Bootstrapping One-sided Flexible Arrays

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
Email: `ralf@informatik.uni-bonn.de`
Homepage: `http://www.informatik.uni-bonn.de/~ralf`

February, 2002

(Pick the slides at `.../~ralf/talks.html#T31`.)

Motivation

One-sided flexible arrays support look-up and update of elements and can grow and shrink at one end.

A variety of tree-based implementations is available.

- ✘ Braun trees:
  all operations in $\Theta(\log\ n)$ time.
- ✘ Binary random-access lists:
  $\Theta(\log\ n)$ access and $\Theta(1)$ list operations (amortized).
- ✘ Skew binary random-access list:
  $\Theta(\log\ n)$ access and $\Theta(1)$ list operations (worst-case).

A common characteristic of the tree-based implementations is the *logarithmic* time bound for the look-up operation.

Motivation—continued

Assume that you have an application that uses indexing a lot but also updates or extends occasionally (ruling out ‘real’ arrays).

There is no data structure available that fits these needs.

Idea: Improve look-up by using *fat* multiway trees trading the running time of look-up operations for the running time of update operations (this idea is due to Chris Okasaki).

Signature

$$\textbf{infixl}\ 9\ !$$

$$\textbf{class}\ \mathit{Array}\ a\ \textbf{where}$$

-- array-like operations

$$
\begin{array}{lll}
(!) & :: & a\ x \to \mathit{Int} \to x \\
\mathit{update} & :: & (x \to x) \to \mathit{Int} \to a\ x \to a\ x
\end{array}
$$

-- list-like operations

$$
\begin{array}{lll}
\mathit{empty} & :: & a\ x \to \mathit{Bool} \\
\mathit{size} & :: & a\ x \to \mathit{Int} \\
\mathit{nil} & :: & a\ x \\
\mathit{copy} & :: & \mathit{Int} \to x \to a\ x \\
\mathit{cons} & :: & x \to a\ x \to a\ x
\end{array}
$$

Signature—continued

$$
\begin{array}{lll}
head & :: & a\ x \to x \\
tail & :: & a\ x \to a\ x \\
& & \text{-- mapping functions} \\
map & :: & (x \to y) \to (a\ x \to a\ y) \\
zip & :: & (x \to y \to z) \to (a\ x \to a\ y \to a\ z) \\
& & \text{-- conversion functions} \\
list & :: & a\ x \to [x] \\
array & :: & [x] \to a\ x
\end{array}
$$

Notational convenience: we write both $map\ f$ and $zip\ f$ simply as f^*.

Multiway trees

Idea: bootstrap an implementation based on multiway trees from a standard implementation of flexible arrays.

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

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.

We will show how to turn $\mathit{Tree}\ a$ into an instance of Array given that a is already an instance.

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

List-like operations

Let us start with the $cons$ operation since this operation will determine the way indexing is done.

Idea: fill up the root node; if it is full up, distribute the elements evenly among the subtrees and start afresh.

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

To make this algorithm work, we have to maintain some invariants.

Invariants

For all nodes $t = \langle array\ [x_1, \ldots, x_m], array\ [t_1, \ldots, t_n] \rangle$:

❶	$m \leqslant n$ and $1 \leqslant n$,
❷	$\lvert t_1 \rvert = \cdots = \lvert t_n \rvert$,
❸	$\lvert t \rvert = 0 \iff m = 0$.

where $|t|$ denotes the size of a tree (the total number of elements).

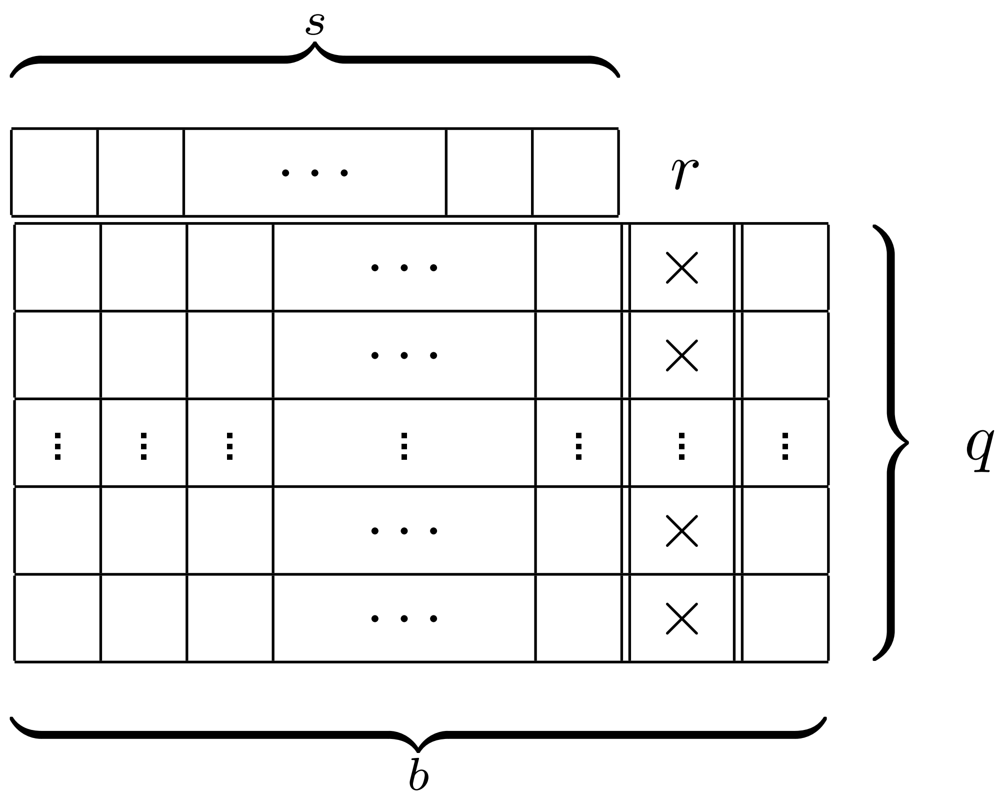
NB. The third invariant is necessary so that we can effectively check whether a given tree is empty.

