Benchmarking Purely Functional Data Structures

Graeme E Moss

Thesis submitted for the degree of DPhil in Computer Science

University of York

Department of Computer Science

July 1999

Abstract

When someone designs a new data structure, they want to know how well it performs. Previously, the only way to do this involves finding, coding and testing some applications to act as benchmarks. This can be tedious and time-consuming. Worse, how a benchmark uses a data structure may considerably affect the efficiency of the data structure. Thus, the choice of benchmarks may bias the results. For these reasons, new data structures developed for functional languages often pay little attention to empirical performance.

We solve these problems by developing a benchmarking tool, *Auburn*, that can generate benchmarks across a fair distribution of uses. We precisely define "the use of a data structure", upon which we build the core algorithms of Auburn: how to generate a benchmark from a description of use, and how to extract a description of use from an application. We consider how best to use these algorithms to benchmark competing data structures.

Finally, we test Auburn by benchmarking several implementations of three common data structures: queues, random-access lists, and heaps. These and other results show Auburn to be a useful and accurate tool. They also reveal areas requiring improvement, which we list as future work.

Contents

List of Tables

List of Figures

Acknowledgements

A studentship from the University of York funded this work, and many people inspired and encouraged me during its development. Firstly, I would like to thank Chris Okasaki, without whom none of this work would have even started. The work Chris and others have done in the field of data structures reveal some of the beauty within Computer Science.

I would also like to thank: Mike Thyer, for countless discussions on many issues relating to this work and to Computer Science in general; Nathan Charles, for keeping me sane whilst working in the office; and those who were foolish enough to ask me about my thesis, for the reminder this gave me about my will to finish.

Most importantly, I would like to thank my supervisor, Colin Runciman, for always being willing to listen, advise, and encourage, regardless of however much other work he had to juggle.

Finally, I would like to thank those who made my stay at York so much more enjoyable: Andrew, Judith, Suzanne, Tony, Nick, Claire, Helen, Randall, Emma, Uzma, Catherine, Kristin, and Marie.

Author's Declaration

I presented a concise and informal version of Chapter 3 at IFL'97 [26] using version 1.0 of Auburn. All of Chapter 5, bar Section 5.4, appears in the draft proceedings of IFL'98 [27]. At PADL'99 [28] I presented brief and formal versions of Sections 3.1 and 3.3, a summary of Section 4.1.1, and an illustration of the use of Auburn version 2.0a.

Chapter 1

Introduction

The importance of efficient data structures is reflected through literature spanning many years [3, 11, 51]. Recently, this has included data structures and complexity models developed specifically for functional languages [14, 38, 40]. But, in practice, what distinguishes a good data structure from a bad data structure? What is the main reason whether a data structure is useful? Empirical performance! Yet most literature has paid little attention to this aspect of data structures. We tackle this deficiency by developing the theory and practice of benchmarking functional data structures.

1.1 Functional Languages

Why use functional languages? Given the amount of literature on data structures for imperative languages, why do we need to bother with functional data structures? There are strong arguments for the functional style of programming [5, 22].

- *Succinctness.* A functional program is typically shorter than its imperative equivalent. This helps reduce development and maintenance costs.

- *Clarity.* The meaning of a functional program is arguably more immediate, by being shorter and by using features like algebraic datatypes and higher order functions.

- *Reasoning.* The lack of state allows referential transparency, which in turn allows the meaning of a program to be independent of its surroundings.

This simplifies any mathematical reasoning on a program, including for example, a proof of its correctness. This also simplifies the programmer's task, by aiding their own mental reasoning about a program.

- *Beauty.* A functional program feels "cleaner" and more aesthetically pleasing. Through aesthetics, this affects the state of the programmer, their enthusiasm to work, and thus the quality of their results.

As a small example, Figure 1.1 and Figure 1.2 show C and Haskell versions respectively of a program to insert and lookup a node in an ordered, unbalanced tree. The most obvious difference between these programs is the difference in size. Figure 1.3 shows a more compact C program, but it is still larger than the Haskell program, and less understandable than the larger C program. The Haskell program is far clearer than either C program. Because of this size difference, and because of the lack of pointers, programming the Haskell version is far less error-prone. The Haskell programmer is free to think about the tree itself, rather than how the tree is represented.

1.2 Functional Data Structures

Given we want to use a functional language, why do we need data structures specifically designed for a functional setting? Will not the vast array of imperative data structures suffice? Unfortunately not, because of the greater demands a functional language places on its data structures: A functional data structure *cannot be destructively updated.* No information can be lost until the program using the data structure no longer requires it. In particular, when a data structure is updated, *both the new and the old* versions of the data structure must be available for further use.

Some imperative data structures *can* be brought across to the functional world with little change. In most cases the design actually becomes *clearer* in a functional setting. Figures 1.1 and 1.2 illustrate this well. Okasaki gives another example by implementing red-black trees in a functional setting [39] and further writes in the conclusions section:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
  int           value;
  struct node  *left, *right;
} node;
typedef node *tree;

int member (int x, tree t) {

  while (t != NULL && t->value != x)
    t = (x < t->value) ? t->left : t->right;
  return (t != NULL);
}

tree mknode (int x) {
  tree t = malloc (sizeof (node));

  t->value = x;
  t->left = t->right = NULL;
  return t;
}

void insert (int x, tree *result) {
  tree t = *result , *tptr = result;

  if (t == NULL) {
    *result = mknode(x);
  } else {
    while (t != NULL && t->value != x) {
      tptr = (x < t->value) ? &t->left : &t->right;
      t = *tptr;
    }
    if (t == NULL) *tptr = mknode(x);
  }
}
```

Figure 1.1: C program to insert and lookup a node in an ordered, unbalanced tree.

```
data Tree a = Empty | Node (Tree a) a (Tree a)

member x Empty = False
member x (Node l y r)
   | x < y     = member x l
   | x > y     = member x r
   | otherwise = True

insert x Empty = Node Empty x Empty
insert x (Node l y r)
   | x < y     = Node (insert x l) y r
   | x > y     = Node l y (insert x r)
   | otherwise = Node l x r
```

Figure 1.2: Haskell program to insert and lookup a node in an ordered, unbalanced tree.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
  int           value;
  struct node  *left, *right;
} node;
typedef node *tree;

tree* find (int x, tree *tp) {
  if (*tp != NULL)
    while (*tp != NULL && (*tp)->value != x)
      tp = (x < (*tp)->value) ? &(*tp)->left : &(*tp)->right;
  return tp;
}

int member (int x, tree t) {return (*find(x,&t) != NULL);}

void insert (int x, tree *tp) {
  if ((tp = find(x,tp)) != NULL) {
    *tp = malloc(sizeof (node));
    (*tp)->value = x; (*tp)->left = (*tp)->right = NULL;
  }
}
```

Figure 1.3: Compact C program to insert and lookup a node in an ordered unbalanced tree.

> When existing imperative algorithms can be implemented in functional languages, the results are often much prettier than the original version. This has been amply demonstrated in the past for various kinds of balanced binary search trees, including 2-3 trees [47], BB-trees [2], and AVL trees [31].

Over the past six or seven years, many papers have given details of new functional data structures [7, 10, 14, 32, 33, 34, 40]. However, these papers only give limited attention to empirical performance. Okasaki writes in an open problems section of his thesis *Purely Functional Data Structures* [36], "The theory and practice of benchmarking [functional] data structures is still in its infancy." This thesis develops the theory and practice of benchmarking functional data structures.

1.3 Benchmarking Functional Data Structures

Suppose we want to measure the efficiencies of some competing data structures. The standard approach is to find a few applications to act as benchmarks, allowing us to measure the efficiency of each data structure when used by each benchmark. Why not do this? Firstly, creating anything but a very artificial benchmark is a substantial task. Secondly, using the results of just a few benchmarks, especially artificial ones, can be very misleading. The efficiency of a data structure may vary heavily according to how it is used, and hence the choice of benchmarks may determine which data structure appears to be the best—see Section 7.2.1 for an example of this. Worse, we will not know if our choice of benchmarks is "fair" or not.

We solve both of these problems by developing a benchmarking tool, *Auburn*, that *creates a benchmark according to a description of use*. By generating a fair distribution of benchmarks over a wide variety of different uses, we not only find which data structure is best *overall*, but also *which data structure is best for a particular use*.

Suppose that we have a single application in mind, and we wish to choose one of many competing data structures to use in our application. Why not simply

measure the performance of our application using each data structure in turn? Unfortunately, this approach does not reveal *why* the data structures perform as they do. If our application changes how it uses the data structures, a different one may now be the most efficient, without us knowing why.

By measuring how our application uses the data structures, and how the data structures' efficiency varies according to this use, we can know why the best data structure is best. Therefore, Auburn also *creates a description of use from an application.*

1.4 Terminology

In order to understand the following chapters, it is necessary to define a few key terms.

- *Benchmark.* A *benchmark* is an application that can use any one of a family of competing data structures. A benchmark is used to measure the performance of such data structures.
- *Abstract Datatype.* An *abstract data type* (ADT) is a type with associated operations manipulating values of that type. A more detailed definition is given in Section 3.1.
- *Implementation.* A data structure that gives a concrete realisation of the type and operations of an ADT is called an *implementation.*
- *Version.* When an application uses a data structure, at any one point in the computation, there exist many different instances of the data structure—for example, a particular list, or a particular queue. Each particular instance of a data structure is called a *version* of the data structure.
- *Persistence. Persistence* is the property of allowing the use of any version of a data structure in its original form after it has been updated. A data structure that supports persistence is called *persistent.* A data structure that is not persistent is called *ephemeral.*
- *Single-Threaded.* An application is *single-threaded* in the use of a data structure if it does not use any persistence supported by the data structure.

- *Amortisation.* When applied to the complexity of an operation, *amortisation* implies that the cost of an operation is considered in the context of a *group* of operations, rather than in isolation. This allows the cost of an expensive operation to be spread over many surrounding inexpensive operations. Note that all complexities are arguably amortised in a lazy language like Haskell.

1.5 Overview

Chapter 2 reviews some implementations of three different ADTs: queues, random-access sequences, and heaps. The details of the implementations provide an example of the different ways of implementing an ADT. They also add meaning to the results of benchmarking the implementations in Chapter 7.

Chapter 3 develops the theory of datatype usage upon which Auburn is based. It defines a *datatype usage graph* (DUG) recording how a data structure is used by an application, and a *profile* summarising the most important aspects of a DUG. This chapter also outlines how we can create a benchmark from a profile, and extract a profile from an application.

Chapter 4 describes the implementation of the core algorithms of Auburn, as outlined in theory in Chapter 3. These involve the creation of benchmarks from profiles through the generation and evaluation of DUGs, and the extraction of profiles from applications through the extraction and profiling of DUGs.

Chapter 5 investigates how we should use Auburn. There are many ways we could use the algorithms of Chapter 4, but we need any method to be efficient, to be accurate, and to produce concise, clear results. This chapter presents a few methods, summarising their advantages and disadvantages, and then recommends one of them.

Chapter 6 outlines the design and use of Auburn. Chapter 4 gives the core algorithms of Auburn, but there are many other design decisions in how to implement and combine these into one package. Most of the decisions relate to the language in which we implement Auburn: Haskell.

Chapter 7 reports the results of using Auburn on the data structures of Chapter 2. We examine the accuracy of these results, and the accuracy of Auburn as

a whole. We also investigate the source of any inaccuracy in Auburn.

Chapter 8 concludes and lists future work.

Appendix A gives the code for the implementations of the data structures detailed in Chapter 2 and used in the final round of benchmarking in Chapter 7.

Appendix B gives the modifications of the implementations of Appendix A used in the fine-tuning section of Chapter 7.

Appendix C details the executables that make up Auburn.

Chapter 2

Implementations of Three ADTs

In Chapter 7, we shall benchmark several implementations of queues, random-access sequences, and heaps. This chapter delivers the key idea behind each implementation. We may then interpret the results of the benchmarking in the light of this review. Without such a review, the results hold little value except towards choosing one over another; with this review, the practical results of design choices become visible and provide insight into their effectiveness.

Each section of this chapter begins with a brief description and formal specification of the ADT. The following subsections review each implementation. We give references to papers describing the implementations in greater detail. As we organise the review by data structure, we can easily compare different implementations of the same data structure. Appendix A gives code for each implementation.

2.1 Queues

Queues are among the simplest of ADTs. They are sequences supporting insertion at the rear, and removal from the front. Figure 2.1 gives the specification of queues. Table 2.1 lists the queue implementations and the complexities of their operations.

$$type\ Queue\ a = [a]$$

$$empty :: Queue\ a$$
$$empty = [\,]$$

$$snoc :: Queue\ a \to a \to Queue\ a$$
$$snoc\ [x_0, \ldots, x_{n-1}]\ x = [x_0, \ldots, x_{n-1}, x]$$

$$head :: Queue\ a \to a$$
$$head\ [x_0, \ldots, x_{n-1}] = x_0 \quad (n \geq 1)$$

$$tail :: Queue\ a \to Queue\ a$$
$$tail\ [x_0, \ldots, x_{n-1}] = [x_1, \ldots, x_{n-1}] \quad (n \geq 1)$$

Figure 2.1: Queue specification. For the purposes of specification, we treat a queue as a list.

Queues			
Name	Lazy	Complexities of Operations	Reference
Naïve	-	*head/tail*: $O(1)$, *snoc*: $O(n)$	n/a
Simple	-	*snoc/head/tail*: $O(1)^{\ddagger}$	[20]
Multihead	-	*snoc/head/tail*: $O(1)$	[20]
Banker's	✓	*snoc/head/tail*: $O(1)^{\dagger}$	[37]
Physicist's	✓	*snoc/head/tail*: $O(1)^{\dagger}$	[38]
Real-time	✓	*snoc/head/tail*: $O(1)$	[34]
Bootstrapped	✓	*snoc/head/tail*: $O(1)^{\dagger}$	[38]
Implicit	✓	*snoc/head/tail*: $O(1)^{\dagger}$	[38]

Table 2.1: Complexities of implementations of queues, including whether lazy evaluation is required. Complexities marked with † are amortized. Complexities marked with ‡ also are amortized, but only under single-threaded use. All other complexities are worst-case.

2.1.1 Naïve Queues

We can represent a queue directly as a list. The normal head and tail operations of lists implement *head* and *tail*. List catenation of a singleton list implements *snoc*.

2.1.2 Batched Queues

Hood and Melville [20] represent a queue as a pair of lists (f, r)—f giving the front portion of the queue and r giving the *reverse* of the rear portion of the queue. The queue of elements $a_1, a_2, \ldots, a_n$ is therefore represented by the lists $f = [a_1, \ldots, a_m]$ and $r = [a_n, \ldots, a_{m+1}]$, $0 \leq m \leq n$ with f empty only when the queue itself is empty. To insert an element onto the queue, simply add an element to the front of r. To remove an element from the queue, take the first element of f; if this leaves f empty, then let the queue become $(\mathit{reverse}\ r, [\,])$.

Every operation except *tail* takes $O(1)$ time. If an application of *tail* causes a reversal of r, it takes $O(n)$ time; otherwise, it also takes $O(1)$ time. For any single-threaded sequence of operations, a reversal of r happens at most once every $A(n)$ operations, where $A(n)$ is $O(n)$. Therefore we can conclude that $A(n)$ single-threaded queue operations take $O(n)$ time—an amortized complexity of $O(1)$. However, persistence destroys this result. Consider an application of *tail* that reverses the rear list. Persistence allows us to repeat this application indefinitely, each application taking $O(n)$ time. Therefore, in a persistent setting, the best complexity we can give to *tail* is $O(n)$.

We take the name of this implementation from [38].

