a_2
t_1 t_2
$Two\ (a_1 \triangleleft N2\ t_1\ a_2\ t_2)\ ds$
ds
a_1
$a_2\ a_3$
t_1 t_2 t_3
$Two\ (a_1 \triangleleft N3\ t_1\ a_2\ t_2\ a_3\ t_3)\ ds$

Fig. 6. The graphical interpretation of the digits 1, 2, and 3.

The first design decision concerns the choice of the underlying number system. The left-spine view of 2-3 trees suggests to base finger search trees on the lazy 1-2 number system. Using the encoding trick of Sec. 5 we arrive at the following definition, which is, in fact, a minor modification of the type *SearchableList* introduced in Sec. 5.2.

```
data FingerSearchTree tree a
  =  Nil
  |  One (Pennant tree a) (FingerSearchTree (Node23 tree) a)
  |  Two (Pennant (Node23 tree) a) (FingerSearchTree (Node23 tree) a)
type OrdSequence  =  FingerSearchTree Empty
```

Recall that a two of weight n is not represented by two pennants of rank n but by a single pennant of rank $n + 1$. But wait! Since *Node23* is not a tuple type, the argument has a slip. A pennant of rank $n + 1$ corresponds either to a two of weight n or to a three of weight n. In other words, the above data structure is based on the lazy 1-2-3 number system! Fortunately, this is exactly what we are after. Finger search trees should be based on a redundant number system in order to achieve the desired amortized time bounds: $\Theta(1)$ for *splitMin* and $\Theta(\log d)$ for *insert*.

To summarize: a finger search tree is like an ordinary 2-3 tree except that each node along the left spine points to its parent instead of its left child. The nodes on the left spine are either 2-, 3-, or 4-nodes; the remaining nodes are either 2- or 3-nodes. Fig. 6 shows the representation of the three digits as Haskell expressions and their graphical interpretation. While the encoding trick proves to be useful for implementing the operations *member*, *insert*, and *delete* it aggravates the readability of those operations which are modelled after arithmetic functions. To restore readability we will make use of so-called *pattern abstractions* (Fähndrich & Boyland, 1997) which allow to name complex patterns. The following three pattern abstractions suggest themselves.

```
pat D1 a1 t1 ds            =  One (a1 ◁ t1) ds
pat D2 a1 t1 a2 t2 ds      =  Two (a1 ◁ N2 t1 a2 t2) ds
pat D2 a1 t1 a2 t2 a3 t3 ds  =  Two (a1 ◁ N3 t1 a2 t2 a3 t3) ds
```

To minimize notation we will use *D1*, *D2*, and *D3* also on the right-hand side of equations.

```
cons                                  ::  a → OrdSequence a → OrdSequence a
cons a s                              =   incr a E s
incr                                  ::  a → t a → FST t a → FST t a
incr a1 t1 Nil                        =   D1 a1 t1 Nil
incr a1 t1 (D1 a2 t2 ds)              =   D2 a1 t1 a2 t2 ds
incr a1 t1 (D2 a2 t2 a3 t3 ds)        =   D3 a1 t1 a2 t2 a3 t3 ds
incr a1 t1 (D3 a2 t2 a3 t3 a4 t4 ds)  =   D2 a1 t1 a2 t2 (incr a3 (N2 t3 a4 t4) ds)
zero                                  ::  FST (Node23 t) a → FST t a
zero Nil                              =   Nil
zero (D1 a1 t1 ds)                    =   T (a1 ◁ t1) (zero ds)
zero (D2 a1 t1 a2 t2 ds)              =   T (a1 ◁ t1) (O (a2 ◁ t2) ds)
zero (D3 a1 t1 a2 t2 a3 t3 ds)        =   T (a1 ◁ t1) (T (a2 ◁ N2 t2 a3 t3) ds)
splitMin                              ::  OrdSequence a → ⟨a, OrdSequence a⟩
splitMin Nil                          =   Null
splitMin (D1 a1 E ds)                 =   ⟨a1, zero ds⟩
splitMin (D2 a1 E a2 E ds)            =   ⟨a1, D1 a2 E ds⟩
splitMin (D3 a1 E a2 E a3 E ds)       =   ⟨a1, D2 a2 E a3 E ds⟩
```

Fig. 7. The implementation of *cons*, *incr*, *zero*, and *splitMin*.

Using these prerequisites the algorithms for adding and removing elements at the front end are easy to code, see Fig. 7. Since a finger search tree represents an ordered sequence, the algorithms are furthermore similar to those on random-access lists. The only difference is that we use the lazy 1-2-3 number system instead of the 1-2 system. Note that *cons*, which adds a new minimum to the front end, is required for the implementation of *insert*. The proof that *cons* and *splitMin* run in $\Theta(1)$ amortized time is identical to the one given in Sec. 3.

7.1 Searching

We study the algorithm for searching an element first because it also lies at the heart of the functions for inserting and deleting elements. The algorithm works in two phases: it first searches the list of trees for the appropriate tree, and then searches the tree for the query element. Note that this procedure is tantamount to an exponential search followed by a binary search (Mehlhorn, 1977). Using type classes the coding of the second phase is relatively straightforward.

```
class Mem tree where
    mem            ::  (Ord a) ⇒ a → tree a → Bool
instance Mem Empty where
    mem a E        =   False
instance (Mem tree) ⇒ Mem (Node23 tree) where
    mem a (N2 t1 a1 t2)
        | a < a1   =   mem a t1
        | a ≡ a1   =   True
        | a > a1   =   mem a t2
```

$$\begin{array}{lll}
\multicolumn{3}{l}{mem\ a\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3)} \\
\quad \mid a < a_1 & = & mem\ a\ t_1 \\
\quad \mid a \equiv a_1 & = & True \\
\quad \mid a < a_2 & = & mem\ a\ t_2 \\
\quad \mid a \equiv a_2 & = & True \\
\quad \mid a > a_2 & = & mem\ a\ t_3
\end{array}$$

When we search the spine list we must take into account that the split keys reside at inconvenient places. Consider, for example, the tree $One\ (a_1 \triangleleft t_1)\ (One\ (a_2 \triangleleft t_2)\ ds)$. In order to determine whether t_1 must be searched we must look one digit ahead: t_1 is the right candidate if $a_1 < a < a_2$. If $a < a_1$, then a must be searched on 'the left'. We encode this information into the result of the search function using the datatype *Loc*.[5]

$$\begin{array}{lll}
\mathbf{data}\ Loc\ a & = & Lt \mid Eq\ a \\
(\star) & :: & (a \to b) \to (Loc\ a \to Loc\ b) \\
f \star Lt & = & Lt \\
f \star Eq\ a & = & Eq\ (f\ a)
\end{array}$$

The operator $(\star)$, which is required in the sequel, corresponds to the *map* function of the data type *Loc*. The search pattern described above is captured by the operator **after**.

$$\begin{array}{lll}
(\mathbf{after}) & :: & a \to Loc\ a \to a \\
a_1\ \mathbf{after}\ Lt & = & a_1 \\
a_1\ \mathbf{after}\ Eq\ a_2 & = & a_2
\end{array}$$

To reduce the number of parenthesis we agree upon that function composition $(\cdot)$ takes precedence over $(\star)$ which in turn takes precedence over **after**.

Now, the function $member'$, which implements the search, is assigned the result type *Loc Bool*. The call $member'\ a\ t$ has consequently two outcomes: *Lt* signals that a must be searched on the left, $Eq\ b$ means that t was searched with b indicating whether the search was successful or not.

$$\begin{array}{lll}
member' & :: & (Ord\ a, Mem\ t) \Rightarrow a \to FST\ t\ a \to Loc\ Bool \\
member'\ a\ Nil & = & Lt \\
\multicolumn{3}{l}{member'\ a\ (One\ (a_1 \triangleleft t_1)\ ds)} \\
\quad \mid a < a_1 & = & Lt \\
\quad \mid a \equiv a_1 & = & Eq\ True \\
\quad \mid a > a_1 & = & Eq\ (mem\ a\ t_1\ \mathbf{after}\ member'\ a\ ds) \\
\multicolumn{3}{l}{member'\ a\ (Two\ (a_1 \triangleleft t_1)\ ds)} \\
\quad \mid a < a_1 & = & Lt \\
\quad \mid a \equiv a_1 & = & Eq\ True \\
\quad \mid a > a_1 & = & Eq\ (mem\ a\ t_1\ \mathbf{after}\ member'\ a\ ds)
\end{array}$$

Again, we have the curious situation that the right-hand sides of the last two

[5] Note that *Loc* is actually isomorphic to the predefined type *Maybe*. We use a self-defined type for reasons of readability and extensibility, see Sec. 8.2.

equations are identical. If we used a more direct representation of the digits — say, $One\ (a_1 \lhd t_1)\ ds$, $Two\ (a_1 \lhd t_1)\ (a_2 \lhd t_2)\ ds$, and $Three\ (a_1 \lhd t_1)\ (a_2 \lhd t_2)\ (a_3 \lhd t_3)\ ds$ — the definition of *member′* would be at least twice as long. This shows that the chosen encoding is far superior because it avoids unnecessary duplication of code. Finally, it is important to note that **after** is non-strict in its *first* argument. Lazy evaluation is vital here to guarantee the $\Theta(\log d)$ running time of *member′*.

The function *member* simply calls *member′* mapping *Lt* to *False* and *Eq b* to *b*.

```
member      :: (Ord a) ⇒ a → OrdSequence a → Bool
member a s  =  False after member′ a s
```

To see that *member* runs in $\Theta(\log d)$ amortized time recall that a digit on the left-spine is assigned a debit of at most one. As *member* processes at most $\log d$ digits, it must discharge at most $\log d$ debits. Furthermore, since its unshared cost for searching a 2-3 tree is $\Theta(\log d)$, we obtain a total amortized running time of $\Theta(\log d)$.

