Bootstrapping One-sided Flexible Arrays

Ralf Hinze
Institut für Informatik III
Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
ralf@informatik.uni-bonn.de

Abstract

The abstract data type *one-sided flexible array*, also called random-access list, supports look-up and update of elements and can grow and shrink at one end. We describe a purely functional implementation based on weight-balanced multiway trees that is both simple and versatile. A novel feature of the representation is that the running time of the operations can be tailored to one's needs—even dynamically at array-creation time. In particular, one can trade the running time of look-up operations for the running time of update operations. For instance, if the multiway trees have a fixed degree, the operations take $\Theta(\log\, n)$ time, where n is the size of the flexible array. If the degree doubles levelwise, look-up speeds up to $\Theta(\sqrt{\log n})$ while update slows down to $\Theta(2^{\sqrt{\log n}})$. We show that different tree shapes can be conveniently modelled after mixed-radix number systems.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; E.1 [**Data**]: Data Structures—*arrays, trees*; I.1.2 [**Computing Methodologies**]: Algorithms—*analysis of algorithms*

General Terms

Algorithms, design, performance

Keywords

Purely functional data structures, flexible arrays, sub-logarithmic look-up, mixed-radix number systems, Haskell

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 ACM 1-58113-487-8/02/0010 ...$5.00

1 Introduction

One-sided flexible arrays, also called random-access lists, are a hybrid of arrays and lists: they support array-like operations such as look-up and update of elements and list-like operations such as *cons*, *head*, and *tail*. Flexible arrays are typically used when efficient random access is required and the arrays are used in a non-single threaded manner *or* when the size of the arrays varies dynamically. A variety of implementations is available. Braun trees [11], for instance, support all operations in $\Theta(\log\, n)$ time. An unrivalled data structure is the skew binary random-access list [15, 16], which provides logarithmic array operations and constant time list operations.

A common characteristic of the tree-based implementations is the logarithmic time bound for the look-up operation. By contrast, pure functional arrays as to be found in the Haskell standard library [19] support constant-time look-up. On the negative side, updating a 'real' array is prohibitively expensive as it takes time linear in the size of the array. This paper describes an alternative, purely functional data structure that mediates between the two extremes.

The data structure itself is quite simple: we employ multiway trees where each node consists of an array of elements and an array of subtrees. Thus, our implementation is bootstrapped from an existing implementation of arrays. The base array type can be chosen at will: it may be a real array, an array of bounded size, a Braun tree, or even an ordinary list. Of course, different choices result in different running times of the bootstrapped operations.

The performance is furthermore influenced by the size of the nodes. Consider bootstrapping from an array with constant time access and linear time update. If the nodes have a fixed degree, then the operations take $\Theta(\log\, n)$ time. However, if the degree doubles levelwise, then look-up speeds up to $\Theta(\sqrt{\log n})$ while update slows down to $\Theta(2^{\sqrt{\log n}})$. A pleasant feature of our implementation is that the structure of a multiway tree is only determined when an array is first created. Much like an egg cell the initial array incorporates the blue print for its future development. We show that a large class of tree shapes can be conveniently modelled after so-called mixed-radix number systems. By choosing an appropriate number system bootstrapped arrays can be tuned for single-threaded or persistent use, for monotonic (arrays may only grow) or non-monotonic use.

The rest of the paper is structured as follows. Sec. 2 introduces the abstract data type of one-sided flexible arrays. Sec. 3 defines multiway trees and describes the implementation of the basic operations. Sec. 4 deals with the creation of arrays and determines the performance of bootstrapped arrays for different choices of tree shapes.

```
infixl 9 !
class Array a where
    -- array-like operations
  (!)      ::  a x → Int → x
  update   ::  (x → x) → Int → a x → a x
    -- list-like operations
  empty    ::  a x → Bool
  size     ::  a x → Int
  nil      ::  a x
  copy     ::  Int → x → a x
  cons     ::  x → a x → a x
  head     ::  a x → x
  tail     ::  a x → a x
    -- mapping functions
  map      ::  (x → y) → (a x → a y)
  zip      ::  (x → y → z) → (a x → a y → a z)
    -- conversion functions
  list     ::  a x → [x]
  array    ::  [x] → a x
```

Figure 1. Signature of one-sided flexible arrays.

Sec. 5 shows how to implement additional operations such as converting from and to lists. Sec. 6 presents preliminary measurements (micro-benchmarks) comparing multiway trees to other implementations of flexible arrays. Finally, Sec. 7 reviews related work and Sec. 8 concludes.

2 One-sided flexible arrays

Fig. 1 lists the signature of one-sided flexible arrays, phrased as a Haskell type class. The operations can be roughly divided into four categories: array-like operations ('!' and *update*), list-like operations (*empty*, *size*, *nil*, *copy*, *cons*, *head*, and *tail*), mapping functions (*map* and *zip*), and conversion functions (*list* and *array*). Most of the operations should be self-explanatory, so we content ourselves with describing the less common ones. The array-operation *update* applies its first argument to the array element at the given position and returns the modified array. The function *copy n x* creates an array of size n that contains n copies of x. App. A contains a reference implementation using lists.

Since we use *map* and *zip* quite heavily in the sequel, we adopt the following notational convenience: we write both *map f* and *zip f* simply as f^* (unless there is danger of confusion).

3 Multiway trees

The data type of multiway trees is parameterized by the type of the base array a and by the element type x.

$$\textbf{data}\ \mathit{Tree}\ a\ x \quad = \quad \langle a\ x, a\ (\mathit{Tree}\ a\ x)\rangle$$

A node $\langle xs, ts\rangle$ is a pair consisting of an array xs of elements, called the *prefix*, and an array ts of subtrees.

In what follows we show how to turn *Tree a* into an instance of *Array* provided that a is already an instance.

$$\textbf{instance}\ (\mathit{Array}\ a) \Rightarrow \mathit{Array}\ (\mathit{Tree}\ a)\ \textbf{where}$$

We require the multiway trees to satisfy a number of invariants. But, rather than stating the conditions from the outset, we will introduce them as we go along.

3.1 List-like operations

Let us start with the list-like operations since they will determine the way indexing is done. The idea for consing elements to an array is as follows: first fill up the element array in the root node until it contains as many elements as there are subtrees. Then, if the root is full up, distribute the elements evenly among the subtrees and start afresh.

$$\begin{array}{lcl} \mathit{cons}\ x\ \langle xs, ts\rangle & & \\ \quad \mid \mathit{size}\ xs < \mathit{size}\ ts & = & \langle \mathit{cons}\ x\ xs, ts\rangle \\ \quad \mid \mathit{otherwise} & = & \langle \mathit{cons}\ x\ \mathit{nil}, \mathit{cons}^*\ xs\ ts\rangle \end{array}$$

To make this algorithm work, we have to maintain the following invariant.

> **Invariant 1:** for all nodes $\langle xs, ts\rangle$:
>
> $$\mathit{size}\ xs \leqslant \mathit{size}\ ts \quad \wedge \quad 1 \leqslant \mathit{size}\ ts.$$

The number of elements must not exceed the number of subtrees. Furthermore, each node must contain at least one subtree so that the second line of *cons* creates a legal node. Clearly, this invariant requires *non-strict evaluation* as the trees are inherently infinite. We will come back to this aspect when we study the creation of arrays in Sec. 4. For the moment, just note that *cons* never changes the number of subtrees.

The *cons* operation maintains a second invariant: since the elements are distributed evenly among the subtrees, each subtree contains the same number of elements (denoted $|t|$). In other words, the multiway trees are perfectly *weight-balanced*.

> **Invariant 2:** for all nodes $\langle xs, \mathit{array}\ [t_0, \ldots, t_n]\rangle$:
>
> $$|t_0| = \cdots = |t_n|.$$

Invariant 1 implies that the array of subtrees is never empty. This raises the question, how we can effectively check whether a given tree is empty. We simply agree upon that a tree is empty if and only if the prefix is empty.

> **Invariant 3:** for all nodes $t = \langle xs, ts\rangle$:
>
> $$|t| = 0 \Longleftrightarrow \mathit{empty}\ xs.$$

This gives a particularly simple implementation of *empty*.

$$\mathit{empty}\ \langle xs, ts\rangle \quad = \quad \mathit{empty}\ xs$$

The definition of *size* builds upon all three invariants.

$$\begin{array}{lcl} \mathit{size}\ \langle xs, ts\rangle & & \\ \quad \mid \mathit{empty}\ xs & = & 0 \\ \quad \mid \mathit{otherwise} & = & \mathit{size}\ xs + \mathit{size}\ ts * \mathit{size}\ (\mathit{head}\ ts) \end{array}$$

Note that *size* takes time linear in the *height* of the tree.

Since a non-empty tree has a non-empty prefix, accessing the first element of an array is straightforward.

$$\begin{array}{lcl} \mathit{head}\ \langle xs, ts\rangle & & \\ \quad \mid \mathit{empty}\ xs & = & \mathit{error}\ \texttt{"head: empty array"} \\ \quad \mid \mathit{otherwise} & = & \mathit{head}\ xs \end{array}$$

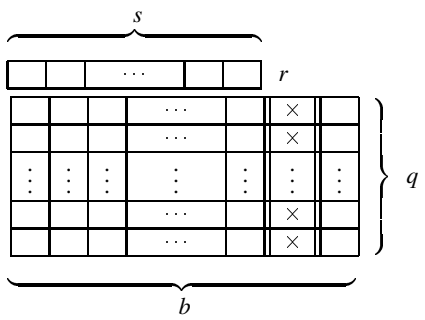

Figure 2. Indexing multiway trees.