2.1.3 Multihead Queues

Hood and Melville [20] improve on the batched implementation of a queue by distributing the reversal of the rear list over a number of operations. This gives real-time queues, that is, the operations run in $O(1)$ worst-case complexity.

In order to continue performing operations whilst reversing the rear list, the reversal begins when the rear list r becomes larger than the front list f. The reversal is spread over the following n operations, where n is the length of the

front list. These n operations create new front and rear lists f_{op} and r_{op} by removing elements from f and by adding elements to the empty list respectively. At the same time, r is reversed onto the end of f to create a new front list f_{new}, taking care to use only elements in f_{op}. The lists f_{new} and r_{op} form the new queue. It is simple to prove that r_{op} is no longer than f_{new}.

To create the list f_{new} over n operations, reverse f to make f_{rev}, and at the same time reverse r to make r_{rev}. Then move elements from the front of f_{rev} onto the front of r_{rev} till an element not in f_{op} is reached, or when all elements have been moved. It is sufficient to move only two elements per operation from f to f_{rev}, from r to r_{rev}, or from f_{rev} to r_{rev}. Hence each operation takes $O(1)$ time.

The name *multihead* derives from the similarity of the solution to how multi-head Turing machines can be simulated. Full details are given in [20]. Note that there are two mistakes in the code given in [20].

- The call `cons[v,T]` on line 4 should read `cons[v,T']`.
- The value `lendiff-1` on line 9 should read `lendiff`.

Appendix A gives the corrected implementation.

2.1.4 Banker's Queues

Okasaki [37] presents an implementation of queues with $O(1)$ amortized complexity. He is able to give an amortized complexity in a persistent setting by appealing to the proof techniques that he develops in [32, 37], and presents in [38]. Representing a queue as a pair of lists is once again the basis of the implementation. Hood and Melville remove the problem of the $O(n)$ persistent complexity of the batched implementation by *explicitly* scheduling a distribution of the work involved in performing the reversal of the rear list. Okasaki gives a much simpler solution that uses lazy evaluation to *implicitly* schedule and share this distribution of work.

The key idea is not to delay more work than a subsequent sequence of operations can pay off. Under single-threaded use, traditional amortization allows us to spread the cost of the reversal of the rear list r of length $|r|$ over the previous $|r|$ applications of *snoc* that built r. With non-single-threaded use however, we

may have several queues sharing the result of a *snoc*. This application of *snoc* can only bear a constant additional cost before losing its $O(1)$ complexity. As an *arbitrary* number of queues may share the result of the *snoc*, the batched implementation of queues cannot have $O(1)$ complexity in a persistent setting.

Okasaki shifts the burden of the reversal from the *preceding* sequence of operations to the *succeeding* sequences of operations—remember that there may be more than one such sequence because of persistence. This is done by insisting that a queue must never engage in a reverse whose cost cannot be spread over operations that occur *after* the reverse is formed but *before* its result is required. The cost of the reverse can then be shared by the operations that occur between suspending an application of the reverse and executing this suspension. The cost of the reverse is considered to be a debt, waiting to be paid off. Lazy evaluation plays a key role here in two respects: a function application can be delayed, and the result of the delayed application can be shared. For further details on persistent amortization, see [38].

So when can we delay a reverse and still be in a position to pay off its debt before its result is needed? Suppose we only reverse the rear list r and append it to the end of the front list f when $|r|$ becomes larger than a constant k times $|f|$. As we apply *tail* to the resulting queue, the new front list will shorten. Until we have removed all of f, the result of the reverse is not required. The number of applications of *tail* required to do this is equal to $|f|$. As $|r|$ is at most a constant k times $|f|$, we can share the cost of the reverse over the $|f|$ applications of *tail* by adding a constant additional cost to each. The operations therefore keep their $O(1)$ complexity.

For a more formal argument using the banker's method of persistent amortization proof techniques, see either [37] or [38]. The name of this implementation is derived from the proof technique used to give it its complexity.

2.1.5 Physicist's Queues

In the same way that Okasaki uses the banker's method to give $O(1)$ amortized bounds to banker's queues, he uses the physicist's method to give $O(1)$ amortized bounds to *physicist's queues* [38].

The major difference between the banker's and physicist's methods is that the banker's method allows the debt of particular suspensions of work to be paid off individually whereas the physicist's method considers the debt of the whole data structure. The idea behind physicist's queues is to make fewer suspensions. For a strict language such as Standard ML where suspensions are explicit and costly, this may reap some rewards. For a lazy language such as Haskell where everything is suspended, the physicist's queues are unlikely to be any more efficient than the banker's queues.

2.1.6 Real-Time Queues

A real-time data structure supports all operations in $O(1)$ worst-case time. Okasaki gives a *real-time* implementation of queues in [34]. We may derive this implementation from the banker's queues by splitting up any monolithic chunks of work into portions taking $O(1)$ time. These portions are spread evenly over every operation. This allows each operation to run in $O(1)$ time.

The only monolithic work suspended by the banker's queues not of $O(1)$ complexity is the reversal of the rear list. This is replaced by the function *rotate* that incrementally reverses the rear list onto the back of the front list. A constant portion of the rotation is done each time the queue is updated.

2.1.7 Bootstrapped Queues

Okasaki [38] offers yet another variation on the banker's queues, this time using the principle of *data-structural bootstrapping* given by Buchsbaum [8]. The basic idea behind bootstrapping is to extend the design of an incomplete or inefficient data structure to use smaller instances of the same data structure.

Recall that banker's queues reverse the rear list onto the end of the front list every time the rear list becomes too large. After a series of such reversals, the front list will look something like this:

$$(\cdots((f \# reverse\ r_1) \# reverse\ r_2) \cdots \# reverse\ r_k)$$

As append is linear in its left argument, such a series of appends is rather expensive since some elements are traversed more than once, eg. every element of

r_1 will be traversed k times. Bootstrapped queues remove this inefficiency by storing the collection $\{reverse\ r_1, \ldots, reverse\ r_k\}$ of reversed rear lists separately, and using them to replace the front list as necessary. This does not then require any applications of append. But how should we store this collection? Noting the *first-in first-out* order in which they are inserted and removed, we shall represent this collection as a queue of lists. This is where bootstrapping is used: A *queue of lists* represents *part* of a queue. The type of a queue becomes:

```
data Queue a = Empty
             | Queue [a] (Queue [a]) Int [a] Int
```

where `Queue f m` $\texttt{fm}_{\texttt{len}}$ `r` $\texttt{r}_{\texttt{len}}$ is a queue with front list `f`, queue `m` of reversed rear lists, and rear list `r`; $\texttt{fm}_{\texttt{len}}$ gives the combined length of `f` and the lists in `m`; and $\texttt{r}_{\texttt{len}}$ gives the length of `r`. The recursive type requires a base case for termination, so an `Empty` constructor is introduced.

The operations of this implementation run in $O(\log^* n)$ time[1], but a simple alteration improves this complexity to $O(1)$. In practice however, this makes little difference.

2.1.8 Implicit Queues

Okasaki [38] describes another implementation of queues, this time based on the principle of *recursive slowdown*. Kaplan and Tarjan first introduced recursive slowdown in [24]. The key observation underlying the technique arises from considering a bootstrapped data structure (for an example of bootstrapping, see Section 2.1.7).

Suppose an operation on a bootstrapped data structure of size n involves a constant amount of work plus that of calling the *same operation* a constant c times on nested data structures of combined size $f(n)$. Let $T(n)$ measure the time taken by this operation. We have:

$$T(n) = O(1) + c\,T(f(n))$$

If we solve this recurrence relation for $c = 1$ and $f(n) = \log n$, we find that $T = O(\log^* n)$. This gives the complexity of the bootstrapped queues of Sec-

[1] $\log^{(1)} k = \log_2 k,\ \log^{(i)} = \log\log^{(i-1)} k\ (i > 1),\ \log^* k = \min\{i \,|\, \log^{(i)} k \leq 1\}$

tion 2.1.7. If however, we solve the relation for $c = 1/2$ and $f(n) = \log n$, we find that $T(n) = O(1)$. Indeed, for $c < 1$ and $f(n) = n - 1$, we still find that $T(n) = O(1)$. But what does performing, say, half an operation mean? Suppose we made sure that only *one* operation was performed on a nested data structure for every *two* operations on the enclosing data structure. This could be seen as performing *half* an operation on the nested data structure for every *one* operation on the enclosing data structure. This is recursive slowdown.

To apply recursive slowdown to queues, we shall represent a queue using a smaller inner queue on which we perform *one* operation for every *two* operations performed on the enclosing queue. If the inner queue is a queue of pairs, we need only insert or remove a pair every two insertions or removals respectively on the enclosing queue. We will keep at least one element at the front of the enclosing queue. This ensures that the enclosing queue is ready to perform an operation and that the inner queue is distinctly smaller. This is Okasaki's implementation, and the type of queues is given by

```
data Queue a = Shallow (ZeroOrOne a)
             | Deep (OneOrTwo a) (Queue (a,a)) (ZeroOrOne a)
data ZeroOrOne a = ZeroInOne | OneInOne a
data OneOrTwo a = OneInTwo a | TwoInTwo a a
```

Whereas Kaplan and Tarjan *explicitly* schedule the work involved in recursive calls to inner data structures, Okasaki uses lazy evaluation to *implicitly* schedule this work, hence the name of this implementation. Data structures using implicit recursive slowdown are typically a lot simpler than their explicit counterparts, but are amortized rather than worst-case.

2.2 Random-Access Sequences

Figure 2.2 specifies sequences that support access to any element. Table 2.2 lists some implementations.

$$type\ RASeq\ a = [a]$$

$$empty :: RASeq\ a$$
$$empty = [\,]$$

$$cons :: a \to RASeq\ a \to RASeq\ a$$
$$cons\ x\ [x_0, \ldots, x_{n-1}] = [x, x_0, \ldots, x_{n-1}]$$

$$head :: RASeq\ a \to a$$
$$head\ [x_0, \ldots, x_{n-1}] = x_0 \quad (n \geq 1)$$

$$tail :: RASeq\ a \to RASeq\ a$$
$$tail\ [x_0, \ldots, x_{n-1}] = [x_1, \ldots, x_{n-1}] \quad (n \geq 1)$$

$$snoc :: RASeq\ a \to a \to RASeq\ a$$
$$snoc\ [x_0, \ldots, x_{n-1}]\ x = [x_0, \ldots, x_{n-1}, x]$$

$$last :: RASeq\ a \to a$$
$$last\ [x_0, \ldots, x_{n-1}] = x_{n-1} \quad (n \geq 1)$$

$$init :: RASeq\ a \to RASeq\ a$$
$$init\ [x_0, \ldots, x_{n-1}] = [x_0, \ldots, x_{n-2}] \quad (n \geq 1)$$

$$lookup :: RASeq\ a \to Int \to RASeq\ a$$
$$lookup\ [x_0, \ldots, x_{n-1}]\ i = x_i \quad (0 \leq i \leq n-1)$$

$$update :: RASeq\ a \to Int \to a \to RASeq\ a$$
$$update\ [x_0, \ldots, x_{n-1}]\ i\ x = [x_0, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{n-1}] \quad (0 \leq i \leq n-1)$$

Figure 2.2: Specification of a sequence supporting random-access. For the purposes of specification, we treat a random-access sequence as a list.

Random-Access Sequences			
Name	Lazy	Complexities of Operations	Reference
Naïve Lists	-	*cons/head/tail*: $O(1)$, *lookup/update*: $O(i)$, *snoc/last/init*: $O(n)$	n/a
Threaded Skew Binary Lists	-	*cons/head/tail*: $O(1)$, *lookup*: $O(\min(i, \log n))$, *update*: $O(i)$	[29]
Balanced Trees	-	*cons/head/tail*: $O(\log n)$, *lookup/update*: $O(\log n)$, *snoc/last/init*: $O(\log n)$	[2, 31]
Braun Trees	-	*head*: $O(1)$, *cons/tail*: $O(\log n)$, *lookup/update*: $O(\log i)$, *snoc/last/init*: $O(\log n)$	[21]
Slowdown Deques	-	*cons/head/tail*: $O(1)$, *lookup/update*: $O(\log d)$, *snoc/last/init*: $O(1)$	[24]
Skew Binary Lists	-	*cons/head/tail*: $O(1)$, *lookup/update*: $O(\min(i, \log n))$	[33]
Elevator Lists	-	*cons/head/tail*: $O(1)$, *lookup/update* : $O(i)$	n/a

Table 2.2: Complexities of implementations of sequences supporting random-access, where n is the length of the sequence, i is the index being accessed by a *lookup* or *update* operation, and d is the distance from the index to the nearest end of the sequence. All complexities are worst-case. None of the implementations require lazy evaluation.

2.2.1 Naïve Lists

An ordinary list provides $O(1)$ access to the front and $O(i)$ access to the i^{th} element.

2.2.2 Threaded Skew Binary Lists

Myers [29] extends the ordinary list implementation with an efficient *lookup* operation, whilst preserving the complexities of the other operations.

Myers uses a number system called *skew binary* that proves very useful in many data structures [7, 32, 38]. The advantage of this system of representing numbers is that no more than a single carry is caused by an addition or subtraction of one. Each digit is either 0 or 1, except the least-significant non-zero digit, which is either 1 or 2. The i^{th} digit has weight $2^{(i+1)} - 1$ as opposed to the usual 2^i of ordinary binary numbers. For example,

$$(120)_{2_s} = (1 \times 7 + 2 \times 3 + 0 \times 1)_{10} = (13)_{10}$$
$$(11111)_{2_s} = (31 + 15 + 7 + 3 + 1)_{10} = (57)_{10}$$

where $(x)_b$ is the number given by x under base notation b, with 2_s standing for skew binary, 2 for binary and 10 for decimal. With skew binary, addition of one produces at most one carry, for example,

$$(120 + 1)_{2_s} = (200)_{2_s}$$

whereas with binary we could have a cascade of carries,

$$(111 + 1)_2 = (1000)_2$$

Removing the possibility of such a cascade allows us to perform an addition or subtraction of one by changing at most two digits, irrespective of the size of the number.

Myers uses the skew binary number system to add auxiliary pointers to ordinary lists. These provide access to elements further down the list. A list of seven elements $[v_7, \ldots, v_1]$, with v_7 at the front is shown in Figure 2.3. Along with the value v_i of each element in the list, we store the position POS of v_i from the end of the list, a pointer NEXT to the next element down from v_i, and a pointer

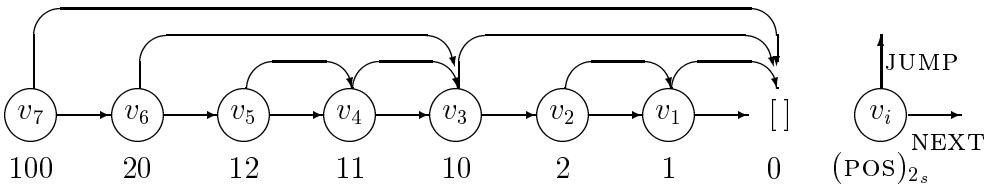


Figure 2.3: An example of a threaded skew binary list. The empty list is represented by [ ].

JUMP to an element further down the list with POS equal to j. The value of j is determined as follows: take the POS of v_i in skew binary, and reduce the least-significant non-zero digit by one. For example, element v_6 has $\text{POS} = (20)_{2_s}$ and hence its JUMP should point to the element with $\text{POS} = (10)_{2_s}$, namely v_3. Using the JUMP pointers where possible, *lookup* now runs in $O(\min(i, \log n))$ time.

As with ordinary lists, however, *update* still runs in $O(i)$ time. There is a series of pointers to the updated element from every preceding element. Therefore each of these elements must have their pointers updated.