7.2 Insertion

Now, things become interesting. The algorithms we have studied so far are passive in the sense that they do not alter the 2-3 trees contained in a finger search tree. If an element is inserted into a 2-3 tree, the tree can possibly grow. More precisely, inserting an element into a 2-3 tree of rank n results either in a tree of rank n or in a tree of rank $n + 1$. In the latter case the root node is always a 2-node which motivates the following definition.

```
data Grown tree a  =  Unchanged (tree a) | Grown (tree a) a (tree a)
class Ins tree where
    ins            :: (Ord a) ⇒ a → tree a → Grown tree a
```

We have two different representations of a 2-node: $U\ (N2\ t_1\ a_1\ t_2)$ is a 2-node of rank n while $G\ t_1\ a_1\ t_2$ is a 2-node of rank $n + 1$. Alternatively, we may view $G\ t_1\ a_1\ t_2$ as a 4-node of rank n (Brown & Tarjan, 1980, p. 596), an interpretation which fits nicely into the framework of numerical representations. The implementation of *ins* is relatively straightforward if one employs so-called *smart constructors* (Adams, 1993).

```
instance Ins Empty where
    ins a E           =  G E a E
instance (Ins tree) ⇒ Ins (Node23 tree) where
    ins a (N2 t₁ a₁ t₂)
        | a ⩽ a₁      =  node2₁ (ins a t₁) a₁ t₂
        | otherwise   =  node2₂ t₁ a₁ (ins a t₂)
    ins a (N3 t₁ a₁ t₂ a₂ t₃)
        | a ⩽ a₁      =  node3₁ (ins a t₁) a₁ t₂ a₂ t₃
        | a ⩽ a₂      =  node3₂ t₁ a₁ (ins a t₂) a₂ t₃
        | otherwise   =  node3₃ t₁ a₁ t₂ a₂ (ins a t₃)
```

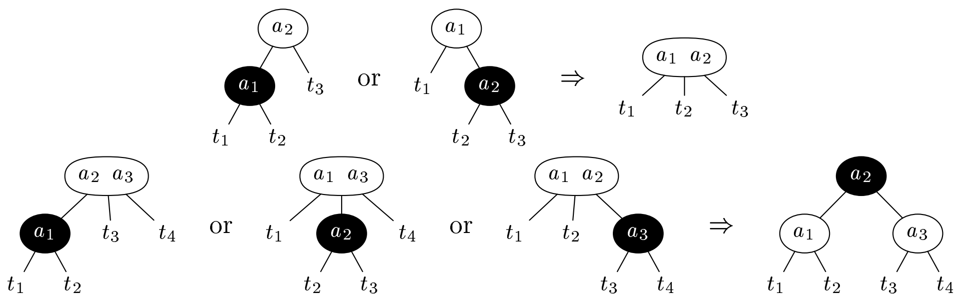


Fig. 8. Rebalancing operations upon insertion (4-nodes are heavily shaded).

The code is quite similar to what one would program for unbalanced trees. The only difference is that the constructors *Node2* and *Node3* have been replaced by functions $node2_i$ and $node3_i$ implementing the rebalancing operations. The index i indicates which subtree was altered. There are essentially two rebalancing operations which are visualized in Fig. 8. If a subtree of a 2-node grows, the node is expanded to a 3-node.

$$\begin{array}{lcl} node2_1\ (U\ t_1)\ a_1\ t_2 & = & U\ (N2\ t_1\ a_1\ t_2) \\ node2_1\ (G\ t_1\ a_1\ t_2)\ a_2\ t_3 & = & U\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3) \end{array}$$

The smart constructor $node2_2$ is defined accordingly. If a subtree of a 3-node grows, the node is split into two 2-nodes. This is the only case where the height of the resulting tree increases.

$$\begin{array}{lcl} node3_1\ (U\ t_1)\ a_1\ t_2\ a_2\ t_3 & = & U\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3) \\ node3_1\ (G\ t_1\ a_1\ t_2)\ a_2\ t_3\ a_3\ t_4 & = & G\ (N2\ t_1\ a_1\ t_2)\ a_2\ (N2\ t_3\ a_3\ t_4) \end{array}$$

The remaining functions, $node3_2$ and $node3_3$, are defined accordingly.

The algorithm for inserting an element into a finger search tree essentially follows the recursion pattern of *member′*. The function *insert′* has result type *Loc* (*FST t a*): *Lt* signals that the element must be inserted on the left, *Eq ds* means that the element has been inserted resulting in the finger search tree *ds*.

```
$insert'$              ::  $(Ord\ a, Ins\ t) \Rightarrow a \to FST\ t\ a \to Loc\ (FST\ t\ a)$
$insert'\ a\ Nil$        =   $Lt$
$insert'\ a\ (One\ p@(a_1 \lhd t_1)\ ds)$
    $|\ a \leqslant a_1$        =   $Lt$
    $|\ otherwise$      =   $Eq\ (one\ a_1\ (ins\ a\ t_1)\ ds$ **after** $One\ p \star insert'\ a\ ds)$
$insert'\ a\ (Two\ p@(a_1 \lhd t_1)\ ds)$
    $|\ a \leqslant a_1$        =   $Lt$
    $|\ otherwise$      =   $Eq\ (two\ a_1\ (ins\ a\ t_1)\ ds$ **after** $Two\ p \star insert'\ a\ ds)$
```

The functions *one* and *two* are smart constructors which continue the balancing along the spine list.

$$\begin{array}{lcl}
\mathit{one}\ a_1\ (U\ t_1)\ t_2 & = & \mathit{One}\ (a_1 \triangleleft t_1)\ t_2 \\
\mathit{one}\ a_1\ (G\ t_1\ a_2\ t_2)\ t_3 & = & \mathit{Two}\ (a_1 \triangleleft \mathit{N2}\ t_1\ a_2\ t_2)\ t_3 \\
\mathit{two}\ a_1\ (U\ t_1)\ t_2 & = & \mathit{Two}\ (a_1 \triangleleft t_1)\ t_2 \\
\mathit{two}\ a_1\ (G\ t_1\ a_2\ t_2)\ t_3 & = & \mathit{Two}\ (a_1 \triangleleft t_1)\ (\mathit{incr}\ a_2\ t_2\ t_3)
\end{array}$$

It remains to implement *insert* which adds an element to an ordered sequence.

$$\begin{array}{lcl}
\mathit{insert} & :: & (\mathit{Ord}\ a) \Rightarrow a \rightarrow \mathit{OrdSequence}\ a \rightarrow \mathit{OrdSequence}\ a \\
\mathit{insert}\ a\ t & = & \mathit{cons}\ a\ t\ \mathbf{after}\ \mathit{insert}'\ a\ t
\end{array}$$

Let us conclude the section with a simple application. We use finger search trees to implement an *adaptive sorting algorithm* which is optimal with respect to the number of inversions. A sorting algorithm is called adaptive if nearly ordered sequences are processed faster than less ordered sequences. To quantify the amount of presortedness several measures have been proposed, see, for instance, (Mannila, 1985; Estivill-Castro & Wood, 1992; Moffat & Petersson, 1992). If the input sequence is $x = a_n a_{n-1} \ldots a_2 a_1$ then the number of inversions is given by

$$\mathit{Inv}(x) = |\{\, (i,j) \mid a_i < a_j \text{ and } i < j \,\}| \ .$$

The measure *Inv* indicates how many exchanges of adjacent elements are needed to sort the input.

The announced algorithm is based on the insertion sort paradigm and works by repeatedly inserting elements into an empty initial tree. The sorted list is obtained by an inorder traversal of the final tree.

$$\begin{array}{lcl}
\mathit{fromList} & :: & (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow \mathit{OrdSequence}\ a \\
\mathit{fromList} & = & \mathit{foldr}\ \mathit{insert}\ \mathit{Nil}
\end{array}$$

The converse of *fromList*, *toList*, is easily programmed. We must essentially adapt the functions of Sec. 5.1 to pennants. Putting things together we obtain an *Inv*-optimal sorting algorithm.

$$\begin{array}{lcl}
\mathit{adaptiveSort} & :: & (\mathit{Ord}\ a) \Rightarrow [\,a\,] \rightarrow [\,a\,] \\
\mathit{adaptiveSort} & = & \mathit{toList} \cdot \mathit{fromList}
\end{array}$$

To see how *adaptiveSort* adapts to the input assume, for example, that the input list is already sorted. In this case *insert* repeatedly calls *cons*, which has an amortized running time of $\Theta(1)$. Consequently, *adaptiveSort* takes time linear to the size of the input sequence. In general we have that the amortized cost of the j-th insertion (given the sequence $x = a_n a_{n-1} \ldots a_2 a_1$) is $\Theta(\log f_j)$ where f_j is the number of elements following a_j that are smaller than a_j, ie $f_j = |\{\, (i,j) \mid a_i < a_j \text{ and } i < j \,\}|$. Since $\mathit{Inv}(x) = \sum_{j=1}^{n} f_j$ we have $\sum_{j=1}^{n} \log f_j = O(n \log(\mathit{Inv}(x)/n))$ which is, in fact, *Inv*-optimal (Mehlhorn, 1979; Mannila, 1985).

7.3 External finger search trees

The decomposition of trees into a base case and a recursion case makes it almost trivial to change the base case. We can use this flexibility, for instance, to improve the space consumption of finger search trees. It is well-known that internal search trees waste memory space since there are more empty nodes than other nodes in a tree. If we estimate the space usage of the constructed value $C\ e_0\ \ldots\ e_{k-1}$ at k cells, the space occupied by empty nodes ranges between $(n+1)/4n \approx 25\%$ for binary trees and $(2n+1)/6n \approx 33\%$ for tertiary trees. This waste of space can be avoided if we replace empty nodes by 2-3 leaves.

```
data Leaf23 a  =  Leaf2 a | Leaf3 a a
```

The constructor *Leaf2* corresponds to an unary tuple and *Leaf3* to a binary tuple. The operation *ins* must be extended to work on leaf nodes.

```
instance Ins Leaf23 where
   ins a (L2 a₁)
      | a ⩽ a₁       =  U (L3 a a₁)
      | otherwise    =  U (L3 a₁ a)
   ins a (L3 a₁ a₂)
      | a ⩽ a₁       =  G (L2 a) a₁ (L2 a₂)
      | a ⩽ a₂       =  G (L2 a₁) a (L2 a₂)
      | otherwise    =  G (L2 a₁) a₂ (L2 a)
```

Since *insert′* is polymorphic with respect to the 'base type' constructor, we can re-use it for the new type. If we attempt to adapt *insert* to the new design, a slight problem shows up. Recall that *insert a t* reduces to $cons\ a\ t = incr\ a\ E\ t$ if a is the new minimal element. The second argument of the auxiliary function *incr* now has type *Leaf23 a* which implies that *incr* can only be used to add two or three elements at the front. A moment's reflection reveals a deficiency of the data structure: it cannot represent a singleton. The smallest trees are *Nil*, $One\ (a_1 \triangleleft L2\ a_2)\ Nil$, and $One\ (a_1 \triangleleft L3\ a_2\ a_3)\ Nil$. It is not hard, however, to remedy this defect. We help ourselves by defining a suitable wrapper datatype.

```
data OrdSequence a  =  Id (FingerSearchTree Leaf23 a)
                    |  Add a (FingerSearchTree Leaf23 a)
```

By convention, a is the smallest element in *Add a t*. The function *insert* is now easily adapted.

```
insert               ::  (Ord a) ⇒ a → OrdSequence a → OrdSequence a
insert a (Id ds)     =   Add a ds after Id ⋆ insert′ a ds
insert a (Add a′ ds)
     | a ⩽ a′        =   Id (incr a (L2 a′) ds)
     | otherwise     =   Id (incr a′ (L2 a) ds) after Add a′ ⋆ insert′ a ds
```

7.4 Deletion

Deletion is usually more intricate than insertion. This general observation also proves to be true for finger search trees. The broad picture should be clear by now: We first decompose the standard algorithm for deletion so that it works on 2-3 trees of rank n. Following the two-level type structure we then define two functions *delete′* and *delete*. The former works for arbitrary 'base type' constructors while the latter is specialized to the type OS.