The *tail* operation is essentially the inverse of *cons*.

$$
\begin{array}{lcl}
\mathit{tail}\ \langle \mathit{xs}, \mathit{ts}\rangle & & \\
\quad \mid \mathit{empty}\ \mathit{xs} & = & \mathit{error}\ \texttt{"tail: empty array"} \\
\quad \mid \mathit{size}\ \mathit{xs} > 1 & = & \langle \mathit{tail}\ \mathit{xs}, \mathit{ts}\rangle \\
\quad \mid \mathit{empty}\ (\mathit{head}\ \mathit{ts}) & = & \langle \mathit{nil}, \mathit{ts}\rangle \\
\quad \mid \mathit{otherwise} & = & \langle \mathit{head}^*\ \mathit{ts}, \mathit{tail}^*\ \mathit{ts}\rangle
\end{array}
$$

Note that we have to be careful to return an empty array if the argument array has size 1 (third equation).

3.2 Array-like operations

The list-like operations have been carefully crafted so that the array operations can be implemented efficiently. To see how indexing works consider the evolution of an initially empty array: After the first overflow, the r-th subtree contains a single element, namely the one at position $s+r$, where s is the size of the prefix. After the second overflow, it contains two elements, the ones at positions $s+r$ and $s+b+r$, where b is the total number of subtrees. In general, after q overflows it comprises the q elements at positions $s+0*b+r$, $s+1*b+r$, …, $s+(q-1)*b+r$. Fig. 2 illustrates the situation. The first row is the prefix of the array; each column of the matrix below corresponds to one subtree. Reading from left to right and from top to bottom we obtain the elements of the array ordered by index.

To determine the location of the i-th element in a given multiway tree, we first check whether the element is contained in the prefix. If this is not the case, then the remainder $\mathit{mod}\ (i-s)\ b$ determines the subtree where the element is to be found and the quotient $\mathit{div}\ (i-s)\ b$ determines the position within the subtree.

$$
\begin{array}{lcl}
\langle \mathit{xs}, \mathit{ts}\rangle\ !\ i & & \\
\quad \mid \mathit{empty}\ \mathit{xs} & = & \mathit{error}\ \texttt{"(!): index out of range"} \\
\quad \mid i < \mathit{size}\ \mathit{xs} & = & \mathit{xs}\ !\ i \\
\quad \mid \mathit{otherwise} & = & (\mathit{ts}\ !\ r)\ !\ q \\
\quad \textbf{where}\ (q, r) & = & \mathit{divMod}\ (i-\mathit{size}\ \mathit{xs})\ (\mathit{size}\ \mathit{ts})
\end{array}
$$

Note that different occurrences of '!' refer to different instances of the overloaded operation: in $(\mathit{ts}\ !\ r)\ !\ q$ the first occurrence operates on the base array while the second constitutes the recursive call.

The update operation is implemented analogously.

$$
\begin{array}{lcl}
\mathit{update}\ f\ i\ \langle \mathit{xs}, \mathit{ts}\rangle & & \\
\quad \mid \mathit{empty}\ \mathit{xs} & = & \mathit{error}\ \texttt{"update: index out of range"} \\
\quad \mid i < \mathit{size}\ \mathit{xs} & = & \langle \mathit{update}\ f\ i\ \mathit{xs}, \mathit{ts}\rangle \\
\quad \mid \mathit{otherwise} & = & \langle \mathit{xs}, \mathit{update}\ (\mathit{update}\ f\ q)\ r\ \mathit{ts}\rangle \\
\quad \textbf{where}\ (q, r) & = & \mathit{divMod}\ (i-\mathit{size}\ \mathit{xs})\ (\mathit{size}\ \mathit{ts})
\end{array}
$$

4 Array creation

The shape of multiway trees and consequently the running time of the operations is solely determined by the array creation functions *nil*, *copy*, and *array*. Conceptually, we may think of an initial array as an infinite tree, whose branching structure is fixed and which will be populated through repeated applications of the *cons* function. An initial array is very much like an egg cell in that it incorporates the blueprint for its future development.

That said, it becomes clear that there is no single collection of constructor functions. Rather, each possible tree shape gives rise to one collection. (Of course, for the instance declaration we have to commit ourselves to one particular, yet arbitrary collection of array creation functions.) Through suitable definitions the user can adjust the running time of the array operations to her needs—even dynamically at array-creation time.

As an aside, note that we do not employ lazy evaluation in an essential way—it is more a matter of convenience. The proposed data structure could be easily implemented in a strict language just by adding a suitable constructor for empty nodes. Of course, the empty constructor has to incorporate information about the 'future' branching structure.

To be able to analyze the running times reasonably well, we make one further assumption: we require that nodes of the same level have the same size. This is quite a reasonable requirement if the elements are accessed with equal probability. If the access characteristic is different, a less regular layout may be advantageous. Given this assumption the structure of trees can be described using a special number system, the so-called *mixed-radix system* [13]. A mixed-radix numeral is given by a sequence of digits d_0, d_1, d_2, … (determining the size of the element arrays) and a sequence of bases b_0, b_1, b_2, … (determining the size of the subtree arrays).

$$
\begin{bmatrix} d_0, d_1, d_2, \ldots \\ b_0, b_1, b_2, \ldots \end{bmatrix} = \sum_{i=0} d_i * w_i \ \textbf{where}\ w_i = b_{i-1} * \cdots * b_1 * b_0
$$

In our case, the bases are positive numbers $1 \leqslant b_i$ and we require the digits to lie in the range $0 \leqslant d_i \leqslant b_i$ (cf Invariant 1). Furthermore, we require $d_i = 0 \Longrightarrow d_{i+1} = 0$ (cf Invariant 3). In other words, if we ignore trailing zeros, then the digits must lie in the range $1 \leqslant d_i \leqslant b_i$. Perhaps surprisingly, each natural number has a unique representation for a fixed sequence of bases. Mixed-radix numerals satisfy the appealing recursion equation

$$
\begin{bmatrix} d_0, d_1, d_2, \ldots \\ b_0, b_1, b_2, \ldots \end{bmatrix} = d_0 + b_0 * \begin{bmatrix} d_1, d_2, \ldots \\ b_1, b_2, \ldots \end{bmatrix},
$$

which corresponds nicely to the definition of *size*. Likewise, *cons* corresponds to incrementing a mixed-radix numeral and *tail* to decrementing one. In Haskell, we can represent mixed-radix numerals by a list of pairs:

$$
\textbf{type}\ \mathit{Mix} \quad = \quad [(\mathit{Int}, \mathit{Int})].
$$

Given a list of bases we can quite easily convert a natural number into a mixed-radix number.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Bases} & = & [\mathit{Int}] \\
\mathit{encode} & :: & \mathit{Bases} \to (\mathit{Int} \to \mathit{Mix}) \\
\mathit{encode}\ (b : \mathit{bs})\ n & & \\
\quad \mid n == 0 & = & \mathit{zip}\ (\mathit{repeat}\ 0)\ (b : \mathit{bs}) \\
\quad \mid \mathit{otherwise} & = & (r+1, b) : \mathit{encode}\ \mathit{bs}\ q \\
\quad \textbf{where}\ (q, r) & = & \mathit{divMod}\ (n-1)\ b
\end{array}
$$

From a list of bases we can also construct an empty array.

$$\begin{array}{lcl} \mathit{gnil} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Bases} \to \mathit{Tree}\ a\ x \\ \mathit{gnil}\ (b : bs) & = & \langle \mathit{nil}, \mathit{copy}\ b\ (\mathit{gnil}\ bs)\rangle \end{array}$$

The function *gnil* can be seen as a *generic* array creation function that is indexed by a mixed-radix number system. In a similar vein we can define a generic *copy* function that creates an array of a certain size.

$$\begin{array}{lcl} \mathit{gcopy} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Mix} \to x \to \mathit{Tree}\ a\ x \\ \mathit{gcopy}\ ((d,b) : \sigma)\ x & = & \langle \mathit{copy}\ d\ x, \mathit{copy}\ b\ (\mathit{gcopy}\ \sigma\ x)\rangle \end{array}$$

The array creation functions for a fixed sequence of bases *bs* are then given by $\mathit{nil} = \mathit{gnil}\ \mathit{bs}$ and $\mathit{copy}\ n\ x = \mathit{gcopy}\ (\mathit{encode}\ \mathit{bs}\ n)\ x$ (the creation function *array* will be dealt with in Sec. 5.3).

Before we look at particular examples of mixed-radix systems, let us first study the running time of look-up and update operations in the general setting. To simplify the analysis, we pretend to work in a strict setting. Furthermore, we assume that the arrays are used in a single-threaded manner (Sec. 4.5 deals with persistent use). The dominant factor of array look-up is the height of the multiway tree. Let $\mathrm{H}(n)$ be the height of the *tallest* tree with size n (counting only non-empty nodes). The running time of '!' and *update* is

$$\begin{array}{lcl} \mathcal{T}_!(n) & = & \sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_!(b_i) \\ \mathcal{T}_{\mathit{update}}(n) & = & \sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_{\mathit{update}}(b_i), \end{array}$$

where $\bar{\mathcal{T}}_{op}$ is the running time of *op* on base arrays. Actually, both operations are slightly faster: the number of accesses is less than the height if the indexed element is located in a prefix high up the tree.

