Date of Acceptance Grade

Instructors

Purely Functional Compressed Bit Vectors with Applications and Implementations

Joel E. Kaasinen

Helsinki 10.7.2011

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytiede

Tekijä — Författare — Author

Joel E. Kaasinen

Työn nimi — Arbetets titel — Title

Purely Functional Compressed Bit Vectors with Applications and Implementations

Oppiaine — Läroämne — Subject

Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro-Gradu -tutkielma	10.07.2011	71

Tiivistelmä — Referat — Abstract

The study of compressed data structures strives to represent information on a computer concisely – using as little space as possible. Compressed bit vectors are the simplest compressed data structure. They are used as a basis for more complex data structures with applications in, for example, computational biology.

Functional programming is a programming paradigm that represents computation using functions without side-effects (such as mutation). Data structures that are representable in and suitable for functional programming are termed functional data structures. Functional data structures are also persistent: operations on them do not destroy previous versions.

This thesis provides implementations of functional compressed bit vectors in the purely functional programming language Haskell. The implemented structures are analyzed and benchmarked against established imperative (C++) implementations.

Applications of compressed bit vectors are also surveyed. This includes compressed wavelet trees, an implementation of which is also presented.

ACM Computing Classification System (CCS):

D.1.1 PROGRAMMING TECHNIQUES: Applicative (Functional) Programming

E.1 DATA STRUCTURES: Arrays, Trees

E.4 DATA CODING AND INFORMATION THEORY: Data compaction and compression

F.2.2 ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY: Nonnumerical Algorithms and Problems: Sorting and searching, Pattern matching

Avainsanat — Nyckelord — Keywords

compressed data structures, functional programming, entropy, wavelet tree

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Resource constraints have always shaped the development of computer science. One of the most basic constraints is that of storage space: while the amount of available space has grown almost exponentially, the amount of data that we want to store has never lagged far behind. Thus compressed representations of data have for long been of interest. While the focus has traditionally been on compressing raw data (images, text, sound), in the past ten years there has been a growing interest in compressing data structures. While (traditional) compressed data can be time-consuming to operate on, compressed data structures try to support efficient access and modification despite their small memory footprint.

In addition to machines, humans have their limits. When software grows too complex, finding bugs and analyzing performance becomes impossible. Functional programming is one possible solution to this problem. By avoiding (global) state and mutation and instead relying on the composition of side-effectless components, many of the problem sources of ordinary imperative programming are eliminated. The programmer is forced to express himself on a higher level, boosting readability of the code and leaving more room for the compiler to optimize.

With the rise of modern functional programming languages such as Haskell, O'Caml and Clojure, the need for efficient functional data structures has grown. Data structures for functional programming need to be designed from a new perspective: the structures need to be immutable but efficient. Immutability means that the data structure operations may not change the structure: thus for example insertions are treated by having the insert operation return a new version of the structure while leaving the original version untouched. Although this sounds horribly inefficient, structures like this can be designed to be efficient.

Already in the 80's there was research into persistent data structures by big names such as Tarjan and Sleator. Persistent data structures are structures that keep track of their change history. They are closely related to immutable data structures since they both retain previous data when an update occurs. Moreover methods of designing persistent and functional data structures overlap (e.g. path copying [DSST89]). Modern research into functional data structures (e.g. Okasaki's book [Oka99] and Finger Trees [HP06]) can be seen as a continuation of research into persistent data structures.

There is an interest in compressed data structures and a need for functional data structures. Also, there are no functional implementations of compressed data structures. This is the gap this thesis aims to fill. The main contribution of this thesis is the *functional* implementation of *static* and *dynamic* bit vectors that are both *compressed* and *persistent*. The implementations are presented using the general framework of *monoidal annotations*.

This thesis starts with a short description of compression and its analysis in sec-

tion 2. This is followed by sections on compressed data structures and their applications (sections 3 and 4). Sections 5 and 6 introduce functional programming and the Haskell language and end the introductory part of this thesis. Sections 7 and 8 describe the Author's functional implementations of compressed data structures. Section 9 describes how the implemented structures were compared against existing implementations.

2 Compression

Compression is the art of representing information concisely. The study of encoding and compression dates back to early communications and signal processing research in the 1940's and 50's by, for example, Shannon, Nyquist and Hartley. In computer science, compression has been studied since the 70's, and numerous methods for compressing strings have been developed.

2.1 Entropy

The key tools for quantifying the performance of compression come from probability theory and statistics. The main factor here is the *unpredictability* of the string to be encoded: the more unpredictable the characters of the string are, the harder it is to compress. However, obtaining suitable measures of unpredictability has been a long process, and the rigorous analysis of some important compression methods have been completed only recently (see e.g. Gagie [Gag07] and Manzini et al [Man01]).

The concept of *information-theoretic entropy*[1] due to Shannon [Sha48] was used in the earliest efforts to analyze rates of information transfer. Shannon's theorem [Sha48, Theorem 9, The Fundamental Theorem for a Noiseless Channel] states that a data stream with an entropy H per symbol cannot be transmitted by using on average under H bits per symbol.

The *entropy* of a discrete random variable X with possible values in U is

$$H(X) = \sum_{x \in U} -p(X = x) \log(p(X = x)).$$

The convention is that $0 \log 0$ is taken to be 0 when calculating the entropy. By the convexity of $x \mapsto -x \log x$ it follows that the entropy of a random variable is maximized when all outcomes are as likely. Correspondingly, the entropy is minimized (in fact it is 0) when $p(X = x) = 1$ for some x. Thus entropy gives as a way of quantifying the inherent *uncertainty* of a random variable.

The problem with entropy as a complexity metric is that it is requires a (probabilistic) model that produces the strings we are interested in. Because of this,

[1] loosely related to the concept of entropy in thermodynamics [Jay57]

several adapted complexity metrics based on entropy that do not need such a model have been proposed [Gag07]. Perhaps the most recent is Kosaraju's and Manzini's *empirical entropy* [Man01].

The 0th-order empirical entropy of a string s is

$$H_0(s) = -\sum_{c \in \Sigma} \frac{n_c}{n} \log\left(\frac{n_c}{n}\right),$$

where Σ is the alphabet, n is the length of the string s and n_c is the number of occurrences of character c in s. Thus the 0th-order entropy depends only on the frequencies of symbols in s. Note that $H_0(s) = H(X_s)$ where X_s is the random variable corresponding to choosing a symbol from s at random.

The kth-order empirical entropy can be defined as

$$H_k(s) = \sum_{x \in \Sigma^k} \frac{n_x}{n} H_0(C_s(x)),$$

where n_x is the number of occurrences of string x as a substring of s and $C_s(x)$ is the string consisting of the characters in s that follow an occurrence of x. Here the string is considered cyclically: that is, `ab` is a substring of the string `baaa` and also $C_{\texttt{baaa}}(\texttt{aa}) = \texttt{ab}$.

The kth-order entropy of a string corresponds roughly to the entropy of the kth order Markov chain approximation for it [Man01]. This can be understood as the average uncertainty of the next symbol in the string when we know the preceding k symbols.

Note that the empirical entropy does not actually depend on the alphabet. It is sufficient to sum over only the k-substrings that occur in s.

As an example, for the string `aababcabcd` we have

$n_{\texttt{aa}} = 1$	$C_s(\texttt{aa}) = \texttt{b}$	$H_0(\texttt{b}) = 0$
$n_{\texttt{ab}} = 3$	$C_s(\texttt{ab}) = \texttt{acc}$	$H_0(\texttt{acc}) = 0.92$
$n_{\texttt{ba}} = 1$	$C_s(\texttt{ba}) = \texttt{b}$	$H_0(\texttt{b}) = 0$
$n_{\texttt{bc}} = 2$	$C_s(\texttt{ba}) = \texttt{ad}$	$H_0(\texttt{ad}) = 1.00$
$n_{\texttt{ca}} = 1$	$C_s(\texttt{ca}) = \texttt{b}$	$H_0(\texttt{b}) = 0$
$n_{\texttt{cd}} = 1$	$C_s(\texttt{cd}) = \texttt{ba}$	$H_0(\texttt{ba}) = 0$
$n_{\texttt{da}} = 1$	$C_s(\texttt{da}) = \texttt{a}$	$H_0(\texttt{a}) = 0$

and thus

$$H_2(\texttt{aababcabcd}) = \frac{3}{10}0.92 + \frac{2}{10}1.00 \approx 0.48.$$

See Figure 1 for additional examples of the empirical entropies of strings.

String	H_0	H_1	H_2	H_3
`aaaaaa`	0	0	0	0
`aaaaab`	0.65	0.60	0.54	0.45
`ababab`	1	0	0	0
`aababcabcd`	1.85	0.80	0.48	0.20
`abracadabra`	2.04	0.55	0	0

Figure 1: Empirical entropies of some strings

2.2 Gap Encoding

In this section we will look at gap encoding, one of the simplest compression methods. It is capable of compressing a binary string (roughly) up to its zeroth-order entropy. However, when we combine gap encoding with the Burrows-Wheeler transform (see Section 2.3) and wavelet trees (see Section 4.3) we will be able to compress strings over arbitrary alphabets up to their kth-order empirical entropy. See Section 4.4 for details.

A binary string with a low zeroth-order entropy H_0 will have an uneven distribution of ones and zeros. This also means that it will have a large number of long *runs*, substrings that contain occurrences only one symbol.

Gap encoding represents the binary string $0^{x_0}10^{x_1}1\dots10^{x_k}$ with $k-1$ ones as $\delta(x_0)\delta(x_1)\dots\delta(x_k)$ where δ is some suitable encoding of integers into binary strings. To make decoding possible, δ has to define a *prefix code*, that is, $\delta(x)$ should be recoverable from $\delta(x)s$ for any integer x and binary string s.

One such encoding is Elias's δ-code [WZ99]. Let $\#x$ denote the number of bits necessary to represent integer x, that is, $\#x = \lceil \log(x+1) \rceil$. Now $\delta(x)$ encodes x in three parts:

1. $\#\#x$ zeros followed by a 1: $0^{\#\#x}1$
2. all but the most significant bit of $\#x$
3. all but the most significant bit of x

The first bits of $\#x$ and x are omitted because they are necessarily 1. The space the encoding requires is $|\delta(x)| = \log x + 2\log\log x + O(1) = \log x + o(\log x)$.

To recover x from $s = \delta(x)s'$ we first count the number of zeros that s starts with. This gives us $\#\#x$. Now we can read the $\#\#x$ bits that encode $\#x$. After this we can read the $\#x - 1$ bits that encode x.

We can now compute the space needed by gap encoding with δ-codes. Let $s = 0^{x_0}10^{x_1}1\dots10^{x_k}$ be a binary string and $G(s) = \delta(x_0)\delta(x_1)\dots\delta(x_k)$ its gap-encoded form. Let $n = |s|$. We now have

$$H_0(s) = -\frac{k}{n}\log\frac{k}{n} - \frac{n-k}{n}\log\frac{n-k}{n} = \frac{k}{n}\log\frac{n}{k} + \frac{n-k}{n}\log\frac{n}{n-k}$$

and thus

$$\begin{aligned}
|G(s)| &= \sum_{i=0}^{k} |\delta(x_i)| = \sum_{i=0}^{k} \log x_i + 2\log\log x + O(1) \\
&\leq (k+1)\log\frac{n-k}{k+1} + (k+1)\log\log\frac{n-k}{k+1} + O(k) \\
&\leq k\log\frac{n}{k} + \log\frac{n}{k} + (k+1)\log\log\frac{n}{k} + O(k) \\
&\leq nH_0(s) + (k+1)\log\log\frac{n}{k} + \log\frac{n}{k} + O(k) \\
&= nH_0(s)(1+o(1)) + O(k+\log n)
\end{aligned}$$

by $\sum_{i=0}^{n} x_i = n-k$ and the convexity of log. Thus when k is small compared to $|s|$ the size of $G(s)$ is close to $|s|H_0(s)$.

A more straightforward encoding for integers is the *nibble encoding*[2]. The nibble encoding stores an integer i by using four bits (a "nibble") to encode every three bits of the binary representation of i. The highest bit of a nibble signals whether the encoding continues to the next nibble. Thus $25 = 11001_2$ would be encoded as 1011 0001. The nibble encoding is straightforward to implement efficiently on a word-based computer. The space requirement is $4\lceil \log x/3 \rceil$. Performing a similar analysis as in the δ case gives the space needed by the nibble gap encoding G_n:

$$\begin{aligned}
|G_n(s)| &= \sum_{i=0}^{k} 4\left\lceil \frac{\log x}{3} \right\rceil \leq 4k + \sum_{i=0}^{k} 4\frac{\log x}{3} \\
&\leq \frac{4(k+1)}{3}\log\frac{n-k}{k+1} + 4k \\
&\leq \frac{4k}{3}\log\frac{n}{k} + 4\log\frac{n}{k} + 4k \\
&\leq \frac{4}{3}nH_0(s) + 4\log\frac{n}{k} + 4k \\
&= nH_0(s)(1+\epsilon) + O(k+\log n).
\end{aligned}$$

2.3 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) [BW94] is a reversible string transform that results in a string that compresses better than the original string. It is an important building block in various compressed data structures.

The BWT operates by lexicographically sorting the cyclic shifts of a string s.[3] To compute the $bwt(s)$, the BWT of a string s:

[2] A variation of the Vbyte encoding [WZ99]

[3] The usual definition of BWT uses left-to-right ordering and forms the BWT into the *last* column of the matrix. This nonstandard definition is due to Manzini et al. [Man01]

Using this definition allows us to consider *preceding* contexts in the analysis, which corre-

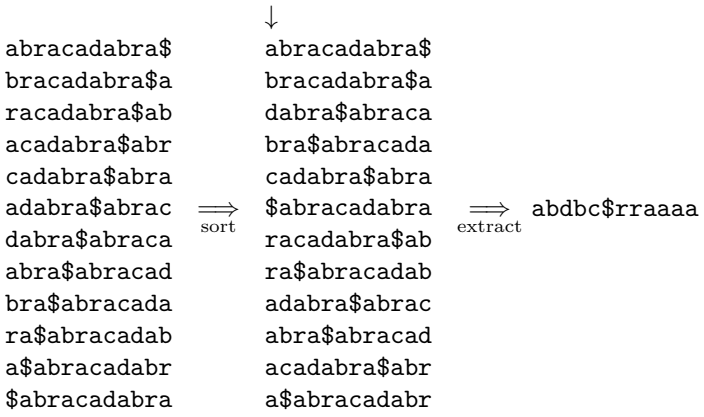


Figure 2: An example of the BWT

1. Append the special symbol `$`, smaller than any other symbol, to s.
2. Form a matrix with the cyclic shifts of s as rows.
3. Sort the rows in *right-to-left lexicographic order*.
4. The first column is the string $bwt(s)$.

See Figure 2 for an illustration.

Somewhat surprisingly, the BWT is reversible. Consider the sorted matrix M in step 3 above. We know the first column of M, $m_1 = bwt(s)$. Sorting the characters of m_1 will give us the last column, m_n. We now know all length 2 substrings of s since the characters of m_n precede the characters of m_1 in s. We can now sort all these 2-substrings (lexicographically) and obtain columns m_n and m_{n-1}. By iteratively applying this procedure we obtain the whole matrix M, from which s is easily extracted.

Both the BWT and its inverse can be computed significantly more efficiently than the algorithms described above do [BW94, section 4]. However the simple algorithms presented above best highlight the nature of the Burrows-Wheeler Transform.

The following lemma relates the BWT to empirical entropy. Note that the lemma holds simultaneously for all k.

Lemma 1. *Let s be a string and $k \leq |s|$ an integer. Now $bwt(s)$ is a catenation of permutations of all the strings $C_s(w)$ such that w is a k-substring of s.*

Proof. Consider the matrix used in BWT after the left-to-right sorting has taken place. The k last columns the matrix contain all the k-substrings of s in lexico-

sponds to our earlier definition of empirical entropy in Section 2.1. Also, with this definition the search algorithm in Section 4.4 can proceed through the pattern in normal order instead of reverse order (cf. [FM05]).

$$\underbrace{\texttt{a}}_{C_s(\$)}\ \underbrace{\texttt{bdbc\$}}_{C_s(\texttt{a})}\ \underbrace{\texttt{rr}}_{C_s(\texttt{b})}\ \underbrace{\texttt{a}}_{C_s(\texttt{c})}\ \underbrace{\texttt{a}}_{C_s(\texttt{d})}\ \underbrace{\texttt{aa}}_{C_s(\texttt{r})}$$

$$\underbrace{\texttt{a}}_{C_s(\texttt{a\$})}\ \underbrace{\texttt{b}}_{C_s(\texttt{\$a})}\ \underbrace{\texttt{d}}_{C_s(\texttt{ca})}\ \underbrace{\texttt{b}}_{C_s(\texttt{da})}\ \underbrace{\texttt{c\$}}_{C_s(\texttt{ra})}\ \underbrace{\texttt{rr}}_{C_s(\texttt{ab})}\ \underbrace{\texttt{a}}_{C_s(\texttt{ac})}\ \underbrace{\texttt{a}}_{C_s(\texttt{ad})}\ \underbrace{\texttt{aa}}_{C_s(\texttt{br})}$$

Figure 3: Splitting $bwt(s)$ into permutations of strings $C_s(w)$ for $s =$ `abracadabra`

String	H_0	H_1	H_2
`abracadabra$`	2.28	0.80	0.17
`abdbc$rraaaa`	2.28	0.60	0.00

Figure 4: A comparison of the empirical entropies of a string and its BWT

graphical order. Also, all the cyclic shifts that end in a given k-substring w are on consecutive rows.