List-like operations—continued

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

Array-like operations

After q overflows the r-th subtree comprises the q elements at positions $s + 0 * b + r$, $s + 1 * b + r$, . . . , $s + (q - 1) * b + r$, where s is the size of the prefix and b is the total number of subtrees.



Array-like operations–continued

This explains the implementation of '!'.

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

NB. *update* is implemented in an analogous fashion.

Array creation

The shape of the multiway trees is solely determined by the array creation functions nil, $copy$, and $array$.

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 $cons$.

☞ 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*.

Given this assumption the structure of trees can be described using a special number system, the so-called *mixed-radix number system*

Mixed-radix number systems

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).

$$\boxed{\left[\begin{array}{l} d_0, d_1, d_2, \ldots \\ b_0, b_1, b_2, \ldots \end{array}\right] = \sum_{i=0} d_i \cdot w_i \textbf{ where } w_i = b_{i-1} \cdot \cdots \cdot b_1 \cdot b_0}$$

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).

☞ Each natural number has a unique representation in this system.

Converting from the mixed-radix number system

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Bases} & = & [\mathit{Int}] \\
\textbf{type}\ \mathit{Mix} & = & [(\mathit{Int}, \mathit{Int})]
\end{array}
$$

Converting a mixed-radix number to a natural number is straightforward (using the Horner's rule).

$$
\begin{array}{lcl}
\mathit{decode} & :: & \mathit{Mix} \rightarrow \mathit{Int} \\
\mathit{decode}\ ((d, b) : \sigma) & & \\
\quad \mid d == 0 & = & 0 \\
\quad \mid \mathit{otherwise} & = & d + b * \mathit{decode}\ \sigma
\end{array}
$$

Converting to the mixed-radix number system

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

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

Generic array creation functions

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

☞ Now, all we have to do is to come up with interesting bases.

Analysis of running times

Let $\mathrm{H}(n)$ be the height of the *tallest* tree with size n.

The running time of '!' and *update* is

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

where $\bar{\mathcal{T}}_{op}$ is the running time of op on base arrays.

Analysis of running times—continued

The *amortized* running-time of $cons$ is given by

$$\boxed{\mathcal{T}_{cons}(n) = \frac{1}{n} \sum_{i=0}^{\mathrm{H}(n)-1} \frac{n}{w_i}\, w_i\, \bar{\mathcal{T}}_{cons}(b_i) = \sum_{i=0}^{\mathrm{H}(n)-1} \bar{\mathcal{T}}_{cons}(b_i).}$$

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: we have a carry every n/w_i steps; if a carry occurs w_i nodes must be rearranged; and the rearrangement of one node takes $\bar{\mathcal{T}}_{cons}(b_i)$ time.

Variant 1: b-ary trees

Mixed-radix numeral:

$$\left[\begin{array}{llllll} d_0, & d_1, & d_2, & \ldots, & d_n, & \ldots \\ b, & b, & b, & \ldots, & b, & \ldots \end{array}\right]$$

The radices are constant.

$$\begin{array}{lcl} \mathit{bary} & :: & \mathit{Int} \rightarrow \mathit{Bases} \\ \mathit{bary}\ b & = & \mathit{repeat}\ b \end{array}$$

b-ary trees—running times

Performance:

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

Of course, the constants hidden in the Θ notation differ widely.

$$\mathcal{T}_{op}(n) \quad \approx \quad \lg n \cdot \bar{\mathcal{T}}_{op}(b) / \lg b.$$

Variant 2: arithmetic progression trees

Mixed-radix numeral:

$$\left[\begin{array}{l} d_0, d_1, \quad\ \ d_2, \qquad\quad \ldots, d_n, \qquad\quad \ldots \\ \alpha,\ \alpha+\beta, \alpha+2\beta, \ldots, \alpha+n\beta, \ldots \end{array}\right]$$