Maintaining the JUMP pointers can be done in $O(1)$ time as follows. Consider a list with head element s. Let the JUMP of s point to t. Let the JUMP of t point to u. To *cons* an element onto the list, compare the distance between s and t, with the distance between t and u. If the two distances are equal, analogous to the least significant non-zero digit of a skew binary number being two, we point JUMP to u, analogous to carrying one in skew binary. If the two distances are not equal then we point JUMP to s.

For example, consider how the JUMP of v_7 was calculated. At the time v_7 was added to the list, the head element was v_6. The JUMP of v_6 points to v_3, and the JUMP of v_3 points to [ ]. The distance between v_6 and v_3 is the same as the distance between v_3 and [ ]. Hence the JUMP of v_7 should point to [ ].

Myers uses pointers to describe and implement his data structures, taking explicit care to ensure that the structures are persistent. With algebraic data-types, the persistent property is enforced and no pointers are mentioned. The type of Myers' list would be given in Haskell by:

```
data RASeq a = Empty
             | Node a (RASeq a) (RASeq a) Int
```

The list with head element `v`, NEXT pointing to the list `next`, JUMP pointing to the list `jump`, and POS equal to `pos` would be given by `Elem v next jump pos`. For example, $\texttt{l}_7$ = `Elem` $\texttt{v}_7$ `Empty` $\texttt{l}_6$ `7` (with a suitable definition of $\texttt{l}_6$, etc.) gives the list $\texttt{l}_7$ in Figure 2.3.

Okasaki [32, 38] gives an implementation of random-access lists that is essentially an unthreaded version of Myers' implementation. See Section 2.2.6 for a comparison of these two data structures. Okasaki constructs his lists with algebraic data-types. Comparing Okasaki's implementation with Myers' illustrates well how algebraic data-types can provide clarity and insight.

Okasaki [32] benchmarks Myers' implementation, improving the code slightly by maintaining the difference between the POS of an element and the POS of the element to which JUMP points. This value is called the *rank* of an element. The POS of each element is no longer maintained and the calculation of the JUMP involved in an application of *cons* is now simpler and more efficient. Appendix A gives this improved implementation.

2.2.3 Balanced Trees

Various forms of balanced tree may be used to implement a random-access sequence. Most of these implementations offer $O(\log n)$ access to any element. Braun trees are a notable exception and offer improved access to the front of the sequence whilst maintaining logarithmic access to any element as an upper bound. They are therefore treated separately in Section 2.2.4.

AVL trees [3, 31] are straightforward but tedious to implement. Okasaki uses an implementation adapted specifically for random-access lists in [33]. Appendix A gives this implementation.

Adams [2] provides an alternative in the form of *BB-trees*. Adams' implementation seems to be quite widely used, so we shall look at it below. Other forms of balanced trees are documented well in imperative literature and most translate across easily to the purely functional or persistent worlds.

Adams gives an implementation of *sets* using BB-trees, which we describe below. The modifications required for implementing random-access sequences are minor (see the code in Appendix A).

BB-Trees

Adams represents a BB-tree as follows:

```
data Set a = Empty
           | Branch Int (Set a) a (Set a)
```

For a non-empty tree `Branch n l x r`, we have:

- A node containing an element `x` and the number `n` of elements in the tree
- The left subtree `l`
- The right subtree `r`

The elements are stored in symmetric order; that is, given any non-empty subtree `Branch n l x r`, every element in the tree `l` is less than or equal to `x`, and `x` is less than or equal to every element in the tree `r`. The following balancing invariant is maintained:

> Given a subtree `Branch n l x r` containing more than two elements, neither `l` nor `r` has more than σ times the number of elements of the other.

To restore the balance of a tree after adding or removing an element, whilst maintaining the order of elements, we need to perform *rotations*. Figure 2.4 shows the four forms of rotation required and Figure 2.5 shows the corresponding code. Note that the trees are constructed using the *function* `branch`, not the data constructor `Branch`, and that branch does not take size as an argument. The function `branch` calculates the size of the tree from the sizes of the left and right subtrees. This avoids unnecessarily verbose code produced by calculating the size separately each time a tree is constructed (as would be necessary if `Branch` was used directly). Adams calls these functions *smart constructors*. Two further smart constructors are given:

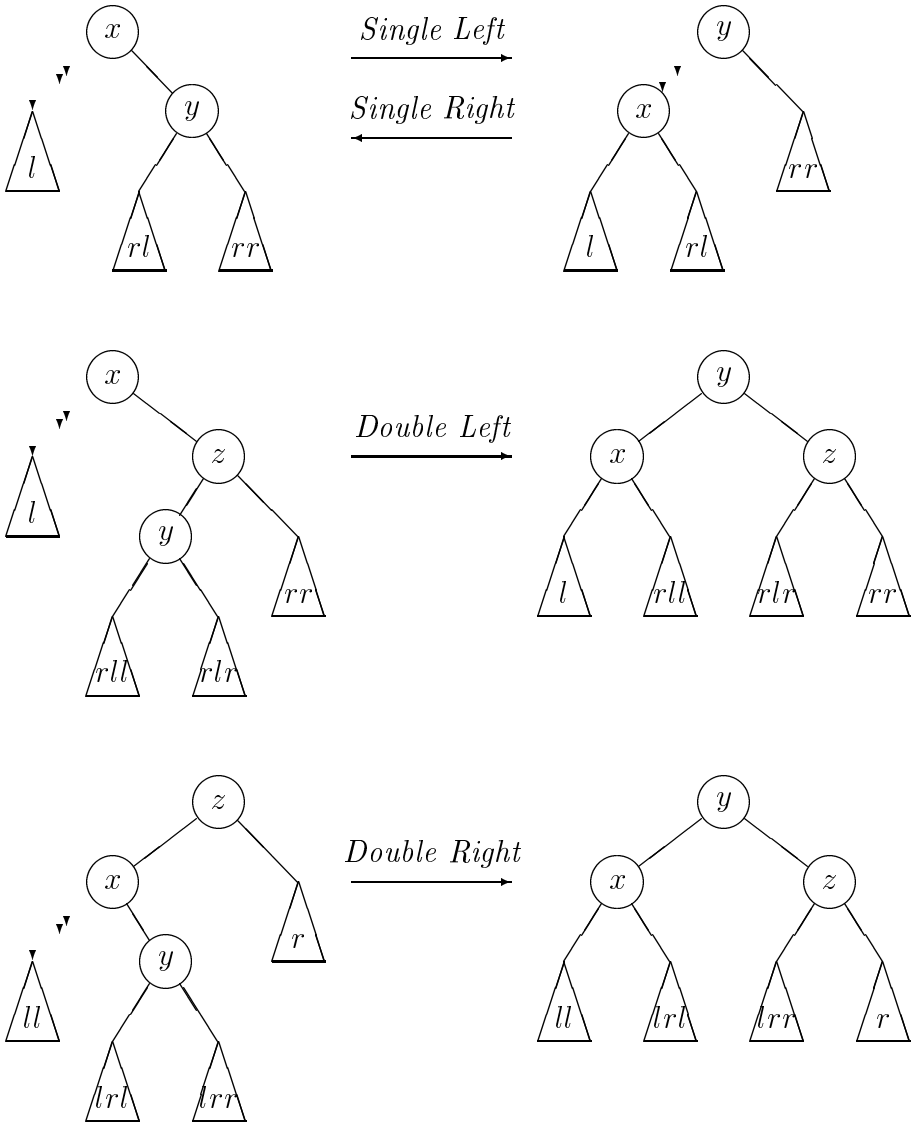


Figure 2.4: Rotations of a binary tree.

- `balBranch`, which constructs a balanced tree from a previously balanced tree that has had at most one element deleted or added to one of its subtrees, both of which are assumed to be now balanced

- `concat3`, which constructs a balanced tree from a node and two subtrees of arbitrary size

Adding or removing a single element to or from a subtree may require a rotation to restore the balancing invariant. An unbalanced tree with a large left or right subtree requires a right or left rotation respectively. Let's suppose that the right subtree r is too large.

```
branch :: Set a -> a -> Set a
branch l x r = Branch (1 + size l + size r) l x r

size Empty = 0
size (Branch n l x r) = n

singleL l x (Branch _ rl y rr) = branch (branch l x rl) y rr
singleR (Branch _ l x rl) y rr = branch l x (branch rl y rr)
doubleL l x (Branch _ (Branch _ rll y rlr) z rr) =
   branch (branch l x rll) y (branch rlr z rr)
doubleR (Branch _ ll x (Branch _ lrl y lrr)) z r =
   branch (branch ll x lrl) y (branch lrr z r)
```

Figure 2.5: Rotating binary trees.

- If the left subtree rl of r is smaller than some constant α times the right subtree rr, then we move rl across to the left subtree l of the main tree to try to restore the balancing invariant whilst preserving the order. This is a *single left* rotation—see Figure 2.4. The rotation also shifts elements x and y round to preserve order.

- If the right subtree rl of r is larger than α times the right subtree rr, then we move only part of rl to restore the balancing invariant. We move the left subtree rll of rl across to the main left subtree l whilst preserving the order of elements—this is what a *double left* rotation does, see Figure 2.4.

The case of the left subtree l being too large is treated symmetrically. The above algorithm can be seen in the code for `balBranch` in Figure 2.6. The function `concat3` simply traverses the tree, restoring balance as necessary by calling `balBranch`.

In a technical report [1], Adams investigates what values of σ and α are sufficient for the algorithm above to maintain the balancing invariant. He produces a graph of suitable combinations of σ and α. As used in Figure 2.6, $\sigma = 5$ and $\alpha = 2$ is one such suitable combination. However, in [2] Adams gives code with

```
sigma :: Int
sigma = 5

alpha :: Int
alpha = 2

balBranch :: Set a -> a -> Set a -> Set a
balBranch l x r
   | sizeL + sizeR < 2 = branch l x r
   | sizeR > sigma * sizeL =
      let (Branch _ rl _ rr) = r
      in if size rl < (size rr) * alpha
         then singleL l x r
         else doubleL l x r
   | sizeL > sigma * sizeR =
      let (Branch _ ll _ lr) = l
      in if size lr < (size ll) * alpha
         then singleR l x r
         else doubleR l x r
   | otherwise = branch l x r
   where sizeL = size l
         sizeR = size r

concat3 :: Ord a => Set a -> a -> Set a -> Set a
concat3 Empty x r = add x r
concat3 l x Empty = add x l
concat3 l@(Branch nl ll x lr) y r@(Branch nr rl z rr)
   | sizeRatio * nl < nr = balBranch (concat3 l y rl) z rr
   | sizeRatio * nr < nl = balBranch ll x (concat3 lr y r)
   | otherwise = branch l y r
```

Figure 2.6: Smart constructors of balanced trees.

$\sigma = 5$ and $\alpha = 1$, which is not suitable. One suspects that the proportion of unbalanced trees is low and the cost of ensuring *all* trees are balanced is greater than the cost taken to navigate the occasional unbalanced tree. However, Adams does not mention this.

Consider the operation `add` that adds an element to a set. The operation `add` descends the tree by recursively calling itself to add the element at the correct position (or returning the tree if the element is present already). As it does so, it may unbalance the tree at each of the nodes lying on its path to the added element's final position. The balancing smart constructor `balBranch` is designed specifically to handle this case by assuming that only a single element has been added or removed since the tree was last in a balanced state and that all subtrees of the two trees it joins are balanced.

```
add :: Ord a => a -> Set a -> Set a
add x Empty = singleton x
add x t@(Branch _ l y r) | x < y = balBranch (add x l) y r
                         | y < x = balBranch l y (add x r)
                         | otherwise = t
```

Other set operations are defined similarly.

2.2.4 Braun Trees

Hoogerwoord [21] uses Braun trees [6] to implement flexible arrays. Braun trees have the following properties:

- For any node of a Braun tree with left subtree l and right subtree r, $|r| \leq |l| \leq |r| + 1$.
- The size of a Braun tree determines its structure exactly.
- Every Braun tree is of minimum height.

Consider the infinite tree of Figure 2.7. Now consider the subtree formed by removing all nodes bar those labelled with numbers in the range $[0..n-1]$ inclusive. This is the Braun tree of size n. For examples of Braun trees, see Figure 2.8. The pattern of how the nodes are labelled is best illustrated by the *lookup* operation.

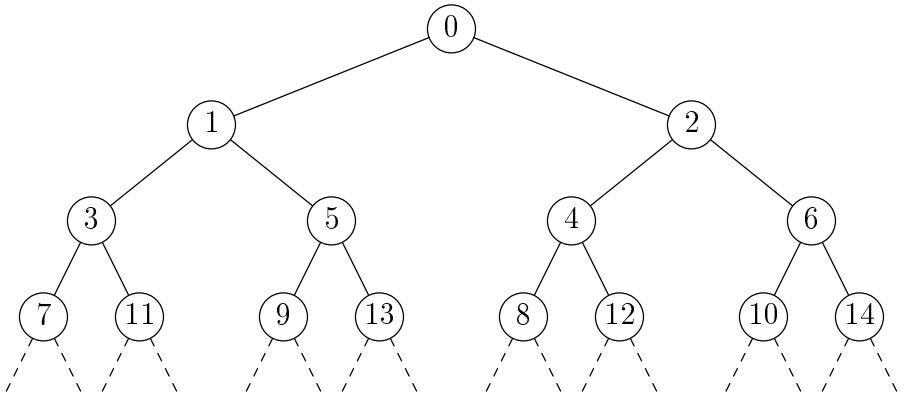


Figure 2.7: The infinite Braun tree.

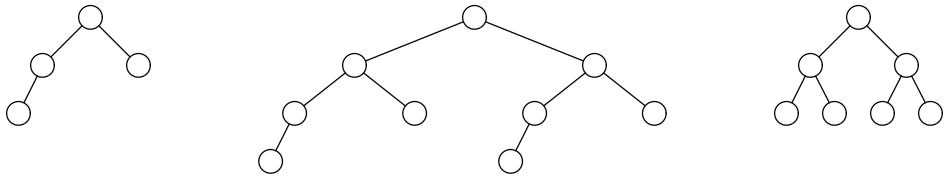


Figure 2.8: The Braun trees of size four, nine and seven.

To *lookup* the n^{th} element of Braun tree T with left subtree l and right subtree r, use the following rules:

- If $n = 0$, then return the root element of T.
- If n is even, then return the $((n/2) - 1)^{\text{th}}$ element of r.
- Otherwise, n is odd, so return the $((n-1)/2)^{\text{th}}$ element of l.

The *update* operation is defined similarly. As every Braun tree is of minimum height, these operations run in $O(\log n)$ time. Treating the trees as lists, it is possible to define *cons* and *tail* to run in $O(\log n)$ time, and *head* in $O(1)$ time.

Hoogerwoord implements flexible arrays, whereas we want random-access lists—we shall now explain the difference. When an element is added or removed from the front of a random-access list, the positions of the other elements in the list shift. If instead positions remain fixed, we have a flexible array. For example, consider applying *cons* to the list $l_1 = [0, \ldots, n]$ to give the list $l_2 = [-1, 0, \ldots, n]$. Both a random-access list and a flexible array give *lookup* l_1 $i = i$. However, a random-access list gives *lookup* l_2 $i = i - 1$, whereas a flexible array gives *lookup* l_2 $i = i$. It is simple to extend an implementation

of a random-access list to give flexible array behaviour, and vice versa. The algorithm we have described above, and the code in Appendix A, both implement random-access lists.

2.2.5 Slowdown Deques

Kaplan and Tarjan [24] introduce the technique of *recursive slowdown* and use it to implement many data structures, including double-ended queues (*deques*). Section 2.1.8 gives a brief explanation of recursive slowdown. The deques can also be made to support random access.