Now if the k-substring w occurs in the end of rows i through j, the first characters of rows i through j give $C_s(w)$. Since we can partition the matrix into ranges of rows that end in the same k-substring, the whole string $bwt(s)$ is given by the strings $C_s(w)$.

For illustration, compare Figure 2 and Figure 3. □

Essentially, the BWT can be seen as a preprocessing step that groups together characters with similar contexts. Thus it is quite natural that the transform enhances compressibility. However, a formal analysis of BWT combined with simple compression was only recently achieved by Manzini and others [Man01] [GM10] [KLV07]. The following short example is due to Manzini.

The compression advantage that the BWT offers can be seen as follows. Suppose we have a compression algorithm A such that for any partition $s_1 s_2 \dots s_k$ of s we have

$$|A(s)| \leq \sum_{i=1}^{k} |s_i| H_0(s_i).$$

Now for an arbitrary k, let $bwt(s) = s_1 s_2 \cdots s_t$ such that each s_i is a permutation of some $C_s(x)$. Such a split is guaranteed to exist by the previous lemma. We

then have

$$\begin{aligned} nH_k(s) &= \sum_{x \in \Sigma^k} n_x H_0(C_s(x)) \\ &= \sum_{x \in \Sigma^k} |C_s(x)| H_0(C_s(x)) \\ &= \sum_i |s_i| H_0(s_i) \\ &\geq |A(bwt(s))| \end{aligned}$$

since $bwt(s)$ consists of permutations of strings $C_s(x)$ for substrings x of s and H_0 does not change if a string is permuted.

Note that this calculation is only a sketch: the algorithm A is impossibly good. In practice we cannot obtain a bound of $nH_k(s)$ since $H_{|s|}(s) = 0$ for any s.

Analyses [Man01] [KLV07] of compressing the BWT of a string with a zeroth-order compressor have produced bounds of the form

$$\alpha |s| H_k(s) + \beta |s| + O(f(\sigma, k)),$$

where α and β are constants, σ is the size of the alphabet and f is a function, for example $f(\sigma, k) = \sigma^k$. Some analyses also produce bounds using *modified empirical entropy* [Man01] [GM10]. For details refer to the cited articles.

3 Compressed Bit Vectors

Compressed bit vectors form the basis of many compressed data structures, especially of those relating to strings [MN08] [Cla96]. Bit vectors are a very versatile structure, allowing efficient embedding of many sorts of structures. Even though domain-specific compressed structures may be more efficient, compressed bit vectors form a valuable baseline for comparison in many cases.

The Dynamic Sequence with Indels problem [MN08] consists of storing a sequence of n symbols $S = s_0 s_1 \cdots s_{n-1}, s_i \in \Sigma$ and supporting the following operations:

- $read(S, i)$ returns s_i;
- $rank_s(S, i)$ returns the number of occurrences of symbol $s \in \Sigma$ in $a_0 \cdots a_i$;
- $select_s(S, i)$ returns the index of the ith occurrence of symbol s;
- $insert(S, i, s)$ inserts symbol $s \in \Sigma$ between s_i and s_{i+1};
- $delete(S, i)$ deletes symbol s_i.

The Dynamic Bit Vector with Indels is a special case of the previous problem, with the two-symbol alphabet $\Sigma = \{0, 1\}$. The Static Bit Vector problem is a restriction of Dynamic Bit Vector with Indels, where *insert* and *delete* are not supported.

The operations *rank* and *select* are related by the equation

$$rank_s(S, select_s(S, i)) = i.$$

This allows one to implement $select_s$ as a binary search over indices using $rank_s$. Also, the following equations hold for the binary alphabet $\Sigma = \{0, 1\}$

$$\begin{aligned} rank_0(S, i) &= i - rank_1(S, i) \\ read(S, i) &= rank_1(S, i) - rank_1(S, i-1) \end{aligned}$$

Using *rank* and *select* we can also implement the operations $next_s(S, i)$ and $prev_s(S, i)$ that return, respectively, the index of the next and previous occurrence of s in S after location i. The implementations are simply:

$$\begin{aligned} next_s(S, i) &= select_s(S, rank_s(S, i) + 1) \\ prev_s(S, i) &= select_s(S, rank_s(S, i) - 1) \end{aligned}$$

Parenthetically, one can see the Dynamic Bit Vector with Indels also as a specialization of the Searchable Partial Sums with Indels [RRR01] problem. The Searchable Partial Sums with Indels problem consists of offering a data structure that stores a sequence $S = (s_1, s_2, \ldots, s_n)$ of n non-negative integers and supports the following operations.

- $sum(S, k)$ returns the sum of the first k numbers: $\sum_{i=1}^{k} s_k$
- $select(S, x)$ searches for a prefix with the given sum. It returns an index k such that $\sum_{i=1}^{k} s_k \geq x$.

When we restrict the stored numbers to be 0 or 1, this is actually the Dynamic Bit Vector with Indels problem: *sum* corresponds to *rank* and *select* corresponds to *select*.

3.1 Prior Work

Raman et al [RRR02] give a solution to Static Bit Vector by dividing the vector into superblocks and the superblocks into blocks. The blocks are then encoded up to their 0th order entropy, and various indices are stored per-block and per-superblock. This allows for constant-time operations (under the cell probe model [PD06]) in $nH_0 + O(n \log \log n / \log n)$ space.

Solutions to the Dynamic Bit Vector with Indels problem tend to utilise some sort of tree to store blocks of bits. Gerlach [Ger07] implements an uncompressed solution in $O(n)$ space by storing blocks of bits in a (red-black search)

tree. Blandford and Blelloch [BB04] store gap-encoded blocks in a "dictionary" (implementable as, say, an ordered search tree) for a dynamic structure in $O(nH_0)$ space. Mäkinen and Navarro [MN08] improve this by eliminating wasted space from the blocks and achieve $nH_0 + o(n)$ space.

4 Applications of Compressed Bit Vectors

4.1 Sets

The classical application for bit vectors is the representations of sets over a finite domain X. Elements of the domain are numbered, and the state of bit i in the vector corresponds to the absence or presence of element number i in the represented set.

In this case the 0th-order entropy of the bit vector s_V that encodes the set $V \subset X$ is

$$H_0(s_V) = -\epsilon \log(\epsilon) - (1-\epsilon)\log(1-\epsilon),$$

where $\epsilon = |V|/|X|$ is the proportion of ones in the vector. This quantity is clearly symmetric on the interval $\epsilon \in [0, 1]$, positive, and 0 when $\epsilon = 0$. Thus the entropy of the bit vector reaches its maximum ($H_0(s_V) = 1$) when V contains half of the elements of the domain ($\epsilon = \frac{1}{2}$).

4.2 Trees

Clark [Cla96] gives an account of representing different trees space-efficiently using bit vectors and the *rank* and *select* operations. The simplest of these representations is Zaks' sequence. It encodes a binary tree into a bit sequence in the following way

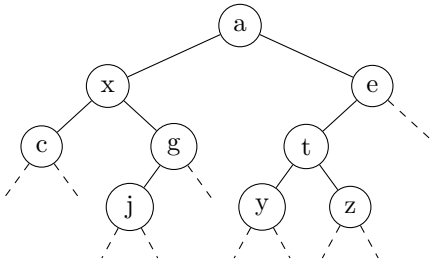
1. Label all nodes with 1
2. Insert children labeled 0 for all nodes missing children
3. List out the labels in level-wise order

An example is given in Figure 5. If the nodes contain additional data, it can be encoded into an array with index i containing the data for the node that occupies index i in the Zaks' sequence.

The operations *rank* and *select* can be used to implement child and parent lookup for the sequence[4]:

- $\mathit{leftchild}(i) = 2\mathit{rank}_1(i)$
- $\mathit{rightchild}(i) = 2\mathit{rank}_1(i) + 1$

[4] note the similarity to representing a binary heap using an array



Zaks' sequence	1	1	1	1	1	1	0 0 0	1	0	1	1	0 0 0 0 0 0
Node data	a	x	e	c	g	t		j		y	z	

Figure 5: A tree and its Zaks' sequence. Dashed lines indicate missing children

- $parent(i) = select_1(\lfloor x/2 \rfloor)$

A binary tree with n inner nodes has $n + 1$ leaves, and thus length of Zaks' sequence for a tree with n nodes is $2n+1$ bits. With an efficient implementation of Static Bit Vector this representation of a tree is quite competitive with the classical pointer-based representation!

4.3 Wavelet Trees

Bit vectors can be used to encode strings over other finite alphabets by using a *wavelet tree* [FGM09]. The wavelet tree is a (balanced) binary tree with symbols of the alphabet in the leaves. The inner nodes encode paths to the leaves such that the ith path leads to the symbol at the ith place in the string (see Figure 6).

The encoding of paths is as follows. Each node contains a bit vector. The ith bit of the vector indicates whether the ith path that reaches this node continues to the left or to the right. Thus descending in the tree works as follows

- When looking for path number i in node with vector w, look at bit $read(w, i)$.
- If it is 0: continue with path number $rank_0(w, i) - 1$ in the left child
- If it is 1: continue with path number $rank_1(w, i) - 1$ in the right child

We need to subtract 1 because rank returns a count (minimum value 1) we need an index (minimum value 0).

We can also implement *rank* and *select* on the encoded string in the following way. To compute $select_x(s, i)$:

1. Start at leaf x of the wavelet tree.

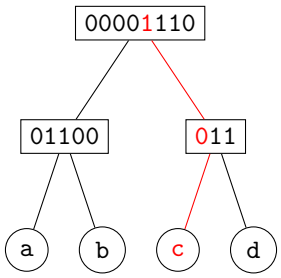


Figure 6: Wavelet tree for the string `abbacdda` and the path corresponding to the occurrence of `c` in the fifth place

2. Move to the parent. Let b be the bit vector in the parent.
3. Set $i := \mathit{select}_a(b, i)$ where a is 0 if we came from the left child and 1 if we came from the right child.
4. If at the root, i contains the answer. Otherwise go back to step 2.

To compute $\mathit{rank}_x(s, i)$:

1. Start at the root.
2. Set $i := \mathit{rank}_a(b, i) - 1$ where b is the bit vector in the current node and a is 0 if x is in the left subtree and 1 if x is in the right subtree.
3. Descend one level towards x.
4. If at x, i contains the answer. Otherwise go back to step 2.

If we use an implementation of Static Bit Vector with constant-time *rank* and *select* queries, we achieve $O(\log \sigma)$ time complexity for *read*, *rank* and *select* on the wavelet tree.

With an uncompressed bit vector implementation with space complexity of $n(1+o(1))$ the wavelet tree takes $n \log \sigma(1 + o(1))$ space.

The following lemma tells us that the wavelet tree is also space efficient if compressed bit vectors are used in the nodes.

Lemma 2. *For a string s, we call $|s|H_0(s)$ the 0th-order information content of s.*

For a wavelet tree of a string s, the information content of s is equal to the sum of the information contents of the bit vectors in the nodes of the tree.

Proof. First note that a subtree of a wavelet tree is also a wavelet tree. We will thus use induction on the wavelet tree height.

Base case: The wavelet tree of height 1 has one node and two leaves. The bit vector in the root node is isomorphic up to choice of symbols to the original

string.

Induction step: Let s be the string represented by the wavelet tree, b be the bit vector in the root of the tree, s_l the string represented by its left subtree and s_r the string represented by its right subtree. Let L be the symbols in the leaves of the left subtree and R respectively the symbols in the right subtree. We know that $s_l \in L^*$ and $s_r \in R^*$.

We know by the induction hypothesis that the sum of the information contents of the nodes of the tree is

$$I = |b|H_0(b) + |s_l|H_0(s_l) + |s_r|H_0(s_r).$$

Now note that the following holds for the information content of any string:

$$\begin{aligned} |s|H_0(s) &= |s| \sum_{x \in \Sigma} -\frac{n_x(s)}{|s|} \log \frac{n_x(s)}{|s|} \\ &= \sum_{x \in \Sigma} |n_x| \log \frac{|s|}{n_x(s)}, \end{aligned}$$

where $n_x(s)$ is the number of occurrences of x in s.

We also know by the definition of the wavelet trees that the number of zeros in b is $|s_l|$ and the number of ones is $|s_r|$. We also know that for $x \in L$, $n_x(s) = n_x(s_l)$, and respectively for $x \in R$, $n_x(s) = n_x(s_r)$.

We thus have

$$\begin{aligned} I =& |s_l| \log \frac{|b|}{|s_l|} + |s_r| \log \frac{|b|}{|s_r|} + \sum_{x \in L} n_x(s_l) \log \frac{|s_l|}{n_x(s_l)} + \sum_{x \in R} n_x(s_r) \log \frac{|s_r|}{n_x(s_r)} \\ =& \left(\sum_{x \in L} n_x(s_l) \right) \log \frac{|b|}{|s_l|} + \left(\sum_{x \in R} n_x(s_r) \right) \log \frac{|b|}{|s_r|} \\ & + \sum_{x \in L} n_x(s_l) \log \frac{|s_l|}{n_x(s_l)} + \sum_{x \in R} n_x(s_r) \log \frac{|s_r|}{n_x(s_r)} \\ =& \sum_{x \in L} n_x(s_l) \left(\log \frac{|s_l|}{n_x(s_l)} + \log \frac{|b|}{|s_l|} \right) + \sum_{x \in R} n_x(s_r) \left(\log \frac{|s_r|}{n_x(s_r)} + \log \frac{|b|}{|s_r|} \right) \\ =& \sum_{x \in L} n_x(s_l) \log \frac{|b|}{n_x(s_l)} + \sum_{x \in R} n_x(s_r) \log \frac{|b|}{n_x(s_r)} \\ =& \sum_{x \in L} n_x(s) \log \frac{|s|}{n_x(s)} + \sum_{x \in R} n_x(s) \log \frac{|s|}{n_x(s)} \\ =& \sum_{x \in \Sigma} n_x(s) \log \frac{|s|}{n_x(s)} \\ =& |s|H_0(s), \end{aligned}$$

which completes the proof. □

An immediate corollary of this is the following. If one uses a compressed bit vector implementation with space complexity $nH_0(1+o(1))$ in the nodes of the wavelet tree, the bit vectors of s require space

$$\begin{aligned}\sum_{b\in B} |b|H_0(b)(1+o(1)) &= \sum_{b\in B} |b|H_0(b) + \sum_{b\in B} o(|b|H_0(b)) \\ &= |s|H_0(s) + o(|s|H_0(s)) = |s|H_0(s)(1+o(1)),\end{aligned}$$

where B is the set of bit vectors stored in the tree. In addition to this the tree itself needs $O(\sigma)$ space bringing the total to

$$|s|H_0(s)(1+o(1)) + O(\sigma).$$

4.4 Indexes

An *index* over a text is a structure that facilitates searches on that text. Indexes have been studied since the early days of computer science. The operations usually required of an index are counting and locating occurrences of a given substring in the indexed text. An index is called a *self-index* if the original text is recoverable from the index (and thus doesn't need to be stored alongside it).

Ferragina and Manzini [FM05] use the BWT of a text as an index. As the BWT can be efficiently compressed (see Section 2.3) this allows for a highly compressed self-index.

Occurrences of a pattern are related to the BWT in the following way: consider the (implicit) sorted matrix in the definition of the BWT. We can identify occurrences of a pattern in the string with ranges of rows in the matrix: the pattern p maps to all rows ending with p. These rows are contiguous in the matrix by construction.

Here is Ferragina's and Manzini's algorithm for counting occurrences of a pattern $p = p_1 p_2 \dots p_k$ in s. In addition to $bwt(s)$, it needs access to a table C: $C(i)$ contains the number of combined number occurrences of symbols $\{\$, 0, \dots, i-1\}$ in s.

1. Initialize $a = 1$, $b = |bwt(s)|$
2. For $i = 1 \dots k$:
   (a) $a = C(p_i) + rank_{p_i}(bwt(s), a-1) + 1$
   (b) $b = C(p_i) + rank_{p_i}(bwt(s), b)$
   (c) If $b < a$, the number of occurrences is 0.
3. The number of occurrences is $b - a + 1$.

See Figure 7 for an illustration.

The table C:

`$`	`a`	`b`	`c`	`d`	`e`
0	1	5	8	11	13

The BWT matrix and the markers a and b:

`a`abcabcdabcde$					
`a`bcabcdabcde$a					
`b`cabcdabcde$aa					
`b`cdabcde$aabca					
`b`cde$aabcab`cda`			a_3,b_3		
`c`abcdabcde$aab					
`c`dabcde$aabcab					
`c`de$aabcab`cdab`					a_4,b_4
`a`bcdabcde$aab`c`	a_1				
`d`abcde$aabcab`c`					
`d`e$aabcabcdab`c`	b_1				
`a`bcde$aabcab`cd`		a_2			
`e`$aabcabcdab`cd`		b_2			
`$`aabcabcdabcde					

Figure 7: An illustration of counting the occurrences of `cdab` in `aabcabcdabcde` via the BWT. The BWT is higlighted in blue. The partial matches found by the search are highlighted in green. a_i and b_i mean the a and b pointers when i characters of the pattern have been processed.