Deletion is opposite to insertion: if we delete an element from a 2-3 tree of rank n we obtain either a tree of rank n or a tree of rank $n-1$. Unfortunately, the type system is not expressive enough to formulate that a rank decreases. This would correspond to removing an application of a type constructor, which is clearly not possible. Hence, we are forced to reformulate the above statement slightly: if we delete an element from a 2-3 tree of rank $n+1$, we obtain either a tree of rank $n+1$ or a tree of rank n. Thereby, we give up the possibility of deleting an element from a rank 0 tree. However, if *Empty* is used as a base type constructor, this makes perfect sense since there is no point in deleting an element from an empty tree. If we use *Leaf23* as a base type, we must accept a little redundancy.

There are different techniques for deleting an element from a 2-3 tree. We adapt the standard algorithm for binary search trees which works by replacing the element by its inorder successor. The reason for choosing this algorithm is simply that we need the auxiliary function which splices out the minimal element anyway. Here are the necessary definitions.

```
data Shrunk tree a  =  Unchanged (Node23 tree a) | Shrunk (tree a)
class Del tree where
    splice          ::  Node23 tree a → (a, Shrunk tree a)
    del             ::  (Ord a) ⇒ a → Node23 tree a → Shrunk tree a
```

The node $S\ t$ may be interpreted as a 0-node which has only a single child (Brown & Tarjan, 1980, p. 602). The function *splice* splices out the leftmost, ie minimal element of its argument. Note that *splice* returns a pair, not an optional pair as *splitMin*. We can use ordinary pairs since the type of *splice* guarantees that the tree contains at least one element. The instance declarations given in Fig. 9 supply the code for deleting an element from a rank 1 and from a rank $n+2$ tree. Note that intensive use of smart constructors is made. Fig. 10 visualizes the rebalancing operations. If a subtree of a 2-3 node shrinks, it is combined with one or two subtrees of its left or its right sibling.

$$\begin{aligned}
&node2'_1\ (U\ t_1)\ a_1\ t_2 &&= U\ (N2\ t_1\ a_1\ t_2)\\
&node2'_1\ (S\ t_1)\ a_1\ (N2\ t_2\ a_2\ t_3) &&= S\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3)\\
&node2'_1\ (S\ t_1)\ a_1\ (N3\ t_2\ a_2\ t_3\ a_3\ t_4) &&= U\ (N2\ (N2\ t_1\ a_1\ t_2)\ a_2\ (N2\ t_3\ a_3\ t_4))\\
&node3'_1\ (U\ t_1)\ a_1\ t_2\ a_2\ t_3 &&= U\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3)\\
&node3'_1\ (S\ t_1)\ a_1\ (N2\ t_2\ a_2\ t_3)\ a_3\ t_4 &&= U\ (N2\ (N3\ t_1\ a_1\ t_2\ a_2\ t_3)\ a_3\ t_4)\\
&node3'_1\ (S\ t_1)\ a_1\ (N3\ t_2\ a_2\ t_3\ a_3\ t_4)\ a_4\ t_5\\
&\quad = U\ (N3\ (N2\ t_1\ a_1\ t_2)\ a_2\ (N2\ t_3\ a_3\ t_4)\ a_4\ t_5)
\end{aligned}$$

```
instance Del Empty where
    splice (N2 E a1 E)           =  (a1, S E)
    splice (N3 E a1 E a2 E)      =  (a1, U (N2 E a2 E))
    del a (N2 E a1 E)
        | a ≡ a1                 =  S E
        | otherwise              =  U (N2 E a1 E)
    del a (N3 E a1 E a2 E)
        | a ≡ a1                 =  U (N2 E a2 E)
        | a ≡ a2                 =  U (N2 E a1 E)
        | otherwise              =  U (N3 E a1 E a2 E)
instance (Del tree) ⇒ Del (Node23 tree) where
    splice (N2 t1 a1 t2)         =  (a, node2'_1 t1' a1 t2)
        where (a, t1')           =  splice t1
    splice (N3 t1 a1 t2 a2 t3)   =  (a, node3'_1 t1' a1 t2 a2 t3)
        where (a, t1')           =  splice t1
    del a (N2 t1 a1 t2)
        | a < a1                 =  node2'_1 (del a t1) a1 t2
        | a ≡ a1                 =  uncurry (node2'_2 t1) (splice t2)
        | a > a1                 =  node2'_2 t1 a1 (del a t2)
    del a (N3 t1 a1 t2 a2 t3)
        | a < a1                 =  node3'_1 (del a t1) a1 t2 a2 t3
        | a ≡ a1                 =  uncurry (node3'_2 t1) (splice t2) a2 t3
        | a < a2                 =  node3'_2 t1 a1 (del a t2) a2 t3
        | a ≡ a2                 =  uncurry (node3'_3 t1 a1 t2) (splice t3)
        | a > a2                 =  node3'_3 t1 a1 t2 a2 (del a t3)
```

Fig. 9. The implementation of *splice* and *del*.

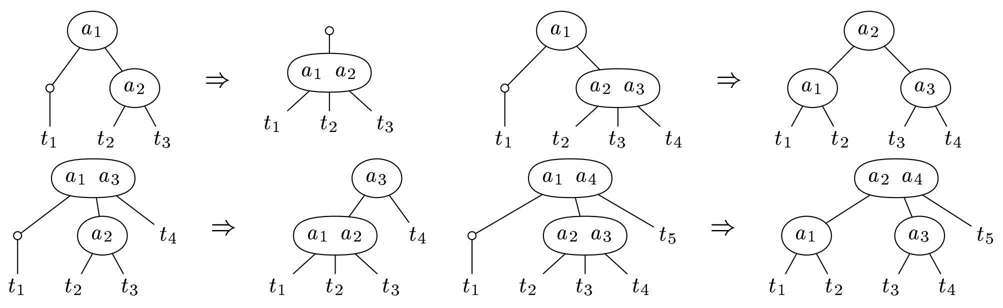


Fig. 10. Rebalancing operations upon deletion.

The functions $node2'_2$, $node3'_2$, and $node3'_3$ are defined accordingly. The fundamental difference between the balancing operations for insertion and those for deletion is that in the latter case the context of a subtree must be taken into account. This has the unfortunate consequence of complicating the definition of *delete'*. Consider, for instance, the finger tree *One* $(a \lhd t)$ *ds* and assume that deleting an element in t causes the tree to shrink. Now, we are stuck because the necessary left context is no longer available. Furthermore, we cannot counterbalance t's loss of height by looking to the right since *ds* may be empty. We help ourselves by wrapping the result of *delete′ a ds* into yet another pair of constructors which signal whether the height of a subtree has decreased or not.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{FingerSearchTree}'\ \mathit{tree}\ a} \\
\quad = & \mathit{Unchanged}'\ (\mathit{FingerSearchTree}\ (\mathit{Node23}\ \mathit{tree})\ a) \\
\quad \mid & \mathit{Shrunk}'\ a\ (\mathit{tree}\ a)\ (\mathit{FingerSearchTree}\ (\mathit{Node23}\ (\mathit{Node23}\ \mathit{tree}))\ a)
\end{array}
$$

The function *delete′* returns an element of type $Loc\ (FST'\ t\ a)$.

$$
\begin{array}{lll}
\mathit{delete}' & :: & (\mathit{Ord}\ a, \mathit{Del}\ t) \Rightarrow a \to \mathit{FST}\ (\mathit{Node23}\ t)\ a \to \mathit{Loc}\ (\mathit{FST}'\ t\ a) \\
\mathit{delete}'\ a\ \mathit{Nil} & = & \mathit{Lt} \\
\multicolumn{3}{l}{\mathit{delete}'\ a\ (\mathit{One}\ p@(a_1 \triangleleft t_1)\ \mathit{ds})} \\
\quad \mid a < a_1 & = & \mathit{Lt} \\
\quad \mid a \equiv a_1 & = & \mathit{Eq}\ (\mathit{uncurry}\ \mathit{one}_1\ (\mathit{splice}\ t_1)\ \mathit{ds}) \\
\quad \mid a > a_1 & = & \mathit{Eq}\ (\mathit{one}_1\ a_1\ (\mathit{del}\ a\ t_1)\ \mathit{ds}\ \textbf{after}\ U' \cdot \mathit{one}_2\ p \star \mathit{delete}'\ a\ \mathit{ds}) \\
\multicolumn{3}{l}{\mathit{delete}'\ a\ (\mathit{Two}\ p@(a_1 \triangleleft t_1)\ \mathit{ds})} \\
\quad \mid a < a_1 & = & \mathit{Lt} \\
\quad \mid a \equiv a_1 & = & \mathit{Eq}\ (U'\ (\mathit{uncurry}\ \mathit{two}_1\ (\mathit{splice}\ t_1)\ \mathit{ds})) \\
\quad \mid a > a_1 & = & \mathit{Eq}\ (U'\ (\mathit{two}_1\ a_1\ (\mathit{del}\ a\ t_1)\ \mathit{ds}\ \textbf{after}\ \mathit{two}_2\ p \star \mathit{delete}'\ a\ \mathit{ds}))
\end{array}
$$

The smart constructors one_i and two_i perform the necessary rebalancing operations.

$$
\begin{array}{lll}
\mathit{one}_1\ a_1\ (U\ t_1)\ t_2 & = & U'\ (\mathit{One}\ (a_1 \triangleleft t_1)\ t_2) \\
\mathit{one}_1\ a_1\ (S\ t_1)\ t_2 & = & S'\ a_1\ t_1\ t_2 \\
\mathit{two}_1\ a_1\ (U\ t_1)\ t_2 & = & \mathit{Two}\ (a_1 \triangleleft t_1)\ t_2 \\
\mathit{two}_1\ a_1\ (S\ t_1)\ t_2 & = & \mathit{One}\ (a_1 \triangleleft t_1)\ t_2
\end{array}
$$

If the subtree of a ‘one’ node shrinks, the auxiliary constructor S' comes into play. It is eliminated by the next enclosing one_2 or two_2 function.