A deque is represented by a prefix of up to five elements, an inner central deque of pairs of elements, and a suffix of up to five elements. A large deque is therefore made up of many deques nested within each other. The outermost level contains simple elements in its prefix and suffix, the second level pairs of elements, the third level pairs of pairs of elements, etc. As with the implicit queues of Section 2.1.8, we make sure that an operation on the inner deque takes place every two operations on the outer deque. To do this, we need to make sure that the prefix and suffix are kept close to being half full to avoid cascades of operations on nested deques. Kaplan and Tarjan introduce a colour scheme to identify prefixes and suffixes with dangerously few or many elements: red for zero or five elements, yellow for one or four elements, and green for two or three elements. A deque is coloured according to the most dangerous colour of its prefix or suffix. The following invariant is then maintained:

> There is a green deque outside of the outermost red deque. There is also a green deque between any two red deques.

This ensures that the outermost deque is always in a state ready to accept a new element or to give up a current element. The details of how to juggle the prefixes and suffixes to maintain this invariant are complex and not given here. Maintaining the invariant may require performing an operation on the inner deque. However, an operation on the inner deque is only necessary if the outer deque is red. The invariant ensures that when the outer deque is red, the inner deque is not red, hence preventing a cascade of operations on nested inner deques. The invariant can be maintained with a constant amount of work per operation. As

the invariant guarantees that the deque is ready to perform an operation in $O(1)$ time, this proves that the deque allows operations on either end to run in $O(1)$ worst-case time.

The operations *lookup* and *update* are implemented by descending the series of nested deques till we reach the prefix or suffix in which the element is stored. If the element is at most d positions from the nearest end of the deque, then the element is at $O(\log d)$ depth since the number of elements stored in each level grows exponentially. As the second level contains pairs of elements, the third level pairs of pairs of elements, and so on, we have to descend this tree-like structure to reach the element. As this tree is also $O(\log d)$ deep, the complexity of *lookup* and *update* is $O(\log d)$.

2.2.6 Skew Binary Lists

Okasaki [32] notes that complete binary trees are a good structure to use for random-access, allowing access and update to any node in $O(\log n)$ time. However, these trees are only found in sizes of the form $2^k - 1$ so the problem remains of how to store lists of arbitrary size. The *skew binary* number system of Section 2.2.2 once more comes to our aid. Recalling that the i^{th} digit represents $2^i - 1$, this number system is ideal for implementing a list of n elements as a collection of complete binary trees according to the representation of n in skew binary (see Figure 2.9). Importantly, the addition or removal of an element involved in the *cons* and *tail* operations is also dealt with in $O(1)$ time thanks to the main property of skew binary numbers: addition or subtraction of one produces at most one carry.

The importance of cheap access to the front of the list for *cons*, *head* and *tail* suggests we order the trees by size, smallest first, and order the elements with left-to-right pre-order.

By analogy with skew binary addition and subtraction, *cons* and *tail* are implemented as follows:

- To *cons* an element onto a list, check if the two smallest trees are the same size. If not, add the new element as a singleton tree. Otherwise, create a larger complete binary tree with the new element as root and the two trees

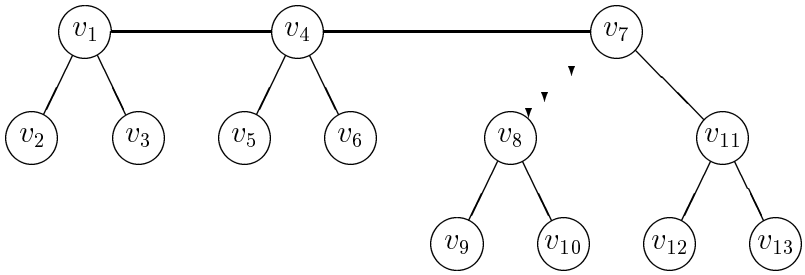


Figure 2.9: A list $[v_1, \ldots, v_{13}]$ represented as a collection of complete binary trees. Number of nodes $= (13)_{10} = 1 \times (2^3 - 1) + 2 \times (2^2 - 1) + 0 \times (2^1 - 1) = (120)_{2_s}$, therefore we have one complete binary tree of depth three, two of depth two and none of depth one.

---

as children—this preserves the ordering and the skew binary form.

- To take the *tail* of a list, simply remove the leading singleton tree if one exists. If not, remove the root of the smallest tree and return both its children to the collection.

These operations are illustrated in Figure 2.10.

The operation *head* is easy to implement in $O(1)$ time. Similarly, *lookup* and *update* are reasonably simple to implement if the size of the tree rooted at each node is stored in the node.

The string representing the number n in the skew binary number system is $O(\log n)$ long. A list of length n is therefore represented by a collection of $O(\log n)$ trees. The largest tree in a list of length n is also $O(\log n)$ deep. The operations *lookup* and *update* traverse the list till the tree containing the desired element is found. This tree is then descended to reach the element. Hence *update* and *lookup* each take $O(\log n)$ time. Upon further examination, we can improve this complexity to $O(\min\{i, \log n\})$ in the worst case and $O(\log i)$ in the expected case, when indexing the i^{th} element.

Parallels can be drawn between Okasaki's lists and Myers' lists (see Section 2.2.2). There are many redundant pointers in Myers' representation, causing *update* to be less efficient, running in $O(i)$ time. The shortest path from the head of the list to any element never uses any of these pointers. By removing them, one

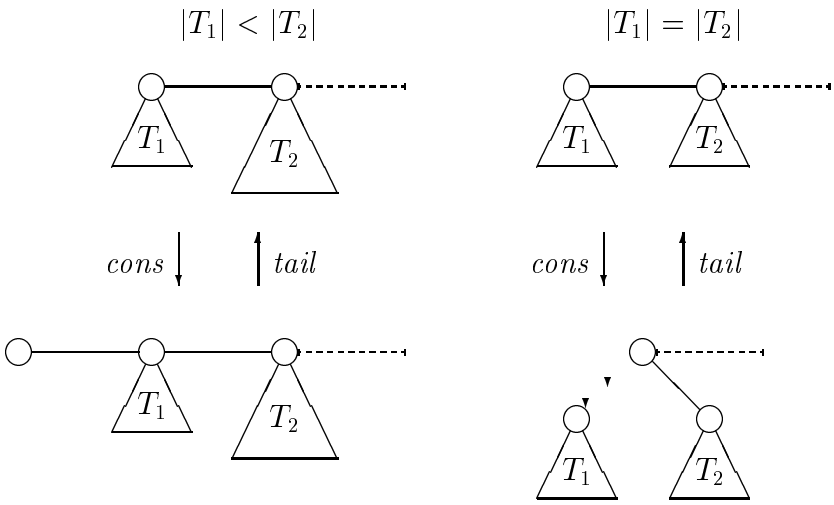


Figure 2.10: The effect of *cons* and *tail* acting on a list represented by a collection of complete binary trees with the smallest two being T_1 and T_2.

---

obtains a structure isomorphic to the same list represented with Okasaki's structure (see Figure 2.11). One can therefore view Okasaki's work as an improvement of Myers' work to gain a more efficient *update*.

Alternatively, one may view Myers' lists as *threaded* versions of Okasaki's lists. A tree is *threaded* when every node contains a pointer to the next element with respect to some traversal order—left-to-right pre-order in this case. This can be seen in Figure 2.11. For example, node v_3 contains a pointer to node v_6. However, for every case where searching through a Myers' list would follow such a pointer, the search in the equivalent list of Okasaki would have followed at least one fewer pointer. For example, the search for v_6 in Okasaki's list moves from v_2 directly to v_6; the search for v_6 in Myers' list moves from v_2 to v_6 via v_3.

2.2.7 Elevator Lists

Preliminary benchmarking results of the implementations of random-access sequences show that the naïve implementation often wins for small lists, and some form of tree wins for large lists. We design an implementation of random-access sequences that is a hybrid of the simple list and the structured tree.

An elevator list is a simple list of *floors*. Each floor is itself a simple list.

```
data List a = Floor Int [a] (List a)
```

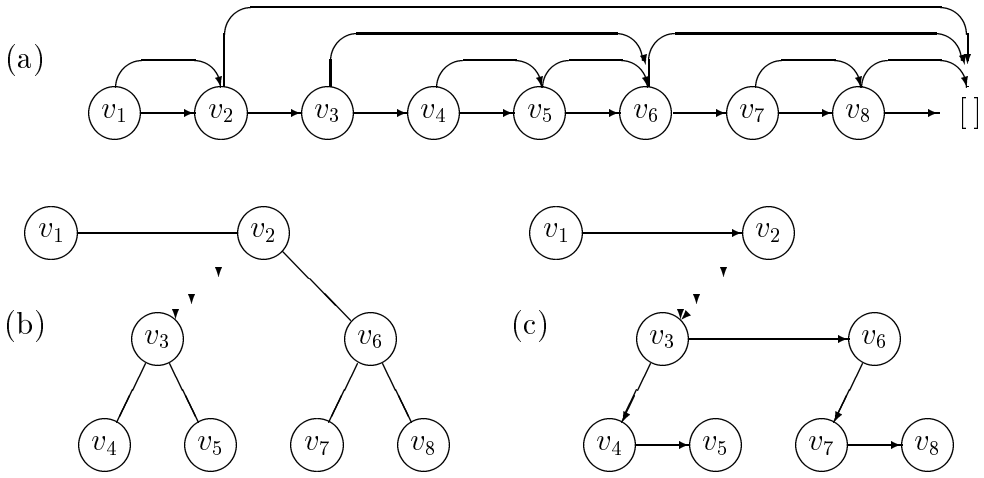


Figure 2.11: A list $[v_1, \ldots, v_8]$ represented (a) by Myers' random-access list, (b) by Okasaki's random-access list, and (c) by Myers' list with redundant pointers removed. Note the similarity between (b) and (c).

We label each floor with its size. There is a fixed "separation" between floors: When the top floor becomes larger than a fixed size, a new floor is built on top.

Ordinary list operations act directly on the top floor. Random-access operations first descend to the correct floor, by subtracting the floor sizes from the index, till the index is less than the floor size, and then use ordinary list lookup and update on this floor.

We represent an empty list by a circular list of empty floors.

```
empty = Floor 0 [] empty
```

For further details, see the code in Appendix A.

2.3 Heaps

Priority queues, or heaps, support an ordered collection of elements. A specification is given in Figure 2.12. A table of implementations can be found at Table 2.3.

$$type\ Ord\ a \Rightarrow Heap\ a = \prec a \succ$$

$$empty :: Ord\ a \Rightarrow Heap\ a$$
$$empty = \prec\succ$$

$$insert :: Ord\ a \Rightarrow\ a \rightarrow Heap\ a \rightarrow Heap\ a$$
$$insert\ x\ h = \prec x \succ \cup\ h$$

$$merge :: Ord\ a \Rightarrow Heap\ a \rightarrow Heap\ a \rightarrow Heap\ a$$
$$merge\ h_1\ h_2 = h_1 \cup\ h_2$$

$$findMin :: Ord\ a \Rightarrow Heap\ a \rightarrow a$$
$$findMin\ h = x\ \wedge\ x \in h\ \wedge\ \forall y \in h \bullet x \leq y \quad (h \neq \prec\succ)$$

$$deleteMin :: Ord\ a \Rightarrow Heap\ a \rightarrow Heap\ a$$
$$deleteMin\ h = h - \prec findMin\ h \succ \quad (h \neq \prec\succ)$$

Figure 2.12: Heap specification. A bag is delimited with $\prec\succ$, $\cup$ is bag union, and $-$ is bag difference.

Heaps			
Name	Lazy	Complexities of Operations	Reference
Naïve	-	*insert/merge*: $O(n)$ *findMin/deleteMin*: $O(1)$	n/a
Binomial	-	*insert/merge*: $O(\log n)$ *findMin/deleteMin*: $O(\log n)$	[38]
Skew Binomial	-	*insert*: $O(1)$, *merge*: $O(\log n)$ *findMin/deleteMin*: $O(\log n)$	[7]
Bootstrapped Skew Binomial	-	*insert/merge*: $O(1)$ *findMin*: $O(1)$, *deleteMin*: $O(\log n)$	[7]
Pairing	-	*insert/merge*: $O(1)$ *findMin*: $O(1)$, *deleteMin*: $O(\log n)$	[35]
Leftist	-	*insert/merge*: $O(\log n)$ *findMin*: $O(1)$, *deleteMin*: $O(\log n)$	[31]
Splay	-	*insert*: $O(\log n)^{\ddagger}$, *merge*: $O(n)^{\ddagger}$ *findMin/deleteMin*: $O(\log n)^{\ddagger}$	[38]

Table 2.3: Complexities of implementations of heaps (priority queues), where n is the size of the heap (the resulting heap in the case of *merge*). Complexities marked with ‡ are amortized under single-threaded use. The complexity of *deleteMin* for pairing heaps is only a conjecture for single-threaded amortized use; this bound has also been conjectured for a persistent version of pairing heaps under amortized persistent use. If lazy evaluation is used, the complexity of *insert* for binomial heaps becomes $O(1)$ amortized. All other complexities are worst-case and none of the implementations require lazy evaluation.

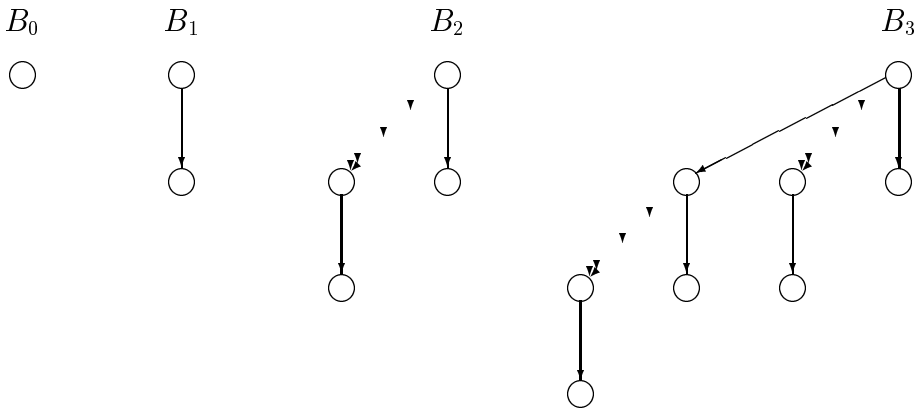


Figure 2.13: The first four binomial trees.

2.3.1 Naïve Heaps

An ordered list implements a heap with *findMin* and *deleteMin* running in $O(1)$ time, and *insert* and *merge* running in $O(n)$ time.

2.3.2 Binomial Heaps

Vuillemin presents *binomial queues* in [51] with every operation running in $O(\log n)$ time. Okasaki [38] preserves this complexity in a purely functional setting. To avoid confusion with ordinary queues, we shall refer to binomial queues as *binomial heaps*.

Binomial Trees

The size of a binomial tree determines its shape exactly: the first four are shown in Figure 2.13. Figure 2.14 shows two equivalent definitions of the binomial tree B_n. The binomial tree B_i has 2^i nodes, ${}^iC_j$ of which are at depth j, where ${}^iC_j = i!(i-j)!/j!$ gives the number of ways of choosing j items from a collection of i items, disregarding order of choice. The name *binomial* derives from the co-efficient of the i^{th} term of a binomial expansion $(x+y)^n$ being given by ${}^{n-i}C_i$.

Given an ordering of elements, a tree is *heap-ordered* if for every node n with parent m, the element stored at n is no smaller than the element stored at m.

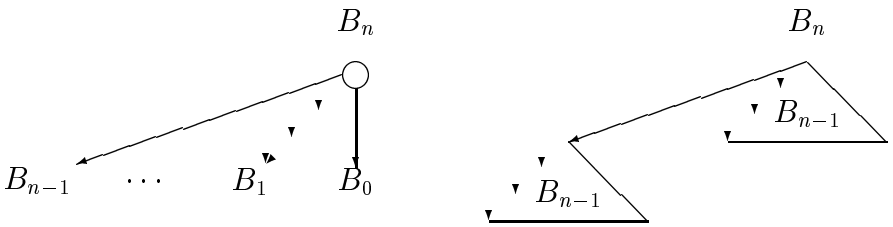


Figure 2.14: Equivalent forms of the binomial tree B_n.