Lemma 3. *The previous algorithm returns the number of occurrences of p in s.*

Proof. We prove by induction that the following invariant is maintained: after i iterations, exactly the rows a through b (inclusive) in the sorted BWT matrix for s end in $p_1 p_2 \ldots p_i$. The matrix is indexed from 1.

Clearly the invariant holds when 0 iterations have been made: every row ends in the empty string, and all the rows are in the range $a = 1$ and $b = |bwt(s)|$.

Let us next assume that the invariant holds after $i-1$ rounds. From the invariant it follows that the characters at indexes a through b in $bwt(s)$ are the characters that have $p' = p_1 p_2 \ldots p_{i-1}$ as a prefix in s. (Remember that the first column of the matrix is $bwt(s)$!)

Consider all the occurrences of p_i in $bwt(s)$. The ones that are before index a in $bwt(s)$ have prefixes in s that are lexicographically smaller than p'. There are $rank_{p_i}(bwt(s), a-1)$ of these. The ones that are in the range $a \ldots b$ have p' as a prefix. There are $rank_{p_i}(bwt(s), b) - rank_{p_i}(bwt(s), a-1)$ of these. Finally the ones that have an index greater than b have a prefix greater than p'.

Now consider all the occurrences of p_i in the last column of the matrix. They appear consecutively and in the order of their prefixes (and thus in the same order as listed in the previous paragraph). Thus if c is the index of the first row that ends in p_i, the rows that end in the string $p' p_i$ have indexes

$$c + rank_{p_i}(bwt(s), a-1) + 1 \ \ldots \ c + rank_{p_i}(bwt(s), b).$$

Noting that $c = C(p_i)$ completes the proof. □

Fetching the locations of the matches is a bit more complicated. Ferragina and Manzini use a method where the positions of some rows of the BWT matrix are remembered, and the positions of matches are found by iterating backwards in the string until a remembered position is found. Refer to the original article for details.

If we store the BWT in an uncompressed wavelet tree, we achieve $O(|p| \log \sigma)$ time for occurrence counting in $|s| \log \sigma (1 + o(1))$ space. This includes the additional $O(\sigma \log |s|)$ space required for storing the array C.

As we saw in Section 4.3, a wavelet tree using compressed bit vectors acts as a zeroth-order compressor. Thus we reach kth order compression when we store the BWT of a string in a compressed wavelet tree (recall the end of Section 2.3). Manzini et al [FGM09] have a detailed analysis of storing the BWT of a text in a wavelet tree.

5 Functional Programming

In general, *functional programming* refers to programming that makes heavy use of functions, especially higher-order functions (functions that take functions as arguments or return them). Beyond this, there are many shades and different definitions for functional programming.

Firstly, there is the distinction between functional programming as a programming paradigm and functional programming languages. (see e.g. [Hud89]) Secondly, some people want their functional programming language to have additional semantic guarantees. These languages are usually termed *purely functional*.

An important aspect of functional programming is that it eschews *state* and *side effects*, preferring to explicitly pass information to functions in arguments instead of via the program's global state. Notable examples of state and state-modifying side effects are variables and assignments. Statelessness and lack of side effects are in a sense conditions that guarantee that a function really is a (mathematical) function.

Adhering to the principles of statelessness and lack of side effects guarantees *referential transparency*, which in turn facilitates *equational reasoning*. [Hud89] Referential transparency means that a term can be always replaced by its definition without affecting semantics. For example, in a functional program the expression

```
let x = f a
  in x + x
```

is equivalent to the expression

```
f a + f a
```

This would not hold if f were allowed to access some global state, for example increment and return the value of a counter. This type of reasoning can be used to formally prove properties about the code, but also helps programmers in their day-to-day reasoning. [Hud89]

The *purely functional programming languages* are programming languages that force the programmer to express himself functionally. Modern purely functional languages include Haskell and Agda. Scheme is one notable example of a language that is not purely functional but encourages functional style.

5.1 Laziness

The evaluation order of ordinary imperative programming languages is designed in order to make the order of side effects predictable. Statements are executed in order, function arguments are evaluated before the function call etc.

In a purely functional language, the situation is radically different: the semantics of the program are oblivious to the order of evaluation. For this reason some purely functional languages choose a radically different evaluation method: *lazy evaluation*.

In lazy evaluation, a value can either be a concrete, evaluated value (like a floating-point number), or a *thunk*, a computation waiting to happen. When the language runtime tries to access a value that turns out to be a thunk, it runs the thunk and replaces it with its result (a value). Lazy evaluation is on-demand evaluation.

The reasons for choosing lazy evaluation stem from theoretical considerations. Lazy evaluation is a way of implementing *normal order evaluation*, an evaluation strategy that halts with the largest set of inputs. This can be seen in the following example. Let `f` be a non-halting function of type `Int->Int`, and

```
g :: Int -> Int -> Int
g 0 x = x
g y _ = y
```

That is, `g` returns its second argument only if the first argument is zero. Otherwise it returns its first argument. Now the term `g 1 (f 0)` has a value under lazy evaluation, since `f 0` never gets evaluated. Under strict (i.e. non-lazy) evaluation, `f 0` would need to be evaluated before the call of `g` could be evaluated.

In addition to the appealing theoretical properties of lazy evaluation, many claim that it makes code clearer and enhances programmer productivity [Hug89].

5.2 Functional Data Structures

Most classical data structures rely on mutation for performance reasons, i.e. their implementations violate referential transparency. (E.g. tree rotations that reuse nodes) *Functional data structures* are data structures that can be implemented in a purely functional language. Functional data structures are by necessity both *immutable* and *persistent*. [Oka99, chapter 1]

Immutable data structures cannot be modified in-place and are thus referentially transparent. This means that once one has obtained a reference (pointer) to a structure, the exact same data will always be reachable through that reference. The benefits of referential transparency in functional programming were already stipulated previously, but immutable data structures also have uses in e.g. concurrent programming in imperative languages [Lea99].

A persistent data structure is one in which older versions of the structure are still accessible after an update. One example of such a structure is a prepend-only linked list. Persistent data structures were researched originally without connection to functional programming. Driscoll et al [DSST89] give a somewhat

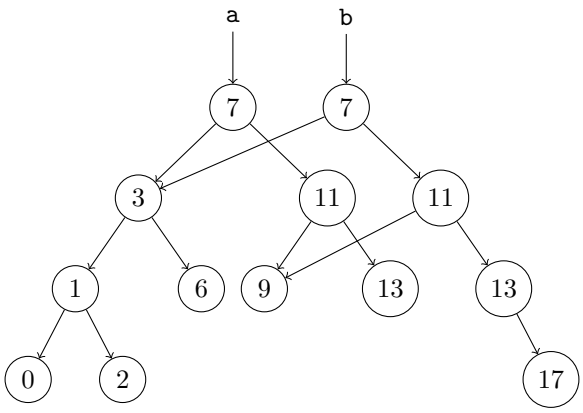


Figure 8: Sharing when adding a new element to an immutable ordered search tree: `b = insert(a,17)`.

outdated but useful summary of the field and discuss a few general methods for obtaining persistence.

There are two distinct flavours of persistence in data structures: explicit and implicit. By *explicit persistence* we refer to structures that use explicit version numbers or in which older versions are reachable by applying some operations to the current version. In contrast functional datastructures are *implicitly persistent*: each instance of the structure is a self-contained version with no relation to other versions. Older versions can be kept available simply by holding on to a reference to them: immutability guarantees this.

The way a functional data structure can be efficient is by *sharing*: the unchanged parts of a linked structure can be shared between different versions of a structure. For example imagine implementing insertions for an immutable tree structure. In order to insert a new leaf, the nodes on the path from the root to the leaf need to be copied. However, the rest of the tree can be reused. See Figure 8.

6 Haskell

Haskell is a pure functional programming language with lazy evaluation. Haskell has an expressive type system to which implementations offer powerful extensions.

This section introduces the reader to the structure of functional programs and Haskell. The purpose of this is purely illustrative: we can only provide a quick overview of the language. Appendix B contains a short summary of the syntax presented here. For additional information on the syntax and features of

Haskell please see The Haskell Report [Mar] or introductory texts (e.g. [Lip11] [OGS08]).

6.1 Expressions

We will start by describing the expression-level syntax of Haskell. All of the examples use functions from the Haskell Prelude, so the reader can try them out using an interactive Haskell interpreter, such as ghci [GHC, chapter 2].

Function application is denoted by juxtaposition: for example the following call to the modulus function evaluates to 1.

```
mod 5 2
```

Function application associates to the left. The previous expression could be written as

```
(mod 5) 2
```

What this means is that all Haskell functions take only one argument. Multi-argument functions can be viewed as functions that return a function. The following sections will clarify this point.

Haskell also has infix operators, for example the usual arithmetic ones: evaluating

```
1 + (5 * 2)
```

yields 11. A nice feature of Haskell is that infix operators can be used as prefix functions by enclosing them in parentheses. The previous example could have been written as

```
(+) 1 ((*) 5 2)
```

Symmetrically, prefix functions can be used as infix operators by enclosing them in backticks (the "`" character):

```
5 `mod` 2
```

Operators may also be partially applied using a syntactic device known as a *section*. For example, the following expressions evaluate to 2:

```
4/2
(/2) 4
(4/) 2
```

One can also use sections with backticked functions. As a summary the following expressions are equivalent (and evaluate to `1`):

```
mod 5 2
(mod 5) 2
5 `mod` 2
(`mod` 2) 5
(5 `mod`) 2
```

The usefulness of sections will become apparent in Section 6.4 when we introduce higher-order functions.

Haskell operators have a precedence order (for operators of the standard library it is documented in the Haskell Report [Mar, Section 4.4.2]). Function application has higher precedence than any infix operations. Thus the following evaluates to 3 and not to 1:

```
mod 5 3 + 1
```

Finally, we can define local variables using the `let...in...` construct. The following expression evaluates to 30:

```
let x = 5 in x + (x * x)
```

One can also define multiple variables, separating the definitions either with `;` or suitable whitespace. The following to expressions both evaluate to 3:

```
let x = 1; y = 2 in x + y

let x = 1
    y = 2
in x + y
```

The final[5] piece of expression syntax is the **`if then else`** expression. Since Haskell does not have assignment or side-effects, the usual **`if`**-statement is next to useless. However, programs need to make choices and thus there is a need for a language construct that chooses. Haskell's **`if`** is an expression that chooses between two values.

```
if True then 1 else 2
let x = 5
  in if mod x 2 == 0 then 2 else 1
```

Both of the above expressions evaluate to `1`

6.2 Definitions

A functional program consists of *definitions*. In Haskell, a definition has two parts: an (optional) *type signature*, and a number of *equations*. Let us begin

[5] actually, we introduce the **`case`** expression a couple of subsections later

with a series of simple examples of calculating the circumference and area of a circle. We will return to type signatures in the next section.

First off, we need π. We define `pi` as a floating point number. We give one equation that tells us how `pi` can be evaluated:

```
pi = 3.1415926
```

We know that the circumference of a circle with radius r is $2\pi r$. Let's write a function `circumference` that maps the radius into the circumference:

```
circumference r = 2 * pi * r
```

Function definitions look just like variable definitions, except multiple argument names (or patterns, which we will return to) follow the name to be defined.

For calculating the area of a circle, we define an auxiliary function that squares a floating point number. The characters `--` indicate a comment that continues to the end of the line.

```
-- raise a number to the second power
square x = x * x

-- compute area of circle, given radius
areaOfCircle r = pi * square r
```

Definitions need not be global. We can make local definitions using the familiar `let...in...` construct. As an example, the function `areaOfCircle` could've been written like this

```
areaOfCircle r =
  let pi = 3.1415926
      square x = x * x
  in pi * square r
```

Haskell allows defining new operators with a syntax similar to defining functions. The following example defines an infix operator `+/` that computes the average of two numbers

```
a +/ b = (a + b) / 2
```

Now the expression `4 +/ 2` evaluates to 3. The set of characters that may be used in infix operators is limited and defined in the Haskell Report [Mar, Section 2.4].

There are two more declaration-level syntactic elements to introduce. *Guards* are a shorthand for dividing function definitions into cases. As an example consider the following recursive definition of factorial:

```
factorial n
  | n == 0    = 1
  | otherwise = n * factorial (n-1)
```

A guard starts with the `|` character, followed by a *guard expression* (of type `Bool`), the `=` character and a *result expression*. The keyword `otherwise` stands for `True`. Cases are considered in order, and the result expression corresponding to the first expression evaluating to `True` is chosen as the return value of the function call.

A construct especially useful with guards is `where`. A `where`-clause introduces local definitions like `let ... in`, but the scope of the definitions range over all the guards. The following contrived definition of the predicate `even` demonstrates this:

```
even n
    | k == 0  = True
    | k == 1  = False
  where k = mod n 2
```

We can also rewrite `areaOfCircle` once more as:

```
areaOfCircle r = pi * square r
  where pi = 3.1415926
        square x = x * x
```

Finally we note that Haskell is sensitive to whitespace. Indentation is required to correspond to the structure of statements in a regular manner. The rules that govern whitespace are called *layout rules* and are documented in the Haskell Report [Mar, Section 2.7]. For the purpose of this text it is enough to remember that expressions and definitions that are on the same level syntactically should start from the same column (c.f. `pi` and `square`, `let` and `in` in the `areaOfCircle` example).

6.3 Types

Haskell is a statically typed language. We did not need to specify any types in the previous examples. This is because Haskell includes a powerful *type inference* algorithm that allows the implementation to ascertain the types of variables in the absence of type signatures. Although the type inference mechanism is powerful, it is customary to give type signatures to top-level definitions. They act as both documentation and help type inference in pinpointing the locations of type errors.

A type signature in Haskell uses is of the form `<name> :: <type>`. For example we could have defined `pi` in the previous section as:

```
pi :: Float
pi = 3.1415926
```

The type `Float` is the type of single-precision floating point numbers. The other basic arithmetic types of Haskell are

- `Int` for signed integers with a finite precision of at least 30 bits
- `Integer` for signed integers with infinite precision
- `Double` for double-precision floating-point numbers

The type of functions is denoted `<argument type> -> <result type>`. The following example adds type signatures to our `areaOfCircle` example:

```
-- raise a number to the second power
square :: Float -> Float
square x = x * x

-- compute area of circle, given radius
areaOfCircle :: Float -> Float
areaOfCircle r = pi * square r
```

As hinted earlier, functions with multiple arguments are equivalent to functions that return functions. Consider the following function on three arguments:

```
f a b c = if b then a else a + c
```

We can give `f` the type `Int -> (Bool -> (Int -> Int))`, as can be seen by the following chain of typings. When `f` is applied to three arguments of suitable types, the result is a number:

```
f 1 True 2 :: Int
```

The last of the arguments given to `f` is a number. Thus `f` applied to two arguments must be a function from numbers to numbers.

```
f 1 True   :: Int -> Int
```

By similar reasoning, `f` applied to one argument must be a function that returns a function from numbers to numbers

```
f 1        :: Bool -> (Int -> Int)
```

Luckily, the arrow `->` associates to the right so we can give `f` and its type signature as

```
f :: Int -> Bool -> Int -> Int
f a b c = if b then a else a + c
```

6.4 Higher-Order Functions

We have seen functions that return functions in the previous section. As a functional programming language Haskell also permits passing functions as arguments to functions.

The simplest higher-order function is perhaps one that applies a given function to an argument, for example

```
applyTo1 f :: (Int -> Int) -> Int
applyTo1 f = f 1
```

Now for example `applyTo1 (*2)` evaluates to 2.

As a more involved example let us consider the function `bothArgs`:

```
bothArgs :: (Float->Float->Float) -> Float -> Float
bothArgs f x = f x x
```

The `bothArgs` function is given a two-parameter function `f` and a value `x`. It returns the value of `f` applied to the two arguments `x` and `x`.

We can use `bothArgs` to for example square a number:

```
bothArgs (*) 3
```

evaluates to `9`. Used like this `bothArgs` seems a bit useless, but when partially applied, `bothArgs` is especially useful. We will demonstrate this with some additional code:

```
-- an approximation of the derivative of f at x
diff :: (Float->Float) -> Float -> Float -> Float
diff f h x = (f (x+h) - f x) / h
```

Now `diff (bothArgs (*)) 0.001 x` gives an approximation of the derivative of the function $t \mapsto t^2$ at `x`. We obtain an approximation of the second derivative of $t \mapsto t^2$ by using

```
diff (diff (bothArgs (*)) 0.001) 0.001 x
```

since we have

```
diff (bothArgs (*)) 0.001  ::  Float -> Float
```

Additional examples of higher-order functions will appear in the following sections.

6.5 Algebraic Data Types and Pattern Matching

Algebraic Data Types (ADTs) and Pattern Matching are devices used in some functional languages. They allow functions to be defined in a very declarative way.