$$
\begin{array}{lll}
\mathit{one}_2\ p\ (U'\ \mathit{ds}) & = & \mathit{One}\ p\ \mathit{ds} \\
\mathit{one}_2\ (a_1 \triangleleft t_1)\ (S'\ a_2\ t_2\ \mathit{ds}) & = & \mathit{Two}\ (a_1 \triangleleft \mathit{N2}\ t_1\ a_2\ t_2)\ (\mathit{zero}\ \mathit{ds}) \\
\mathit{two}_2\ p\ (U'\ \mathit{ds}) & = & \mathit{Two}\ p\ \mathit{ds} \\
\mathit{two}_2\ (a_1 \triangleleft \mathit{N2}\ t_1\ a_2\ t_2)\ (S'\ a_3\ t_3\ \mathit{ds}) & = & \mathit{One}\ (a_1 \triangleleft t_1)\ (\mathit{One}\ (a_2 \triangleleft \mathit{N2}\ t_2\ a_3\ t_3)\ \mathit{ds}) \\
\multicolumn{3}{l}{\mathit{two}_2\ (a_1 \triangleleft \mathit{N3}\ t_1\ a_2\ t_2\ a_3\ t_3)\ (S'\ a_4\ t_4\ \mathit{ds})} \\
\multicolumn{3}{l}{\quad =\ \mathit{One}\ (a_1 \triangleleft t_1)\ (\mathit{One}\ (a_2 \triangleleft \mathit{N3}\ t_2\ a_3\ t_3\ a_4\ t_4)\ \mathit{ds})}
\end{array}
$$

Note that *delete′* expects a finger tree of type $FST\ (Node23\ t)\ a$, ie the first 2-3 tree in the spine list must be of rank 1. Thus we are forced to treat the first digit separately.

```
delete             ::  (Ord a) ⇒ a → OrdSequence a → OrdSequence a
delete a Nil        =  Nil
delete a (One p@(a1 ◁ E) ds)
        | a < a1    =  One p ds
        | a ≡ a1    =  zero ds
        | a > a1    =  One p ds after one2 p ⋆ delete' a ds
delete a (Two p@(a1 ◁ t1) ds)
        | a < a1    =  Two p ds
        | a ≡ a1    =  uncurry two1 (splice t1) ds
        | a > a1    =  two1 a1 (del a t1) ds after two2 p ⋆ delete' a ds
```

This duplication of code is in a sense the price one has to pay for using the type system to check structural constraints.

Let us conclude the section with a simple application. Dijkstra's algorithm for single-source shortest paths and Prim's algorithm for minimum-weight spanning trees both require a *decreaseKey* operation, which replaces an element of a given sequence by a smaller one. Building upon *insert* and *delete* its definition is straightforward.

```
decrease                      ::  (Ord a) ⇒ a → a → OS a → OS a
decrease a a' q
        | member a q ∧ a' < a  =  insert a' (delete a q)
        | otherwise            =  q
```

7.5 Ordinary 2-3 trees

We have shown in the introduction that ordinary 2-3 trees can also be expressed as a nested datatype. Let us briefly consider how the operations of search, insertion, and deletion can be realized for this data structure. Recall that the structure of 2-3 trees is captured by the following datatype definition.

```
data Tree23 tree k  =  Zero (tree k) | Succ (Tree23 (Node23 tree) k)
```

Building upon the constructor class *Mem* the implementation of *member* is entirely straightforward.

```
member               ::  (Ord a, Mem t) ⇒ a → Tree23 t a → Bool
member a (Zero t)     =  mem a t
member a (Succ t)     =  member a t
```

Insertion and deletion are equally straightforward to implement. The change of height must merely be encoded into the prefix of *Succ* and *Zero* constructors. This is easily accomplished using smart constructors.

```
insert               ::  (Ord a, Ins t) ⇒ a → Tree23 t a → Tree23 t a
insert a (Zero t)     =  zero (ins a t)
insert a (Succ t)     =  Succ (insert a t)
```

```
zero              :: Grown t a → Tree23 t a
zero (U t)        =  Zero t
zero (G t1 a1 t2) =  Succ (Zero (N2 t1 a1 t2))
```

For deletion we look one constructor ahead to be able to counterbalance a possible loss of height.

```
delete                    :: (Ord a, Del t) ⇒ a → Tree23 t a → Tree23 t a
delete a (Zero t)         =  Zero t
delete a (Succ (Zero t))  =  succ (del a t)
delete a (Succ (Succ t))  =  Succ (delete a (Succ t))
succ                      :: Shrunk t a → Tree23 t a
succ (S t)                =  Zero t
succ (U t)                =  Succ (Zero t)
```

8 2-3 trees under the double-spine view

2-3 trees under the left-spine view are biased towards the front end: extracting the minimum is cheap and simple to implement; extracting the maximum is expensive and non-trivial to implement. By symmetrizing the design we obtain a data structure which treats both ends on an equal basis. 2-3 trees under the double-spine view are the best known purely functional implementation of ordered sequences:

member, *insert* and *delete*	take	$\Theta(\log(\min\{d, n-d\}))$ steps,
splitMin and *splitMax*	take	$\Theta(1)$ steps,
$(\bowtie)$	takes	$\Theta(\log(\min\{n_1, n_2\}))$ steps,
partition	takes	$\Theta(\log(\min\{d, n-d\}))$ steps, and
merge	takes	$\Theta(n_s \log(n_\ell / n_s))$ steps,

where d is the distance from the smallest element, n_s is the size of the shorter and n_ℓ the size of the longer sequence.

The type of 2-3 trees under the double-spine view is a slight modification of random-access deques introduced in Sec. 5.2.

```
data FingerSearchTree tree a
       = Simple2 a
       | Simple3 a (tree a) a
       | Composite (Digit Front tree a)
                   (FingerSearchTree (Node23 tree) a)
                   (Digit Rear tree a)
data Digit pennant tree a
       = One (pennant tree a)
       | Two (pennant (Node23 tree) a)
data Front tree a       =  a ◁ tree a
data Rear tree a        =  tree a ▷ a
data OrdSequence a      =  Nil | Id (FingerSearchTree Empty a)
```

Two points are worth mentioning. First of all, *FST* is not capable of representing the empty sequence. For that reason the wrapper datatype *OS* is introduced. Second, we use different types of pennants for the digits on the left and for those on the right spine. The types have been chosen so that the order of elements within an expression of type *OS* reflects the order of elements within the sequence represented. Thus the smallest finger search trees are *Nil*, $Id\ (S2\ a_1)$, $Id\ (S3\ a_1\ E\ a_2)$, and $Id\ (C\ (One\ (a_1 \lhd E))\ (S2\ a_2)\ (One\ (E \rhd a_3)))$. An ordered sequence of size 4 is either represented by

$$Id\ (C\ (Two\ (a_1 \lhd N2\ E\ a_2\ E))\ (S2\ a_3)\ (One\ (E \rhd a_4)))$$

or by

$$Id\ (C\ (One\ (a_1 \lhd E))\ (S2\ a_2)\ (Two\ (N2\ E\ a_3\ E \rhd a_4)))\ .$$

As in the preceding section we employ pattern abstractions to improve the readability of the code.

```
pat F1 a₁ t₁              =  One (a₁ ◁ t₁)
pat F2 a₁ t₁ a₂ t₂        =  Two (a₁ ◁ N2 t₁ a₂ t₂)
pat F3 a₁ t₁ a₂ t₂ a₃ t₃  =  Two (a₁ ◁ N3 t₁ a₂ t₂ a₃ t₃)
```

The pattern abstraction for the rear digits, *R1*, *R2*, and *R3*, are defined accordingly.

8.1 Deque operations

The operations *cons*, *incr*, *zero*, *splitMin* and their colleagues *snoc*, *rcni*, *eroz*, *splitMax* can be easily adapted to the new design. Here, we show the modified versions of *cons* and *incr* only.

```
cons                                 ::  a → OrdSequence a → OrdSequence a
cons a Nil                           =   Id (S2 a)
cons a (Id s)                        =   Id (incr a E s)
incr                                 ::  a → t a → FST t a → FST t a
incr a₁ t₁ (S2 a₂)                   =   S3 a₁ t₁ a₂
incr a₁ t₁ (S3 a₂ t₂ a₃)             =   C (F1 a₁ t₁) (S2 a₂) (R1 t₂ a₃)
incr a₁ t₁ (C (F1 a₂ t₂) m r)        =   C (F2 a₁ t₁ a₂ t₂) m r
incr a₁ t₁ (C (F2 a₂ t₂ a₃ t₃) m r)  =   C (F3 a₁ t₁ a₂ t₂ a₃ t₃) m r
incr a₁ t₁ (C (F3 a₂ t₂ a₃ t₃ a₄ t₄) m r)
     =  C (F2 a₁ t₁ a₂ t₂) (incr a₃ (N2 t₃ a₄ t₄) m) r
```

It is instructive to relate the equations to the rebalancing operations on 2-3 trees. The first equation of *incr* corresponds to an expansion of a 2-node to a 3-node, the second equation realizes a split of a 3-node into two 2-nodes. If an ordered sequence is built by repeatedly calling *cons*, we obtain a quite regularly shaped tree: the 2-3 trees are complete binary trees and on the right spine only the digit 1 appears.

The operations *cons* and *snoc* (and *splitMin* and *splitMax*) operate largely independent on both ends. This independence also simplifies the proof that the deque operations run in $\Theta(1)$ amortized time, see (Okasaki, 1998, Sec. 11.1). In essence, the debit allowances of the front digit and those of the rear digit are simply added. Define the debit allowance of a digit by $\sharp(1) = 0$, $\sharp(2) = 1$, and $\sharp(3) = 0$. Then the delayed call m in $C\ f\ m\ r$ is allowed $\sharp(f) + \sharp(r)$ debits. A routine proof shows that the operations do, in fact, preserve this debit invariant.