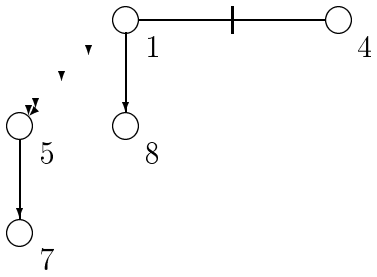


Figure 2.15: An example of a binomial heap: $[B_2, B_0]$. There is no B_1 tree and its absence is indicated by a vertical dash.

A binomial heap is a list of heap-ordered binomial trees: $[B_{i_0}, B_{i_1}, \ldots, B_{i_n}]$ with $i_0 < i_1 < \cdots < i_n$. The size of a binomial heap determines its structure exactly. The binomial tree B_i appears in a binomial heap either once or not at all. An example of a binomial heap can be seen in Figure 2.15.

A useful property of binomial heaps is that the binary representation of the number of nodes within the heap corresponds exactly with the heap representation. For example, the heap in Figure 2.15 has five nodes and its binary equivalent is indeed the number five: "1 B_2, 0 B_1 and 1 B_0" giving "101". The length of the binary representation of the number n is $O(\log n)$. Hence a binomial heap of n elements is a list of length $O(\log n)$.

Operations on Binomial Heaps

An example of a *merge* can be seen in Figure 2.16. Merging binomial heaps is strongly analogous to binary addition. Trees or digits of equal weight are added

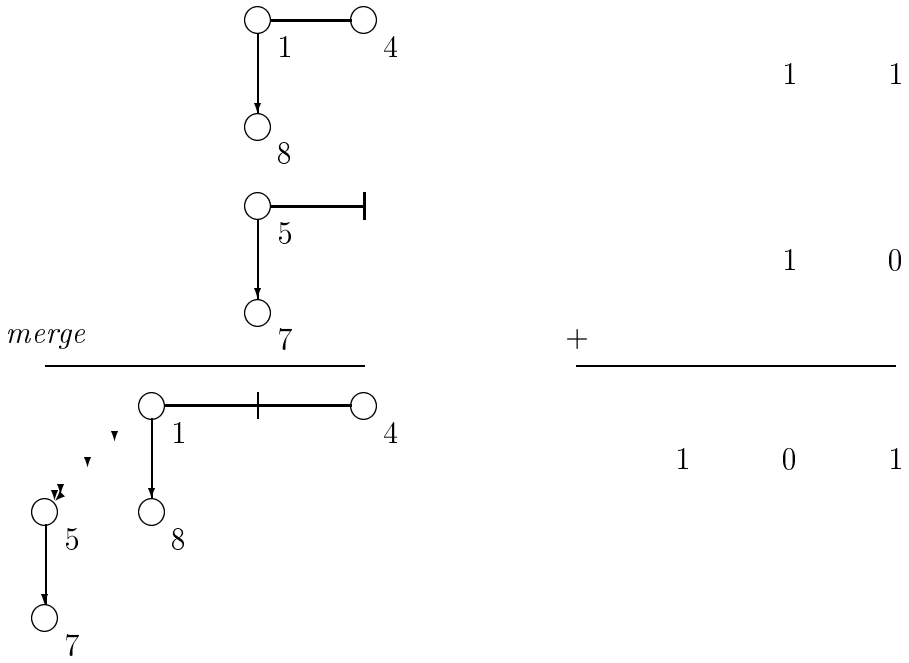


Figure 2.16: A *merge* of two binomial heaps and the corresponding binary addition.

---

together to produce a tree or digit of the next heaviest weight. Two binomial trees of equal weight are added together by making the tree with the larger root the leftmost child of the other tree.

The operation *findMin* simply scans the roots of the binomial trees to be added. The other operations are defined in terms of merge: *deleteMin* q scans for the minimum root, removes it, and merges its children with the remainder of q (the children of the root of a binomial tree always form a binomial heap, as can be seen in Figure 2.14); *insert* i q simply merges q with the singleton queue containing i. As there are $O(\log n)$ binomial trees in a binomial heap of size n, each operation takes $O(\log n)$ time.

2.3.3 Skew Binomial Heaps

Brodal and Okasaki [7] adapt the binomial heap implementation to use *skew binary* arithmetic (see Section 2.2.2) in place of ordinary binary arithmetic. Recall that the addition or subtraction of one takes $O(1)$ time using the skew binary number system. In the case of heaps, this allows *insert* to run in $O(1)$ time. The

other operations maintain their $O(\log n)$ complexity.

A *skew binomial* heap is a list of *skew binomial* trees. Unfortunately, skew binomial trees are not as neat as their binomial counterparts. This is because we must use some form of addition to implement *merge*. Skew binary addition is rather awkward in general and so we choose to use ordinary binary addition. The conflict between using skew binary addition to implement *insert* and ordinary binary addition to implement *merge* reduces the elegance of the implementation. However, making *insert* run in $O(1)$ time allows heaps of *optimal* complexity to be built—see Section 2.3.4.

2.3.4 Bootstrapped Skew Binomial Heaps

Brodal and Okasaki [7], after adding the skew binary number system to binomial heaps, add yet another feature: bootstrapping (see Section 2.1.7). This gives heaps of *optimal* complexity: *deleteMin* runs in $O(\log n)$ time and *findMin*, *insert* and *merge* run in $O(1)$ time. It is easy to show these bounds are optimal using the $\Omega(n \log n)$ bound on sorting n items.

Recall that bootstrapping extends the design of an incomplete or inefficient data structure by using smaller instances of the same data structure. We shall let heaps contain other heaps as elements. This allows *merge* to be implemented by the more efficient *insert*.

Suppose we import a heap implementation that runs *insert* in $O(1)$ time. In the Haskell notation, let the type of these heaps be given by `Old.Heap a`. We wish to create bootstrapped heaps that can contain other heaps. We might consider the type:

```
data Heap a = Heap (Old.Heap (Old.Heap a))
```

Here we have applied a single level of bootstrapping. But the top-level heap contains elements of type `Old.Heap a`. These old heaps contain simple elements of type `a`, and so we cannot insert heaps into them; we need to be able to insert heaps at an arbitrary depth of nesting. We need a recursive definition:

```
data Heap a = Heap (Old.Heap (Heap a))
```

However, we do not have anywhere to store the simple elements of type a with this definition. So instead we store the minimum element at the root as follows:

```
data Heap a = Empty
            | Root a (Old.Heap (Heap a))
```

The old heap implementation will require an ordering of its elements: bootstrapped heaps in this case. This is given by an ordering of the roots.

As bootstrapped heaps are old heaps of bootstrapped heaps, we can *merge* two bootstrapped heaps by using *Old.insert* to insert one into the other. As *Old.insert* is $O(1)$, *merge* is $O(1)$. We can define *insert* in terms of *merge* as usual, and so *insert* is still $O(1)$. The operation *findMin* simply looks at the root. The operation *deleteMin* is implemented in terms of *Old.merge*, *Old.findMin* and *Old.deleteMin* and therefore remains $O(\log n)$ (assuming that the old heaps implement these operations in $O(\log n)$ time).

2.3.5 Pairing Heaps

Okasaki [35] presents a functional translation of *pairing heaps* which were first described by Fredman, Sedgewick, Sleator, and Tarjan [15]. A heap is represented by a heap-ordered multi-way tree:

```
data Heap a = Empty
            | Node a [Heap a]
```

The operation *findMin* simply looks at the root. Two heaps are merged by making the heap with the largest root the leftmost child of the other heap. An element is inserted by merging with a heap containing the single element. Pairing heaps derive their name from the implementation of *deleteMin*: the root is removed and the children are combined in two passes. The first pass working left-to-right merges successive pairs of children together. The second pass working right-to-left merges the results of the first pass into one heap.

Although pairing heaps are quite well-known, no one has established tight bounds on their complexity. It is clear that all operations beside *deleteMin* run in $O(1)$ time. In an ephemeral setting, it has been conjectured that *deleteMin* runs in $O(\log n)$ amortized time. In a persistent setting however, the above

implementation certainly does not meet these bounds. Consider successively inserting the elements $0, 1, \ldots, n$ into an empty heap. The result will be a heap with root 0 and children $[n, \ldots, 1]$. Now perform *deleteMin* on the same heap m times. Each *deleteMin* will repeat the same work taking $O(n)$ time each. The amortized cost of *deleteMin* is therefore $O(n)$ in a persistent setting.

Okasaki [35] also presents a persistent version of pairing heaps using lazy evaluation, which should not be subject to a similar refutation of $O(\log n)$ amortized complexity. However, as with their ephemeral counterparts, a proof is not known. Appendix A gives the ephemeral version.

2.3.6 Leftist Heaps

A *leftist heap* [25] is a heap-ordered binary tree satisfying the *leftist property*:

> The r-height of every left child is greater than or equal to the r-height of its right sibling.

The *r-height* of a binary tree is the number of internal nodes on the path from the root to the rightmost external node—this path is called the *right spine*. One may prove by induction that the r-height of any leftist heap of size $n > 0$ is bounded above by $\log_2 n + 1$.

Leftist heaps are an example of a data structure that translates across easily from the imperative to the persistent or functional world. Núñez et al. present a functional implementation in [31].

To *merge* two leftist heaps, view their right spines as ordered lists. Merging these ordered lists ensures the resulting tree is heap-ordered. This constructs the right-spine from top to bottom. On the way back up, the leftist property is preserved by making the child with the largest r-height the left child. As each pass runs in time proportional to the combined length of the right spines of the arguments of *merge*, the operation runs in $O(\log n)$ time. The remaining operations are straightforward.

2.3.7 Splay Heaps

Okasaki [38] presents an implementation of heaps using *splay trees* [49]. A splay tree is a binary tree that does not maintain any balance information but consistently re-structures itself in a manner that tends to balance the tree. For example, as the elements are stored in symmetric order, the *deleteMin* operation must remove the leftmost node. After this node is removed, the leftmost path is ascended, re-structuring the tree as it does so by shifting elements from left subtrees over to right subtrees. This tends to shorten the leftmost path, improving the time taken for subsequent applications of *deleteMin*.

To *insert* a node x, the tree is split into nodes smaller than x, and nodes larger than x. These subtrees then form the left and right children of x respectively. As the tree is split, it is once again re-structured: if x splits the tree somewhere in the left subtree of the root, then elements are moved over to the right subtree and vice versa. This tends to balance the tree.

The operation *findMin* simply finds the leftmost node. This takes $O(\log n)$ time. If every application of *deleteMin* is accompanied by at most one application of *findMin*, as is often the case, we may amortize the cost of *findMin* to $O(1)$. Otherwise, we may store the minimum element separately from the tree. This may be done without increasing the complexity of the other operations. As this causes more work, this is only advisable when *findMin* is called often.

2.4 Summary

This chapter shows there are many ways to implement the same ADT. But which implementation is best? Does it depend on how we use the data structure? Calculating the complexities of the operations gives us a theoretical answer, but empirical performance may give a different picture.

Therefore, after developing the benchmarking procedures motivated in Section 1.3, we benchmark all of the implementations of this chapter in Chapter 7.

Chapter 3

Datatype Usage Graphs

In Section 1.3 we identified a need to qualify the performance of a data structure by how it is used. We can do this by creating benchmarks whose use of the data structure is well-defined. This information is useless unless we can find out how an application uses a data structure. This chapter outlines a theoretical framework for (a) creating a benchmark from a description of use, and for (b) creating a description of use from an application. Chapter 6 builds on this framework to provide a practical tool to do both (a) and (b).

The ADT framework has a solid basis of literature [52] and is very convenient for abstracting over many data structures—an ADT abstracts over many data structures implementing the same operations. We shall therefore insist on every data structure we deal with being an implementation of some ADT.

The ambiguity of the phrase "how an ADT is used" presents an obstacle. Without an exact definition of this property, we would find it hard to talk about the efficiency of an implementation of an ADT according to how it is used, or indeed about how a particular application uses an ADT. Consider the two applications of queues in Figure 3.1 (see Section 2.1 for a definition of queues). Inspecting the code for each application allows us to see what operations are being performed, in what order, and how the result of one operation may rely on the result of another. But the task is by no means straightforward. With more complicated applications, the task would become extremely difficult. We need a simple record of how an ADT is used by an application.

We use a labelled directed graph. See Figure 3.2 for examples that describe

```
apply :: Int -> (a -> a) -> a -> a
apply n f q = (iterate f q) !! n

snocTrue :: Queue Bool -> Queue Bool
snocTrue q = snoc q True

app1 :: Int -> Bool
app1 n = (head . apply (n-1) tail . apply n snocTrue) empty

app2 :: Int -> Bool
app2 n = (and . map (head . tail) . take n . repeat) nSnocs
   where nSnocs = apply n snocTrue empty
```

Figure 3.1: Two artificial simple applications of queues: `app1` and `app2`. Note that `app2` uses a `where` clause to share the value of `nSnocs`.

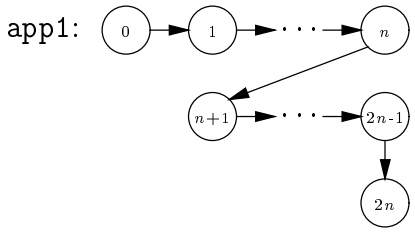
---

how the queue ADT is used by the two applications of Figure 3.1. The nodes are labelled with partially applied operations of the ADT, with the remaining arguments supplied by the arcs. There is an arc from u to v if the result of the operation at u is taken as an argument by the operation at v. The nodes are numbered according to the order of evaluation. Such a graph is a *datatype usage graph* (DUG). We shall make the definition of a DUG precise in the following section.

A DUG is closely related to both an *execution trace* [38] and a *version graph* [13]. An execution trace without cycles and with every operation returning a single result is a DUG. A DUG with every operation returning an ADT value is a version graph. Execution traces have been used as a model on which to explain persistent amortized complexity via lazy evaluation [38]. Version graphs have been used to explain the design of persistent data structures [12, 13, 40].

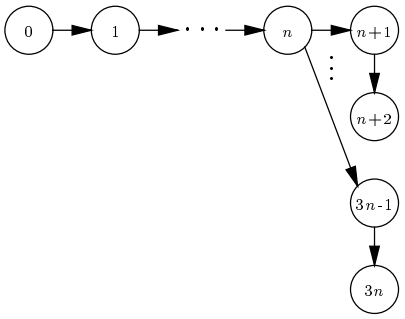
During the run of an application, many different instances of an ADT will exist. For example, whilst running queue application `app1` there will exist at some time an empty queue, a queue containing just `True`, a queue containing two

app1:



Node	Operation
0	*empty*
$1 \ldots n$	$\lambda l \cdot snoc\ l$ True
$n+1 \ldots 2n-1$	$\lambda l \cdot tail\ l$
$2n$	$\lambda l \cdot head\ l$

app2:



Node	Operation
0	*empty*
$1 \ldots n$	$\lambda l \cdot snoc\ l$ True
$n+1$	$\lambda l \cdot tail\ l$
$n+2$	$\lambda l \cdot head\ l$
$\vdots$	$\vdots$
$3n-1$	$\lambda l \cdot tail\ l$
$3n$	$\lambda l \cdot head\ l$

Figure 3.2: Graphs showing how the queue ADT is used by the different applications given in Figure 3.1. Note that node n of `app2` corresponds to the value `nSnocs` shared by n applications of *tail*.

copies of `True`, and so on. Each of these particular instances of the ADT is called a *version* [38] (as defined in Section 1.4). A node of a DUG is called a *version node* if it is labelled with an operation that results in a version. The subgraph of a DUG containing just the version nodes is called the *version graph*. This is consistent with the definition of a version graph given by Driscoll et al. [13].