An *Algebraic Data Type* is a type, consisting of a number of constructors, each with some number of (typed) fields. The syntax for ADTs used here resembles Haskell. ADT declarations are of the form

```
data <typename> =
   <constructorname> <fieldtype> <fieldtype> ...
 | <constructorname> <fieldtype> <fieldtype> ...
 | ...
```

An ADT with a number of zero-field constructors corresponds to an enumeration, e.g:

```
data Bool = True | False   -- actual Prelude definition of Bool
data TrafficLight = Red | Yellow | Green
```

An ADT with only one constructor with a number of fields corresponds to a C-like struct:

```
-- a report has an id, an author and contents
data Report = Report Int String String
```

The corresponding C-code is:

```
struct Report {
    int id;
    char *author;
    char *contents;
}
```

The fields of an ADT are not named by default, but we can give the relevant *accessor functions* names using *record syntax*. Here is the definition of `Report` using record syntax:

```
data Report =
  Report { getId :: Int,
           getAuthor :: String,
           getContents :: String }
```

This exposes the following functions that return the values of the corresponding fields.

```
getId :: Reoprt -> Int
getAuthor :: Report -> String
getContents :: Report -> String
```

ADTs can also be recursive, as this example of a binary tree with integer data in the nodes shows:

```
data Tree = Leaf | Node Int Tree Tree
```

An example value of type `Tree` is

```
Node 3 (Node 1 Leaf (Node 2 Leaf Leaf)) Leaf
```

a tree of height 3 with 3 inner nodes and 4 leaves.

Pattern matching is a way of defining functions case-by-case with the cases corresponding to constructors of the argument type. We'll start with some examples:

```
canMove :: TrafficLight -> Bool
canMove Green  = True
canMove Yellow = False
canMove Red    = False

treeHeight :: Tree -> Int
treeHeight Leaf                      = 0
treeHeight (Node value left right)  =
   1 + max (treeHeight left) (treeHeight right)
```

Thus a pattern both constitutes a check that the value in question was constructed with the given constructor, and binds the possible arguments to that constructor to names (c.f. `value`, `left` and `right` above). The special pattern "`_`" stands for no pattern: nothing is required of the argument and no bindings are made. That is, "`_`" marks an unused argument. For example

```
-- takes a TrafficLight and a speed, returns a speed
drive :: TrafficLight -> Float -> Float
drive Green  speed = speed  -- keep on driving
drive Yellow _     = 0      -- stop
drive Red    _     = 0      -- stop
```

The `case...of...` structure allows for pattern matching in expressions. It takes a sequence of cases in the form `<pattern> -> <expression>`. One could rewrite the definition of `treeHeight` using `case` as:

```
treeHeight t =
  case t of
    Leaf              -> 0
    Node _ left right -> 1 + max (treeHeight left) (treeHeight right)
```

Section 8 provides additional examples of algebraic types and pattern matching, especially the Wavelet tree implementation in Section 8.10.

Note that when using pattern matching, each case is an equation of its own. In contrast, cases defined by guards are all contained in the same equation. Indeed,

when guards and pattern matching are used together, pattern matching takes effect first. Again, see the code in Section 8.10 for examples.

Functional languages tend to support pattern matching against literals. We can replace guards with pattern matching in our previous `factorial` example:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

6.6 Strictness Annotations

Haskell is a lazy language. However it does offer primitives for strict (a.k.a. eager) evaluation. The most important of these is the `!` annotation when defining a data type. When the field type is prefixed with `!` this means that the value of the field is calculated immediately when the constructor is evaluated. Consider the two pair types defined below:

```
data Pair a = Pair a a
data SPair a = SPair !a !a
```

The following expression with the ordinary pair leaves the call to `fib` unevaluated:

```
let fstPlusOne :: Pair Int -> Int
    fstPlusOne (Pair a b) = a + 1
in fstPlusOne (Pair 0 (fib 1111111))
```

whereas if we substitute `SPair` for `Pair`, the call to `fib` will get evaluated right away when `fstPlusOne` pattern matches on its argument.

Strictness annotations are especially useful with data structures. Consider the strict version of the `Tree` type defined earlier:

```
data Tree = Leaf | Node Int !Tree !Tree
```

When the root of this tree is accessed, the whole structure of the tree gets evaluated (due to the strictness of the left and right child fields). This makes the performance of the tree datatype much more predictable, but also circumvents a nasty leak. The leak works like this. Consider a program that builds a set of integers over its course and in the end queries it. With a lazy tree implementation this would actually delay the building of the tree until it gets queried: the first query would trigger computing all the insertions and deletions that the program had performed. Additionally, the unevaluated operations would take up much more space than the tree-based set: linear in the amount of operations instead of the number of elements. For this reason both the Haskell standard library's `Data.Map` tree implementation and the `Tree` described in Section 8.7 later use strictness annotations.

For additional information on strictness, see Chapter 25 of Real World Haskell [OGS08].

6.7 Type System Features

In this section we introduce language devices which are used in Section 8: parametrized types and type classes. The code snippets in Section 8 also double as additional examples of the features discussed here.

The Haskell type system is based on the Hindley-Milner type system [Mar, Section 4.1]. Without going into further details the most important features of the Hindley-Milner system are parametrized types and (parametric) polymorphism (for more information on type systems, see e.g. Pierce [Pie02]). Hindley-Milner is popular among functional languages of the ML family since it offers relatively high expressiveness combined with efficient type inference.

Parametrized types are a common feature of ML-style functional programming languages. [CW85, Section 4.2] They allow one to define algebraic data types that are parametrized over some other type. As a simple example we offer `Maybe` from the Haskell Prelude:

```
data Maybe a = Just a | Nothing
```

Here `a` is a *type parameter*, that can be instantiated by applying the *type constructor* `Maybe` to some type. Thus for example the datatype `Maybe Int` behaves like the datatype

```
data IMaybe = IJust Int | INothing
```

In other words, `Maybe Int` has as values `Int`s tagged with the constructor `Just`, and also the special value `Nothing`. The `Maybe` type can be seen as a functional analog of indicating invalid return values with null as is usually done in C and Java.

One may draw a parallel between the usual use cases of C++ templates [Str00] and parametrized types. Both of these can be used to abstract data types over types they contain. For example the type of lists of integers might be denoted `List Int` in haskell and `List<Int>` in C++.

Polymorphism means that a function can accept values of different types as its argument. A classic example is subclass polymorphism in object-oriented languages: if C is a subclass of D, one can use an object of class C whenever an object of class D is required (this is the Liskov Substitution Principle [Lis87] rephrased). *Parametric polymorphism* [CW85] is polymorphism over type ranges with similar structure. For example the Haskell Prelude function `length` has a type of `[a]->Int`. In this type `a` is a *type variable*, which can be *instantiated* to any type. Thus `length` gives the length of any list, regardless of the element type. A type variable can be repeated, which simply means that the

type it is instantiated with occurs multiple times in the resulting type. Some examples of polymorphic types follow:

```
-- we can look at the constructor without
-- caring about the arguments
isJust :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
-- pairs and their accessors
data Pair a b = Pair a b
fst :: Pair a b -> a
snd :: Pair a b -> b
-- function composition
(.) :: (a->b) -> (c->a) -> (c->b)
```

We also could have defined the function `bothArgs` from Section 6.4 as

```
bothArgs :: (a->a->b) -> a -> b
bothArgs f x = f x x
```

making it polymorphic.

We can again compare these to C++ templates. For example the C++ equivalent of the `isJust` function might be declared

```
template <class A>
bool isJust(Maybe<A> x) ...
```

A useful feature of Haskell's type system is the `newtype` declaration. It introduces a wrapper type which is only present at compile-time: unwrapping and wrapping it generates no code at all. The syntax is like for `data`, but only one constructor with one field is allowed:

```
newtype Foo a = Foo (Bar Int a)
```

One of the most notable extensions Haskell makes to the standard Hindley-Milner system is *type classes*. Type classes are a feature unique to Haskell that allows ad-hoc overloading of functions or a simple form of dispatch by type. A type class is simply a collection of types that have an implementation for a certain set of operations. Type classes do not despite their name have any connection with the classes of object oriented programming languages.

As an example we provide the (somewhat contrived) type class `Empty` that provides a common abstraction for containers that can be empty. We define instances of `Empty` for lists and the data type `Maybe` introduced above.

```
class Empty a where     -- for every type a in the empty class
  empty :: a            -- there is an empty element (of type a)
  isempty :: a -> Bool  -- and also an isempty predicate

-- regardless of a we can tell whether Maybe a is empty
instance Empty (Maybe a) where
```

```
  empty = Nothing
  isempty (Just x) = False
  isempty Nothing  = True

instance Empty [a] where
  empty = []
  isempty []     = True
  isempty (x:xs) = False
```

A type class is somewhat like an interfrace in object-oriented languages. It defines a set of operations, whose implementations are then defined per type that implements the interface.

Wrapper types defined with **`newtype`** are especially useful in conjunction with type classes. They allow defining multiple instances over the same type as the following contrived example shows

```
newtype IMaybe = IMaybe (Maybe Int)
instance Empty IMaybe where
  empty = IMaybe (Just 0)
  isempty (IMaybe Nothing) = True
  isempty (IMaybe (Just 0) = True
  isempty _                = False
```

This instance does not overlap with the previous `Empty (Maybe a)` instance. Additionally we can switch between instances simply by wrapping or unwrapping.

Type classes are used to structure our implementation in Section 8.

When using functions from a type class, a *type class constraint* (also known as a *context*) must be added to the type of the function. A constrained type is of the form

```
(<classname> <type>, ...) => <type>
```

As an example here is the function `makeEmpty`:

```
makeEmpty :: (Empty a) => a -> a
makeEmpty x
  | isempty x  = x
  | otherwise  = empty
```

The type

```
(Empty a) => a -> a
```

can be read as "for all instances `a` of the type class `Empty`, this is a function from `a` to `a`". That is, the constraint `(Empty a) =>` constrains the concrete types with which the type variable `a` can be instantiated to.

Algebraic datatypes can also be constrained by type classes, but the semantics are slightly complicated. Refer to the Haskell Report [Mar, Section 4.2.1] for information.

Related to constraining is *superclassing*, which is not to be confused with object-oriented inheritance. The definition of a type class can have a type class constraint, contents of which are called superclasses. If a type is to be declared an instance of a type class it must also be an instance of the class's superclasses. One example is the `Measured` class presented later. A more contrived example can be obtained by extending the type class `Empty`:

```
class Empty a => Countable a where
  count :: a -> Int
```

Here the intention is of course that

```
count empty == 0
```

In simple use cases superclassing behaves much like inheritance between interfaces in object oriented languages.

Functions in type classes may also have default implementations (usually in terms of other functions in the class). For example we could have defined the `Countable` class in the following way:

```
class Empty a => Countable a where
  count :: a -> Int
  count x = if isempty x then 0 else 1
```

An oft-implemented extension to Haskell's basic type class mechanism is *multi-parameter type classes* [GHC, Section 7.6.1.1], in which a type class can have multiple parameters, and thus describe a connection between the classes.

```
class Bijection a b where
  in  :: a -> b
  out :: b -> a
```

6.8 Lists

Lists are important in functional programming. They are easy to handle recursively and additionally have nice operational properties under lazy evaluation: the list's contents can get computed as we iterate through it.

A polymorphic linked list can be implemented as a recursive algebraic datatype:

```
data List a = Cons a (List a) | Empty
```

That is, a list is either empty or a *cons cell*[6], containing a value – called the *head* – and a pointer to the *tail* – the rest of the list.