In many cases, the height function can be conveniently derived from the size function. Let $\mathrm{S}(h)$ be the size of the *smallest* tree with height h. The two functions are related by

$$\mathrm{S}(h) \leqslant n \Longleftrightarrow h \leqslant \mathrm{H}(n). \tag{1}$$

In math speak, $\mathrm{S}{:}\mathbb{N} \to \mathbb{N}$ and $\mathrm{H}{:}\mathbb{N} \to \mathbb{N}$ form a *Galois connection* between the orders $(\mathbb{N}, \leqslant)$ and $(\mathbb{N}, \leqslant)$. In the case of multiways trees, $\mathrm{S}(h)$ equals the mixed-radix numeral whose first h digits are ones.

$$\mathrm{S}(h) = \begin{bmatrix} 1, & \ldots, 1, & 0, & \ldots \\ b_0, \ldots, & b_{h-1}, & b_h, & \ldots \end{bmatrix} = \sum_{i=0}^{h-1} w_i.$$

Turning to the *cons* operation let us first note that its *worst-case* running time is proportional to the size of the tree: if we have a cascading carry as in

$$\begin{bmatrix} b_0, \ldots, b_{n-1}, 0, & \ldots \\ b_0, \ldots, b_{n-1}, b_n, & \ldots \end{bmatrix} + 1 = \begin{bmatrix} 1, & \ldots, 1, & 1, & 0, & \ldots \\ b_1, \ldots, & b_{n-1}, & b_n, & b_{n+1}, & \ldots \end{bmatrix},$$

then the whole tree must be rearranged. However, this worst-case only happens once in a while. We obtain a much better estimate of the running time if we conduct an *amortized analysis*. The amortized running-time of *cons* is given by

$$\begin{array}{lcl} \mathcal{T}_{\mathit{cons}}(n) & = & \frac{1}{n} \sum_{i=0}^{\mathrm{H}(n)-1} \frac{n}{w_i}\, w_i\, \bar{\mathcal{T}}_{\mathit{cons}}(b_i) \\ & = & \sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_{\mathit{cons}}(b_i). \end{array}$$

The formula on the first line can be understood as follows: the sum calculates the costs of n successive *cons* operations. If we divide the result by n, we obtain the amortized running-time. Each summand describes the total costs at level i: the i-th level is touched every n/w_i steps; if it is touched, then w_i nodes must be rearranged; and the rearrangement of one node takes $\bar{\mathcal{T}}_{\mathit{cons}}(b_i)$ time in the worst-case. Perhaps surprisingly, the amortized running time of *cons* is given by the same formula as the worst-case running time of look-up and update. Note, however, that the analysis assumes that there are no interfering *tail* operations. Otherwise, cascading carries or borrows may occur in every step. Sec. 4.5 deals with the general case.

In the sequel, we study three instances of mixed-radix systems. For each instance, we will analyze the asymptotic growth of the sum $\sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_{op}(b_i)$ for different choices of $\bar{\mathcal{T}}_{op}$. Keep in mind that this sum captures the *worst-case* running time of '!' and *update*, but the *amortized* running time of *cons*.

4.1 b-ary trees

The simplest case of a mixed-radix number system is the *positional system* where the radix is the same for all positions.

$$\begin{bmatrix} d_0, d_1, d_2, \ldots, d_n, \ldots \\ b, \ b, \ b, \ \ldots, b, \ \ldots \end{bmatrix}$$

It is worth noting, however, that this number system is still somewhat unusual as we disallow the digit zero except in 'rightmost' positions—we have a so-called *zeroless* representation [16]. As an example, in radix 2 the decimal number 27 is 11220… rather than 110110….

Since the radix is fixed, the bootstrapped arrays are just b-ary trees. The array creation functions are given by

$$\begin{array}{lcl} \mathit{bary} & :: & \mathit{Int} \to \mathit{Bases} \\ \mathit{bary}\ b & = & \mathit{repeat}\ b \\ \mathit{nil}_1\ b & = & \mathit{gnil}\ (\mathit{bary}\ b) \\ \mathit{copy}_1\ b\ n\ x & = & \mathit{gcopy}\ (\mathit{encode}\ (\mathit{bary}\ b)\ n)\ x. \end{array}$$

Actually, in this simple case the definition of nil_1 can be slightly improved. The following implementation uses only constant space.

$$\mathit{nil}_1\ b \quad = \quad t\ \textbf{where}\ t = \langle \mathit{nil}, \mathit{copy}\ b\ t\rangle$$

Turning to the performance, the size of the smallest tree of height h is given by the sum of the geometric progression: $\sum_{i=0}^{n-1} b^i = (b^n - 1)/(b-1)$. The height function can be easily calculated from S using relation (1).

$$\begin{array}{lcl} \mathrm{S}(h) & = & (b^h - 1)/(b-1) \\ \mathrm{H}(n) & = & \lfloor \log_b(n(b-1)+1) \rfloor \end{array}$$

For the running time of the bootstrapped operations we calculate

$$\begin{array}{lcl} \mathcal{T}_{op}(n) & = & \sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_{op}(b_i) \\ & = & \lfloor \log_b(n(b-1)+1) \rfloor * \bar{\mathcal{T}}_{op}(b) \\ & \approx & \lg n * \bar{\mathcal{T}}_{op}(b) / \lg b, \end{array}$$

where lg is the binary logarithm. As an example, assume that the base operation takes logarithmic time $\bar{\mathcal{T}}_{op}(n) = c * \lg n$. In this case, the running time of the bootstrapped operation, $\mathcal{T}_{op}(n) = c * \lg n$, is

exactly the same. The formula above shows that the dependence of the bootstrapped operation on the base operation is linear: if we improve the running time of the base operation by a constant factor, then the bootstrapped operation speeds up by exactly the same factor. It is instructive to take a look at some concrete figures. Say, we want to represent an array of size $2^{28} = 268,435,456$ elements.[1] The following table displays $\mathcal{T}_{op}(2^{28})$ for different choices of b and $\bar{\mathcal{T}}_{op}$.

b	2	4	16	256	1024
$\bar{\mathcal{T}}_{op}(n) = 1$	28	14	7	4	3
$\bar{\mathcal{T}}_{op}(n) = \lg n$	29	29	29	29	29
$\bar{\mathcal{T}}_{op}(n) = n$	29	43	106	781	2302

If we bootstrap from a real array and if we use radix-256 trees, then we need at most 4 steps to access an arbitrary element. Updating an element is considerably more expensive: roughly 800 steps are required on an average.

Of course, if we abstract away from multiplicative constants, we obtain logarithmic asymptotic running times irrespective of the underlying base array.

base array	bootstrapped array
$\Theta(1)$	$\Theta(\log n)$
$\Theta(\log n)$	$\Theta(\log n)$
$\Theta(n)$	$\Theta(\log n)$

4.2 Arithmetic progression trees

Can we improve upon the logarithmic worst-case complexity? The answer is an emphatic "Yes!". The idea is to steadily increase the size of nodes as we move down a tree: the root node has α descendants, the nodes on the second level have $\alpha+\beta$ descendants, the nodes on the third level $\alpha+2\beta$ and so forth.

$$\begin{bmatrix} d_0, d_1, & d_2, & \ldots, d_n, & \ldots \\ \alpha, & \alpha+\beta, \alpha+2\beta, \ldots, \alpha+n\beta, & \ldots \end{bmatrix}$$

Since the radices form an arithmetic progression, we call the corresponding trees *arithmetic progression trees*. The array creation functions enjoy straightforward definitions.

$$\begin{array}{lcl} \mathit{arithmetic} & :: & \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\ \mathit{arithmetic}\ \alpha\ \beta & = & \alpha : \mathit{arithmetic}\ (\alpha+\beta)\ \beta \\ \mathit{nil}_2\ \alpha\ \beta & = & \mathit{gnil}\ (\mathit{arithmetic}\ \alpha\ \beta) \\ \mathit{copy}_2\ \alpha\ \beta\ n\ x & = & \mathit{gcopy}\ (\mathit{encode}\ (\mathit{arithmetic}\ \alpha\ \beta)\ n)\ x \end{array}$$

For the asymptotic analysis let us consider one particular instance of arithmetic progression trees fixing $\alpha = \beta = 1$. For this choice, the mixed-radix number system specializes to a variant of the so-called *factorial number system* (the standard factorial number system requires the digits to lie in the range $0 \leqslant d_i < b_i = i+1$).

$$\begin{bmatrix} d_0, d_1, d_2, d_3, d_4, \ldots \\ 1, \ 2, \ 3, \ 4, \ 5, \ \ldots \end{bmatrix}$$

The decimal number 271965, for instance, is represented by $123214650\ldots$. The size of the smallest tree is given by the so-called *left factorial* function $!n = \sum_{k=0}^{n-1} k!$. The height function is roughly the inverse of the left factorial function denoted ¡n.

$$\mathrm{S}(h) \quad = \quad !h$$

$$¡n - 1 \leqslant \mathrm{H}(n) \leqslant ¡n$$

Let $n¡$ be the inverse of the factorial function. Note that $n¡ = \Theta(\log n / \log\log n)$. Since $(h-1)! \leqslant !h \leqslant h!$ and consequently $n¡ \leqslant ¡n \leqslant (n+1)¡$, we have

$$\mathrm{H}(n) \quad = \quad \Theta(\log n / \log\log n).$$

Using the summation formulas $\sum_{i=0}^{n} \log i = \Theta(n \log n)$ and $\sum_{i=0}^{n} i = \Theta(n^2)$ we can estimate the running time of the bootstrapped operations.

base array	bootstrapped array
$\Theta(1)$	$\Theta(\log n / \log\log n)$
$\Theta(\log n)$	$\Theta(\log n)$
$\Theta(n)$	$\Theta((\log n)^2 / (\log\log n)^2)$

Since $\log n / \log\log n$ grows more slowly than $\log n$, we have beaten the logarithmic look-up time of 'traditional' tree-based implementations. Figure 3 inserts the functions above into the asymptotic hierarchy. Alas, the function $\log\log n$ grows very, very slow, so, in practice, the speed-up boils down to constant factor.