8.2 Bag operations

For reasons of space we will only consider the re-implementation of *member*. The modification of *insert* and *delete* is left as an instructive exercise to the reader. To adapt *member* we must first symmetrize the auxiliary datatype *Loc*.

```
data Loc a                  =   Lt | Eq a | Gt
between                     ::  a → Loc a → a → a
between a₁ Lt a₃            =   a₁
between a₁ (Eq a₂) a₃       =   a₂
between a₁ Gt a₃            =   a₃
```

The function *between* generalizes the operator **after** of Sec. 7.1. Depending on the value of the second argument the search is continued to the left or to the right. The function *member′* implements a quasi-parallel search along the two spines. If d is the distance from the smallest element, then *member* runs in $\Theta(\log(\min\{d, n-d\}))$ amortized time.

```
member′               ::  (Ord a, Mem t) ⇒ a → FST t a → Loc Bool
member′ a (S2 a₁)
    | a < a₁          =   Lt
    | a ≡ a₁          =   Eq True
    | a > a₁          =   Gt
member′ a (S3 a₁ t₁ a₂)
    | a < a₁          =   Lt
    | a ≡ a₁          =   Eq True
    | a < a₂          =   Eq (mem a t₁)
    | a ≡ a₂          =   Eq True
    | a > a₂          =   Gt
member′ a (C f m r)
    | a < min′ f      =   Lt
    | a > max′ r      =   Gt
    | otherwise       =   Eq (between (mem a f) (member′ a m) (mem a r))
```

Note that we assume that *Digit Front* and *Digit Rear* are instances of *Mem*. The auxiliary function *min′* determines the leftmost, ie smallest element contained in a digit, *max′* accordingly yields the rightmost, ie largest element.

```
min′ (One (a₁ ◁ t₁))   =   a₁        max′ (One (t₁ ▷ a₁))   =   a₁
min′ (Two (a₁ ◁ t₁))   =   a₁        max′ (Two (t₁ ▷ a₁))   =   a₁
```

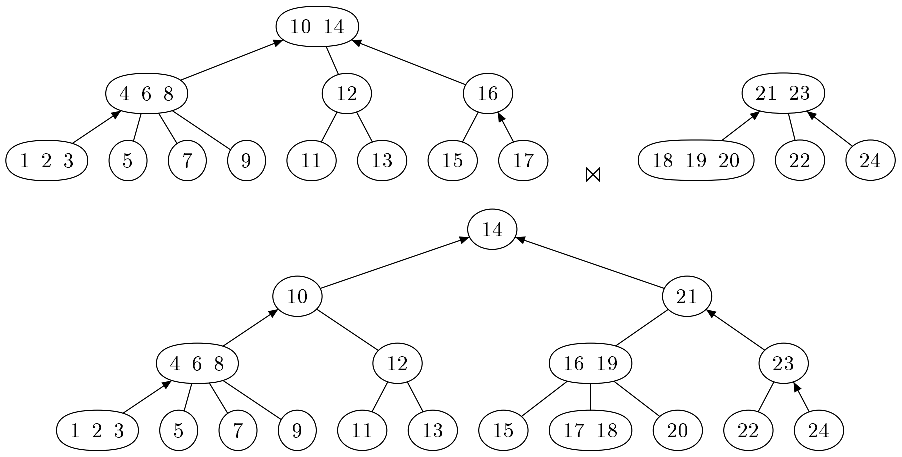


Fig. 11. The concatenation of *fromList* [1 . . 17] and *fromList* [18 . . 24].

The function *member* calls *member′* mapping *Lt* and *Gt* to *False* and *Eq b* to *b*.

```
member            :: (Ord a) ⇒ a → OrdSequence a → Bool
member a Nil       = False
member a (Id s)    = between False (member′ a s) False
```

The adaptive sorting algorithm described in Sec. 7.2 has the irritating property that the worst case is a list in descending order which is arguably almost sorted. In a sense this is due to the measure *Inv* which yields the largest value for lists in descending order. If we represent ordered sequences by 2-3 trees under the double-spine view, we obtain a sorting algorithm which is optimal with respect to $\overline{Inv}$ given by

$$\overline{Inv}(x) = \min\{Inv(x), Inv(reverse\ x)\}\ .$$

The new worst-case is an interleaving of an ascending and a descending list:

$$1, 2n, 2, 2n-1, 3, 2n-2, \ldots, n-2, n+1, n-1, n\ .$$

Again, one can argue that this sequence is almost sorted. In Sec. 8.5 we will introduce yet another sorting algorithm based on finger search trees which sorts the above sequence in linear time.

8.3 Concatenation

It is well-known that 2-3 trees support the operations concatenation and splitting in $\Theta(\log n)$ time. Is it possible to adapt these algorithms to the double-spine view without loss of efficiency? The answer is in the affirmative. It turns out that the runtime complexity is even slightly better since both algorithms profit from the bottom-up representation.

The concatenation of two ordered sequences is depicted in Fig. 11. Note that $s_1 \bowtie s_2$ is only well-defined if every element of s_1 is smaller than every element of

s_2. The operation works by traversing the spines of both trees until the root of the smaller tree is reached. Along the walk the pennants on the right spine of the first argument are joined with the pennants on the left spine of the second argument. The remaining pennants form the spine lists of the concatenated sequence. The function *join*, which is explained below, implements a single step.

```
join   ::   D R t a → G t a → D F t a → G (Node23 t) a
join (R1 t1 a1) (U t2) (F1 a2 t3)
        =   U (N3 t1 a1 t2 a2 t3)
join (R1 t1 a1) (G t2 a2 t3) (F1 a3 t4)
        =   G (N2 t1 a1 t2) a2 (N2 t3 a3 t4)
join (R1 t1 a1) (U t2) (F2 a2 t3 a3 t4)
        =   G (N2 t1 a1 t2) a2 (N2 t3 a3 t4)
...
join (R2 t1 a1 t2 a2) (G t3 a3 t4) (F2 a4 t5 a5 t6)
        =   G (N3 t1 a1 t2 a2 t3) a3 (N3 t4 a4 t5 a5 t6)
```

The function *join* takes three arguments: the rear digit of the first sequence, the join of the previous step, and the front digit of the second sequence. Since a join may produce a tree of a higher rank, the second argument and the result are of type *Grown t a* and *Grown (Node23 t) a*, respectively. Inspecting the left-hand sides of the equations we see that elements and subtrees are always strictly alternating, ie we never have two elements or two subtrees in succession. This renders *join*'s implementation almost trivial: if the second argument is of the form $G\ t_1\ a_1\ t_2$, then both digits are converted to nodes of corresponding sizes, ie the digit i becomes an $(i+1)$-node. Otherwise, the larger digit, say, i is converted to an i-node (the only exception to this rule is the first equation of *join*). That said it becomes clear that care must be taken to ensure that *join* does not receive the digit three for which there is no corresponding node. This is easily achieved by applying the transformation $3a = 1(inc\,a)$ prior to a call of *join*, see below.

The function *app* iterates *join* until one of its arguments becomes simple. The terminating cases correspond to simple, double and triple increments.

```
app                               ::  FST t a → G t a → FST t a → FST t a
app (S2 a1) (U t1) x              =   incr a1 t1 x
app (S2 a1) (G t1 a2 t2) x        =   incr a1 t1 (incr a2 t2 x)
app (S3 a1 t1 a2) (U t2) x        =   incr a1 t1 (incr a2 t2 x)
app (S3 a1 t1 a2) (G t2 a3 t3) x  =   incr a1 t1 (incr a2 t2 (incr a3 t3 x))
```

The symmetric cases are defined accordingly. If both arguments are composite, *app* recurses.

```
app (C f1 m1 r1) t (C f2 m2 r2)  =  C f1 (app m1' (join r1' t f2') m2') r2
    where (m1', r1')             =  mron m1 r1
          (f2', m2')             =  norm f2 m2
```

The helper function *norm* implements the above mentioned normalization step

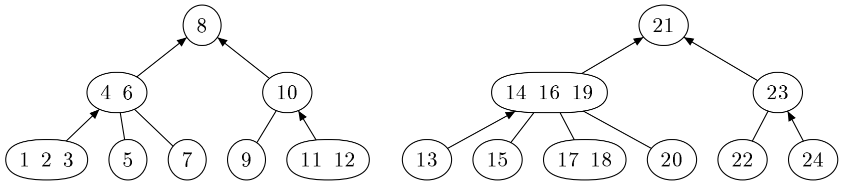


Fig. 12. The concatenated sequence of Fig. 11 partitioned at 13.

$3a = 1(inc\, a)$ and is defined as follows (*mron* is defined symmetrically).

$$\begin{array}{lcl} norm\ (F3\ a_1\ t_1\ a_2\ t_2\ a_3\ t_3)\ m & = & (F1\ a_1\ t_1, incr\ a_2\ (N2\ t_2\ a_3\ t_3)\ m) \\ norm\ f\ m & = & (f, m) \end{array}$$

Finally, $(\bowtie)$ checks for empty arguments and calls *app* with the second argument set to $U\ E$.

$$\begin{array}{lcl} (\bowtie) & :: & OrdSequence\ a \to OrdSequence\ a \to OrdSequence\ a \\ Nil \bowtie s_2 & = & s_2 \\ s_1 \bowtie Nil & = & s_1 \\ Id\ s_1 \bowtie Id\ s_2 & = & Id\ (app\ s_1\ (U\ E)\ s_2) \end{array}$$

Since *app* recurses to the height of the smaller tree, $s_1 \bowtie s_2$ runs in $\Theta(\min\{h_1, h_2\})$ amortized time where h_i is the height of s_i. By contrast the top-down concatenation of 2-3 trees takes $\Theta(\max\{h_1, h_2\})$ time.

8.4 Splitting

The operation *partition* $a\ s$ splits s into two ordered sequences s_1 and s_2 with $max\ s_1 \leqslant a \leqslant min\ s_2$. For simplicity, we allow both s_1 and s_2 to contain elements equal to a. The strict variant of *partition*, which yields two sequences with $max\ s_1 \leqslant a < min\ s_2$, can be defined within the same time bounds. However, its definition is more awkward since a large number of cases must be considered. Fig. 12 displays the result of *partition* 13 applied to the concatenated sequence of Fig. 11.