[6] a term originating from LISP

In Haskell lists have special syntax, but are operationally equivalent with the above ADT. The type `List a` is denoted `[a]`, the empty list constructor is `[]` and the infix operator `:` is the equivalent of `Cons`. The following code snippet gives implementations of some of the core list operations from the Haskell Prelude.

```
-- the accessors head and tail
head :: [a] -> a
head (x:xs) = x
tail :: [a] -> [a]
tail (x:xs) = xs

-- testing whether a list is empty
null :: [a] -> Bool
null []     = True
null (x:xs) = False

-- apply a function to each element of the list
map :: (a->b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs

-- return those elements for which a predicate is true
filter :: (a->Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

7 Monoidal Annotations and Annotated Trees

Suppose you need to design a generic module that combines a sequence of elements using a given combining function. This module could then be used for example to compute the sum of a sequence of numbers or the intersection of sets of words. The module needs to be efficient: you want to parallelize as much of the computation as possible, maybe even distributing large combination tasks over the network.

However, there is a problem: the result of the combination might depend on the ordering of the combination operations. You can guard against this by requiring that the supplied operation doesn't care about the order the operations are carried out in. The name of this property is *associativity*. For example almost all of the common arithmetic operations are associative[7]. Once one assumes that the combining operation is associative, one has free hands to organize

[7] exponentiation being the most notable exception: $a^{b^c} \neq (a^b)^c$

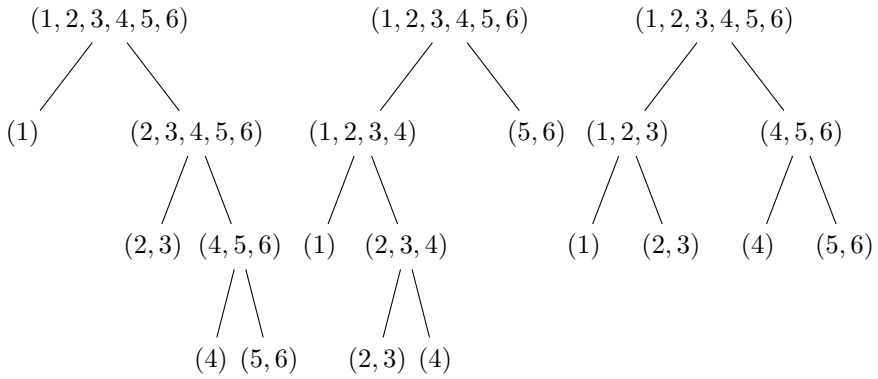


Figure 9: The associativity of catenation: different ways of computing $(1) \oplus (2,3) \oplus (4) \oplus (5,6)$

the computation of the combining in any way you see fit. See for example Figure 9.

Usually of course the sequence of values one wishes to combine is calculated from another sequence: we sum the lengths of the contents of a sequence of files or compute the maximum of the priorities of elements of a queue. Thus we have a general recipe for computing properties over a sequence of things:

1. Assign a value to each thing
2. Combine the values using an associative operation

This idea is called *monoidal annotations*.[8]

It is interesting to note that for example the basic operation of Google's processing framework MapReduce [DG08] resembles the recipe above. The *map* operation transforms a value into a pair $(id, annotation)$, after which the different annotations related to the same *id* are combined using the *reduce* operation. The *reduce* operation is required to be associative, for example a sum of integers (e.g. counting occurrences) or a union of sorted lists (e.g. aggregating words that occur in a given context).

Let us go through monoidal annotations once more, this time formally. A monoid is the mathematical term for an associative binary operation with a neutral element. For example, addition and maximum are monoids over the non-negative integers with 0 as the neutral element.[9] Stated more formally, $(X, \circ)$ is a monoid if

[8] Monoidal annotations as a term is quite well-known in functional programming folklore but does not have much exposure in the academic setting. However Steele's talk [Ste09] on the subject is excellent.

[9] Hinze and Paterson [HP06] offer multiple additional examples of useful monoidal annotations in their paper.

1. $\circ : X \times X \to X$ is a binary function on X
2. $x \circ (y \circ z) = (x \circ y) \circ z$ for all $x, y, z \in X$.
3. There exists an $e \in X$ such that $x \circ e = e \circ x = x$ for all $x \in X$.

A monoidal annotation a is a function that maps a set of elements into a monoid. These annotations can be extended to sequences of elements in a natural way: the annotation for the sequence $(x_0, x_1, \ldots, x_n)$ is

$$a((x_0, x_1, \ldots, x_n)) = a(x_0) \circ a(x_1) \circ \cdots \circ a(x_n).$$

We can omit parentheses on the right side of this equation because of the associativity of $\circ$.

As an example, consider a set of processes with priorities. We can consider the priorities as annotations from the $(\mathbb{N}, \max)$ monoid. The annotation for a sequence of processes is simply the maximum priority over that sequence.

The generic operation *find* is helpful in implementing queries over annotated sequences:

find$_a(S, p)$ Given an annotation a and a predicate p over the annotation type X find a split $S = bxr$ where x is a single element such that $\neg p(a(b))$ and $p(a(bx))$.

Consider the previous example of a sequence S of processes annotated with their priorities. Let $p_k(n)$ be the predicate $n \geq k$. Now if *prio* is the priority annotation, $\mathit{find}_{\mathit{prio}}(S, p)$ returns a split bxr such that $\max_{x \in B} \mathit{prio}(x) < k$ and $\mathit{prio}(x) \geq k$.

A predicate p over a monoid $(X, \circ)$ is called monotonic if $p(x)$ implies $p(x \circ y)$ for all $y \in X$. For a monotonic predicate p, there exists at most one split for $\mathit{find}(S, p)$ to uncover. Limiting ourselves to monotonic predicates allows for a clean and efficient implementation for *find*. For example the predicate $x \mapsto x \geq k$ is monotonic over the monoids $(\mathbb{N}, +)$, $(\mathbb{N}, \max)$ and $(\mathbb{Q}_+, \cdot)$. Also, the predicate p_k from the previous example is monotonic. As a final example, the predicate "the element x occurs in the given list" over the monoid of lists with catenation as the operation is monotonic.

The benefit of monotonic predicates is that one can easily perform $\mathit{find}(S, p)$ by a binary search over prefixes of S when p is monotonic.

Often one is interested in multiple annotations on the same data, such as the *size* and *ones* annotations in the next section. The mathematical abstraction corresponding to this is the *product monoid*. The product of the monoids $(A, +)$ and $(B, \circ)$ is the monoid $(X, \oplus)$ with the following properties:

1. The set of elements $X = A \times B$ is the set of tuples (a, b) such that $a \in A$ and $b \in B$.

2. The operation $\oplus$ is defined by

$$(a, b) \oplus (a', b') = (a + a', b \circ b')$$

Similarly, the we can define the product of the annotations $a : Y \to A$ and $b : Y \to B$ as the annotation

$$x : Y \to X,\ x(y) = (a(y), b(y)).$$

Thus the product annotation into the product monoid just keeps track of both annotations side-by-side.

For example by combining the $(\mathbb{N}, +)$-annotation $\mathit{size}(x) = 1$ and a $(\mathbb{N}, \max)$-annotation *priority* we get a *priority search queue*: we can index the queue and retrieve the element with the maximum priority. We can even obtain the maximum priority over the first k elements: first we get the k-length prefix of the string using *find* over the *size* annotation and then we measure the result with the *priority* annotation.

7.1 Rank and Select via Monoids

We can reformulate the operations on bit vectors from Section 3 using annotations. Let us consider two monoidal annotations for binary elements $\{0, 1\}$.

1. The *size* annotation: each element has measure 1 and measurements are combined with simple addition. Thus the measure of a binary sequence is its length.
$$\mathit{size} : \{0, 1\} \to \mathbb{N},\ \mathit{size}(x) = 1$$

2. The *rank* annotation for binary sequences: the measure is 0 for bits with value 0 and 1 for bits with value 1. Measures are again combined with addition. The measure of a sequence becomes the number of 1-bits it holds.
$$\mathit{rank} : \{0, 1\} \to \mathbb{N},\ \mathit{rank}(1) = 1,\ \mathit{rank}(0) = 0$$

We define the predicate s_i over the size annotation and r_i over the rank annotation. Both $s_i(k)$ and $r_i(k)$ are true when $k > i$. That is, s_i is the predicate "there are at least i bits" and r_i is the predicate "there are at least i ones".

We can implement *query*, *rank* and *select* using *find* and the predicates s_i and r_i.

- To compute $\mathit{query}(S, i)$, perform $\mathit{find}_{\mathit{size}}(S, s_i)$ to find the element x with index i in S.
- To compute $\mathit{rank}(S, i)$, perform $\mathit{find}_{\mathit{size}}(S, s_i)$ to find a split $S = bxa$. Now x is the element in S at index i, so $\mathit{rank}(bx)$ is the answer.

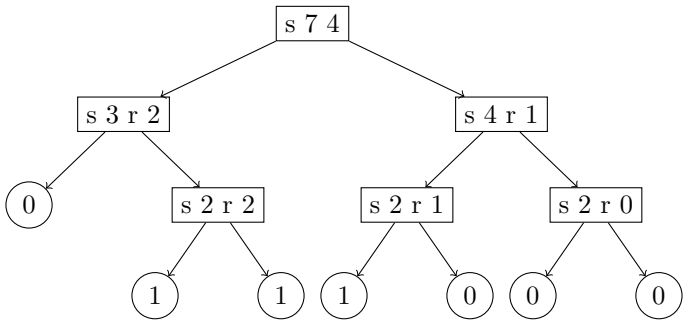


Figure 10: The annotated tree of the sequence 0111000 under the *sizerank* annotation.

- To compute $select(S, i)$, perform $find_{rank}(S, r_i)$ to find a split $S = bxa$. Now x is the ith one in S, so $size(bx)$ is the answer.

Thus to support all of these operations we will use the product annotation *sizerank*:

$$sizerank : \{0,1\} \to \mathbb{N},\ sizerank(x) = (size(x), rank(x))$$

7.2 Data Structures

We mentioned that *find* for monotonic predicates is merely a binary search over prefixes. A natural data structure that uses this idea is the *annotated tree*. It is a (binary) tree in which the data (elements of the sequence) are stored with their annotations in the leaves. Each inner node stores the monoid sum of the annotations of its children. This means that each inner node stores the annotation of the subsequence rooted at that node. See Figure 10 for an example.

Now the binary search over prefixes turns into a standard tree search. The algorithm for *find* is as follows. Let $(\oplus, e)$ be the annotation monoid.

1. Start at the root node. Assign $a := e$.
2. If at a leaf node with annotation l and element x: if $p(a \oplus l)$ then x is the element at which to split the sequence.
3. Otherwise, let l be the annotation of the left child and r the annotation for the right child.
   - If $p(a \oplus l)$, then continue search at the left child.
   - If $p(a \oplus l \oplus r)$, then set $a := a \oplus l$ and continue search at the right child.
   - Otherwise fail

Another way to implement annotated sequences is the *finger tree*. Finger trees are a functional data structure invented by Hinze and Paterson [HP06]. They resemble annotated 2-3-trees but in addition support efficient access to both ends of the sequence and efficient catenation of sequences.

We will use both annotated trees and finger trees to implement dynamic bit vectors in Section 8.

8 Implementations

In this section we aim to give a somewhat detailed overview of the Haskell library implemented as part of this thesis. Functions from the Haskell standard library The Prelude are explained as needed.

This section omits most details of constructing the implemented structures. Naive construction algorithms were of course implemented but they are relatively straightforward and would only clutter the presentation of the main matter. Efficient construction of compressed bit vector structures is a research topic of its own.

The implementation is available in full at

`http://github.com/opqdonut/bitvectors` [10]

The version of GHC used was 6.12.

8.1 Overview

We aim to implement static and dynamic bit vectors as outlined in Section 3. These will serve as a basis for implementing some of the applications described in Section 4.

Our static bit vector implementation is based loosely on ideas by Raman et al [RRR02] and the implementation in the RLCSA library [MNSV10]. Another good source is Mäkinen et al [MN07]. The basic idea is to store the compressed bit vector along with a number of array-based indices that make fast queries possible.

Implementations for dynamic bit vector use ideas from Blandford and Blelloch [BB04] (another source is Gerlach [Ger07]). They store a compressed bit vector as a sequence of gap-encoded blocks contained in a tree structure. We implemented two versions of this idea. The first one uses a custom balanced annotated tree implementation. The second one uses a finger tree (see section 7), as implementated by Hinze and Paterson and described in their paper [HP06].

[10] revision `449c6a6dc5456e7b2b2c`

The implementation is split into modules as outlined below. The structure of this chapter also follows the module structure.

module `BitVector` defines the `BitVector` and `DynamicBitVector` type classes that describe operations on bit vectors. It also implements the `Gap` type for representing gap-encoded bit vectors.

module `Measure` implements the `SizeRank` monoid (see Section 7.1) for monoidal rank and select.

module `Encoding2` implements integer codes (see Section 2.2) and the following bit containers

type `Block` a packed bit array

type `UBlock` an uncompressed bit vector (simple wrapper around `Block`)

type `EBlock` a gap-encoded bit vector using δ-encoding

type `NBlock` a gap-encoded bit vector using nibble encoding

module `Static2` implements `SuccinctArray` and using it a static compressed bit vector `Static`

module `SmallBlock` implements efficient constant-size bit vectors:

type `SmallBlock` is uncompressed

type `SmallElias` is δ-encoded

module `Tree` implements a balanced binary tree with monoidal annotations (see Section 7) (type `Tree`), and a dynamic bit vector, `Dynamic` on top of this.

module `FingerTreeDynamic` implements a dynamic bit vector `FDynamic` using Finger Trees (see Section 7).

module `Wavelet` implements a wavelet tree (type `Wavelet`) that can use any `BitVector` instance.

8.2 Bit Vectors and Gaps

The interfaces to static and dynamic bit vectors are captured by the type classes `BitVector` and `DynamicBitVector`, as shown below. The default implementation of `queryrank0` is given by an equation in Section 3. The default implementation of `deconstruct` uses the `[a..b]` syntax to produce a list of all integers from `a` to `b`.

```
class BitVector a where
  query :: a -> Int -> Bool
  queryrank  :: a -> Int -> Int
  select :: a -> Int -> Maybe Int
```

```
  queryrank0 :: a -> Int -> Int
  queryrank0 a i = i - queryrank a i + 1

  querysize :: a -> Int

  deconstruct :: a -> [Bool]
  deconstruct b = map (query b) [0 .. querysize b - 1]

class DynamicBitVector a where
  -- returns version of structure with element added:
  insert :: a -> Int -> Bool -> a
  -- returns version of structure with element deleted:
  delete :: a -> Int -> a
```

The return type of `select` is `Maybe Int`: asking for the index of the ith 1 when the vector only has $k < i$ 1's is undefined. In this case `select` returns `Nothing`. It is also of course possible that a `query` or `queryrank` is given an invalid index but it is more reasonable to expect the user of these functions to know the length of the vector than the number of ones in it.

The type class `Construct` is for constructing bit vectors. Note that the operations `construct` (which takes a length argument) and `construct'` (which doesn't) have default implementations in terms of each other. The type class `BlockSize` is for bit vector implementations that are paremeterized by a block size.

```
class Construct a where
  construct  :: Int -> [Bool] -> a
  construct _ xs = construct' xs

  construct' :: [Bool] -> a
  construct' xs = construct (length xs) xs

class BlockSize a where
  queryBlockSize :: a -> Int
  constructWithBlockSize :: Int -> [Bool] -> a
```

We wish to support efficient operations of gap-encoded bit vectors. This is why we implement a simple gap-based representation for bit strings.

A single gap is represented by a value of type `Gap`. The type `Gap` is just a wrapper around `Int`. The value `Gap n` represents the bit sequence 0^n1. A bit vector

$$0^{x_0}10^{x_1}1\ldots10^{x_k}$$

can be represented as the list `[Gap x0,Gap x1,...,Gap xk]`. Note how the final gap consists of only zeros without the terminating 1.

The function `gapify` translates between a bit vector (represented as `[Bool]`) and its gap representation (`[Gap]`).

```
newtype Gap = Gap {unGap :: Int}

gapify :: [Bool] -> [Gap]
gapify xs = loop xs 0
  where loop [] acc         = [Gap acc]
        loop (True:xs) acc  = Gap acc : loop xs 0
        loop (False:xs) acc = loop xs (acc+1)
```

The `BitVector` operations are simple loops. The loops are written out as recursive helper functions named `loop`. Recall that the function `null :: [a] -> Bool` tests whether a list is empty. The impure `error` function terminates the program with the given message.

```
instance BitVector [Gap] where

  querysize gs = sum (map ((+1).unGap) gs) - 1

  query gaps index = loop index gaps
    where loop left (Gap gap:gaps)
            | gap<left  = loop (left-gap-1) gaps
            | gap==left && not (null gaps) = True
            | gap>left  = False
            | otherwise = error "Query␣past␣end"

  queryrank gaps index = loop index 0 gaps
    where loop left ones (Gap gap:gaps)
            | gap<left  = loop (left-gap-1) (ones+1) gaps
            | gap==left && not (null gaps) = ones+1
            | gap>left  = ones
            | otherwise = error "Rank␣past␣end"

  select gaps index = loop 0 index gaps
    where loop _    _    [] = Nothing
          loop bits ones (Gap gap:gaps)
            | ones>0  = loop (bits+gap+1) (ones-1) gaps
            | ones==0 && not (null gaps) = Just (bits+gap)
            | otherwise                  = Nothing

  deconstruct gs = unGapify gs
```

The `DynamicBitVector` instance functions are of the same form. Note that we handle the quite different cases of inserting a 1 and inserting a 0 separately.

```
instance DynamicBitVector [Gap] where
  insert gaps index False = loop gaps index
    where loop (Gap gap:gaps) index
            | gap <  index = Gap gap : loop gaps (index-gap-1)
            | gap >= index = Gap (gap+1) : gaps
          loop [] _ = error "Insert␣past␣end!"
```

```
  insert gaps index True = loop gaps index
    where loop (Gap gap:gaps) index
            | gap <  index = Gap gap : loop gaps (index-gap-1)
            | gap >= index = Gap index : Gap (gap-index) : gaps
          loop [] _ = error "Insert␣past␣end!"

  delete gaps index = loop gaps index
    where loop (Gap gap:gaps) index
            | gap <  index = Gap gap : loop gaps (index-gap-1)
            | gap == index =
                case gaps of
                  [] -> error "Delete␣past␣end!"
                  (Gap gap' : gaps') -> Gap (gap+gap') : gaps'
            | gap >  index = Gap (gap-1) : gaps
          loop [] _ = error "Delete␣past␣end!"
```

8.3 The `SizeRank` Monoid

The `Monoid` type class is used to represent monoids in Haskell. The `Measured` type class describes a monoidal annotation (see Section 7).

```
class Monoid a where
  -- the identity element for mappend
  mempty :: a
  -- an associative operation
  mappend :: a -> a -> a
  -- sum over list, default implementation with mappend
  mconcat :: [a] -> a

class (Monoid v) => Measured v a where
  measure :: a -> v
```

As discussed in Section 7.1, we use the combined `SizeRank` annotation to support rank and select queries. Here is the type and the relevant instances.

```
-- this is essentially "data SizeRank = SizeRank Int Int"
data SizeRank = SizeRank {getSize :: !Int,
                          getRank :: !Int}

instance Monoid SizeRank where
    mappend (SizeRank a a') (SizeRank b b') =
        SizeRank (a+b) (a'+b')
    mempty = SizeRank 0 0

instance Measured SizeRank Bool where
    measure True  = SizeRank 1 1
    measure False = SizeRank 1 0

instance Measured SizeRank [Bool] where
```

```
  measure xs = mconcat (map measure xs)
```

We measure a bit string represented as a list of booleans (type `[Bool]`) simply by measuring each bit and combining the results using `mconcat`.

We also have instances for gap representations. Lists of gaps are measured by measuring each gap and summing the results using `mconcat`, after which the absence of the final 1 must be accounted by subtracting 1 from both size and rank.

```
instance Measured SizeRank Gap where
  measure (Gap gap) = SizeRank (gap+1) 1

instance Measured SizeRank [Gap] where
  -- the last gap has no final 1
  measure gs = let SizeRank s r = mconcat (map measure gs)
               in SizeRank (s-1) (r-1)
```

The predicates we use for searching `SizeRank`-annotated sequences are the ones mentioned in Section 7. These definitions use the function composition operator `(.)` and a section on the operator `>` (see Section 7).

```
index :: Int->SizeRank->Bool
index i = (>i) . getSize
rank :: Int->SizeRank->Bool
rank i = (>i) . getRank
```

A less idiomatic (but equivalent) way of writing these functions would have been

```
index i sr = getSize sr > i
rank i sr  = getRank sr > i
```

Partially applying these functions yields the needed predicates. For example,

```
rank 10 :: SizeRank -> Bool
```

is the predicate that finds locates the 10th 1 in the bit vector.

In some cases we want to cache measurements, and thus we implement the type `Cached`. It is simply bundles together a reference to the value and its measurement. The function `cached` builds a `Cached` and the accessor function `unCached` retrieves the wrapped object. The measurement can be retrieved simply with a call to `measure` since we implement the natural instance of `Measured`.

```
data Cached a v = Cached {cmeasure :: !a, unCached :: !v}
instance Monoid a => Measured a (Cached a v) where
  measure = cmeasure

cached :: Measured a v => v -> Cached a v
cached x = Cached (measure x) x
```

8.4 Codes and Blocks

This section summarises the types defined for bit-level manipulations in the module `Encoding2`.

Codes are short bit strings that are used as the basis of the various implemented encodings. Codes can be catenated (`+++`) and sliced (`takeCode` and `dropCode`).

The following definition of `Code` and related functions uses the following functions:

```
shiftL :: Bits a => a -> Int -> a -- left shift by given amount
shiftR :: Bits a => a -> Int -> a -- right shift
(.|.)  :: Bits a => a -> a -> a   -- bitwise or
ones   :: Bits a => Int -> a  -- a value with the n lowest bits set
```

```
-- A Code is a smallish chunk of bits
data Code = Code {codelength :: !Word8, code :: !Word64}

(+++) :: Code -> Code -> Code
a +++ b
  | codelength a + codelength b > 64 = error "out␣of␣space"
  | otherwise = Code
                (codelength a + codelength b)
                (shiftL (getCode b) (fromIntegral $ codelength a)
                 .|. getCode a)

takeCode :: Int -> Code -> Code
takeCode a (Code l c) = Code (fromIntegral a `min` l) (c .&. ones a)

dropCode a (Code l c)
  | a > fromIntegral l  = Code 0 0
  | otherwise =  Code (l-(fromIntegral a)) (c `shiftR` a)
```

Blocks represent bit strings of arbitrary length. They can be built from codes. A `UArray Int Word8` means an unboxed array of bytes, i.e. a contigous segment of memory.