4.3 Geometric progression trees

We have seen in the previous section that array look-up speeds up if we steadily increase the size of nodes. Now, instead of enlarging the nodes by a constant amount, we can alternatively enlarge them by a constant *factor*: the root node has α descendants, the nodes on the second level have $\alpha\beta$ descendants, the nodes on the third level $\alpha\beta^2$ and so forth.

$$\begin{bmatrix} d_0, d_1, \ d_2, & \ldots d_n, & \ldots \\ \alpha, \ \alpha\beta, \alpha\beta^2, \ldots \alpha\beta^n, & \ldots \end{bmatrix}$$

The radices now form the elements of a geometric progression. Accordingly, the corresponding trees are called *geometric progression trees*. Here are the array creation functions.

$$\begin{array}{lcl} \mathit{geometric} & :: & \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\ \mathit{geometric}\ \alpha\ \beta & = & \alpha : \mathit{geometric}\ (\alpha * \beta)\ \beta \\ \mathit{nil}_3\ \alpha\ \beta & = & \mathit{gnil}\ (\mathit{geometric}\ \alpha\ \beta) \\ \mathit{copy}_3\ \alpha\ \beta\ n\ x & = & \mathit{gcopy}\ (\mathit{encode}\ (\mathit{geometric}\ \alpha\ \beta)\ n)\ x \end{array}$$

Turning to the asymptotic analysis we will again consider one particular instance fixing $\alpha = 1$ and $\beta = 2$. Since $w_i = 2^{i(i-1)/2}$, the size function is

$$\mathrm{S}(h) \quad = \quad \sum_{i=0}^{n-1} 2^{i(i-1)/2}.$$

The size of a geometric progression tree is dominated by the size of the nodes on the lowest level. We have

$$2^{(h-1)(h-2)/2} \leqslant \mathrm{S}(h) \leqslant 2 * 2^{(h-1)(h-2)/2}.$$

A little calculation yields the following estimation.

$$\sqrt{2\lg n} - 1 \leqslant \mathrm{H}(h) \leqslant \sqrt{2\lg n} + 2. \qquad (2)$$

Consequently,

$$\mathrm{H}(h) \quad = \quad \Theta(\sqrt{\log n}).$$

[1] If we assume that every element fits into 64 bits, then a conventional array of that size would require 1GB of main memory.

$$\log\log n \prec \sqrt{\log n} \prec \log n/\log\log n \prec \log n \prec (\log n)^2/(\log\log n)^2 \prec 2^{\sqrt{\log n}} \prec n$$

Figure 3. Logarithmico-exponential functions ranked by order of growth.

Using (2) we can furthermore determine the running time of the bootstrapped operations.

base array	bootstrapped array
$\Theta(1)$	$\Theta(\sqrt{\log n})$
$\Theta(\log n)$	$\Theta(\log n)$
$\Theta(n)$	$\Theta(2^{\sqrt{\log n}})$

As to be expected, geometric progression trees are more extreme than arithmetic progression trees: look-up is considerably faster but update is also considerably slower (see also Figure 3).

4.4 Fat nodes

Using mixed-radix number systems we can nicely steer the out-degree of nodes, that is, the size of the subtree arrays. However, the size of the element arrays escapes our control, since the size is determined by the overall number of elements. In the worst case the prefixes are singletons, whereas in the best case they are as large as the subtree arrays. We use the terms 'worst case' and 'best case' because large prefixes are generally preferable as they improve both *locality* and *space usage*.

As an example, consider representing an array of 1099 elements using geometric progression trees ($\alpha = 1$ and $\beta = 2$). The tree corresponding to

$$\begin{bmatrix} 1,2,4,8,16,0, & 0, & \ldots \\ 1,2,4,8,16,32,64,\ldots \end{bmatrix}$$

consumes 1174 cells (simply summing up the array sizes). Now, if we add one element, we obtain

$$\begin{bmatrix} 1,1,1,1,1, & 1, & 0, & \ldots \\ 1,2,4,8,16,32,64,\ldots \end{bmatrix},$$

which consumes 2199 cells (87% more). Furthermore, adjacent array elements are always located in different parts of the tree. Note in this respect, that indexing works in little-endian order (from least to most significant bits).

Now, to ensure that the prefixes are reasonably large, we can simply use larger digits. Currently, we require the significant digits to lie in the range $1 \leqslant d_i \leqslant b_i$. We generalize this condition by shifting the range to

$$\delta * b_i + 1 \leqslant d_i \leqslant (\delta+1) * b_i$$

for some fixed natural number δ. Of course, to be able to represent all natural numbers we allow the rightmost significant digit to be smaller than $\delta * b_i + 1$.

The following function converts a natural number into the new number system given some δ and a list of bases.

$$\begin{array}{lcl} \mathit{encode} & :: & \mathit{Int} \to \mathit{Bases} \to (\mathit{Int} \to \mathit{Mix}) \\ \mathit{encode}\ \delta\ (b:bs)\ n & & \\ \quad \mid n < d_{\mathit{min}} & = & (n,b) : \mathit{zip}\ (\mathit{repeat}\ 0)\ bs \\ \quad \mid \mathit{otherwise} & = & (r + d_{\mathit{min}}, b) : \mathit{encode}\ \delta\ bs\ q \\ \quad \textbf{where}\ d_{\mathit{min}} & = & \delta * b + 1 \\ \qquad (q,r) & = & \mathit{divMod}\ (n - d_{\mathit{min}})\ b \end{array}$$

As an example, fixing $\delta = 16$ we have ($1099 + 1 = 1100$)

$$\begin{bmatrix} 17,34,68,114,0, & \ldots \\ 1, \ 2, \ 4, \ 8, \quad 16,\ldots \end{bmatrix} + 1 = \begin{bmatrix} 17,33,65,115,0, & \ldots \\ 1, \ 2, \ 4, \ 8, \quad 16,\ldots \end{bmatrix},$$

Though there is a cascading carry, the second tree requires only one additional memory cell (1110 versus 1111 cells).

On the downside, since the prefixes are larger, *cons*ing slows down by a constant factor. The asymptotic running times are not affected, though. Note that the list operations must be slightly adapted to work with the new number system (the array operations work without change). The details are left to the reader.

4.5 Redundant and lazy number systems

Recall that the amortized analysis of *cons* assumed that there were no interleaving *cons* and *tail* operations. If both operations are allowed, then *cons*'s running time degrades to $\Theta(n)$. Consider, for instance, an array that cycles between

$$\begin{bmatrix} b_0,\ldots,b_{n-1},0, & \ldots \\ b_0,\ldots,b_{n-1},b_n,\ldots \end{bmatrix} \text{ and } \begin{bmatrix} 1, & \ldots,1, & 1, & 0, & \ldots \\ b_1,\ldots,b_{n-1},b_n,b_{n+1},\ldots \end{bmatrix}.$$

This problem is typical of non-redundant number systems, where each number has a unique representation. The cure is to switch to a *redundant number system*, for instance, by using digits in the range $1 \leqslant d_i \leqslant b_i + 1$.

A similar performance problem shows up if the arrays are used persistently. Consider an array of size

$$\begin{bmatrix} b_0,\ldots,b_{n-1},0, & \ldots \\ b_0,\ldots,b_{n-1},b_n,\ldots \end{bmatrix}$$

and suppose that we repeatedly *cons* elements to this array. Again, *cons* takes time linear in the size of the array. In this case, *lazy evaluation* saves the day. If we switch to a lazy setting (or add explicit delays to the definitions), then the calculated time bounds hold regardless of whether the arrays are used persistently. For a more in-depth treatment of amortization, persistence and lazy evaluation the interested reader is referred to Okasaki's excellent textbook [16].

5 Mapping and conversion functions

5.1 Mapping functions

The implementation of *map* and *zip* is entirely straightforward.

$$\begin{array}{lcl} \mathit{map}\ f\ \langle xs, ts\rangle & = & \langle f^*\ xs, (\mathit{map}\ f)^*\ ts\rangle \\ \mathit{zip}\ f\ \langle xs_1, ts_1\rangle\ \langle xs_2, ts_2\rangle & = & \langle f^*\ xs_1\ xs_2, (\mathit{zip}\ f)^*\ ts_1\ ts_2\rangle \end{array}$$

Note that *zip* expects its arguments to have the same size.