The rest of this chapter is organised as follows. Section 3.1 defines a DUG precisely. Section 3.2 defines the *evaluation* of a DUG, effectively creating a benchmark. Section 3.3 defines a *profile* of a DUG, summarising the main characteristics. Section 3.4 defines a *shadow data structure*, useful for creating a DUG that matches a given profile, and for adding information to a profile.

3.1 Definition

We should first define what we mean by an ADT. An ADT provides operations to create, manipulate, and observe values of some new type. The only way to interact with values of this type is through the ADT operations. This allows the *implementation* of the ADT to be removed from its *use*—we may exchange implementations without changing how we use the ADT. We have therefore abstracted away from the implementation.

We shall restrict ourselves to *container types*, that is, ADTs that contain elements of some other type. For example, a list ADT allows lists of integers, lists of characters, etc. For any such ADT, we may consider the ADT as defining a type constructor T. For example, a list ADT may be taken as defining a type constructor *List* taking a type t to the type *List t*. A list of integers would then have the type *List Int*. We shall restrict T to be unary. Most common ADTs satisfy these restrictions.

Definition 3.1 (ADT)

For any type constructor T, and any set of functions F, the pair (T, F) is an ADT if the following are satisfied:

- T is unary.
- Each function in F takes at least one argument of type $T\ a$, or returns a result of type $T\ a$, where a is a type variable.

For the sake of simplicity we shall further restrict the ADTs considered by giving the following definitions.

Definition 3.2 (Simple Type)

For any type constructor T of arity one, we say that the type t is *simple over* T if t

- Can be formed as *type* by the grammar

$$\begin{aligned} \mathit{type} &::= \mathit{argument\_type} \to \mathit{type} \mid \mathit{result\_type} \\ \mathit{argument\_type} &::= T\ a \mid a \mid \mathit{Int} \\ \mathit{result\_type} &::= T\ a \mid a \mid \mathit{Int} \mid \mathit{Bool} \end{aligned}$$

  where a is a type variable

- Contains at least one occurrence of $T\ a$

We shall abbreviate this to saying that t is *simple* where the context makes it unambiguous over which type constructor t is simple.

Example 3.2

The following types are simple over the type constructors *Queue*, *List* and *Set* respectively:

- $\mathit{Queue}\ a \to a \to \mathit{Queue}\ a$
- $\mathit{List}\ a \to \mathit{Int} \to a$
- $\mathit{Set}\ a$

The following are *not* simple over *any* type constructor:

- $\mathit{List}\ a \to \mathit{Queue}\ a$
- $(a \to a) \to \mathit{List}\ a \to \mathit{List}\ a$
- a

Definition 3.3 (Simple ADT)

We define the ADT $\mathcal{A} = (T, \{f_1, \ldots, f_n\})$ to be *simple* if the type of each operation f_i is simple.

```
module List (List,empty,catenate,cons,tail,head,lookup,isEmpty)
   where
empty    :: List a
catenate :: List a -> List a -> List a
cons     :: a -> List a -> List a
tail     :: List a -> List a
head     :: List a -> a
lookup   :: List a -> Int -> a
isEmpty  :: List a -> Bool
```

Figure 3.3: Haskell code giving the signature of a simple list ADT $\mathcal{A}_{List}$ providing normal list operations, catenation and indexing. The exported type constructor is `List`. The type of each operation is simple over `List`.

Example 3.3

The signature of a simple ADT $\mathcal{A}_{List}$ is given in Figure 3.3.

Many ADTs are simple: queues, deques, lists, random-access sequences, heaps, sets, integer finite maps, etc. However, any higher-order operations such as *map*, or any operations converting from one data structure to another such as *fromList*, need to be excluded.

When talking about DUGs we shall find it useful to classify the operations according to the different roles they play. We therefore make the following definition.

Definition 3.4 (Generator, Mutator, Observer, Role, Version Arity)

For any operation f of type t, where t is of the form

$$t = t_1 \to t_2 \to \cdots \to t_m$$

and is simple over the type constructor T, f is classified as follows:

Generator	If $t_m = T\ a$ and $(\forall j, 1 \leq j < m)\ t_j \neq T\ a$
Mutator	If $t_m = T\ a$ and $(\exists j, 1 \leq j < m)\ t_j = T\ a$
Observer	If $t_m \neq T\ a$ and $(\exists j, 1 \leq j < m)\ t_j = T\ a$

Note that the categorisation is complete and any operation of simple type is exactly one of: generator, mutator or observer. This is called the *role* of the operation. We define the *version arity* of an operation to be the number of version arguments taken by that operation. Therefore, every generator has version arity 0, and every mutator and observer has version arity greater than or equal to 1.

Example 3.4

Looking at the signature of the simple ADT $\mathcal{A}_{List}$ in Figure 3.3, `empty` is a generator; `catenate`, `cons` and `tail` are mutators; `head`, `lookup` and `isEmpty` are observers. Every mutator and observer has version arity 1, apart from `catenate`, which has version arity 2.

Look at the DUGs in Figure 3.2. The label attached to a DUG node is a partial application of an ADT operation. For simplicity, the arguments used to partially apply the operation are restricted to atomic values—nested function applications are not allowed. The remaining arguments are supplied by the arcs. We shall now define the functions that label DUG nodes.

Definition 3.5 (Partial Application, $Pap(\mathcal{A})$)

Given a simple ADT $\mathcal{A} = (T, \{f_1, \ldots, f_n\})$, a *partial application* of f_i is any function of the following form:

$$\lambda x_1 \cdot \lambda x_2 \cdot \ldots \cdot \lambda x_k \cdot f_i \; a_1 \; a_2 \; \ldots \; a_m, \quad 0 \leq k \leq m$$

Here, m is the arity of f_i, each x_j occurs exactly once in the sequence $[a_1, \ldots, a_m]$, and every other element of this sequence is an atomic value. To avoid duplication, we further insist that $x_1, \ldots, x_k$ occur in order in the sequence $[a_1, \ldots, a_m]$, that is, x_{j_1} occurs before x_{j_2} for $j_1 < j_2$. The set of all partial applications of any function of a simple ADT $\mathcal{A}$ is denoted by $Pap(\mathcal{A})$.

Example 3.5

For the list ADT $\mathcal{A}_{List}$, whose signature is given in Figure 3.3, the following functions are in $Pap(\mathcal{A}_{List})$:

- $\lambda l \cdot cons$ 'a' l
- $empty$
- $\lambda l_1 \cdot \lambda l_2 \cdot catenate\ l_1\ l_2$

Whereas, the following functions are not:

- $\lambda l \cdot catenate\ l\ l$
- $\lambda l_1 \cdot \lambda l_2 \cdot catenate\ l_2\ l_1$
- $\lambda l_1 \cdot \lambda l_2 \cdot cons\ (lookup\ l_1\ 2)\ l_2$

We may use a partial application to assign a role to a node: For a node v labelled with a partial application of the operation f, the role of v is defined to be the role of f. For example, looking at the DUG for `app1` in Figure 3.2, node 0 is a generator, nodes 1 to $2n-1$ are mutators, and node $2n$ is an observer.

We are now in a position to give a definition of a DUG. For nodes with more than one incoming arc, we need to identify which arc corresponds to which argument. We therefore label every arc to such a node with an argument position.

Definition 3.6 (DUG)

Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a simple ADT $\mathcal{A} = (T, \{f_1, \ldots, f_n\})$, a total mapping $\eta : \mathcal{V} \to Pap(\mathcal{A})$, and a bijection $\sigma : \mathcal{V} \to \{1..|\mathcal{V}|\}$, let $\mathcal{E}_P \subset \mathcal{E}$ be those arcs incident to a node with more than one incoming arc, and let $\tau : \mathcal{E}_P \to \mathbb{N}$ be a total mapping. The 4-tuple $(\mathcal{G}, \eta, \sigma, \tau)$ is a DUG for $\mathcal{A}$, if for every $v \in \mathcal{V}$ the following properties are satisfied:

1. The arity of $\eta(v)$ equals the in-degree of v.
2. If v has more than one incoming arc, τ restricted to the incoming arcs is a bijection with the set $\{1..indegree(v)\}$.
3. The application of $\eta(v)$ to the arguments given by $\mathcal{E}$ and τ is type consistent.
4. If v has successor $w \in \mathcal{V}$, $\sigma(v) < \sigma(w)$.
5. The type of every argument of $\eta(v)$ is $T\ a$.

Properties 1–3 ensure the DUG is well-defined. Properties 4–5 impose restrictions on DUGs to make generating DUGs easier: Property 4 orders the arguments of an operation before the operation itself—note that this forces the graph to be acyclic—see the problem *Choosing the operation before the arguments* of Section 4.1.1 for justification of this restriction; Property 5 ensures only version arguments are taken from the results of other operations—see the problem *Choosing non-version arguments from the graph* of Section 4.1.1 for justification.

Example 3.6

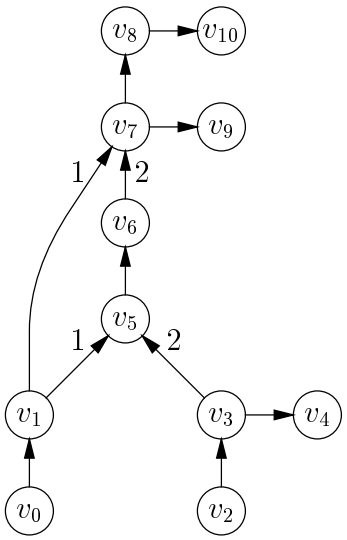
Once again using the ADT $\mathcal{A}_{List}$, whose signature is given in Figure 3.3, an example of a DUG is shown in Figure 3.4. A table defines η. The ordering σ of the evaluation of the nodes is given by: $\sigma(v_i) = i$. Labels assigned by τ are written beside the relevant arcs: v_5 catenates v_1 onto the front of v_3, and v_7 catenates v_1 onto the front of v_6. The type variable a can be substituted by the type *Char* to obtain type consistency for every function application.

As each operation returns only a single value, we may associate each node with the value it produces. The nodes of the version graph are associated with versions formed by either generating a fresh version or by mutating one or more previous versions. The arcs *within* the version graph represent the flow of data *within* the privacy of the ADT framework. The arcs going *out* from the version graph represent the flow of data *out* of the privacy of the ADT framework.

3.2 Evaluation

We have so far presented a DUG as a record of how an application uses an implementation of an ADT. We can reverse this process. By creating an *evaluator* of DUGs, we create an application that uses an ADT implementation in the manner given by the DUG it evaluates. We can then use this application as a benchmark with a known pattern of use.

For example, evaluating the DUG for `app1` of Figure 3.2 should create an *empty* queue, then *snoc* the value *True* onto the queue n times, then take the *tail* of the



	η
v_0	*empty*
v_1	$\lambda l \cdot$ *cons* 'c' l
v_2	*empty*
v_3	$\lambda l \cdot$ *cons* 'h' l
v_4	$\lambda l \cdot$ *head* l
v_5	$\lambda l_1 \cdot \lambda l_2 \cdot$ *catenate* l_1 l_2
v_6	$\lambda l \cdot$ *tail* l
v_7	$\lambda l_1 \cdot \lambda l_2 \cdot$ *catenate* l_1 l_2
v_8	$\lambda l \cdot$ *tail* l
v_9	$\lambda l \cdot$ *lookup* l *1*
v_{10}	$\lambda l \cdot$ *isEmpty* l

Figure 3.4: A DUG for the list ADT $\mathcal{A}_{List}$ (see Figure 3.3).

---

queue $(n-1)$ times, and finally apply *head*. We will define evaluation by first defining how we may associate each node with a function application.

Definition 3.7 (Interpretation of Partial Applications)

Let $\mathcal{A}$ be any simple ADT. Let f be an operation of $\mathcal{A}$. Let $g \in Pap(\mathcal{A})$ be any partial application of f. Let $\mathcal{I}$ be an implementation of $\mathcal{A}$. The interpretation of g under $\mathcal{I}$, denoted by $[\![g]\!]_{\mathcal{I}}$, is the value of g using the implementation of f in $\mathcal{I}$.

Example 3.7

Let $\mathcal{L}$ be the ordinary Haskell implementation of lists, then

- $[\![\lambda l \cdot cons\ True\ l]\!]_{\mathcal{L}}$ = `\l -> (True:l)`
- $[\![\lambda l \cdot head\ l]\!]_{\mathcal{L}}$ = `\(x:xs) -> x`
- $[\![empty]\!]_{\mathcal{L}}$ = `[]`

Definition 3.8 (Interpretation of Nodes)

Let $(\mathcal{G}, \eta, \sigma, \tau)$ be any DUG for the ADT $\mathcal{A}$, let v be any node of $\mathcal{G}$, and let

$\mathcal{I}$ be an implementation of $\mathcal{A}$. Let $e_1, \ldots, e_k$, $k \geq 0$, be the arcs incident to v, ordered by τ, from the nodes $v_1, \ldots, v_k$ respectively. The interpretation of v under $\mathcal{I}$, denoted by $[\![v]\!]_{\mathcal{I}}$, is the following expression:

$$[\![v]\!]_{\mathcal{I}} = [\![\eta(v)]\!]_{\mathcal{I}}\ [\![v_1]\!]_{\mathcal{I}}\ \ldots\ [\![v_k]\!]_{\mathcal{I}}$$

where the right-hand side is an application of the function $[\![\eta(v)]\!]_{\mathcal{I}}$. Note that as $\mathcal{G}$ is acyclic, this recursive definition is sound.

Example 3.8

Using the DUG shown in Figure 3.4, and the ordinary Haskell implementation $\mathcal{L}$ of lists,

- $[\![v_1]\!]_{\mathcal{L}}$ =`(\l -> ('c':l)) []`
- $[\![v_4]\!]_{\mathcal{L}}$ =`(\(x:xs) -> x) ((\l -> ('h':l)) [])`

3.2.1 Order of Evaluation

The order of evaluating the interpretations of the DUG nodes can significantly affect efficiency. Within functional languages there are two main schemes for deciding the order of evaluation of an expression: lazy and eager. We shall accomodate *both* schemes by using the node ordering of a DUG $(\mathcal{G}, \eta, \sigma, \tau)$ given by σ in two separate ways.

Lazy Evaluation

If we consider how a function is applied under lazy evaluation, we see that a closure representing the application is first formed, then its value is perhaps demanded one or more times, and then it is garbage collected. The formation of the closure can be a separate incident to its value being demanded. The order of the formation of the closures can also affect efficiency. Hence we shall order the forming of the closures of the expressions given by the interpretations of each DUG node.

Under lazy evaluation, only the work required to form the demanded result is performed. We must demand a result or no work will be done. Within the ADT framework, we cannot look within an ADT value, so we instead demand the

values that are of some other type. Looking at a DUG, only the values given by the observer nodes have such a type. The order in which we demand these values will affect efficiency.

Within the current framework we shall insist that the order in which we demand the evaluation of the observer nodes coincides with the order of the formation of the closures associated with observer nodes, ie. as soon as we form a closure for an observer node, we demand it. There is the possibility for an extension here to allow for these to occur at different times.

Definition 3.9 (Lazy Evaluation of a DUG)

Given a DUG $(\mathcal{G}, \eta, \sigma, \tau)$ for an ADT $\mathcal{A}$, and an implementation $\mathcal{I}$ of $\mathcal{A}$, the *lazy evaluation* of the DUG with respect to $\mathcal{I}$ is the process of performing the following steps on each node $\sigma(i)$ in order:

- Form the closure given by $[\![\sigma(i)]\!]_{\mathcal{I}}$.
- If the node is an observer, demand the value of this closure.