```
newtype Block = Block (UArray Int Word8)

-- catenate codes into a block
makeBlock :: [Code] -> Block
-- implementation omitted

-- read code of given length from given index
readCode :: Block -> Int -> Int -> Code
-- implementation omitted
```

For performance reasons we also implemented so called small blocks of a constant bit length. A `SmallBlock` consists of one 64 bit code, and can thus be efficiently

manipulated as a whole on modern computers. `SmallBlock` acts as a counterpart to the decision in some implementations to fix $\log n$ as a compile-time constant (e.g. Gerlach's [Ger07]).

```
newtype SmallBlock = SmallBlock Code
                   deriving Show

instance BitVector SmallBlock where
  -- omitted, low-level code
```

8.5 Encoded Blocks

For compressing bit vectors we implemented gap encoding coupled with Elias δ-encoding and with the nibble encoding (see Section 2.2). These encodings are implemented as the following functions

```
elias_encode :: Gap -> Code
nibble_encode :: Gap -> Code
```

We use a set of simple wrappers around `Block` to represent simple compressed sequences of bits. For benchmarking purposes we also offer an unencoded bit vector. The types and their various instances are summarized below. The type `UBlock` needs an additional bitlength field since `Block`s consist of whole bytes. We implemented terminators for the encodings, the unencoded bit needs an explicit length.

```
-- gap + elias
newtype EBlock = EBlock {unEBlock :: Block}
-- gap + nibble
newtype NBlock = NBlock {unNBlock :: Block}
-- unencoded
data UBlock = UBlock {umeasure :: !SizeRank, unUBlock :: !Block}
```

Naturally we know how to create and open encoded blocks. The class `Encoded` embodies provides methods for this. The methods `encodedSize`, `combine`, `cleave` are used when balancing the dynamic structure described in Section 8.9. The function `encodeMany` takes a length and encodes a sequence of gaps into blocks of the given length. It is used for constructing bit vectors.

```
class Encoded a where
  decode :: a -> [Gap]
  encode :: [Gap] -> a
  encodedSize :: a -> Int

  encodeMany :: Int -> [Gap] -> [a]

  combine :: a -> a -> a
  cleave :: a -> (a,a)
```

with relevant instances for `EBlock`, `NBlock` and `UBlock`.

Using the functions that read the two encodings,

```
readEliass :: Block -> [Gap]
readNibbles :: Block -> [Gap]
```

the following instances for `EBlock` and `NBlock` are easy to define. These code snippets make extensive use of the function composition operator

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

The unaccustomed reader may want to expand definitions of the form

```
f = a . b . c
```

to the equivalent form

```
f x = a (b (c x))
```

in his or her head.

The instances simply delegate to the `[Gap]` instances.

```
instance Measured SizeRank EBlock where
  measure = measure . readEliass . unEBlock

instance Measured SizeRank NBlock where
  measure = measure . readNibbles . unNBlock

instance Measured SizeRank UBlock where
  measure = umeasure
```

```
instance BitVector EBlock where
  query     (EBlock b) i = query (readEliass b) i
  queryrank (EBlock b) i = queryrank (readEliass b) i
  select    (EBlock b) i = select (readEliass b) i
  querysize = querysize . readEliass . unEBlock

instance DynamicBitVector EBlock where
  insert (EBlock b) i val = encode newGaps
    where newGaps = insert (readEliass b) i val
  delete (EBlock b) i = encode newGaps
    where newGaps = delete (readEliass b) i

instance BitVector NBlock where
  -- just uses readNibbles instead of readEliass

instance DynamicBitVector NBlock where
  -- just uses readNibbles instead of readEliass

instance BitVector UBlock where
```

```
  -- omitted

instance DynamicBitVector UBlock where
  -- omitted
```

The query operations for `EBlock` and `NBlock` directly use the gap lengths instead of fully decoding the vector. This means that the operations only have to iterate through the $O(nH_0)$ gaps instead of the n bits.

The space requirement for `EBlock` is $nH_0 + O(1)$ and for `NBlock` $\frac{4}{3}nH_0 + O(1)$, as the analysis of the encodings in Section 2.2 shows.

8.6 Small Encoded Blocks

We also implemented gap-encoded variants of `SmallBlock`.

```
newtype SmallBlock = SmallBlock Word64
```

Constructing small encoded blocks is done in a different fashion from the `encode` function from the `Encoded` type class:

```
-- encodes a list of Gaps into Codes that are as fully populated as
-- possible
packElias :: [Gap] -> [Code]
```

The `BitVector` instance again simply delegates to the instance for `[Gap]`

```
smallEliasToGaps :: SmallElias -> [Gap]
-- implementation omitted

instance Measured SizeRank SmallElias where
  measure = measure . smallEliasToGaps

instance BitVector SmallElias where
  deconstruct = unGapify . smallEliasToGaps

  query s i = query (smallEliasToGaps s) i
  queryrank s i = queryrank (smallEliasToGaps s) i
  select s i = select (smallEliasToGaps s) i
  querysize s = querysize (smallEliasToGaps s)
```

8.7 Dynamic Bit Vector

The `Tree` type is an annotated binary tree (see Section 7). Here are the data definition and the straightforward `Measured` instance:

```
data Tree a v = Empty
              | Leaf {measureLeaf :: !a, val :: v}
              | Node {left :: !(Tree a v),
                      right :: !(Tree a v),
                      measureNode :: !a}
              deriving Show

instance Measured a v => Measured a (Tree a v) where
  measure Empty = mempty
  measure (Leaf a _) = a
  measure (Node _ _ a) = a
```

The functions `leaf` and `node` are "smart constructors" for `Leaf` and `Node`: they calculate the stored annotation using `measure`.

```
leaf :: Measured ann val => val -> Tree ann val
leaf v = Leaf (measure v) v

node :: Measured ann val =>
        Tree ann val -> Tree ann val -> Tree ann val
node Empty r = r
node l Empty = l
node l r = Node l r (measure l +++ measure r)
```

The function `find` performs annotation-based lookups and is used to implement the bit vector operations. For our purposes it is enough to return the found element v and the annotation of the sequence of elements preceding v. As `find` is a straightforward tree search, it takes $O(h)$ time where h is the height of the tree.

```
find :: Measured a v => (a -> Bool) -> Tree a v -> Maybe (a,v)
find p t = go mempty t
  where
    go acc (Leaf ann v)
        | p (acc +++ ann) = Just (acc,v)
        | otherwise = Nothing
    go acc (Node l r ann)
        | p (acc +++ measure l) = go acc l
        | p (acc +++ ann) = go (acc +++ measure l) r
        | otherwise = Nothing
```

A `BitVector` instance for `Tree` is straightforward to define. We simply use `find` and delegate to the `BitVector` instance of the element type. The implementation of `select` does slightly more checking. The operations *query*, *rank* and *select* achieve a time performance of $O(\log(n/b)+\log b)$ where b is the block size.

```
instance (Measured SizeRank a, BitVector a) =>
         BitVector (Tree SizeRank a) where


  query t i = query block (i-s)
```

```
    where Just ((SizeRank s r),block) = find (index i) t

  queryrank t i = r + queryrank block (i-s)
    where Just ((SizeRank s r),block) = find (index i) t

  select t i
    | i >= getRank (measure t) = Nothing
    | otherwise =
      case find (rank i) t
      of Just (SizeRank s r, block) -> fmap (+s) $ select block (i-r)
         Nothing -> Nothing

  querysize = getSize . measure
```

To implement insertions and deletions we need the operation `modify` that searches for a leaf using annotations like `find` but also rebuilds the tree on the way. The parameter `f` is a function that transforms the found leaf. Using `modify` we can implement simple non-balancing insertions and deletions for `Tree`-based bit vectors: we find the leaf containing the index to be inserted/deleted, do the insertion and rebuild the tree to update annotations. Both *insert* and *delete* are take time $O(\log(n/b) + \log b)$, just like the previous operations.

```
modify :: Measured a v =>
          (a -> Bool) -> ((a,v) -> v) -> Tree a v -> (Tree a v)
modify p f t = go mempty t
  where
    go acc (Leaf ann v)
      | p (acc +++ ann) = leaf $ f (acc,v)
      | otherwise = error "modify␣failed!"
    go acc (Node l r ann)
      | p (acc +++ measure l) = node (go acc l) r
      | p (acc +++ ann)       = node l (go (acc +++ measure l) r)
      | otherwise = error "modify␣failed!"


instance (Measured SizeRank a, DynamicBitVector a) =>
         DynamicBitVector (Tree SizeRank a) where

  insert t i v = modify (index i) insertIntoLeaf t
    where insertIntoLeaf (SizeRank s r,block) = insert block (i-s) v

  delete t i = modify (index i) deleteFromLeaf t
    where deleteFromLeaf (SizeRank s r,block) = delete block (i-s)
```

We introduce the type alias

```
type Dynamic a = Tree SizeRank a
```

Now the types `Dynamic UBlock` and `Dynamic SmallBlock` represent uncompressed bit vectors and `Dynamic EBlock`, `Dynamic NBlock` and `Dynamic SmallElias` repre-

sent compressed bit vectors.

The space required by the tree is $O(m)$ where m is the amount of leaves. Thus the total space requirement of `Dynamic EBlock` is

$$2nH_0 + O(n/\log n)$$

since the leaves can be half-full in the worst case.

8.8 Static Compressed Bit Vector

Our static bit vector, `Static`, stores a `Block` along with indices that facilitate fast queries. The idea of the indices is to store the results of *rank* and *select* queries every $\log n$ bit positions, so that the queries can be completed by iterating over only $\log n$ bits.

The problem one encounters is that the space needed by the indices becomes $O(n/\log n)O(\log n) = O(n)$ if implemented naively – ruining the compressed aspect of the vector.

For this reason we implement a difference-encoded succinct array, `SuccinctArray`. A `SuccinctArray` stores a sequence of integers by storing a number of base values from the sequence, and representing the other integers as differences from a base value.[11]

In practical terms we choose a *stride*. Every stride'th value in the sequence is stored in full, and the values in between are stored as differences to the previous full value.

This is the implementation of `SuccinctArray`. The Prelude type `UArray i a` is an unboxed array of values of type `a`, indexed by a range of values of type `i`. The operator `!` is the indexing operator for `UArray`s. The Prelude function `divMod` returns the result of integer division and the modulus. The Prelude function `fromIntegral` performs conversions between instances of the `Integral` class, `Int` and `Int16` in this case.

```
-- we store data in unboxed arrays
type V = UArray Int

-- Big is used to store the full values
type Big = Int
-- Small is used to store the differences
type Small = Int16

data SuccinctArray =
  SuccinctArray
  {stride :: !Int,
```

[11] This is closely related to the block-superblock idea used in other implementations (e.g. [RRR02] [MN07])

```
   big :: !(V Big),
   small :: !(V Small)}
  deriving Show

mkSuccinctArray ::
  Int -> [Big] -> SuccinctArray
mkSuccinctArray stride vals = -- omitted

(!-) :: SuccinctArray -> Int -> Big
(SuccinctArray stride big small) !- i =
  base + fromIntegral offset
  where (bigI,smallI) = i `divMod` stride
        base = big ! bigI
        offset = small ! (bigI*stride + smallI)
```

We can now implement the static bit vector. We store a encoded `Block` along with three indices:

`locations` maps positions in the original bit vector to the beginnings of codewords in the compressed bit vector;

`offsets` tells how many bits to skip after decoding the codeword indicated by `locations` in order to get to the indexed bit;

`ranks` tells the rank of the indexed bit.

All of these indices are stored every `blockSize` $(10 \log n)$ bit locations. In addition they are represented using a `SuccinctArray` with a stride of $\log n$. Thus full rank, location and offset values are stored every $10(\log n)^2$ locations, and difference values are stored every $10 \log n$ locations. This brings the total space usage to

$$nH_0 + O\left(b\frac{n}{(\log n)^2} + s\frac{n}{\log n}\right)$$

where b is bitsize of `Big` and s the bitsize of `Small`. This gives $nH_0 + o(n)$ when b and s are chosen suitably. In the code however they are constants: $b = 32$ and $s = 16$.

The datatype `Static` and the implementations of `query` and `queryrank` are given below. The operations merely read the relevant indices, use the function

```
readEliass' :: Block -> Int -> [Gap]
```

to read the gaps from a block starting at the given index, and then delegate to the `BitVector` instance of `[Gap]`. The Prelude function `div` is for truncated integer division.

```
data Static =
  Static {
   sbitlength :: !Int,
   compressed :: !Block,
```

```
   blockSize :: !Int,
   ranks :: !SuccinctArray,      -- i -> rank(B,i*blockSize)
   locations :: !SuccinctArray,  --\ mapping from unencoded locations
   offsets :: !SuccinctArray     --/ to encoded locations
 }
 deriving Show

instance BitVector Static where
  query = _query
  queryrank = _queryrank
  select = _select
  querysize = sbitlength
```

The *query* and *rank* operations for `Static` are straightforward lookups. We find the block that contains the sought index and then delegate to the `BitVector [Gap]` instance.

```
_query :: Static -> Int -> Bool
_query static i =
  let arrayIndex = i `div` blockSize static
      -- i' is the index for which we can get location and offset
      i' = arrayIndex * blockSize static
      -- we start decoding here
      location = locations static !- arrayIndex
      -- total number of bits to skip from decoded stream
      offset = (offsets static !- arrayIndex) + (i-i')
      gaps = readEliass' (compressed static) location
  in
   query gaps offset

_queryrank :: Static -> Int -> Int
_queryrank static i =
  let arrayIndex = i `div` blockSize static
      i' = arrayIndex * blockSize static
      location = locations static !- arrayIndex
      offset = (offsets static !- arrayIndex) + (i-i')
      baseRank = ranks static !- arrayIndex
      gaps = readEliass' (compressed static) location
  in
   baseRank + queryrank gaps offset
```

The time complexity of the *query* and *rank* operations is $O(b)$ where b is the block size. They do a constant amount of work and then iterate through at most a blockful of gaps.

The *select* operation is implemented as a binary search over the `ranks` index. This is slightly more efficient than a straightforward binary search over *rank* queries.

```
binarySearch :: (Int -> Bool) -> Int -> Int -> Int
```

```
binarySearch tooBig min max
  | max==min   = min
  | max-min==1 = min
  | tooBig mid = binarySearch tooBig min mid
  | otherwise  = binarySearch tooBig mid max
    where mid = (min + max) `div` 2

_select :: Static -> Int -> Maybe Int
_select static i =
  let tooBig ind = ranks static !- ind >= i
      arrayIndex = binarySearch tooBig 0 (saLength $ ranks static)
      baseRank = ranks static !- arrayIndex
      baseIndex = blockSize static * arrayIndex
      location = locations static !- arrayIndex
      offset = offsets static !- arrayIndex
      gaps = readEliass' (compressed static) location
  in
   --- this actually works because the offset bits that should be
   --- discarded are always zeros
   do blockInd <- select gaps (i - baseRank)
      return $ baseIndex + blockInd - offset
```

The time complexity of *select* is $O(\log(n/\log n))$ since the indexes are stored every $O(\log n)$ locations.

8.9 Finger Tree -based Dynamic Bit Vector

As outlined previously, in this second Dynamic Bit Vector we store blocks measured by `SizeRank` annotations in a finger tree.

We use a length of $8 \log n$ for the blocks. This means that the finger tree has $n/(8 \log n)$ elements. We achieve $O(\log n)$ time for *query*, *rank* and *select* since finding the wanted `Block` takes $O(\log(n/\log n)) = O(\log n)$ time and decoding and iterating through the block requires linear time wrt. the block length. The same applies for *insert* and *delete*.

The space complexity of the finger tree is not analyzed in the original paper, but it seems to be $O(m)$ where m is the number of elements in the sequence. This brings the total space required by `FDynamic EBlock` to

$$2nH_0 + O(n/\log n)$$

since the blocksize is $O(\log n)$ and again the leaves can be half-full.

The data type `FDynamic` simply encapsulates the suitable `FingerTree`. The type variable `a` is intended to range over the different blocks, allowing us to choose the underlying bit storage according to usage.

```
data FDynamic a =
```

```
  (Measured SizeRank a, BitVector a) =>
  FDynamic {blocksize :: Int,
            unwrap :: FingerTree SizeRank (Cached SizeRank a)}
```

The definitions for the basic operations become pleasantly succinct, as the following snippet shows. The `viewl` function observes the leftmost element of the sequence, returning either a pair `head :< rest` or `EmptyL` if the sequence was empty.

```
-- a wrapper around the split method for finger trees.
-- returns singled-out element plus SizeRank sum of
-- all preceding elements.
find :: FDynamic a -> (SizeRank->Bool) -> Maybe (SizeRank,a)
find (FDynamic _ f) p =
  let (before,after) = split p f
      m = measure before
  in case viewl after of
    elem :< _ -> Just (m, unCached elem)
    EmptyL    -> Nothing

_query :: BitVector a => FDynamic a -> Int -> Bool
_query f i = query block i'
  where Just (SizeRank s r, block) = find f (index i)
        i' = i-s

_queryrank :: BitVector a => FDynamic a -> Int -> Int
_queryrank f i = r + queryrank block i'
  where Just (SizeRank s r, block) = find f (index i)
        i' = i-s
```

Insertions and deletions are implemented like in the `Tree` case. We define the function `modify` that seeks out a location in the finger tree and performs the given update operation. The difference is that now we balance the blocks after modification using the function `balanceAt`. The operator `><` is the catenation of finger trees.