Splitting a sequence proceeds in two steps. First a 2-3 tree with $min\ t \leqslant a \leqslant max\ t$ is determined conducting a quasi-parallel, exponential search along the spines. On the way back the 2-3 tree is successively split into two halves which are combined with the pennants on the spines to form the two ordered sequences. The function *part* which implements both stages is dual to the function *app* of Sec. 8.3: it takes an element of type $FST\ t\ a$ and returns a triple of type $(FST\ t\ a, t\ a, FST\ t\ a)$.

```
part                  :: (Ord a) ⇒ a → FST t a → Loc (FST t a, t a, FST t a)
part a (S2 a₁)
      | a ⩽ a₁         = Lt
      | otherwise      = Gt
part a (S3 a₁ t₁ a₂)
      | a ⩽ a₁         = Lt
      | a ⩾ a₂         = Gt
      | otherwise      = Eq (S2 a₁, t₁, S2 a₂)
part a (C f m r)
      | a ⩽ min′ f     = Lt
      | a ⩾ max′ r     = Gt
      | otherwise      = Eq (between (cutf a f m r) (cut ⋆ part a m) (cutr a f m r))
    where cut s        = split a f s r
```

The recursion terminates if either the root of the tree is reached or a tree with $min\ t \leqslant a \leqslant max\ t$ is found. In the latter case one of the auxiliary functions *cutf* or *cutr* is called which performs the initial cut. Here is the code for *cutf* with *cutr* being defined symmetrically.

```
cutf a (F1 a₁ t₁) m r    = (S2 a₁, t₁, zero m r)
cutf a (F2 a₁ t₁ a₂ t₂) m r
      | a ⩽ a₂            = (S2 a₁, t₁, C (F1 a₂ t₂) m r)
      | otherwise         = (zero (S2 a₁) (R1 t₁ a₂), t₂, zero m r)
cutf a (F3 a₁ t₁ a₂ t₂ a₃ t₃) m r
      | a ⩽ a₂            = (S2 a₁, t₁, C (F2 a₂ t₂ a₃ t₃) m r)
      | a ⩽ a₃            = (zero (S2 a₁) (R1 t₁ a₂), t₂, C (F1 a₃ t₃) m r)
      | otherwise         = (zero (S2 a₁) (R2 t₁ a₂ t₂ a₃), t₃, zero m r)
```

On the way back the helper function *split* is called which realizes a single partitioning step. It takes four arguments: the split key, the front digit, the result of the previous step, and the rear digit.

```
split a f (m₁, N2 t₁ a₁ t₂, m₂) r
      | a ⩽ a₁        = (eroz f m₁, t₁, C (F1 a₁ t₂) m₂ r)
      | otherwise     = (C f m₁ (R1 t₁ a₁), t₂, zero m₂ r)
split a f (m₁, N3 t₁ a₁ t₂ a₂ t₃, m₂) r
      | a ⩽ a₁        = (eroz f m₁, t₁, C (F2 a₁ t₂ a₂ t₃) m₂ r)
      | a ⩽ a₂        = (C f m₁ (R1 t₁ a₁), t₂, C (F1 a₂ t₃) m₂ r)
      | otherwise     = (C f m₁ (R2 t₁ a₁ t₂ a₂), t₃, zero m₂ r)
```

Finally, *partition* calls *part* and takes care of the various special cases.

```
partition                :: (Ord a) ⇒ a → OS a → (OS a, OS a)
partition a Nil          = (Nil, Nil)
partition a (Id t)       = between (Nil, Id t) (wrap ⋆ part a t) (Id t, Nil)
  where wrap (t₁, E, t₂) = (Id t₁, Id t₂)
```

The running time of *partition* is similar to that of *member* and *insert*: it takes $\Theta(\log(\min\{d, n-d\}))$ amortized time. By contrast, the top-down variant runs in $\Theta(\log n)$ time.

8.5 Merging and Margesort

Building upon $(\bowtie)$ and *partition* we can define the *merge* of two ordered sequences.

```
merge                :: (Ord a) ⇒ OS a → OS a → OS a
merge x y            =  case splitMin y of
  Null               →  x
  ⟨a, y′⟩            →  x₁ ⋈ cons a (merge y′ x₂)
      where (x₁, x₂) =  partition a x
```

The algorithm is reminiscent of the usual *merge* on lists. It is, however, more efficient since it uses exponential and binary search rather than linear search to partition the two input sequences into the minimum number of segments that must be reordered to obtain the output sequence. It can be shown that *merge* is asymptotically optimal, ie it takes $\Theta(n_s \log(n_\ell/n_s))$ amortized time where n_s is the size of the shorter and n_ℓ the size of the longer sequence (Moffat *et al.*, 1992) — provided the strict variant of partition is used in its definition. The worst-case for *merge* are sequences which must be interleaved to obtain the ordered output sequence: $merge\ [a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_n] = [a_1, b_1, a_2, b_2, \ldots, a_n, b_n]$. In this case *merge* does not profit from the exponential and binary search.

Besides being asymptotically optimal *merge* also exhibits an adaptive behaviour. Consider, for instance, the sequences s_1 and s_2 with $max\ s_1 \leqslant min\ s_2$. In this special case *merge* degenerates to $(\bowtie)$ and the running time consequently amounts to $\Theta(\min\{h_1, h_2\})$. Note that a number of measures of *premergedness* have been proposed in order to characterize simple problem instances for *merge*. The interested reader is referred to (Carlsson *et al.*, 1993) for an in-depth treatment of the subject.

The idea suggests itself to use the above implementation of *merge* as the basis for an adaptive merge sort algorithm. The resulting algorithm is termed *Margesort* by its inventors A. Moffat, O. Petersson, and N. Wormald (1992).

```
margeSort  :: (Ord a) ⇒ [a] → [a]
margeSort  =  toList · foldm merge Nil · map single

single     :: a → OrdSequence a
single a   =  Id (S2 a)
```

Margesort employs a standard divide and conquer strategy. This recursion pattern is implemented by the higher-order function *foldm* which is a colleague of *foldr* and *foldl*. Its name was chosen to indicate that it builds a balanced expression tree: for $f = foldm\ (*)\ e$ we have $f\ [\,] = e$, $f\ [a] = a$, and $f\ [a_1, ..., a_n] = f\ [a_1, ..., a_m] * f\ [a_{m+1}, ..., a_n]$ where $m = \lceil n/2 \rceil$.

The Margesort algorithm is remarkable because it is optimally adaptive to a range of measures of presortedness, namely *Inv*, *Rem*, and *SMS*. No other algorithm with

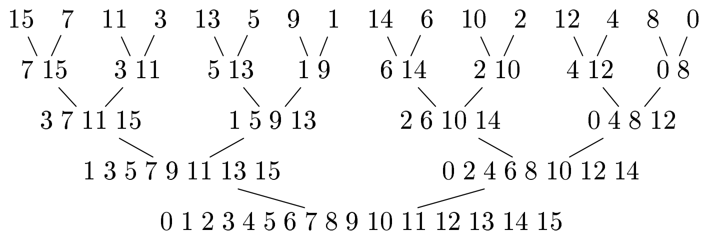


Fig. 13. The worst-case for Margesort.

this behaviour is known to the author. The measure $Rem(x)$ counts the minimum number of elements that must be removed from x to leave a sorted sequence.

$$Rem(x) = n - \max\{\, n \mid x \text{ has an ascending subsequence of size } n \,\}$$

An algorithm is *Rem*-optimal if it sorts in $\Theta(n + Rem(x)\log(Rem(x)))$ time. The measure $SMS(x)$ is defined as the minimum number of monotone subsequences into which x can be decomposed.

$$SMS(x) = \min\{\, k \mid x \text{ can be composed of } k \text{ monotone sequences} \,\}$$

An algorithm is *SMS*-optimal if it sorts in $\Theta(n\log(SMS(x)))$ time. Note that *SMS* is superior to the measure *Runs* which counts the number of ascending runs in the input, ie every *SMS*-optimal algorithm is also *Runs*-optimal. Putting things together we have that the running time of Margesort is

$$\Theta(\min\{n + Rem(x)\log(Rem(x)),\ n\log(Inv(x)/n),\ n\log(SMS(x))\})\ ,$$

see (Moffat *et al.*, 1992). The worst-case for Margesort arises if every *merge* operation must interleave its arguments. Fig. 13 illustrates the worst-case for a sequence of size 16. Such a sequence can be systematically constructed if one repeatedly 'uninterleaves' an ascending sequence. An uninterleaving step reverses a *merge* operation, ie the sequence $[a_1, b_1, a_2, b_2, \ldots, a_n, b_n]$ is split into $[a_1, a_2, \ldots, a_n]$ and $[b_1, b_2, \ldots, b_n]$. The uninterleaving of $[0\,..\,15]$, for instance, can be understood simply by turning Fig. 13 upside down. It turns out that the sequence thus obtained corresponds to the so-called *bit-reversal permutation* (Cormen *et al.*, 1991, Problem 18.1) of the original sequence. A bit-reversal permutation operates on sequences whose size is $n = 2^k$ for some natural number k and swaps elements whose indices have binary representations that are the reverse of each other.

8.6 Variations

If 2-3 finger search trees are augmented with size fields, they can additionally support order statistic operations. Selecting the i-th smallest element, for instance, can be implemented in $\Theta(\log(\min\{i, n - i\}))$ time. Let us briefly sketch the necessary changes to the data structure. It suffices to add size fields to 2-3 nodes and to maintain a 'global' size field.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Node23}\ \mathit{tree}\ a & = & \mathit{Node2}\ \mathit{Int}\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a) \\
 & \mid & \mathit{Node3}\ \mathit{Int}\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a)\ a\ (\mathit{tree}\ a) \\
\textbf{data}\ \mathit{OrdSequence}\ a & = & \mathit{Nil} \mid \mathit{Id}\ \mathit{Int}\ (\mathit{FingerSearchTree}\ \mathit{Empty}\ a)
\end{array}$$

The size of a 2-3 tree can be determined simply by inspecting the size field. As usual, we define a suitable type class for this purpose.

$$\begin{array}{lcl}
\textbf{class}\ \mathit{Size}\ \mathit{tree}\ \textbf{where} & & \\
\quad \mathit{size} & :: & \mathit{tree}\ a \to \mathit{Int} \\
\textbf{instance}\ \mathit{Size}\ \mathit{Empty}\ \textbf{where} & & \\
\quad \mathit{size}\ E & = & 0 \\
\textbf{instance}\ \mathit{Size}\ (\mathit{Node23}\ \mathit{tree})\ \textbf{where} & & \\
\quad \mathit{size}\ (\mathit{N2}\ s\ t_1\ a_1\ t_2) & = & s \\
\quad \mathit{size}\ (\mathit{N3}\ s\ t_1\ a_1\ t_2\ a_2\ t_3) & = & s
\end{array}$$

In order to maintain the size fields it is useful to define *smart constructors* which do the necessary calculations behind the scenes.

$$\begin{array}{lcl}
\mathit{n2} & :: & (\mathit{Size}\ t) \Rightarrow t\ a \to a \to t\ a \to \mathit{Node23}\ t\ a \\
\mathit{n2}\ t_1\ a_1\ t_2 & = & \mathit{N2}\ (\mathit{size}\ t_1 + 1 + \mathit{size}\ t_2)\ t_1\ a_1\ t_2 \\
\mathit{n3} & :: & (\mathit{Size}\ t) \Rightarrow t\ a \to a \to t\ a \to a \to t\ a \to \mathit{Node23}\ t\ a \\
\mathit{n3}\ t_1\ a_1\ t_2\ a_2\ t_3 & = & \mathit{N3}\ (\mathit{size}\ t_1 + 1 + \mathit{size}\ t_2 + 1 + \mathit{size}\ t_3)\ t_1\ a_1\ t_2\ a_2\ t_3
\end{array}$$

Now, the code of the previous sections can be easily adapted by systematically replacing the constructors *N2* and *N3* by their smart counterparts. The implementation of $\mathit{select} :: \mathit{OS}\ a \to \mathit{Int} \to a$ which determines the i-th smallest element is left as an instructive exercise to the reader. Note that the size field in *Id* is necessary to be able to conduct a quasi-parallel search along the spines.