Example 3.9

The lazy evaluation of the DUG of Figure 3.4 would form the closures $[\![v_i]\!]$ for $0 \leq i \leq 10$ in order. When the closures for the observer nodes are formed, namely $[\![v_4]\!]$, $[\![v_9]\!]$, and $[\![v_{10}]\!]$, their value is demanded at the same time.

Eager Evaluation

Whereas with lazy evaluation many applications of functions may remain unevaluated closures, under eager evaluation they will always be reduced. Hence the eager evaluation of a DUG will evaluate every node and there is no distinction between forming a function application and evaluating it.

Definition 3.10 (Eager Evaluation of a DUG)

Given an ordered DUG $(\mathcal{D}, \sigma)$, and an implementation $\mathcal{I}$ of $\mathcal{A}$, the evaluation of the DUG with respect to $\mathcal{I}$ is the process of taking each node $\sigma(i)$ in order and evaluating the application given by $[\![\sigma(i)]\!]_{\mathcal{I}}$.

Example 3.10

The eager evaluation of the DUG of Figure 3.4 would simply evaluate each $[\![v_i]\!]$ for $0 \leq i \leq 10$ in order.

3.2.2 Abstract Evaluation

The most abstract implementation of an ADT is the ADT itself. We use the abstract operations to create, manipulate, and observe abstract values. These abstract values only exist within the abstract world of mathematics, not within any machine.

Definition 3.11 (Abstract Evaluation)

The *abstract evaluation* of a DUG for the ADT $\mathcal{A}$ is a mapping ξ that takes a node v to the result of evaluating $[\![v]\!]_{\mathcal{A}}$.

Example 3.11

The abstract evaluation ξ of the DUG of Figure 3.4 is given by the following table, using $[x_0, \ldots, x_n]$ to denote a list of elements x_0, ..., x_n:

v_i	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
$\xi(v_i)$	[ ]	['c']	[ ]	['h']	'h'	['c', 'h']	['h']	['c', 'h']	['h']	'h'	*False*

3.3 Profile

Recall from the introduction of this chapter that we want to create a benchmark from a DUG, and that we want to extract a DUG from an application. However, a DUG may be very large, and hence difficult to give or inspect, so we shall now define the *profile* of a DUG. The profile will condense the most relevant characteristics of a DUG into a few numbers. We can use pseudo-random numbers to generate a family of DUGs that *on average* have a given profile. The initial seed given to the pseudo-random number generator determines which one is chosen. We can now create a benchmark from a profile, and extract a profile from an application.

We should first give some justification of using pseudo-random numbers. Why do we need a random element to our DUG generation? This is because there are many DUGs that match a single profile, and without an element of randomness we

will always pick the same one. But why cannot we just generate this one DUG? Because fixing ourselves to just one of these invites bias into our results. Such a bias may favour one ADT implementation over another, unfairly representing their performance. Picking several of these DUGs at random combats this bias.

So what characteristics do we choose to record in a profile? One obvious choice is the fraction of *persistent* applications of operations. An application of an operation is persistent if one of the version arguments has already been mutated—that is, a mutator has already been applied to this argument. However, considering the application of an operation as a whole causes problems with the generation of DUGs. Specifically, we will find that it is easier to choose the arguments independently of each other before applying the operation—see the problem *Choosing the operation before the arguments* of Section 4.1.1.

To solve this problem, we split an application into the parts represented by the *arcs*: One arc identifies one application. This allows us to identify whether an application is persistent according to whether the source of the arc has been previously mutated. With this definition of persistence we can identify which applications of operations to an argument are persistent independently of the other arguments. Note that the order associated with the targets of the arcs indicates the order of the applications.

Definition 3.12 (Mutation, Observation)

For any node v of the version graph of a DUG, a *mutation* of v is an arc from v to a mutator node. Note that an n-ary mutator creates n mutations. An observation is defined similarly. Mutations and observations inherit the ordering given to the nodes *to* which they point.

Example 3.12

Looking at the DUG in Figure 3.4, the arc from v_7 to v_8 is a mutation, and the arc from v_7 to v_9 is an observation. As v_9 is ordered after v_8, the observation $v_7 \to v_9$ is ordered after the mutation $v_7 \to v_8$.

Definition 3.13 (Persistent, Ephemeral)

For any node v of the version graph of a DUG with node ordering σ, a mutation or observation of v is *persistent* if it is ordered by σ *after the*

earliest mutation of v. This captures the notion of persistence: mutating or observing the previous value of a mutated data structure. A mutation or observation that is not persistent is called *ephemeral*.

Example 3.13

As in Example 3.12, looking at the DUG in Figure 3.4, we see that the observation $v_7 \to v_9$ occurs after the mutation $v_7 \to v_8$. As this mutation is the only mutation of v_7, it is also the earliest. Thus the observation occurs after the earliest mutation, and so is persistent. The mutation $v_1 \to v_7$ is also persistent. The observation $v_3 \to v_4$ is ephemeral.

Another obvious characteristic of DUGs is the ratio of how many times we apply one operation relative to another.

Definition 3.14 (Weight)

For any DUG $\mathcal{D}$, the *weight* of a mutator f in $\mathcal{D}$ is the number of mutations that apply f to nodes in $\mathcal{D}$. The weight of an observer is defined similarly. The weight of a generator f is simply the number of nodes that are generated by f. To unify these two definitions, one might imagine a single void node with arcs to each generator node.

Example 3.14

The weights of the operations in the DUG in Figure 3.4 are given below.

Role	Generator	Mutator			Observer		
Operation	*empty*	*catenate*	*cons*	*tail*	*head*	*lookup*	*isEmpty*
Weight	2	4	2	2	1	1	1

We can localise the weight of a mutator or of an observer to just a *subgraph*. This allows us to see how this ratio might change from one region of the DUG to another.

Definition 3.15 (Weight in $\mathcal{H}$)

For any subgraph $\mathcal{H}$ of a version graph, the *weight* of a mutator f in $\mathcal{H}$ is the number of mutations that apply f to nodes in $\mathcal{H}$. The weight of an observer is defined similarly.

Example 3.15

Looking at the DUG in Figure 3.4, let the subgraph $\mathcal{H}$ include just the nodes v_0, v_1, v_2 and v_3. The weights of the mutators and observers in $\mathcal{H}$ are given below.

Role	Mutator			Observer		
Operation	*catenate*	*cons*	*tail*	*head*	*lookup*	*isEmpty*
Weight in $\mathcal{H}$	3	2	0	1	0	0

Information such as the average number of mutations of a node is not only useful for summarising DUGs, it also provides a very convenient way to generate a DUG with a given profile (see ahead to Section 4.1.1).

From the fraction of mutations that are persistent, we can calculate the average number of mutations of previously mutated nodes as follows. Let p_m be the fraction of mutations that are persistent. Take any node v_i that is mutated at least once. The first mutation of v_i is ephemeral, and the remaining n_i mutations are persistent. Averaging over all j mutated nodes, we have

$$p_m = \frac{\sum_{i=1}^{j} n_i}{\sum_{i=1}^{j} (n_i + 1)}, \quad \overline{n} = \frac{\sum_{i=1}^{j} n_i}{j} \;\Rightarrow\; \overline{n} = \frac{p_m}{1 - p_m}$$

If we know the fraction m of nodes that are not mutated at all, we can calculate the average number $\overline{\mu}$ of mutations of a node:

$$\overline{\mu} = 0m + \left(1 + \frac{p_m}{1 - p_m}\right)(1 - m) = \frac{1 - m}{1 - p_m}$$

We call p_m the *persistent mutation factor* (PMF), and m the *mortality*.

If we calculate the ratio r of mutations to observations, we can also estimate the average number of observations of a node. Making the assumption that a node was made by a mutator, then the average number of observations of a node is $1/r$. As we have excluded nodes made by generators, this is only an estimate. From the fraction p_o of observations that are persistent, we can calculate the average number of observations made before the first mutation at $(1 - p_o)/r$, and the average number of observations made after the first mutation at p_o/r. We call p_o the *persistent observation factor* (POF).

Later we shall wish to calculate the profile of a subgraph of a DUG. As the weight of a generator cannot be localised to a subgraph, we separate out

generation weights from the weights of mutators and observers. To allow the calculation of the ratio r of mutations to observations, we group the mutation and observation weights together to form the *mutation-observation weights*.

Definition 3.16 (DUG Profile)

The profile of a DUG $\mathcal{D}$ with version graph $\mathcal{G}_V$ is given by the following:

- *Generation weights*: The ratio of the weights of each generator.
- *Mutation-observation weights*: The ratio of the weights of each mutator and observer in $\mathcal{G}_V$.
- *Mortality*: The fraction of nodes in $\mathcal{G}_V$ that are not mutated.
- PMF: The fraction of mutations of nodes in $\mathcal{G}_V$ that are persistent.
- POF: The fraction of observations of nodes in $\mathcal{G}_V$ that are persistent.

Example 3.16

The DUG shown in Figure 3.4 has the following profile:

- *Generation weights*: As there is only one generator, *empty*, this property is redundant at: $empty = 1$.
- *Mutation–observation weights*: We have

  $$catenate : cons : tail : head : lookup : isEmpty = 4 : 2 : 2 : 1 : 1 : 1$$

  Note that each application of *catenate* carries double the weight of an application of one of the other operations because each application of *catenate* creates two mutations.
- *Mortality*: Of the eight version nodes, only one (v_8) is not mutated, so the mortality is 1/8.
- PMF: There are eight mutations, one of which ($v_1 \to v_7$) is persistent, so the PMF is 1/8.
- POF: There are three observations, one of which ($v_7 \to v_9$) is persistent, so the POF is 1/3.

If the PMF and POF of a DUG are both zero, then we know that there are no persistent applications of an operation. Therefore, we make the following definition.

Definition 3.17 (Single-Threaded)

An application using an implementation of a simple ADT $\mathcal{A}$ in a manner recorded by the DUG $\mathcal{D}$ is *single-threaded* for $\mathcal{A}$ if the PMF and POF of $\mathcal{D}$ are both zero. A single-threaded application does not require a persistent implementation of the ADT.

Example 3.17

The DUG of application `app1` shown in Figure 3.2 has PMF and POF both zero and is therefore single-threaded.

3.4 Shadow Data Structure

To aid the generation of DUGs, and to add information to profiles, we use a *shadow data structure*. A shadow data structure maintains a *shadow* of every version. This shadow contains information about the version. A shadow data structure does not depend on any implementation of the ADT, but is instead abstract and applicable to *any* implementation of the same ADT.

As a running example, for the ADT $\mathcal{A}_{List}$, whose signature is given in Figure 3.3, and for which each version is a list, let the shadow of a version contain the length of the list. Below we give an overview of the uses of a shadow data structure.

Guarding Against Undefined Function Applications

When generating a DUG from a profile, if we blindly choose to label a node with any operation, we may create an application that is undefined: for example, most list ADTs would not define the value of *head empty*. Such applications of *partial operations* need to be excluded from a DUG generated at random. We need to have a *guard* around the partial operation telling us which applications of the operation we can form. We can use the shadow of a version to store enough information to allow decisions about whether a particular operation may be applied to that version. For example, for $\mathcal{A}_{List}$, if we maintain the length of a list in the shadow, we can prevent the application *head empty* by only allowing *head* to be applied to lists of length 1 or more.

Phasing Profiles

We can also use the shadow data structure to split a profile into *phases*. The shadow of a version node will determine which phase the node is in. This is useful for giving a more specific profile. For example, we might wish to make a DUG for $\mathcal{A}_{List}$ where the average length of the list is n elements. We can do this if we make *cons* more likely than *tail* on lists shorter than n elements, and vice versa for lists longer than n elements. This is possible if we maintain the length of the list in the shadow, and give a different profile for each of the two phases: lists no longer than n elements, and lists longer than n elements.

Shadow Profiling

The shadow could also store any other useful information about what operations were performed. This *shadow profile* information would allow profile information specific to an ADT to be collected, along with the general profile information already described in this chapter. For example, by maintaining the length of a list, we can calculate the average length of a list per mutation or observation.

Note that a shadow data structure is only used for the generation or analysis of DUGs, and need not be involved in applications using an ADT implementation.

We shall later use a further restriction on DUGs to aid both DUG generation and DUG extraction: Versions may only contain integer elements. Introducing this restriction here also simplifies the definition of a shadow data structure. See Section 4.1.1 for a discussion of this restriction. This restriction implies that the type variable a in the type of an operation becomes instantiated to *Int*.

We shall now define a shadow data structure precisely.

3.4.1 Shadowing

We should first define the shadows themselves. The shadows are maintained by the shadow operations.

Definition 3.18 (Shadow Operation)

For any simple ADT (T, F), and for any generator or mutator $f \in F$, let t

be the type of f with type variable a instantiated to Int. For any type s, the function g is an *s-shadow* of f if g has the type $shadow_s(t)$ given by

$$\begin{aligned} shadow_s(t_1 \to t_2) &= shadow_s(t_1) \to shadow_s(t_2) \\ shadow_s(T\ Int) &= s \\ shadow_s(Int) &= Int \end{aligned}$$

The shadows maintained by this shadow operation have type s. There are no shadows of observers as they do not return versions.

Example 3.18

For any type s, an s-shadow of the *lookup* operation of $\mathcal{A}_{List}$ (see Figure 3.3) has the following type:

$$shadow_s(T\ Int \to Int \to Int \to T\ Int) = s \to Int \to Int \to s$$

Definition 3.19 (Shadowing)

Let $\mathcal{A} = (T, \{f_1, \ldots, f_n\})$ be any simple ADT. Let $\{f_{i_1}, \ldots, f_{i_m}\}$ be the generators and mutators of $\mathcal{A}$. For any set $F' = \{f'_{i_1}, \ldots, f'_{i_m}\}$ of operations, and any type s, the pair (s, F') is a *shadowing* of $\mathcal{A}$ if the following hold:

- Each f'_{i_j} is an s-shadow of f_{i_j}.
- There exists a homomorphism $\phi :: T\ Int \to s$; that is,
  for all $f_{i_j}, x_1, \ldots, x_k$, where $k \geq 0$ is the arity of f_{i_j},
  if $f_{i_j}\ x_1\ \ldots\ x_k$ is well-defined, then the following holds:

$$\phi\ (f_{i_j}\ x_1\ \ldots\ x_k) = f'_{i_j}\ (\phi'\ x_1)\ \ldots\ (\phi'\ x_k)$$

  where for all x,

$$\phi'\ x = \begin{cases} \phi\ x, & \text{if } x \text{ has type } T\ Int \\ x, & \text{otherwise} \end{cases}$$

Example 3.19

The Haskell code of Figure 3.5 is a shadowing $\mathcal{S}_{List}$ of the ADT $\mathcal{A}_{List}$ (see Figure 3.3). In this case, the type s shadowing $List\ Int$ is of type Int, and the homomorphism $\phi :: List\ Int \to Int$ is the function that returns the length of a list.

```
type Shadow = Int

empty_Shadow :: Shadow
empty_Shadow = 0

catenate_Shadow :: Shadow -> Shadow -> Shadow
catenate_Shadow s0 s1 = s0 + s1

cons_Shadow :: Int -> Shadow -> Shadow
cons_Shadow i0 s0 = s0 + 1

tail_Shadow :: Shadow -> Shadow
tail_Shadow s0 = s0 - 1
```

Figure 3.5: A shadowing of ADT $\mathcal{A}_{List}$ (see Figure 3.3).

Definition 3.11 assigns an abstract ADT value to every version node of a DUG; the following definition assigns the shadow of the ADT value.