```
modify :: (DynamicBitVector a, Measured SizeRank a, Encoded a) =>
          (SizeRank -> Bool) ->
          ((SizeRank,a) -> a) ->
          FDynamic a -> FDynamic a
modify pred f (FDynamic size t) =
  FDynamic size (before >< balanced)
    where (before', after') = split pred t

          (before, block, after) =
            case viewl after' of
              b :< bs -> (before', unCached b, bs)
              EmptyL ->
                case viewr before' of
                  bs :> b -> (bs, unCached b, empty)
```

```
                  EmptyR -> error "modify:␣This␣shouldn't␣happen!"

          sr = measure before
          newblock = f (sr,block)

          balanced = balanceAt size newblock after

_insert f i val = modify (index i) insertIntoLeaf f
  where insertIntoLeaf (SizeRank s r, a) = insert a (i-s) val

_delete f i = modify (index i) deleteFromLeaf f
  where deleteFromLeaf (SizeRank s r, a) = delete a (i-s)
```

Balancing is performed by splitting blocks that are over twice the blocksize and combining blocks that are under half the block size with their siblings. Combining can produce blocks that are too large and need to be split. The function `balanceAt` handles this logic. It is given the block size, the block to insert, and the sequence of blocks following the insertion location.

In addition to the functions of the `Encoded` type class (`combine`, `cleave` and `encodedSize`), this code uses the finger tree operations `viewL`, `<|` (add an element to the left end of the sequence) and `singleton` (produce a sequence containing only the given element).

```
balanceAt :: (Measured SizeRank a, Encoded a) =>
             Int -> a ->
             FingerTree SizeRank (Cached SizeRank a) ->
             FingerTree SizeRank (Cached SizeRank a)
balanceAt lim elem after
  | encodedSize elem > 2*lim
    = let (a,b) = cleave elem in cached a <| cached b <| after
  | encodedSize elem < lim`div`2
    = case (viewl after)
      of EmptyL -> singleton (cached elem)
         ((Cached _ a) :< after') ->
           balanceAt lim (combine elem a) after'
  | otherwise = cached elem <| after
```

8.10 A Simple Wavelet Tree

We implemented the `WaveletTree a` type that denotes a wavelet tree using a bit vector of type `a` to implement the bit data in the inner nodes.

```
-- A type synonym
type Symbol = Char

data WaveletTree a
  = Leaf Symbol
```

```
  | Node [Symbol] a (WaveletTree a) (WaveletTree a)
  deriving Show
```

Constructing the wavelet tree is done with a simple recursion. The function `alphabetSplit` performs the core operation of the recursion: it splits the string into two parts with disjoint alphabets. Since we split the alphabet in half (the function `halve`), this produces a balanced wavelet tree.

It uses two core list processing functions from the Prelude introduced in Section 6.8: `map :: (a->b) -> [a] -> [b]` applies a function to each element of a list and `filter :: (a->Bool) -> [a] -> [a]` returns those elements of a list for which the given predicate is true. The Prelude function `elem :: a->[a]->Bool` returns whether a list contains a given element.

```
alphabetSplit :: [Symbol] ->  -- left alphas
                 [Symbol] ->  -- right alphas
                 [Symbol] ->  -- data
                 ([Bool],     -- guide
                  [Symbol],   -- left data
                  [Symbol])   -- right data
alphabetSplit left right xs = (guide,l,r)
  where guide = map (`elem` right) xs
        l = filter (`elem` left) xs
        r = filter (`elem` right) xs

halve :: [a] -> ([a],[a])
halve xs = splitAt (length xs `div` 2) xs

data WaveletTree a
  = Leaf Symbol
  | Node [Symbol] a (WaveletTree a) (WaveletTree a)
  deriving Show

symbols :: WaveletTree a -> [Symbol]
symbols (Leaf s) = [s]
symbols (Node ss _ _ _) = ss

mkWavelet :: Construct a =>
             [Symbol] ->     -- alphabet
             [Symbol] ->     -- data
             WaveletTree a
-- base case: alphabet of size one gives a leaf
mkWavelet [x] xs =
  if all (==x) xs
  then Leaf x
  else error ("Bad␣leaf!␣" ++ show x ++ "␣" ++ show xs)
-- otherwise split the alphabet and recurse
mkWavelet symbs xs = Node symbs vec left right
  where (lsymbs,rsymbs) = halve symbs
        (guide,lxs,rxs) = alphabetSplit lsymbs rsymbs xs
```

```
        vec = construct' guide
        left  = mkWavelet lsymbs lxs
        right = mkWavelet rsymbs rxs
```

The function `mkWavelet'` is a wrapper around `mkWavelet` that produces a *left-heavy* wavelet tree: the symbols in the left subtree of a node have more occurrences than the symbols in the right subtree. This is achieved by ordering the alphabet based on the frequencies of the symbols and the fact that `halve` maintains the ordering. The purpose of left-heavy wavelet trees is to make the bit vectors in the nodes have more zeros than ones, thus making them suitable for gap encoding (cf. [FGM09, Section 3.2]).

`Data.Map.Map` is the standard library ordered search tree type. Here it used by an alias `M.Map`.

```
histogram :: [Symbol] -> M.Map Symbol Int
histogram = -- omitted

mkWavelet' :: Construct a => [Symbol] -> WaveletTree a
mkWavelet' xs = -- omitted
```

The implementations of the operations `wread` and `wrank` are faithful encodings of the algorithms in Section 4.3, except that `wrank` is written in top-down instead of bottom-up form. Additionally, we return an answer of 0 from `wrank` as early as possible (see the first equation for `wrank`). The alternative would have been to define `queryrank guide (-1) = 0`.

```
wread :: BitVector a => WaveletTree a -> Int -> Symbol
wread (Leaf symbol) i = symbol
wread (Node _ b left right) i
  | val == False = wread left  (queryrank0 b i - 1)
  | val == True  = wread right (queryrank  b i - 1)
    where val = query b i
```

```
-- a utility function
symbols :: WaveletTree a -> [Symbol]
symbols (Leaf s) = [s]
symbols (Node ss _ _ _) = ss

wrank :: BitVector a => WaveletTree a -> Symbol -> Int -> Int
-- we fell of the tree while hunting for occurrences:
wrank _              _       (-1) = 0
wrank (Leaf symbol) symbol' i =
  if symbol==symbol'
  then i+1
  else error "This␣shouldn't␣happen!"
wrank (Node _ guide left right) symbol i
  | symbol `elem` symbols left
    = wrank left  symbol (queryrank0 guide i - 1)
  | symbol `elem` symbols right
```