9 Related work

Numerical representations The idea of numerical representations traces back to 'A programming and problem-solving seminar' conducted by M.J. Clancy and D.E. Knuth (1977). In *loc. cit.* they describe their attempt to find a data structure for which the operations of search, insertion, and deletion all take $O(\log d)$ *worst-case* time. A solution is sketched which amounts to a balanced search tree under the left-spine view. The major insight is, however, that rebalancing operations along the spine can be modelled after a *segmented, redundant binary number system* where the digits are drawn from the set $\{0, 1, 2\}$. The digit 2 may be interpreted as a carry in progress. The numbers satisfy the invariant that any pair of 2's is separated by at least one 0. Using this representation the binary increment can be implemented in constant time.

1. Find the leftmost 2 if there is one and change $2a$ to $0(a + 1)$.
2. Increment the leftmost digit (which cannot be a 2).

Interestingly, nearly the same system was re-invented eighteen years later by H. Kaplan and R.E. Tarjan (1995) who call the underlying design principle *recursive slowdown*. To be able to support both increment and decrement in constant time the digits are drawn from the set $\{0, \ldots, 5\}$. A simpler system using the digits $\{-1, 0, 1, 2\}$ can be found in (Clancy & Knuth, 1977). In contrast to the lazy number systems used in this paper segmented, redundant binary numbers guarantee worst-case rather than amortized time bounds. Their implementation is, however, more costly as consecutive 1s must be grouped in blocks to be able to execute step 1 in constant time.

The archetypical example for binary numerical representations are probably binomial queues (Vuillemin, 1978). K. Mehlhorn (1984) employs number systems to dynamize static data structures which do not support insertion and deletion efficiently. To exemplify, consider an ordered table which can be built in time $O(n \log n)$ and searched in time $O(\log n)$. In order to support modifications efficiently one uses a sequence of tables of non-decreasing size — the size of a table corresponds to the weight of a digit. If this data structure is modelled after the binary number system, searching slows down to $O(\log^2 n)$, but insertion and deletion can be supported in time $O(\log n)$. A comprehensive treatment of number systems can be found in the recent textbook by C. Okasaki (1998) who develops this algorithmic design principle to a fine art. In particular, lazy number systems are presented as a simple alternative to segmented, redundant number systems. The underlying design principle is called *implicit recursive slowdown*.

Finger search trees Finger search trees were invented by L.J. Guibas, E.M. McCreight, M.F. Plass, and J.R. Roberts (1977) who employ level-linked B-trees of order at least 25. In order to achieve the desired time bound of $O(\log d)$ they impose so-called regularity constraints on the search paths, which are closely related to segmented, redundant number systems. M.R. Brown and R.E. Tarjan (1980) proposed to simplify the design by using level-linked 2-3 trees. Since this choice bases their data structure on the 1-2 number system, the resulting time bounds are amortized rather than worst-case. Furthermore, the data structure cannot handle arbitrary sequences of insertions and deletions efficiently. The latter deficiency can be overcome by using a redundant number system. This change was, in fact, advocated by S. Huddleston and K. Mehlhorn (1982) who base their data-structure on level-linked *weak B-trees* (the simplest weak B-trees are 2-3-4 trees). Level-linked trees allow to maintain an arbitrary set of dynamic fingers. If one confines oneself to a static finger at the front and/or at the rear end, simpler data structures may be used. K. Mehlhorn (1979) describes a data structure which is similar to the one introduced in Sec. 7. Instead of 2-3 trees he uses sequences of AVL-trees. The details of insertion are quite complicated in order to achieve worst-case bounds. In essence the representation is based on a segmented number system where the digits are drawn from the set $\{\frac{1}{2}, 1, 2\}$. AVL-trees offer a slightly greater degree of freedom since also trees of successive heights may be linked, ie $\frac{1}{2}$ and 1 may be 'added' to 2.

H. Kaplan and R.E. Tarjan (1996) informally describe three purely functional implementations of 2-3 finger search trees. All three solutions are based on the double-spine view and the first two are superficially similar to the data structure of Sec. 8. The first implementation is modelled after the recursive-slowdown number system and uses leaf-oriented 2-3 trees, which contain additional split keys to conduct the search. Due to the segmentation the two spines must be managed as a unit. By contrast, the data structure of Sec. 8 treats both spines largely independent, ie a *cons* only touches the left spine. The second implementation separates the two spine lists and uses the digits $\{0, \ldots, 3\}$. It is, however, unclear how unbalanced situations (one spine list is empty, the other is not) are avoided. Finally, an implementation is sketched which uses one level of structural bootstrapping to obtain a double-logarithmic concatenation time. The principle idea is to represent the spine lists by one of the first two structures.

Purely functional implementations Random-access lists were originally presented by V.J. Dielissen and A. Kaldewaij (1995) as leftist left-complete leaf trees. Based on this representation P.R. Borges (1997, Sec. 6.2) gives an implementation in Haskell. C. Okasaki (1998) recasts this data structure as a binary numerical representation and specifies it both as a regular (Sec. 9.2.1) and as a non-regular datatype (Sec. 10.1.2). Binomial queues have been implemented several times in Haskell (King, 1994; Okasaki, 1996b; Hinze, 1998a). The formulation as a higher-order nested datatype, however, seems to be original. The first functional implementation of 2-3 trees is due to C.M. Hoffmann and M.J. O'Donnell (1982). They use equational rewrite rules to specify insertion and rebalancing. Later C.M.P Reade (1992) extended the approach to cover deletions. It is interesting to compare this implementation with the one based on nested datatypes. C.M.P Reade uses extra constructors (called *Put* and *Taken*) to signal that the height of a subtree has changed. These constructors are, however, elements of the datatype *Tree23*. This mingling undermines the type system and in turn complicates the proof of correctness, which is sketched in *loc. cit.* By contrast, we use separate types for this purpose. Furthermore, using nested datatypes all structural invariants can be made manifest. C.M.P Reade establishes these properties by hand using so-called subtype sets. Recently, in independent work R. Paterson (1998) has defined 2-3 trees as a first-order nested datatype. Insertion is implemented using higher-order functions, very much as in the first definition of *listify* in Sec. 5.1.

We have noted in the introduction that functional implementations of priority queues do not support the operations *delete* and *decreaseKey* efficiently. This deficiency has been noted by several researchers. D.J. King (1994) proposes to combine a priority queue with a set data structure (actually, it should be a bag) which records the valid elements of the queue. Deletion works by deleting the element from the set. Unfortunately, the running times are determined by the set operations so that it seems preferable to use a set data structure only. An alternative approach suggested by C. Okasaki (1996) is to use two priority queues, one containing 'positive' occurrences of an element and one containing 'negative' occurrences. Deletion is implemented by insertion into the negative queue. However, since deletion does not actually remove an element, the running times are influenced by the complete history of the data structure.

Nested datatypes R.H. Connelly and F.L. Morris (1995) have generalized the concept of a trie to permit indexing by elements of an arbitrary monomorphic datatype. The implementation of generalized tries requires both nested datatypes and polymorphic recursion. Recently, the author has extended the approach to arbitrary first-order polymorphic datatypes (1998b). Implementing these tries places even greater demands on the type system: it requires rank-2 type signatures and higher-order nested datatypes. The textbook on functional data structures by C. Okasaki (1998) contains a wealth of examples of first-order nested datatypes, which serve as implementations of several sequence abstractions: indexable sequences, queues, deques, and catenable deques. An unusual application of nested datatypes was given by R. Bird and R. Paterson (1998a) who express de Bruijn notation as a nested datatype. In a companion paper (1998b) they furthermore show how to define fold operators, so-called generalized folds, on first-order nested datatypes.

10 Conclusion

The main contributions of this paper are as follows. We have shown that numerical representations can be nicely captured by nested declarations. In contrast to a regular datatype a nest can often express all structural constraints of a data structure, so that violations of these constraints can be detected at compile-time. While developing the Haskell code presented in the paper higher-order polymorphism and constructor classes have proven their value. Higher-order nested datatypes are in many cases preferable to first-order nests because they allow to make the connection between container types and element types explicit. Constructor classes in turn simplify programming with higher-order nests by building and passing access functions (like *mem*, *ins*, or *del*) automatically and implicitly. Since a part of the work is done behind the scenes, the resulting code is more concise and probably also clearer.

The main innovation from a data-structural point of view is certainly the purely functional implementation of 2-3 finger search trees which appears to be the best known functional data structure for ordered sequences. By maintaining a static finger at the front and one at the rear end 2-3 finger search trees support efficient access to the minimum and the maximum of the sequence. Furthermore, insertions and deletions are cheap in the vicinity of a finger. In detail, the operations *splitMin* and *splitMax* are supported in $\Theta(1)$ amortized time, *member*, *insert*, and *delete* in $\Theta(\log(\min\{d, n-d\}))$ amortized time where d is the distance from the smallest element. Concatenation is supported in $\Theta(\log(\min\{n_1, n_2\}))$, splitting in $\Theta(\log(\min\{d, n-d\}))$, and merge in $\Theta(n_s \log(n_\ell/n_s))$ amortized time where n_s is the size of the shorter and n_ℓ the size of the longer sequence. Since 2-3 finger search trees are based on a lazy number system, these bounds remain valid even if the data structure is used in a persistent setting.

Finally, we have shown that various adaptive sorting algorithms, most notably *Margesort*, can be implemented in a functional setting without sacrificing the known time bounds.

Let us conclude with some remarks on the expressibility of nested declarations.

We have seen that nested datatypes can, for instance, capture the structure of 2-3 trees. Now, 2-3 trees belong to the class of 0-balanced trees[6]. Other instances of that class, 1-2 brother trees (Ottmann & Wood, 1980), 2-3-4 trees or red-black trees (Guibas & Sedgewick, 1978), can be equally easily encoded as a nested datatype. For 1-balanced trees, which are also known as AVL-trees (Adel'son-Vel'skiĭ & Landis, 1962), a slightly more elaborate scheme is required. The idea is to maintain two type parameters which correspond to AVL-trees of successive heights.