The radices form the elements of an arithmetic progression.

$$\begin{array}{lcl} \mathit{arithmetic} & :: & \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bases} \\ \mathit{arithmetic}\ \alpha\ \beta & = & \alpha : \mathit{arithmetic}\ (\alpha+\beta)\ \beta \end{array}$$

Arithmetic progression trees—running times

If we fix $\alpha = \beta = 1$, we obtain the so-called *factorial number system*.

$$\begin{bmatrix} d_0, d_1, d_2, \ldots, d_n, & \ldots \\ 1,\ 2,\ 3,\ \ldots, n+1, & \ldots \end{bmatrix}$$

Performance ($\alpha = \beta = 1$):

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

An excerpt of the asymptotic hierarchy

$$
\begin{aligned}
&\log\log n \\
&\quad \prec \sqrt{\log n} \\
&\quad\quad \prec \log n/\log\log n \\
&\quad\quad\quad \prec \log n \\
&\quad\quad\quad\quad \prec (\log n)^2/(\log\log n)^2 \\
&\quad\quad\quad\quad\quad \prec 2^{\sqrt{\log n}} \\
&\quad\quad\quad\quad\quad\quad \prec n
\end{aligned}
$$

Variant 3: Geometric progression trees

Mixed-radix numeral:

$$\left[\begin{array}{l} d_0, d_1,\ d_2,\quad \ldots d_n,\quad \ldots \\ \alpha,\ \alpha\beta, \alpha\beta^2, \ldots \alpha\beta^n, \ldots \end{array}\right]$$

The radices form the elements of a geometric progression.

$$\boxed{\begin{array}{lcl} \mathit{geometric} & :: & \mathit{Int} \to \mathit{Int} \to \mathit{Bases} \\ \mathit{geometric}\ \alpha\ \beta & = & \alpha : \mathit{geometric}\ (\alpha * \beta)\ \beta \end{array}}$$

Arithmetic progression trees—running times

Performance ($\alpha = 1$ and $\beta = 2$):

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

Conversion functions

The conversion functions make use of the following helper functions.

Destructors:

$$
\begin{aligned}
&\mathit{elements} && :: && (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \rightarrow [x] \\
&\mathit{elements}\ \langle xs, ts\rangle && = && \mathit{list}\ xs \\
&\mathit{subtrees} && :: && (\mathit{Array}\ a) \Rightarrow \mathit{Tree}\ a\ x \rightarrow [\mathit{Tree}\ a\ x] \\
&\mathit{subtrees}\ \langle xs, ts\rangle && = && \mathit{list}\ ts
\end{aligned}
$$

Constructor:

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

Conversion functions: $list$

A straightforward recursive implementation of $list$.

$$
\begin{array}{lcl}
list & :: & (Array\ a) \Rightarrow Tree\ a\ x \rightarrow [x] \\
list\ t & & \\
\quad \mid empty\ t & = & [\,] \\
\quad \mid otherwise & = & elements\ t \mathbin{+\!\!+} riffle\ (list^{*}\ (subtrees\ t)) \\
riffle & :: & [[x]] \rightarrow [x] \\
riffle\ x & & \\
\quad \mid all\ empty\ x & = & [\,] \\
\quad \mid otherwise & = & head^{*}\ x \mathbin{+\!\!+} riffle\ (tail^{*}\ x)
\end{array}
$$

NB. $riffle = concat \cdot transpose.$

Conversion functions: *array*

A generic version of *array*.

$$
\begin{array}{lcl}
garray & :: & (Array\ a) \Rightarrow Mix \rightarrow [x] \rightarrow Tree\ a\ x \\
garray\ ((d, b) : \sigma)\ xs & & \\
\quad \mid d == 0 & = & gnil\ (b : map\ snd\ \sigma) \\
\quad \mid otherwise & = & node\ ys\ ((garray\ \sigma)^{*}\ (unriffle\ b\ zs)) \\
\quad \textbf{where}\ (ys, zs) & = & splitAt\ d\ xs \\
unriffle & :: & Int \rightarrow [x] \rightarrow [[x]] \\
unriffle\ n\ xs & & \\
\quad \mid empty\ xs & = & replicate\ n\ [\,] \\
\quad \mid otherwise & = & cons^{*}\ ys\ (unriffle\ n\ zs) \\
\quad \textbf{where}\ (ys, zs) & = & splitAt\ n\ xs
\end{array}
$$

Programming challenge

$\boxed{\text{Give linear-time implementations of } list \text{ and } garray.}$

Conclusion

- Bootstrapped arrays are simple to implement.
- Again, number systems have proven their worth in designing purely functional data structures.
- One can nicely trade the running time of look-up operations for the running time of update operations.
- Sensible choices for the base arrays are ‘real’ arrays or lists (‘logarithmic base arrays’ don’t lead to improvements).
- Preliminary measurements show that bootstrapped arrays perform well for random access, the main factor being the underlying base array.