5.2 Array destruction

The array conversion function *list* makes use of the following destructor functions.

$$\begin{array}{lcl} \mathit{elems} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \to [x] \\ \mathit{elems}\ \langle \mathit{xs}, \mathit{ts} \rangle & = & \mathit{list}\ \mathit{xs} \\ \mathit{subs} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \to [\mathit{Tree}\ a\ x] \\ \mathit{subs}\ \langle \mathit{xs}, \mathit{ts} \rangle & = & \mathit{list}\ \mathit{ts} \end{array}$$

Recursive solution

Here is the vanilla implementation of *list*: the subtrees are flattened recursively; the lists thus obtained are riffled (cf Fig. 2) and the result is appended to the prefix.

$$\begin{array}{lcl} \mathit{list} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \to [x] \\ \mathit{list}\ t & & \\ \quad \mid \mathit{empty}\ t & = & [\,] \\ \quad \mid \mathit{otherwise} & = & \mathit{elems}\ t +\!\!+ \mathit{riffle}\ (\mathit{list}^*\ (\mathit{subs}\ t)) \end{array}$$

The auxiliary function *riffle* merges n lists of length m into a single list of length mn.

$$\begin{array}{lcl} \mathit{riffle} & :: & [\,[x]\,] \to [x] \\ \mathit{riffle}\ x & & \\ \quad \mid \mathit{all}\ \mathit{empty}\ x & = & [\,] \\ \quad \mid \mathit{otherwise} & = & \mathit{head}^*\ x +\!\!+ \mathit{riffle}\ (\mathit{tail}^*\ x) \end{array}$$

Note that $\mathit{riffle} = \mathit{concat} \cdot \mathit{transpose}$.

The recursive implementation of *list* takes *super-linear* time in general. As an example, listifying b-ary trees has a running time of $\Theta(n \log n)$.

Iterative solution

If we know that the nodes of one level have the same size (so that there is an underlying number system), then *list* can be made to run in linear time. The principle idea is to transform the given tree levelwise by working on a list of subtrees.

$$\begin{array}{lcl} \mathit{list} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \to [x] \\ \mathit{list}\ t & = & \mathit{ilist}\ [t] \\ \mathit{ilist} & :: & (\mathit{Array}\ a) \Rightarrow [\mathit{Tree}\ a\ x] \to [x] \\ \mathit{ilist}\ \mathit{ts} & & \\ \quad \mid \mathit{empty}\ (\mathit{head}\ \mathit{ts}) & = & [\,] \\ \quad \mid \mathit{otherwise} & = & \mathit{riffle}\ (\mathit{elems}^*\ \mathit{ts}) \\ & +\!\!+ & \mathit{ilist}\ (\mathit{riffle}\ (\mathit{subs}^*\ \mathit{ts})) \end{array}$$

Note that the subtrees are riffled before they are passed to the recursive call. It is not at all clear why and how this scheme works. Fortunately, one can derive the iterative implementation from the recursive one. However, since the derivation proceeds in a point-free style, the calculations are relegated to an appendix (see App. B).

5.3 Array construction

The smart constructor *node* needed below is the inverse of the destructor functions *elems* and *subs*.

$$\begin{array}{lcl} \mathit{node} & :: & (\mathit{Array}\ a) \Rightarrow [x] \to [\mathit{Tree}\ a\ x] \to \mathit{Tree}\ a\ x \\ \mathit{node}\ \mathit{xs}\ \mathit{ts} & = & \langle \mathit{array}\ \mathit{xs}, \mathit{array}\ \mathit{ts} \rangle \end{array}$$

Recursive solution

As for the other array creation functions we define a *generic* version of *array* that is parametric in the underlying number system. The generic version takes as an additional argument the size of the input list represented as a mixed-radix numeral.

$$\begin{array}{lcl} \mathit{garray} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Mix} \to [x] \to \mathit{Tree}\ a\ x \\ \mathit{garray}\ ((d,b) : \sigma)\ \mathit{xs} & & \\ \quad \mid d == 0 & = & \mathit{gnil}\ (b : \mathit{snd}^*\ \sigma) \\ \quad \mid \mathit{otherwise} & = & \mathit{node}\ \mathit{xs}_1\ ((\mathit{garray}\ \sigma)^*\ (\mathit{unriffle}\ b\ \mathit{xs}_2)) \\ \quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) & = & \mathit{splitAt}\ d\ \mathit{xs} \end{array}$$

The helper function *unriffle* n splits a list of length mn into n lists of length m. In a sense which is made precise in App. B *unriffle* is the inverse of *riffle*.

$$\begin{array}{lcl} \mathit{unriffle} & :: & \mathit{Int} \to [x] \to [\,[x]\,] \\ \mathit{unriffle}\ n\ \mathit{xs} & & \\ \quad \mid \mathit{empty}\ \mathit{xs} & = & \mathit{copy}\ n\ [\,] \\ \quad \mid \mathit{otherwise} & = & \mathit{cons}^*\ \mathit{xs}_1\ (\mathit{unriffle}\ n\ \mathit{xs}_2) \\ \quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) & = & \mathit{splitAt}\ n\ \mathit{xs} \end{array}$$

Iterative solution

Again, we can achieve a linear time behaviour if we turn the recursive solution into an iterative one. The following variant of *garray* builds the tree levelwise.

$$\begin{array}{lcl} \mathit{garray} & :: & (\mathit{Array}\ a) \Rightarrow \mathit{Mix} \to [x] \to \mathit{Tree}\ a\ x \\ \mathit{garray}\ \sigma\ \mathit{xs} & = & \mathit{head}\ (\mathit{iarray}\ 1\ \sigma\ \mathit{xs}) \end{array}$$

$$\begin{array}{l} \mathit{iarray} :: (\mathit{Array}\ a) \Rightarrow \mathit{Int} \to \mathit{Mix} \to [x] \to [\mathit{Tree}\ a\ x] \\ \mathit{iarray}\ w\ ((d,b) : \sigma)\ \mathit{xs} \\ \quad \mid d == 0 \quad = \quad \mathit{copy}\ w\ (\mathit{gnil}\ (b : \mathit{snd}^*\ \sigma)) \\ \quad \mid \mathit{otherwise} \quad = \quad \mathit{node}^*\ (\mathit{unriffle}\ w\ \mathit{xs}_1) \\ \qquad\qquad\qquad\qquad\quad (\mathit{unriffle}\ w\ (\mathit{iarray}\ (w * b)\ \sigma\ \mathit{xs}_2)) \\ \quad \textbf{where}\ (\mathit{xs}_1, \mathit{xs}_2) \quad = \quad \mathit{splitAt}\ (w * d)\ \mathit{xs} \end{array}$$

Note that *unriffle* is applied only 'locally' before the trees of one level are constructed. Furthermore, note that the iterative version also improves *sharing*: the empty tree $\mathit{gnil}\ (b : \mathit{snd}^*\ \sigma)$ is constructed only once being shared among all nodes on the lowest level. That said it becomes clear that there is further room for improvement. Observe that the nodes on the lowest level contain identical subtree arrays, each of which consists of b copies of the empty tree. These subtree arrays can be shared, as well. The relevant laws for improving *iarray* are

$$\begin{array}{lcl} \mathit{unriffle}\ m\ (\mathit{copy}\ (m * n)\ x) & = & \mathit{copy}\ m\ (\mathit{copy}\ n\ x) \\ f^*\ (\mathit{copy}\ n\ x) & = & \mathit{copy}\ n\ (f\ x). \end{array}$$

The details of the modification are left to the reader.

6 Benchmarks

This section presents preliminary measurements comparing various instances of multiway trees to Haskell 98 arrays, *Int*-indexed arrays (starting at zero), lists, Braun trees [11], and skew binary random-access lists [15, 16]. The programs were compiled with Version 5.04 of the Glasgow Haskell Compiler (`-O2`); the executables were run on a Pentium III (645 MHz) with 192 MB of main memory. All multiway tree implementations are bootstrapped from *Int*-indexed arrays using Haskell's type class mechanism. Despite appearance, the parameters of the different variants (b, α, β, and δ) were chosen at will. We plan to conduct more systematic measurements in the future.

$n =$	1e3	5e3	1e4	5e4	1e5	5e5	1e6
standard array	0.05	0.37	1.38	9.25	18.95	105.08	230.48
Int-indexed array	0.05	0.27	1.29	8.00	16.08	83.57	170.98
list	1.01	23.52	121.12	–	–	–	–
Braun tree	0.24	1.65	4.33	26.85	56.99	324.02	685.17
random-access list	0.18	1.29	3.42	21.47	46.56	263.14	567.01
b-ary ($b = 28$, $\delta = 0$)	0.11	0.64	2.00	16.44	33.81	177.31	478.79
b-ary ($b = 256$, $\delta = 0$)	0.07	0.49	1.65	9.59	27.71	151.63	330.90
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 0$)	0.11	0.63	2.02	15.35	33.18	177.82	362.67
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 0$)	0.10	0.62	1.89	16.23	33.24	161.52	354.16
b-ary ($b = 28$, $\delta = 32$)	0.05	0.46	1.58	10.53	21.93	128.81	300.08
b-ary ($b = 256$, $\delta = 32$)	0.05	0.33	1.52	9.44	19.55	114.95	245.46
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 32$)	0.05	0.46	1.61	10.89	22.28	118.58	245.47
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 32$)	0.07	0.46	1.76	10.85	22.15	130.95	272.22

Figure 4. Repeated random indexing ($100 * n$).