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Bin\ tree\ tree'\ a} & = & \mathit{Leftbiased}\ (\mathit{tree'}\ a)\ a\ (\mathit{tree}\ a) \\
 & | & \mathit{Balanced}\ (\mathit{tree'}\ a)\ a\ (\mathit{tree'}\ a) \\
 & | & \mathit{Rightbiased}\ (\mathit{tree}\ a)\ a\ (\mathit{tree'}\ a) \\
\textbf{data}\ \mathit{AVL\ tree\ tree'\ a} & = & \mathit{Zero}\ (\mathit{tree}\ a) \\
 & | & \mathit{Succ}\ (\mathit{AVL\ tree'}\ (\mathit{Bin\ tree\ tree'})\ a) \\
\textbf{type}\ \mathit{OrdSequence} & = & \mathit{AVL\ Empty\ Leaf}
\end{array}$$

Note that the definition of *AVL* follows the recursion pattern of the function *fibs* given by

$$\mathit{fibs}\ a\ a' \quad = \quad a : \mathit{fibs}\ a'\ (a + a') \ ,$$

which generates the infinite list of Fibonacci numbers if called with 0 and 1. Other tree schemes which have been shown to be expressible by nested declarations include Braun trees and left-complete trees (Paterson, 1998).

Numerical representations fit particularly well into the framework of nested datatypes. The examples we have studied in this paper are based on variations of the lazy binary number system. The use of the binary system is, however, by no means compelling. Alternative choices which are also definable as nests include ternary or quaternary number systems (Okasaki, 1998, Sec. 9.4) or the Fibonacci number system (Knuth, 1997, Ex. 1.2.8-34). On the negative side, nested datatypes are not expressive enough to capture sparse or segmented representations. Skew-binary random-access lists (Okasaki, 1995b), for instance, are based on a sparse representation. Briefly, in a sparse representation the digit 0 is not represented explicitly, ie the weights of two successive digits may differ by an arbitrary amount which renders an implementation using a nested datatype impossible.

References

Adams, Stephen. (1993). Functional Pearls: Efficient sets—a balancing act. *Journal of functional programming*, **3**(4), 553–561.

Adel'son-Vel'skiĭ, G.M., & Landis, Y.M. (1962). An algorithm for the organization of information. *Doklady akademiia nauk SSSR*, **146**, 263–266. English translation in Soviet Math. Dokl. 3, pp. 1259–1263.

Aho, Alfred V., Hopcroft, John E., & Ullman, Jeffrey D. (1983). *Data structures and algorithms*. Addison-Wesley Publishing Company.

[6] In general, a binary tree is called a-balanced if for each node the heights of the left and the right subtree differ by at most a.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J. (ed), *Fourth international conference on mathematics of program construction, MPC'98, Marstrand, Sweden*. Lecture Notes in Computer Science, vol. 1422. Springer Verlag.

Bird, Richard, & Paterson, Ross. (1998a). De Bruijn notation as a nested datatype. *Journal of functional programming*. To appear.

Bird, Richard, & Paterson, Ross. (1998b). *Generalised folds for nested datatypes*. Submitted for publication.

Bird, Richard S. (1997). Functional Pearls: On building trees with minimum height. *Journal of functional programming*, **7**(4), 441–445.

Brodal, Gerth Stølting, & Okasaki, Chris. (1996). Optimal purely functional priority queues. *Journal of functional programming*, **6**(6), 839–857.

Brown, Mark R., & Tarjan, Robert E. (1980). Design and analysis of a data structure for representing sorted lists. *SIAM journal on computing*, **9**(3), 594–614.

Carlsson, Svante, Levcopoulos, Christos, & Petersson, Ola. (1993). Sublinear merging and natural mergesort. *Algorithmica*, **9**, 629–648.

Clancy, Michael J., & Knuth, Donald E. 1977 (April). *A programming and problem-solving seminar*. Technical Report CS-TR-77-606. Stanford University, Department of Computer Science.

Connelly, Richard H., & Lockwood Morris, F. (1995). A generalization of the trie data structure. *Mathematical structures in computer science*, **5**(3), 381–418.

Dielissen, Victor J., & Kaldewaij, Anne. (1995). A simple, efficient, and flexible implementation of flexible arrays. *Pages 232–241 of: Third international conference on mathematics of program construction (MPC'95)*. Lecture Notes in Computer Science, vol. 947. Springer Verlag.

Dijkstra, E.W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, **1**, 269–271.

Estivill-Castro, Vladimir, & Wood, Derick. (1992). A survey of adaptive sorting algorithms. *ACM computing surveys*, **24**(4), 441–476.

Fähndrich, Manuel, & Boyland, John. 1997 (June). Statically checkable pattern abstractions. *Pages 75–84 of: Proceedings of the 1997 ACM SIGPLAN international conference on functional programming*.

Guibas, Leo J., & Sedgewick, Robert. (1978). A diochromatic framework for balanced trees. *Pages 8–21 of: Proceedings of the 19th annual symposium on foundations of computer science*. IEEE Computer Society.

Guibas, Leo J., McCreight, Edward M., Plass, Michael F., & Roberts, Janet R. 1977 (May). A new representation for linear lists. *Pages 49–60 of: Conference record of the ninth annual ACM symposium on theory of computing*.

Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM transactions on programming languages and systems*, **15**(2), 253–289.

Hinze, Ralf. (1998a). Functional Pearls: explaining binomial heaps. *Journal of functional programming*. To appear.

Hinze, Ralf. 1998b (November). *Generalizing generalized tries*. Tech. rept. IAI-TR-98-11. Institut für Informatik III, Universität Bonn.

Hoffmann, Christoph M., & O'Donnell, Michael J. (1982). Programming with equations. *ACM transactions on programming languages and systems*, 83–112.

Huddleston, Scott, & Mehlhorn, Kurt. (1982). A new data structure for representing sorted lists. *Acta informatica*, **17**, 157–184.

Hughes, R. John Muir. (1986). A novel representation of lists and its application to the function "reverse". *Information processing letters*, **22**(3), 141–144.

Jones, Mark P. (1995a). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Jones, Mark P. (1995b). Functional programming with overloading and higher-order polymorphism. *Pages 97–136 of: First international spring school on advanced functional programming techniques.* Lecture Notes in Computer Science, vol. 925. Springer Verlag.

Kaplan, Haim, & Tarjan, Robert E. 1995 (May). Persistent lists with catenation via recursive slow-down. *Pages 93–102 of: Proceedings of the twenty-seventh annual ACM symposium on the theory of computing.*

Kaplan, Haim, & Tarjan, Robert E. 1996 (May). Purely functional representations of catenable sorted lists. *Pages 202–211 of: Proceedings of the twenty-eighth annual ACM symposium on the theory of computing.*

King, D. J. (1994). Functional binomial queues. Hammond, K., Turner, D. N., & Sansom, P. M. (eds), *Glasgow functional programming workshop.* Ayr, Scotland: Springer Verlag.

Knuth, Donald E. (1997). *The art of computer programming, Volume 1: Fundamental algorithms.* 3rd edn. Addison-Wesley Publishing Company.

Kubiak, Ryszard, Hughes, John, & Launchbury, John. (1992). Implementing projection-based strictness analysis. Heldal, Rogardt, Kehler Holst, Carsten, & Wadler, Philip (eds), *Functional programming, Glasgow 1991: Proceedings of the 1991 workshop, Portree, Isle of Skye.* Workshops in Computing. London: Springer Verlag.

Mannila, Heikki. (1985). Measures of presortedness and optimal sorting algorithms. *IEEE transactions on computers*, **34**, 318–325.

Mehlhorn, K. (1977). *Effiziente Algorithmen.* Stuttgart, Germany: B.G. Teubner.

Mehlhorn, Kurt. (1979). Sorting presorted files. *Pages 199–212 of: Proceedings 4th GI conference on theoretical computer science.* Lecture Notes in Computer Science, no. 67. Springer-Verlag.

Mehlhorn, Kurt. (1984). *Data structures and algorithms 3: Multi-dimensional searching and computational geometry.* Berlin: Springer Verlag.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(3), 348–375.

Moffat, A., & Petersson, O. (1992). An overview of adaptive sorting. *Australian computer journal*, **24**(2), 70–77.

Moffat, A., Petersson, O., & Wormald, N.C. (1992). Sorting and/by merging finger trees. *Pages 499–509 of: ISAAC: 3rd international symposium on algorithms and computation*, vol. 650.

Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. Paul, M., & Robinet, B. (eds), *International symposium on programming, 6th colloquium Toulouse.* LNCS 167.

Okasaki, Chris. (1995a). Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. *Pages 646–654 of: 36th annual symposium on foundations of computer science (FOCS'95).* Los Alamitos: IEEE Computer Society Press.

Okasaki, Chris. (1995b). Purely functional random-access lists. *Pages 86–95 of: Proceedings of the seventh international conference on functional programming languages and computer architecture (FPCA'95).* La Jolla, California: ACM Press, for ACM SIGPLAN/SIGARCH and IFIP WG2.8.

Okasaki, Chris. (1995c). Simple and efficient purely functional queues and deques. *Journal of functional programming*, **5**(4), 583–592.

Okasaki, Chris. (1996a). Functional data structures. *Pages 131–158 of: The second international summer school on advanced functional programming techniques.* Lecture Notes in Computer Science, vol. 1129.

Okasaki, Chris. 1996b (May). The role of lazy evaluation in amortized data structures. *Pages 62–72 of: ACM SIGPLAN international conference on functional programming.*

Okasaki, Chris. 1997 (June). Catenable double-ended queues. *Pages 66–74 of: Proceedings of the 1997 ACM SIGPLAN international conference on functional programming. ACM SIGPLAN Notices,* 32(8), August 1997.

Okasaki, Chris. (1998). *Purely functional data structures.* Cambridge University Press.

Ottmann, T., & Wood, D. (1980). 1-2 brother trees or AVL trees revisited. *The computer journal*, **23**(3), 248–255.

Paterson, Ross. 1994 (April). *Control structures from types.* ftp://santos.doc.ic.ac.uk/pub/papers/R.Paterson/folds.dvi.gz.

Paterson, Ross. 1998 (October). *Private communication.*

Paulson, L. C. (1996). *ML for the working programmer.* 2nd edn. Cambridge University Press.

Peterson, J., & Hammond, K. 1997 (March). *Report on the programming language Haskell 1.4, a non-strict, purely functional language.* Research Report YALEU/DCS/RR-1106. Yale University, Department of Computer Science.

Peyton Jones, Simon, Jones, Mark, & Meijer, Erik. (1997). Type classes: an exploration of the design space. *Haskell workshop.*

Prim, R.C. (1957). Shortest connection networks and some generalizations. *Bell systems technical journal*, November, 1389–1401.

Reade, C.M.P. (1992). Balanced trees with removals: an exercise in rewriting and proof. *Science of computer programming*, **18**(2), 181–204.

Snyder, Lawrence. (1977). On uniquely represented data structures (extended abstract). *Pages 142–146 of: 18th annual symposium on foundations of computer science, Providence.* Long Beach, Ca., USA: IEEE Computer Society Press.

Vuillemin, Jean. (1978). A data structure for manipulating priority queues. *Communications of the ACM*, **21**(4), 309–315.