```
|| = wrank right symbol (queryrank  guide i - 1)
```

Now for example `WaveletTree NBlock` gives us a compressed wavelet tree with roughly linear-time operations whereas `WaveletTree Static` gives a compressed wavelet tree with logarithmic time queries.

8.11 Remarks

The resulting Haskell codebase is pleasantly small: 2500 lines of code contain multiple compressed bit vectors, their tests and a number of utilities for benchmarking. Structuring the implementation into modules with clean interfaces also helps the readability of the code. Even the longest module (`Encoding2`) is under 500 lines. All in all the implementation feels clean and concise, very much thanks to the choice of Haskell as the implementation language.

The `QuickCheck` [CH00] test framework was used as an implementation aid and also to verify the correctness of the operations. Surprisingly large test inputs were needed to dig out some of the bugs in the implementation.

The biggest bottleneck in the implementation at the moment are the low-level bit operations (e.g. `readCode`). Fully optimising them is out of the scope of this work but is an interesting topic of its own. However the profiling utilities that GHC provides have proven invaluable in nailing down the performance issues and fixing those that were fixable.

9 Benchmarks

We benchmarked the implemented structures against some existing imperative implementations. The static structure used as a benchmark is the the one included in the RLCSA library [MNSV10]. It is a compressed static bit vector that implements *rank* and *select* using indices on top of a δ-encoded bit vector. The dynamic implementation used as a benchmark is Wolfgang Gerlach's dynfmi library [Ger07] (already mentioned previously) which is an *uncompressed* dynamic bit vector based on a red-black tree.

9.1 Static Operations

Each structure was benchmarked by constructing the structure with a bit sequence loaded from a file and then performing a number of *query* and *rank*[12] operations for pseudo-random indices. To eliminate one-off costs and construction time we ran the benchmark first with 100 queries and then with 100 000

[12] *select* was not benchmarked since it is either as fast as *rank* or significantly slower, depending on the structure

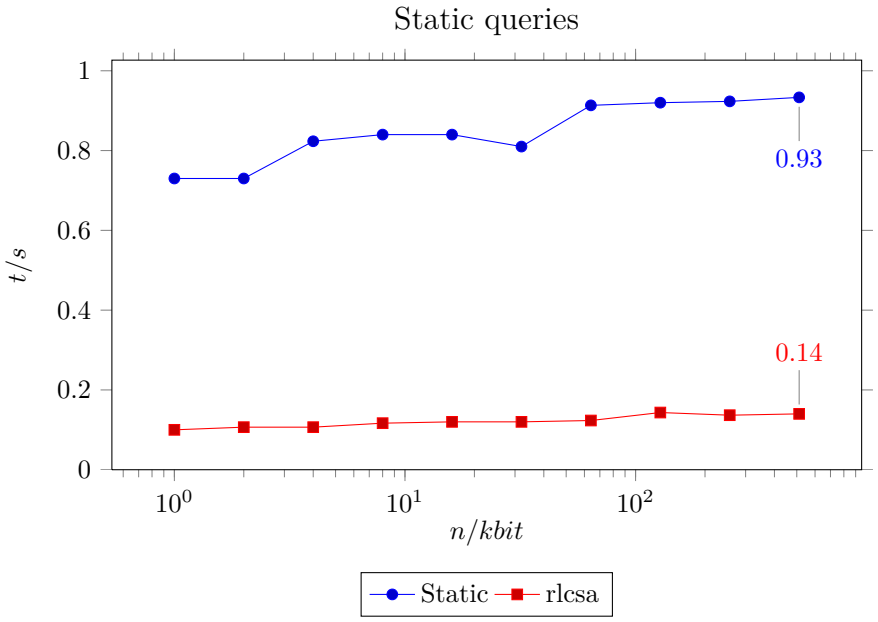


Figure 11: Performance of implemented static bit vector versus two C++ implementations. The y-axis shows time consumed by 100 000 queries.

queries and subtracted the running times. The tests were done with pseudo-random data with a H_0 of 0.2.

The static structure behaved reasonably well compared to C++ implementations. See Figure 11. The dynamic structures showed a logarithmic running time as expected. See Figure 12 and Figure 13. The tree-based structures proved faster than the finger tree based ones. This is in line with Hinze and Paterson's observation of Finger Trees being 3-5 time slower than search trees for indexing [HP06]. Also the constant-size `SmallBlock`s proved faster than variable-length `Block`s by a factor of 2 in finger trees and 4 in trees.

When compared to dynfmi, `Tree SmallBlock` is slower only by a factor of 5. The comparison is apt since both structures store constant-sized uncompressed blocks of bits in a tree structure. Encouragingly adding compression (`SmallElias`) only costs an additional 20%.

9.2 Dynamic Operations

Dynamic implementations were benchmarked by loading data (pseudo-random, $H_0 = 0.2$) from a file and then performing a number of the following operations:

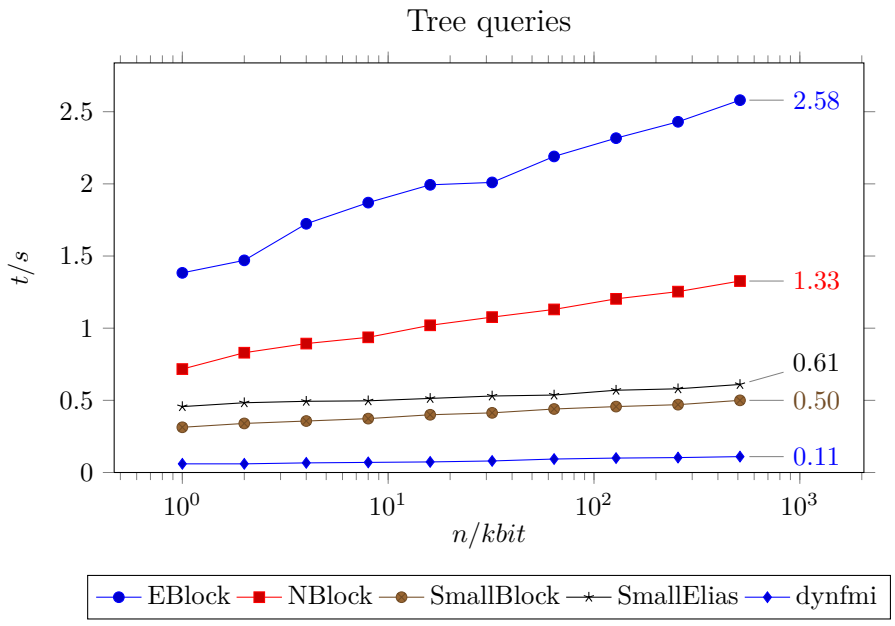


Figure 12: Performance of the tree-based dynamic bit vectors against dynfmi under queries.

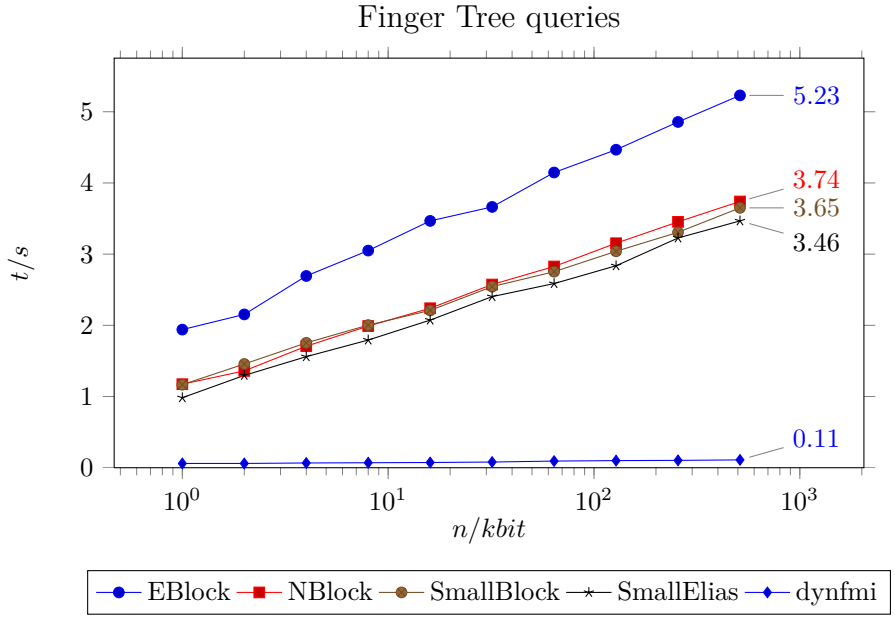


Figure 13: Performance of the Finger Tree -based dynamic bit vectors against dynfmi under queries.

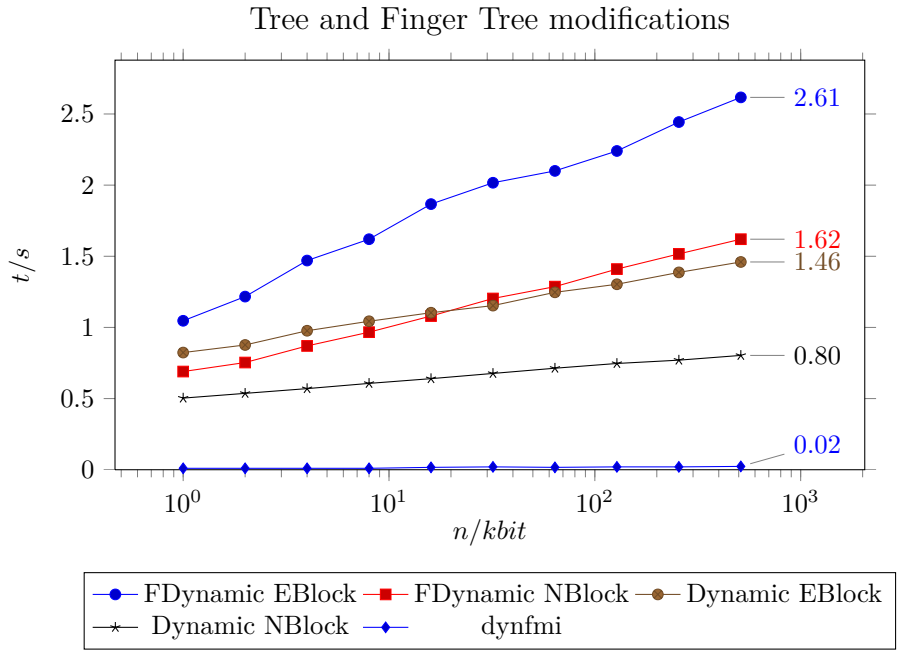


Figure 14: Performance of the dynamic implementations under insertions and deletions.

1. Insert a random bit at a random index
2. Delete a random index
3. Query a random index

Again, to discount one-off costs the benchmark was run first for 100 and then 10 000 iterations. See Figure 14 for results. Once more, `Tree` proved faster than finger trees and `NBlock` faster than `EBlock`. However, the factor between `Dynamic NBlock` and dynfmi grows from about 10 to about 40 compared to queries. This is due to the inefficiency of the `Block` operations.

9.3 Memory Use

In addition to running time, the memory use of the structures was also benchmarked. The memory usage numbers for the implemented structures are based on the total number of allocated bytes as reported by the GHC profiling tools. The memory usage of dynfmi was benchmarked using the valgrind massif tool [NS07].

Memory usage for all the structures exhibited a sub-linear trend. See Figure 15.

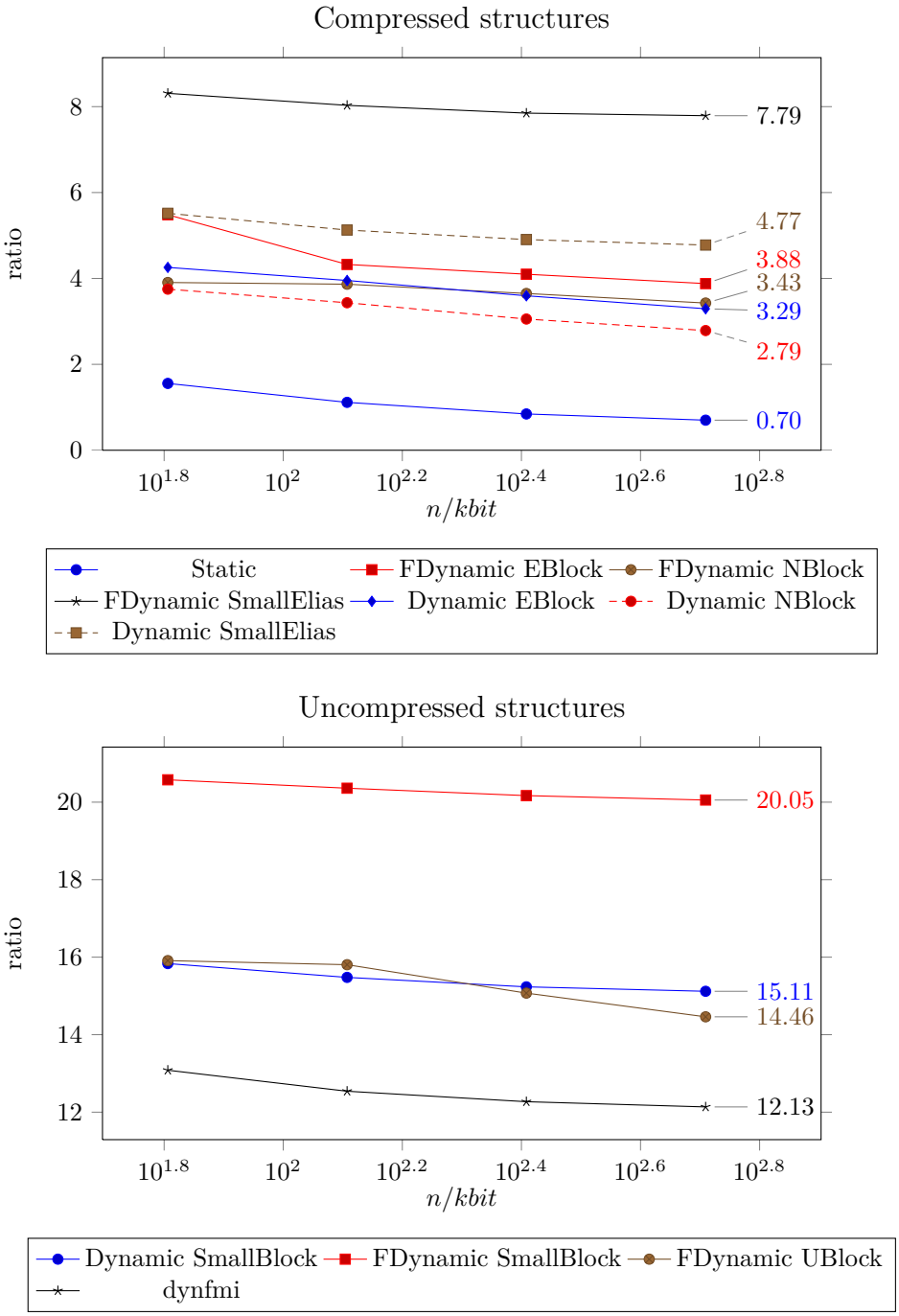


Figure 15: Memory use of the implemented structures. The y-axis plots the ratio of the memory used to the size of the uncompressed data. The data was pseudo-random with $H_0 = 0.2$.

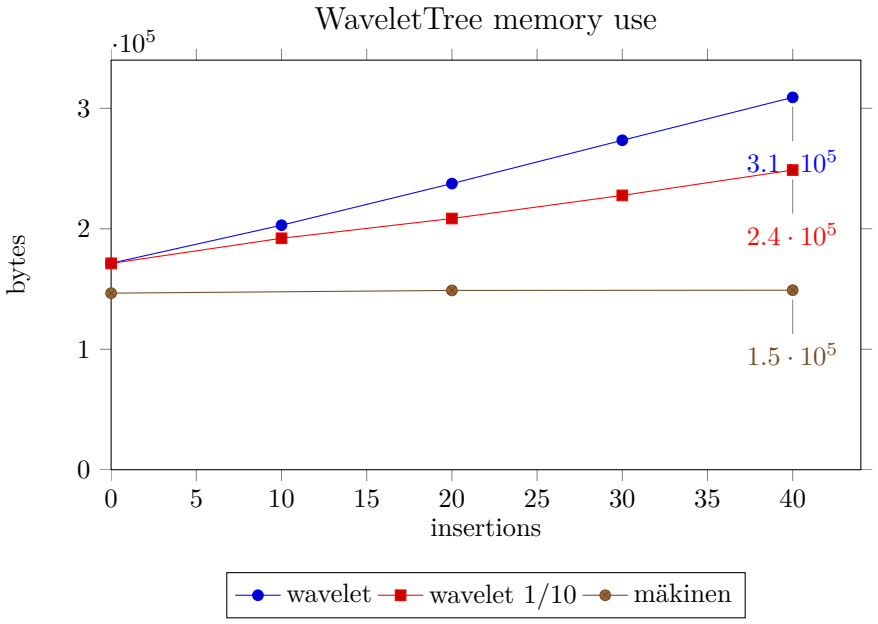


Figure 16: Memory use of persistent wavelet tree under random insertions with $n_0 = 16384$, $\sigma = 26$, $H_0 = 4.07$. For "wavelet 1/10" only every tenth modified version is retained.

Finally, we benchmarked the memory use of the implemented wavelet tree under modifications. In the benchmark a wavelet tree was constructed from data, after which a number of insertions were made into it sequentially, retaining the intermediate versions. The comparison was to a *static* persistent structure by Mäkinen et al [MNSV10, Theorem 25 and section 7], which can be used to support the same set of queries. However, unlike the dynamic wavelet tree, once their structure has been built it needs to be compressed, after which no more insertions can be done. Before compression the structure uses roughly 10 times more space.

See Figure 16 for the measurements of the wavelet tree benchmark. The memory use of the wavelet tree can be seen to rise only mildly: 40 related versions occupy less than double the space of one version. Additionally, if access to some intermediate versions is not needed they can be garbage collected. This is also demonstrated in Figure 16. All in all, Mäkinen's structure wins asymptotically but offers less features.

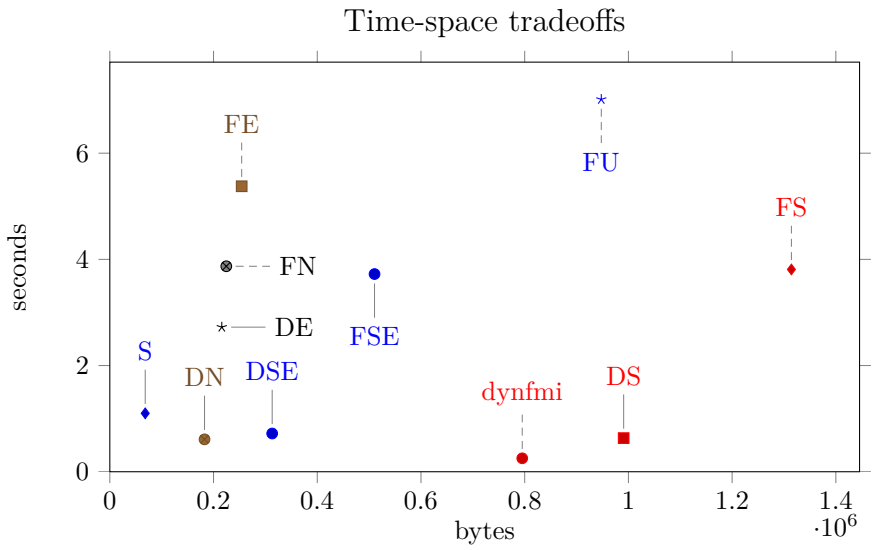


Figure 17: A time-space plot of the implemented and existing bit vectors. Values are for queries with $n = 512 \cdot 1024$, $H_0 = 0.2$. The first letters of the abbreviations are D for `Dynamic`, F for `FDynamic` and S for `Static`. The rest denotes the block implementation: E for `EBlock`, N for `NBlock`, U for `UBlock`, S for `SmallBlock` and SE for `SmallElias`

9.4 Space-Time Tradeoffs

Some benchmarks were made in order to explore the various time-space tradeoffs available. Figure 17 shows an overview of the different implementations. `Dynamic NBlock` is by far the most competitive dynamic structure, and almost the best overall if it were not for the better memory usage of `Static`. The uncompressed `Dynamic SmallBlock` is also the best of its pack, but loses to dynfmi.

Figure 18 shows the effect of changing the block size for `FDynamic EBlock`. Block-size choices of 360 (i.e. $19 \log n$) and 184 (i.e. $9.5 \log n$) seem to give a good choice of time vs. space. Recall that the implementation actually uses $8 \log n$, leaning slightly more towards speed.

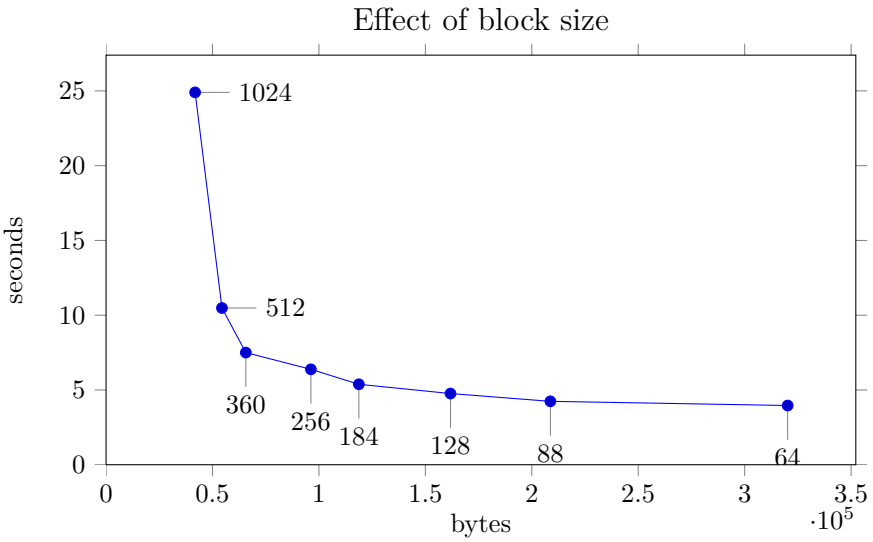


Figure 18: The time-space tradeoff for FingerTree+EBlock under different block sizes. $n = 256 \cdot 1024, H_0 = 0.2$.

10 Conclusions

This thesis has presented the first functional implementations of compressed bit vectors. The implementations are unfortunately significantly slower than the imperative implementations they were benchmarked against. However, their performance is understandable given the unique combination of features: compression and persistence. The implementation is also self-contained and relatively small, leaving the door open for future development.

Simplicity is also the key virtue of the presented wavelet tree implementation. It fared quite well against a specialized static structure while supporting more operations. The wavelet tree implementation could easily be extended into a full-fledged indexing library with the addition of an efficient implementation of BWT and some utilities.

Many practical issues relating to the data structures were not discussed in this thesis. These include serialization (for storing the structures outside working memory) and efficient construction. Also, alternative compression schemes such as run length encoding were not implemented. On the other hand this has kept the codebase smaller. Also, new encodings are relatively easy to implement using the infrastructure already in place.

The idea of monoidal annotations proved a useful abstraction, and the idea clearly needs more attention. The author was unable to find any article that concentrated on monoidal annotations and their various applications, though many do hint at the framework's generality. Relatedly, this thesis proves once again the versatility of the finger tree (and the robustness of its Haskell implementation).

References

[BB04] Daniel K. Blandford and Guy E. Blelloch. Compact representations of ordered sets. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 11–19, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[BW94] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, 35:268–279, September 2000.

[Cla96] David Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, December 1985.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[FGM09] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Inf. Comput.*, 207(8):849–866, 2009.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52:552–581, July 2005.

[Gag07] Travis Gagie. Empirical entropy in context. *CoRR*, abs/0708.2084, 2007.

[Ger07] Wolfgang Gerlach. Dynamic FM-Index for a Collection of Texts with Application to Space-efficient Construction of the Compressed Suffix Array. Diplomarbeit, Faculty of Technology, Bielefeld University, 2007.

[GHC] The GHC Team. *The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2*. `http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/index.html`.

[GM10] Travis Gagie and Giovanni Manzini. Move-to-front, distance coding, and inversion frequencies revisited. *Theor. Comput. Sci.*, 411(31-33):2925–2944, 2010.

[HP06] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[Hud89] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989.

[Hug89] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Jay57] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106(4):620–630, May 1957.

[KLV07] Haim Kaplan, Shir Landau, and Elad Verbin. A simpler analysis of Burrows-Wheeler-based compression. *Theor. Comput. Sci.*, 387:220–235, November 2007.

[Lea99] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[Lip11] Miram Lipovača. *Learn You a Haskell for Great Good!* No Starch Press, Inc., San Francisco, CA, USA, 2011. Available on-line at `http://learnyouahaskell.com/`.

[Lis87] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23:17–34, January 1987.

[Man01] Giovanni Manzini. An analysis of the Burrows–Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[Mar] Simon Marlow, editor. *Haskell 2010 Language Report*. `http://www.haskell.org/onlinereport/haskell2010/`.

[MN07] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007.

[MN08] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3):1–38, 2008.

[MNSV10] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[OGS08] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1999.

[PD06] Mihai Patrascu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35:932–963, April 2006.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[RRR01] Rajeev Raman, Venkatesh Raman, and S. Rao. Succinct dynamic data structures. In Frank Dehne, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures*, volume 2125 of *Lecture Notes in Computer Science*, pages 426–437. Springer Berlin / Heidelberg, 2001.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

[Ste09] Guy L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. *SIGPLAN Not.*, 44:1–2, August 2009.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[WZ99] Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

A Notations

log	the base-2 logarithm
$s_1 s_2$	the catenation of strings s_1 and s_2
s^k	the string consisting of k copies of s
$\|s\|$	the length of the string s
$H_k(s)$	the kth-order empirical entropy of string s
Σ	the alphabet
Σ^k	the strings of length k over alphabet Σ
σ	the size of the alphabet, $\|\Sigma\|$
$C_s(x)$	a string containing the characters immediately following an occurrence of substring x in string s

B Haskell Syntax

- Comments

```
a = 1  -- two dashes start a comment that continues to the end of the line
```

- Expression syntax

```
<function> <argument> <argument>...
<argument> <operator> <argument>
let <pattern> = <expression> in <expression>
if <expression> then <expression> else <expression>

case <expr>
  of <pattern> -> <expression>
     <pattern> -> <expression>
     ...
```

- Tuples (can be pattern matched)

```
(<value>,<value>)
```

- Lists (can be pattern matched)

```
[<value>,...]
[]
<value>:<list>
```

- Function definition

```
<name> :: <type>
<name> <pattern>... = <expression>

<name> <pattern>... = <expression>
   where <pattern> = <expression>
```

```
<name> <pattern>...
   | <expression> = <expression>
   | <expression> = <expression>
   ...
   | otherwise = <expression>
```

- Data type definitions

```
data <typename> <typevariable>... =
   <constructorname> <fieldtype>...
 | <constructorname> <fieldtype>...
 | ...

newtype <typename> <typevariable>... =
   <constructorname> <fieldtype>...
```

- Type aliases

```
type <typename> <typevariable>... = <type>
```

- Type classes and instances

```
class <classname> <typevariable>... where
  <functionname> :: <functiontype>
  <functionname> :: <functiontype>
  ...
```

```
instance <classname> <type>... where
  <function definitions>
```