The benchmarks are so-called *micro-benchmarks*: a certain operation or a certain sequence of operations is repeated a number of times. In the first test, an array of size n is created and subsequently indexed $100 * n$ times in a random fashion. The results of this benchmark are displayed in Fig. 4. As to be expected, standard arrays are the data structure of choice when only look-up is used. Perhaps surprisingly, however, arrays are only three times faster than skew binary random-access lists, which perform amazingly well. Braun trees are slightly slower as they exhibit poor locality, see [15]. The multiway tree implementations show the expected behaviour: the larger the nodes, the better the running time. In particular, trees with fat nodes (cf Sec. 4.4) perform very well—they consistently outperform random-access lists by a factor of two.

The second and the third benchmark feature a combination of random indexing and consing: starting with an array of a given size we repeatedly perform a *cons* operation followed by 10 or 100 look-ups, respectively. The results are displayed in Fig. 5 and 6. Random-access lists are superior up to an array size of $100,000$ elements. For larger arrays arithmetic or geometric progression trees are preferable. Braun trees are competitive for larger trees, as well (though this may be an artifact of lazy evaluation as not every element is accessed). Fig. 7 and 8 display the result of the fourth and the fifth benchmark, which test the combination of random indexing and updating. The results are similar to the two previous benchmarks (perhaps slightly more in favour of bootstrapped arrays).

Of course, the measurements are far from being conclusive. There is some indication that skew binary random-access lists are the data structure of choice if little is known about the requirements of a particular application. If the arrays are reasonably large and if updates are less frequent than look-ups, then multiway trees have something to offer—in particular, as they can be adapted to the needs of an application. As a rule of thumb, the size of the nodes or the rate of growth should be chosen according to the look-up/update ratio.

7 Related work

The data structure of multiway trees generalizes *fork-node trees* introduced by Hinze [9] and discovered independently by Xi (private communication). Fork-node trees correspond to the simplest instance of multiway trees, namely 2-ary trees. The idea of trading the running time of look-up for the running time of update is due to Okasaki (private communication). Okasaki proposed to generalize binary tries [17] to n-ary tries. Both data structures, however, do not support list-like operations.

Virtually all implementations of flexible arrays are based on tree structures. The first design due to Braun and Rem [2] uses binary, node-oriented trees featuring a rigid balancing scheme: for any given node, the left subtree has size $\lceil (n-1)/2 \rceil$ and the right subtree has size $\lfloor (n-1)/2 \rfloor$. Braun trees support look-up and update in $\Theta(\log i)$ time and *cons* and *tail* in $\Theta(\log n)$ time. Skew binary random-access lists [15] meet the time bound for array operations (albeit in the expected case) and allow the list operations to be implemented in $\Theta(1)$ time. Random-access lists are sequences of *complete* binary, node-oriented trees; they are modelled after the skew-binary number system. Several variants of this data structure with different underlying number systems can be found in the textbook [16]. Random-access lists based on binary numbers correspond to the leaf trees of Dielissen and Kaldewaij [3, 4], which are leftist left-perfect trees (the offsprings of the nodes on the right spine form a sequence of complete leaf trees). A detailed comparison of these tree structures can be found in [9].

Number systems serve admirably as templates for data structures. Various examples of this cross-fertilization can be found in the literature [21, 6, 16, 8]. To the best of the author's knowledge the use of mixed-radix number systems is original.

Radically different implementations are available for *rigid* arrays that cannot grow or shrink. *Version tree arrays* [10, 1] are a blend of association lists and 'real' arrays: the initial version is represented by a mutable array; update operations build a difference list in front of this array. Thus, updates take $\Theta(1)$ time, but look-ups degrade to $\Theta(u)$ time where u is the number of updates between the original version and the most recent one. By dynamically restructuring the trailers look-up and update can be improved to $\Theta(1)$ for the special case of *single-threaded access*.

A different approach that provides better support for *persistent* use represents an array by a pair consisting of a unique version stamp and a pointer to a master array, whose elements are finite maps from version stamps to associated values. The implementation by O'Neill and Burton [18] uses splay trees [20] for the finite maps and a version stamp scheme based on the *list order problem* [5]. Their implementation exhibits $\Theta(1)$ bounds for single-threaded use and $\Theta(\log n)$ amortized bounds for persistent use.

A common characteristic of the two approaches above is the internal use of destructive updates. Purely functional implementations of rigid arrays include AVL trees [14] and binary tries [17]. In fact, since an array can be seen as a finite map from indices to values, every implementation of *finite maps* will do. Similarly, one may

$n =$	1e3	5e3	1e4	5e4	1e5	5e5	1e6
standard array	5.31	26.56	–	–	–	–	–
Int-indexed array	0.07	0.25	0.55	4.51	13.56	69.65	137.37
list	0.13	0.54	2.59	24.24	–	–	–
Braun tree	0.04	0.09	0.14	0.56	1.00	4.21	9.13
random-access list	0.02	0.04	0.05	0.14	0.30	3.89	13.94
b-ary ($b = 28$, $\delta = 0$)	0.03	0.07	0.10	0.59	0.96	3.86	11.00
b-ary ($b = 256$, $\delta = 0$)	0.03	0.06	0.10	0.34	1.35	4.59	10.50
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 0$)	0.03	0.08	0.10	0.69	0.88	3.15	8.31
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 0$)	0.03	0.07	0.10	0.76	1.08	3.36	6.30
b-ary ($b = 28$, $\delta = 32$)	0.06	0.12	0.16	0.51	0.83	3.81	9.24
b-ary ($b = 256$, $\delta = 32$)	0.08	0.29	0.51	0.86	1.29	5.11	11.26
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 32$)	0.05	0.11	0.15	0.71	1.10	3.85	7.84
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 32$)	0.04	0.11	0.17	0.51	0.92	4.71	10.77

Figure 5. Random indexing and consing (one *cons* followed by 10 look-ups repeated 1000 times).

$n =$	1e3	5e3	1e4	5e4	1e5	5e5	1e6
standard array	4.97	434.74	–	–	–	–	–
Int-indexed array	0.18	0.39	0.68	4.68	13.75	69.61	137.16
list	1.11	4.91	15.37	302.75	–	–	–
Braun tree	0.32	0.46	0.58	1.21	2.00	5.12	11.06
random-access list	0.24	0.32	0.36	0.70	0.89	5.16	14.60
b-ary ($b = 28$, $\delta = 0$)	0.20	0.26	0.31	1.07	1.60	4.84	15.03
b-ary ($b = 256$, $\delta = 0$)	0.16	0.23	0.27	0.69	1.76	5.26	9.71
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 0$)	0.20	0.27	0.31	1.04	1.79	4.17	9.71
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 0$)	0.20	0.27	0.31	1.34	1.75	4.89	7.31
b-ary ($b = 28$, $\delta = 32$)	0.18	0.27	0.33	0.84	1.29	5.12	8.84
b-ary ($b = 256$, $\delta = 32$)	0.19	0.44	0.66	1.00	1.50	6.27	13.60
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 32$)	0.17	0.28	0.33	1.06	1.69	4.54	9.08
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 32$)	0.18	0.27	0.36	0.82	1.25	5.56	11.33

Figure 6. Random indexing and consing (one *cons* followed by 100 look-ups repeated 1000 times).

$n =$	1e3	5e3	1e4	5e4	1e5	5e5	1e6
standard array	2.64	15.76	–	–	–	–	–
Int-indexed array	0.05	0.20	0.43	4.20	13.13	67.94	133.36
list	0.26	1.72	5.23	51.70	–	–	–
Braun tree	0.04	0.10	0.15	0.55	1.03	4.05	8.44
random-access list	0.04	0.06	0.07	0.17	0.33	4.55	14.13
b-ary ($b = 28$, $\delta = 0$)	0.03	0.07	0.13	0.61	0.95	3.41	12.68
b-ary ($b = 256$, $\delta = 0$)	0.03	0.08	0.11	0.35	1.41	4.36	11.05
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 0$)	0.04	0.08	0.12	0.70	0.93	3.02	8.37
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 0$)	0.04	0.08	0.11	0.80	1.16	4.18	6.71
b-ary ($b = 28$, $\delta = 32$)	0.05	0.08	0.11	0.45	0.85	3.56	7.74
b-ary ($b = 256$, $\delta = 32$)	0.05	0.24	0.40	0.53	1.02	4.86	10.17
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 32$)	0.05	0.08	0.11	0.51	0.85	4.02	7.15
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 32$)	0.03	0.10	0.14	0.43	0.88	4.47	11.30

Figure 7. Random indexing and updating (one update followed by 10 look-ups repeated 1000 times).

$n =$	1e3	5e3	1e4	5e4	1e5	5e5	1e6
standard array	2.24	15.17	–	–	–	–	–
Int-indexed array	0.15	0.33	0.58	4.44	13.39	67.80	133.58
list	1.40	7.16	19.09	229.30	–	–	–
Braun tree	0.32	0.47	0.57	1.08	1.70	5.18	9.53
random-access list	0.27	0.36	0.39	0.68	1.03	5.18	16.72
b-ary ($b = 28$, $\delta = 0$)	0.21	0.27	0.31	1.03	1.52	4.94	13.62
b-ary ($b = 256$, $\delta = 0$)	0.18	0.23	0.27	0.69	1.81	5.28	13.37
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 0$)	0.21	0.28	0.31	1.22	1.77	4.17	8.01
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 0$)	0.21	0.26	0.30	1.27	1.72	4.85	9.14
b-ary ($b = 28$, $\delta = 32$)	0.16	0.23	0.28	0.73	1.12	4.21	9.45
b-ary ($b = 256$, $\delta = 32$)	0.15	0.38	0.59	0.82	1.17	5.44	10.78
arithmetic ($\alpha = 28$, $\beta = 4$, $\delta = 32$)	0.16	0.24	0.28	0.82	1.21	3.83	7.31
geometric ($\alpha = 6$, $\beta = 6$, $\delta = 32$)	0.17	0.25	0.32	0.73	1.19	5.42	12.03

Figure 8. Random indexing and updating (one update followed by 100 look-ups repeated 1000 times).

adapt an implementation of *ordered lists*. Instances based on finger search trees [12, 7], for example, support *cons* in $\Theta(1)$ time and allow to access or update the i-th element in $\Theta(\log i)$ time.

8 Conclusion and future work

We have presented a purely functional implementation of one-sided flexible arrays based on weight-balanced multiway trees. The data structure is simple to implement and fully persistent. Multiway trees are bootstrapped from an existing implementation of arrays. The performance of the underlying base array type and the branching structure of the multiway trees determine the overall performance. Both parameters can be chosen dynamically at array-creation time. We have shown that a large class of tree shapes can be conveniently modelled after mixed-radix number systems and we have analysed three obvious choices (b-ary trees, arithmetic progression trees, and geometric progression trees) in detail. Preliminary measurements show that multiway trees perform reasonably well in practice.

The work described here is the end-product of several abstractions generalizing fork-node trees [9] to b-ary trees and b-ary trees to multiways trees based on mixed-radix number systems. Each generalization had a tremendous effect on both implementation and presentation: without exception, the generalized operations were simpler to implement and easier to understand than their instances. In particular, the introduction of mixed-radix number systems unified and simplified the implementation of the array creation functions.

Much is left to be done. The generality of the data structure opens up quite a huge design space. Clearly, a more systematic exploration of this space is desirable: What is the optimal tree shape for a given look-up/update ratio? Likewise, what is the preferred shape if the size of the array is known to lie in a certain range? To obtain more convincing figures, we also plan to port the implementation to other languages such as O'Caml, Standard ML, and Clean.

Acknowledgements

I am indebted to Chris Okasaki for a very stimulating discussion on functional arrays (the idea of trading the running time of look-up operations for the running time of update operations is due to Chris). Thanks are due to Peter Thiemann for spotting an error in a previous version of *tail*. Furthermore, thanks go to the ART team at York and the IFIP 2.8 working group for valuable feedback. Finally, I would like to thank four anonymous referees for their constructive comments.

References

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT*, 28(3):490–503, 1988.

[2] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4, Eindhoven University of Technology, 1983.

[3] Victor J. Dielissen and Anne Kaldewaij. A simple, efficient, and flexible implementation of flexible arrays. In *Third International Conference on Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany*, volume 947 of *Lecture Notes in Computer Science*, pages 232–241. Springer-Verlag, July 1995.

[4] Victor J. Dielissen and Anne Kaldewaij. Leaf trees. *Science of Computer Programming*, 26(1–3):149–165, May 1996.

[5] Paul Dietz and Daniel Sleator. Two algorithms for maintaining order in a list. In Alfred Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 365–372, New York City, NY, May 1987. ACM Press.

[6] R. Fagerberg. A generalization of binomial queues. *Information Processing Letters*, 57(2):109–114, 1996.

[7] Ralf Hinze. Numerical representations as higher-order nested datatypes. Technical Report IAI-TR-98-12, Institut für Informatik III, Universität Bonn, December 1998.

[8] Ralf Hinze. Constructing red-black trees. In Chris Okasaki, editor, *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99, Paris, France*, pages 89–99, September 1999. The proceedings appeared as a technical report of Columbia University, CUCS-023-99, also available from `http://www.cs.columbia.edu/~cdo/waaapl.html`.

[9] Ralf Hinze. Manufacturing datatypes. *Journal of Functional Programming, Special Issue on Algorithmic Aspects of Functional Programming Languages*, 11(5):493–524, September 2001.

[10] Sören Holmström. A simple and efficient way to handle large datastructures in applicative languages. In *Proceedings of the Joint SERC/Chalmers Workshop on Declarative Programming, University College*, London, UK, 1983.

[11] R.R. Hoogerwoord. A logarithmic implementation of flexible arrays. In R.S. Bird, C.C. Morgan, and J.C.P. Woodcock, editors, *Proceedings of the Second International Conference on Mathematics of Program Construction*, pages 191–207. LNCS, Springer-Verlag, 1992.

[12] Haim Kaplan and Robert E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 202–211, Philadelphia, Pennsylvania, May 1996.

[13] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Publishing Company, 3rd edition, 1997.

[14] Eugene W. Myers. Efficient applicative data types. In Ken Kennedy, editor, *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, Salt Lake City, UT, January 1984. ACM Press.

[15] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 86–95, La Jolla, California, June 1995. ACM SIGPLAN/SIGARCH and IFIP WG2.8, ACM Press.

[16] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[17] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Maryland*, pages 77–86, 1998.

[18] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997.

[19] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Si-

mon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Standard libraries for the Haskell 98 programming language. Available from `http://www.haskell.org/definition/`, February 1999.

[20] Daniel D Sleator and Robert E Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

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

A A reference implementation of *Array*

The declaration below makes the list data type an instance of the *Array* class. Its main purpose is to specify the intended behaviour of the member functions.

instance *Array* [ ] **where**

$$
\begin{array}{lcl}
[\,]\,!\,i & = & \mathit{error}\ \texttt{"index out of range"} \\
(x:xs)\,!\,0 & = & x \\
(x:xs)\,!\,(i+1) & = & xs\,!\,i \\
\mathit{update}\ f\ i\ [\,] & = & \mathit{error}\ \texttt{"index out of range"} \\
\mathit{update}\ f\ 0\ (x:xs) & = & f\ x:xs \\
\mathit{update}\ f\ (i+1)\ (x:xs) & = & x:\mathit{update}\ f\ i\ xs \\
\mathit{empty}\ [\,] & = & \mathit{True} \\
\mathit{empty}\ (x:xs) & = & \mathit{False} \\
\mathit{size}\ [\,] & = & 0 \\
\mathit{size}\ (x:xs) & = & 1+\mathit{size}\ xs \\
\mathit{nil} & = & [\,] \\
\mathit{copy}\ 0\ x & = & [\,] \\
\mathit{copy}\ (n+1)\ x & = & x:\mathit{copy}\ n\ x \\
\mathit{cons} & = & (:) \\
\mathit{head}\ (x:xs) & = & x \\
\mathit{tail}\ (x:xs) & = & xs \\
\mathit{map}\ f\ [\,] & = & [\,] \\
\mathit{map}\ f\ (x:xs) & = & f\ x:\mathit{map}\ f\ xs \\
\mathit{zip}\ f\ [\,]\ [\,] & = & [\,] \\
\mathit{zip}\ f\ (x:xs)\ (y:ys) & = & f\ x\ y:\mathit{zip}\ f\ xs\ ys \\
\mathit{list} & = & \mathit{id} \\
\mathit{array} & = & \mathit{id}
\end{array}
$$

B Derivation of *ilist*

This section derives the iterative definition of *list* from the straightforward recursive definition.

Functions

The recursive definition of *list* uses lists a lot: as an intermediate result it produces a list of lists, which is then riffled into a list of elements. It is important to note that *riffle* only works if the inner lists have the same length. Fortunately, this property is guaranteed by Invariant 2. Now, for the derivation it is useful, even vital to keep track of this size information. For that reason, we replace the ubiquitous list type $[X]$ by tuple (or vector) types of the form X^n. In a sense, we view the type of lists as a family of tuple types. Likewise, a list-processing function is seen as a family of functions operating on tuples. We require the following operations.

$$
\begin{array}{lcl}
\mathit{wrap} & :: & X \to X^1 \\
\mathit{unwrap} & :: & X^1 \to X \\
\rho_{m,n} & :: & (X^n)^m \to X^{mn} \\
\check{\rho}_{m,n} & :: & X^{mn} \to (X^n)^m \\
\mathit{cat}_{m,n} & :: & X^m \times X^n \to X^{m+n} \\
\mathit{uncat}_{m,n} & :: & X^{m+n} \to X^m \times X^n \\
\mathit{zip}_n & :: & X^n \times Y^n \to (X \times Y)^n \\
\mathit{unzip}_n & :: & (X \times Y)^n \to X^n \times Y^n
\end{array}
$$

Here $\rho_{m,n}$ denotes one instance of *riffle* and $\check{\rho}_{m,n}$ denotes one instance of *unriffle*. For reasons of readability, we will usually omit the second index writing ρ_m instead of $\rho_{m,n}$ and likewise for $\check{\rho}_{m,n}$, $\mathit{cat}_{m,n}$ and $\mathit{uncat}_{m,n}$.

Properties

The functions satisfy a variety of properties that are needed in the derivation.

The functions come in pairs. Since they operate on tuple types, each operation has a natural inverse: *wrap* and *unwrap* are inverse to each other ($\mathit{wrap} \cdot \mathit{unwrap} = \mathit{id}$ and $\mathit{unwrap} \cdot \mathit{wrap} = \mathit{id}$), $\rho_{m,n}$ and $\check{\rho}_{m,n}$, $\mathit{cat}_{m,n}$ and $\mathit{uncat}_{m,n}$, zip_n and unzip_n.

All the functions are polymorphic in the element type(s). As such they enjoy quite general properties, often referred to as *naturality laws*.

$$
\begin{array}{lcl}
f^1 \cdot \mathit{wrap} & = & \mathit{wrap} \cdot f \\
f^{mn} \cdot \rho_{m,n} & = & \rho_{m,n} \cdot (f^n)^m \\
(f^n \times g^n) \cdot \mathit{unzip}_n & = & \mathit{unzip}_n \cdot (f \times g)^n
\end{array}
$$

Here, $(\cdot)^n$ is the mapping function on n-tuples and '$\times$' is the mapping function on pairs.

The following law states a basic property of ρ, which could be put to use in a divide and conquer implementation of *riffle*.

$$\rho_{m,n+n'} \cdot (\mathit{cat}_{n,n'})^m \cdot \mathit{zip}_m = \mathit{cat}_{mn,mn'} \cdot (\rho_{m,n} \times \rho_{m,n'}) \qquad (3)$$

Given two matrices of type $(X^n)^m$ and $(X^{n'})^m$ we can either join them horizontally (catenating the rows) and riffle the result or else riffle them separately and catenate the resulting lists. The final result is the same in both cases.

Finally, ρ satisfies two monad-like properties.

$$\rho_{1,n} \cdot \mathit{wrap} = \mathit{id} \qquad (4)$$

$$\rho_{m,nn'} \cdot (\rho_{n,n'})^m = \rho_{mn,n'} \cdot \rho_{m,n} \qquad (5)$$

If we ignore the size constraints, we obtain $\rho \cdot \mathit{wrap} = \mathit{id}$ and $\rho \cdot \rho^* = \rho \cdot \rho$, which are two of the monad laws with *concat* replaced by ρ. That said note that the input for the last equation is a *three-dimensional* matrix of type $((X^{n'})^n)^m$.

Specification

Before we specify the iterative version of *list* let us first rephrase *Tree* and recursive version in the current framework.

The data type of multiway trees is indexed by a mixed-radix numeral, which specifies the size of the prefixes and the degree of the

nodes.

$$Tree\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} X \quad = \quad X^{d_0} \times (Tree\ \begin{bmatrix} d_1,\ldots \\ b_1,\ldots \end{bmatrix} X)^{b_0}$$

Let

$$n \quad = \quad \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix}.$$

The type of *list* is then

$$list\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} \quad :: \quad Tree\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} X \to X^n.$$

The straightforward recursive implementation is given by

$$list\ \begin{bmatrix} 0,\ 0,\ \ldots \\ b_0,b_1,\ldots \end{bmatrix} \quad = \quad const\ ()$$

$$list\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} \quad = \quad cat_{d_0} \cdot (id \times \rho_{b_0} \cdot (list\ \begin{bmatrix} d_1,\ldots \\ b_1,\ldots \end{bmatrix})^{b_0}).$$

Now, the idea of the iterative version is to replace the b_0 recursive calls to *list* by a single recursive call that operates on b_0 trees simultaneously. This motivates the following specification (we renamed b_0 to w).

$$ilist_w\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} \quad :: \quad (Tree\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} X)^w \to X^{wn}$$

$$ilist_w\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix} \quad = \quad \rho_w \cdot (list\ \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix})^w$$

Derivation

Let us introduce the following abbreviations:

$$\sigma_0 = \begin{bmatrix} 0,\ 0,\ \ldots \\ b_0,b_1,\ldots \end{bmatrix},\ \sigma = \begin{bmatrix} d_0,d_1,\ldots \\ b_0,b_1,\ldots \end{bmatrix},\ \text{and}\ \sigma' = \begin{bmatrix} d_1,\ldots \\ b_1,\ldots \end{bmatrix}.$$

For the base case we calculate.

$$\begin{array}{ll} & ilist_w\ \sigma_0 \\ = & \quad \{\ \text{specification of } ilist\ \} \\ & \rho_w \cdot (list\ \sigma_0)^w \\ = & \quad \{\ \text{definition of } list\ \} \\ & \rho_w \cdot (const\ ())^w \\ = & \quad \{\ const\ () = f^0 \text{ and naturality of } \rho\ \} \\ & const\ () \cdot \rho_w \\ = & \quad \{\ const\ () \cdot f = const\ ()\ \} \\ & const\ () \end{array}$$

For the inductive case we reason.

$$\begin{array}{ll} & ilist_w\ \sigma \\ = & \quad \{\ \text{specification of } ilist\ \} \\ & \rho_w \cdot (list\ \sigma)^w \\ = & \quad \{\ \text{definition of } list\ \} \\ & \rho_w \cdot (cat_{d_0} \cdot (id \times \rho_{b_0} \cdot (list\ \sigma')^{b_0}))^w \\ = & \quad \{\ (\cdot)^n \text{ functor}\ \} \\ & \rho_w \cdot (cat_{d_0})^w \cdot (id \times \rho_{b_0} \cdot (list\ \sigma')^{b_0})^w \\ = & \quad \{\ zip_n \cdot unzip_n = id\ \} \\ & \rho_w \cdot (cat_{d_0})^w \cdot zip_w \cdot unzip_w \cdot (id \times \rho_{b_0} \cdot (list\ \sigma')^{b_0})^w \\ = & \quad \{\ \text{property (3)}\ \} \\ & cat_{wd_0} \cdot (\rho_w \times \rho_w) \cdot unzip_w \cdot (id \times \rho_{b_0} \cdot (list\ \sigma')^{b_0})^w \\ = & \quad \{\ \text{naturality of } unzip\ \} \\ & cat_{wd_0} \cdot (\rho_w \times \rho_w) \cdot (id^w \times (\rho_{b_0} \cdot (list\ \sigma')^{b_0})^w) \cdot unzip_w \\ = & \quad \{\ \text{functor `}\times\text{' and } (\cdot)^n\ \} \\ & cat_{wd_0} \cdot (\rho_w \times \rho_w \cdot (\rho_{b_0})^w \cdot ((list\ \sigma')^{b_0})^w) \cdot unzip_w \\ = & \quad \{\ \text{property (5)}\ \} \\ & cat_{wd_0} \cdot (\rho_w \times \rho_{wb_0} \cdot \rho_w \cdot ((list\ \sigma')^{b_0})^w) \cdot unzip_w \\ = & \quad \{\ \text{naturality of } \rho\ \} \\ & cat_{wd_0} \cdot (\rho_w \times \rho_{wb_0} \cdot (list\ \sigma')^{wb_0} \cdot \rho_w) \cdot unzip_w \\ = & \quad \{\ \text{specification of } ilist\ \} \\ & cat_{wd_0} \cdot (\rho_w \times ilist_{wb_0}\ \sigma' \cdot \rho_w) \cdot unzip_w \end{array}$$

Noting that $unzip_w \approx elems^* \triangle subs^*$ we have derived the point-free counterpart of the definition given in Sec. 5.2.

It remains to define *list* in terms of *ilist*.

$$\begin{array}{ll} & list\ \sigma \\ = & \quad \{\ \text{property (4)}\ \} \\ & \rho_1 \cdot wrap \cdot list\ \sigma \\ = & \quad \{\ \text{naturality of wrap}\ \} \\ & \rho_1 \cdot (list\ \sigma)^1 \cdot wrap \\ = & \quad \{\ \text{specification of } ilist\ \} \\ & ilist_1\ \sigma \cdot wrap \end{array}$$

C Derivation of *iarray*

Since $list\ \sigma$ and $ilist_w\ \sigma$ are isomorphisms, we can specify $garray\ \sigma$ and $iarray_w\ \sigma$ simply as function inverses.

$$list\ \sigma \cdot garray\ \sigma = id = garray\ \sigma \cdot list\ \sigma$$

$$ilist_w\ \sigma \cdot iarray_w\ \sigma = id = iarray_w\ \sigma \cdot ilist_w\ \sigma$$

As an illustration, here is the straightforward derivation of *iarray* (inductive case only).

$$\begin{array}{ll} & ilist_w\ \sigma \cdot iarray_w\ \sigma = id \\ \iff & \quad \{\ \text{definition of } ilist\ \} \\ & cat_{wd_0} \cdot (\rho_w \times ilist_{wb_0}\ \sigma' \cdot \rho_w) \cdot unzip_w \cdot iarray_w\ \sigma = id \\ \iff & \quad \{\ cat \text{ and } uncat \text{ are inverses}\ \} \\ & (\rho_w \times ilist_{wb_0}\ \sigma' \cdot \rho_w) \cdot unzip_w \cdot iarray_w\ \sigma = uncat_{wd_0} \\ \iff & \quad \{\ \rho \text{ and } \check{\rho} \text{ are inverses, specification of } ilist_w\ \} \\ & unzip_w \cdot iarray_w\ \sigma = (\check{\rho}_w \times \check{\rho}_w \cdot iarray_{wb_0}\ \sigma') \cdot uncat_{wd_0} \\ \iff & \quad \{\ zip \text{ and } unzip \text{ are inverses}\ \} \\ & iarray_w\ \sigma = zip_w \cdot (\check{\rho}_w \times \check{\rho}_w \cdot iarray_{wb_0}\ \sigma') \cdot uncat_{wd_0} \end{array}$$

Noting that $zip_w \approx node^*$ we have derived the point-free counterpart of the definition given in Sec. 5.3.