Definition 3.20 (Shadow Evaluation)

Let $\mathcal{D}$ be any DUG for ADT $\mathcal{A}$, and $\mathcal{S} = (s, F)$ be any shadowing of $\mathcal{A}$. The *shadow evaluation* of $\mathcal{D}$ is a mapping ζ that takes a version node v to the result of evaluating $[\![v]\!]_{\mathcal{S}}$, where an operation is interpreted by its shadow.

Example 3.20

Taking the DUG of Figure 3.4 with the shadowing $\mathcal{S}_{List}$ of Figure 3.5, the shadow evaluation ζ of the DUG is given below:

v_i	v_0	v_1	v_2	v_3	v_5	v_6	v_7	v_8
$\zeta(v_i)$	0	1	0	1	2	1	2	1

Note from Examples 3.11 and 3.20 that the evaluation of each version node under $\mathcal{S}_{List}$ equals the length of the list produced by the evaluation under $\mathcal{A}_{List}$. This results from the condition that a shadowing defines a homomorphism from the ADT values to the shadow values. This is now proved.

Lemma 3.1 *For any* DUG $\mathcal{D}$ *for* ADT $\mathcal{A}$*, any version node* v *in* $\mathcal{D}$*, and any shadowing* $\mathcal{S}$ *defining a homomorphism* ϕ*, if* $[\![v]\!]_{\mathcal{A}}$ *is well-defined, then* $\phi\ [\![v]\!]_{\mathcal{A}} = [\![v]\!]_{\mathcal{S}}$.

Proof: We shall proceed by induction on n, the number of nodes of in the version graph.

- For $n = 0$ the lemma is satisfied trivially.
- We shall assume that the lemma is true for all DUGs with no greater than n version nodes. We claim the lemma is true for any DUG with $n + 1$ version nodes. Take such a DUG $\mathcal{D}$. Take any version node v with zero out-degree within the version graph. There must be at least one such node as the graph is acyclic. As v has no successors within the version graph, we may remove v and any successors outside of the version graph from $\mathcal{D}$ to obtain another DUG $\mathcal{D}'$. As $\mathcal{D}'$ has n version nodes, the inductive hypothesis states that for any version node v' in $\mathcal{D}'$, $\phi\ [\![v']\!]_{\mathcal{A}} = [\![v']\!]_{\mathcal{S}}$. Therefore we need only prove that the lemma is true for v. Let $e_1, \ldots, e_k$, $k \geq 0$, be the arcs incident to v, ordered by τ, from the nodes $v_1, \ldots, v_k$ respectively. Let f be the operation from which $\eta(v)$ is derived, and let f' be the shadow of f given by $\mathcal{S}$.

$$\begin{aligned} [\![v]\!]_{\mathcal{S}} &= [\![\eta(v)]\!]_{\mathcal{S}}\ [\![v_1]\!]_{\mathcal{S}}\ \ldots\ [\![v_k]\!]_{\mathcal{S}} \\ &= (\lambda x_1 \cdot \ldots \cdot \lambda x_k \cdot f'\ a_1\ \ldots\ a_m)\ (\phi\ [\![v_1]\!]_{\mathcal{A}})\ \ldots\ (\phi\ [\![v_k]\!]_{\mathcal{A}}) \end{aligned}$$

  Without loss of generality, we shall assume that for $1 \leq i \leq k$, $a_i = x_i$.

$$\begin{aligned} [\![v]\!]_{\mathcal{S}} &= f'\ (\phi\ [\![v_1]\!]_{\mathcal{A}})\ \ldots\ (\phi\ [\![v_k]\!]_{\mathcal{A}})\ a_{k+1}\ \ldots\ a_m \\ &= \phi\ (f\ [\![v_1]\!]_{\mathcal{A}}\ \ldots\ [\![v_k]\!]_{\mathcal{A}}\ a_{k+1}\ \ldots\ a_m) \\ &= \phi\ ((\lambda x_1 \cdot \ldots \cdot \lambda x_k \cdot f\ a_1\ \ldots\ a_m)\ [\![v_1]\!]_{\mathcal{A}}\ \ldots\ [\![v_k]\!]_{\mathcal{A}}) \\ &= \phi\ (\eta(v)\ [\![v_1]\!]_{\mathcal{A}}\ \ldots\ [\![v_k]\!]_{\mathcal{A}}) \\ &= \phi\ [\![v]\!]_{\mathcal{A}} \end{aligned}$$

□

This lemma shows that we can have access to the shadow of a version, as defined by the homomorphism of the shadowing, by using just the shadow operations. We do not need a version to create a shadow, we need only know which operations created the version. This abstracts us away from any concrete representation of the version.

For example, the shadowing of Figure 3.5 defines a homomorphism from a version, which is a list, to its length. Lemma 3.1 shows we can calculate the shadow of a version v, namely its length, without having access to v itself. All we need to know is which operations created v. To construct the length of v, we use shadows of the same operations, with the same arguments.

3.4.2 Guarding

Using the information stored in the shadows, we wish to define a guard of an operation f that indicates which applications of f are allowed. We could make a guard take the same arguments as f and return true or false, according to whether the application is allowed or not. However, when generating an application at random, this would force every argument of an operation to be chosen *before* passing these arguments to the relevant guard. With an application such as *lookup* l i, this means guessing which indices are available for *lookup* before testing the validity of the application. This would be very inefficient.

The definition of a DUG already restricts arguments supplied by the result of another operation to just version arguments. This allows non-version arguments to be chosen independently of the results of other operations. Suppose we pass the guard *only the version arguments* of an operation. The valid ranges of remaining arguments could be returned as the result. One argument could then be chosen from each range with the resulting application guaranteed to be valid. For example, the guard for *lookup* could return a range of indices up to the length of the list.

This works only if we make the further restriction that the guard returns independent ranges of non-version arguments. Where the ranges of valid non-version arguments are dependent, the guard must return some independent subset of ranges. As we have ensured that every non-version argument is of type *Int*, a guard may return a range using the type *IntSubset*.

Definition 3.21 (IntSubset, member)

The type *IntSubset* is given by

$$\begin{aligned} data\ IntSubset &= All \\ &\mid Pool \\ &\mid Int\ ..:\ Int \\ &\mid FiniteSet\ (Set\ Int) \\ &\mid None \end{aligned}$$

and represents subsets of integers in the sense made precise by the following definition of the membership operation:

$$\begin{aligned} member &:: Int \to IntSubset \to Bool \\ member\ i\ All &= True \\ member\ i\ Pool &= (1 \le i \le poolSize) \\ member\ i\ (l\ ..:\ u) &= (l \le i \le u) \\ member\ i\ (FiniteSet\ s) &= member_{FS}\ i\ s \\ member\ i\ None &= False \end{aligned}$$

where $member_{FS}$ is the membership operation on the type *Set Int*, and *poolSize* is some constant. We assume the availability of a suitable ADT to manipulate values of type *Set Int*.

The definition of *IntSubset* allows the same set to be given in more than one way; in fact, only the *FiniteSet* constructor is needed. However, the other constructors provide dynamic, more efficient, or shorter alternatives:

- The set of all possible integers is more efficiently given as *All* than as *FiniteSet* (*foldr add empty* [*minBound..maxBound*]).
- The constant *poolSize* can be given at run-time of the generation of a DUG. The constructor *Pool* therefore gives a set of dynamic size. This is useful in assessing the effect of equal elements on efficiency of ADT implementations—see the problem *Choosing non-version arguments from the graph* of Section 4.1.1 for further details.
- The set $\{1, \ldots, n\}$ is more easily and more efficiently given as $1 : .. : n$ than as *FiniteSet* (*foldr add empty* [*1..n*]).

- The *None* is included to complement *All* and as a shorter alternative to *FiniteSet empty*.

Using *IntSubset*, we can now give the type of a guard.

Definition 3.22 (Guard Type)

Let T be any type constructor of arity one. Let t be any simple type over T with type variable a instantiated to *Int*. Let n be the number of arguments of an operation of type t, v of which are version arguments. For any type s, the type $guard_s(t)$ is given by

$$guard_s(t) = \overbrace{s \to \cdots \to s}^{v \text{ times}} \to \begin{cases} [IntSubset]_{n-v} & \text{if } v < n \\ Bool & \text{if } v = n \end{cases}$$

where $[a]_n$ is the type of lists of n elements of type a, and where s represents the type of shadows. This replaces every version argument with a shadow, and moves every non-version argument over to the result type. There are $n - v$ non-version arguments; if $n - v = 0$, then the result type is *Bool*, otherwise it is a list of length $n - v$ of elements of type *IntSubset*.

Example 3.22

Consider the ADT $\mathcal{A}_{List}$, whose signature is in Figure 3.3. For any type s, any guard of the operation *head* using shadows of type s must be of type

$$guard_s(T\ Int \to Int) = s \to Bool$$

If we add the operation *update* of type

$$update :: List\ a \to Int \to a \to List\ a$$

to $\mathcal{A}_{List}$, then any guard of *update* must be of type

$$guard_s(T\ Int \to Int \to Int \to T\ Int) = s \to [IntSubset]_2$$

As the type $[a]_n$ cannot be written in Haskell, one might ask why we have chosen it over an i-tuple. Unfortunately, Haskell does not support functions over tuples of arbitrary size. We must work with the result of any guard in general, and thus we are forced to use lists. However, the type of lists does not express their

length. Hence lists with types that do specify length were chosen as a compromise between expressibility and practicality.

We can now define a guard itself. We ensure the guard is of the correct type, and that it correctly indicates when a function application is well-defined.

Definition 3.23 (Guard)

Let $\mathcal{S} = (s, F')$ be a shadowing of the ADT $\mathcal{A} = (T, F)$ defining a homomorphism $\phi :: T\ Int \to s$. For any operation $f \in F$ of type t, the function g is an $\mathcal{S}$-*guard* of f if the following hold:

- The type of g is $guard_s(t)$.
- For all $x_1, \ldots, x_n$, where $x_{i_1}, \ldots, x_{i_k}$ are each of type $T\ Int$ and $x_{j_1}, \ldots, x_{j_l}$ are the rest, we have:
  - If $l = 0$, $f\ x_1\ \ldots\ x_n$ is well-defined if

$$g\ (\phi\ x_{i_1})\ \ldots\ (\phi\ x_{i_k}) = True$$

  - If $l \geq 1$, $f\ x_1\ \ldots\ x_n$ is well-defined if

$$g\ (\phi\ x_{i_1})\ \ldots\ (\phi\ x_{i_k}) = [s_1, \ldots, s_l]$$

    and for all $1 \leq t \leq l$,

$$member\ x_{j_t}\ s_t = True$$

Example 3.23

The Haskell code of Figure 3.6 defines $\mathcal{S}_{List}$-guards of every operation of $\mathcal{A}_{List}$. Recall that the homomorphism given by $\mathcal{S}_{List}$ is the length function. The guards of *empty*, *catenate*, and *isEmpty*, are trivial as these operations are total. The guard of *cons* allows any element to be added to the front of a list. The guard of *tail* will return *True* when and only when the list whose shadow it is being applied to is non-empty, ie. exactly when it is safe to apply *tail* to the list. The guard of *head* is identical. The guard of *lookup* will return the range of indices over which *lookup* is well-defined: any index from first to last element inclusive.

```
empty_Guard :: Bool
empty_Guard = True

catenate_Guard :: Shadow -> Shadow -> Bool
catenate_Guard s0 s1 = True

cons_Guard :: Shadow -> [IntSubset]
cons_Guard s0 = [All]

tail_Guard :: Shadow -> Bool
tail_Guard s0 = s0>0

head_Guard :: Shadow -> Bool
head_Guard s0 = s0>0

lookup_Guard :: Shadow -> [IntSubset]
lookup_Guard s0 = [0:..:(s0-1)]

isEmpty_Guard :: Shadow -> Bool
isEmpty_Guard s0 = True
```

Figure 3.6: Haskell code for $\mathcal{S}_{List}$-guards of the operations of $\mathcal{A}_{List}$ (see Figure 3.5). Note that as Haskell does not have a type for lists of a given length, and as tuples are awkward to manipulate in the general case, lists of arbitrary length are used.

We shall generate a DUG by adding one node at a time. We shall choose an operation and predecessors for a new node, and then decide using the guards which integer arguments, if any, will produce a new DUG with a valid evaluation. We now wish to prove that the guards do allow us to make this decision.

Lemma 3.2 *Suppose we have a* DUG $\mathcal{D} = (\mathcal{G}, \eta, \sigma, \tau)$ *for* ADT $\mathcal{A}$*, a shadowing* $\mathcal{S}$ *of* $\mathcal{A}$*, and an* $\mathcal{S}$*-guard for every operation of* $\mathcal{A}$*, with every node of* $\mathcal{D}$ *having a well-defined evaluation under* $\mathcal{A}$*. Now propose an extension of* $\mathcal{D}$ *by one node* v *using operation* f *and predecessors* $v_1, \ldots, v_k$*. The guards can use* just *the information provided by the shadow evaluation of* $\mathcal{D}$ *to give sets of integer arguments. Choosing any integer from each will provide a well-defined evaluation of* v *under* $\mathcal{A}$*.*

Proof: Let g be the $\mathcal{S}$-guard of f. If valid, the evaluation of v under $\mathcal{A}$ is given by the result of evaluating the following:

$$\begin{aligned} [\![v]\!]_{\mathcal{A}} &= \eta(v)\ [\![v_1]\!]_{\mathcal{A}}\ \ldots\ [\![v_k]\!]_{\mathcal{A}} \\ &= f\ x_1\ \ldots\ x_n \end{aligned}$$

where $x_{i_m} = [\![v_m]\!]_{\mathcal{A}}$, $\{x_{i_m}\}_{m=1}^{k} \cup \{x_{j_m}\}_{m=1}^{l} = \{x_m\}_{m=1}^{n}$ and $k+l = n$. As each v_m is in $\mathcal{D}$, $[\![v_m]\!]_{\mathcal{A}}$ is well-defined. By Lemma 3.1, $\phi\ x_{i_m} = \phi\ [\![v_m]\!]_{\mathcal{A}} = [\![v_m]\!]_{\mathcal{S}}$. Therefore, given the shadow evaluation of $\mathcal{D}$, we can determine the value of each $\phi\ x_{i_m}$, and hence the integer sets given by $g\ (\phi\ x_{i_1})\ \ldots\ (\phi\ x_{i_k})$. From Definition 3.23, choosing any $x_{j_1}, \ldots, x_{j_l}$ from these sets gives a well-defined evaluation of v under $\mathcal{A}$. □

Note that, in general, it may not be possible to define a guard that gives *every* well-defined application—for example, where the integer arguments cannot be independently chosen. However, for all of the ADTs in this thesis, it is possible to define guards which *do* give every well-defined application.

3.4.3 Phasing

It is useful to be able to identify different *phases* of an application. The profile of each phase may be given separately. For example, an application could have a growth phase where the data structures are being built, and a decay phase

where the data structures are being examined and taken apart. The profile of the growth phase would show more applications of building operations than of deconstructing operations, and vice versa for the decay phase.

Thus a profile split into phases reveals more about an application's use of the ADT than just the whole profile. Additionally, when generating a DUG according to profiles for each phase, there is more control over the generation process.

We assign each ADT version to a phase. Note, however, that at any one point in the computation, there may be many versions in different phases. For example, using the growth and decay phase example above, there may be some versions being built in the growth phase, whilst some are being taken apart in the decay phase.

Information stored in the shadow determines which phase a version is in. The phases *partition* the version graph; that is, each version node will belong to a single phase. The non-version nodes are not shadowed and will not belong to any phase. We will identify a phase by a value of the type *PhaseId* which we will define as a type synonym with *Int*. The first phase is phase 1. Letting s be the type of a shadow, we may suppose that the following simple function would suffice:

$$phaser \;::\; s \to PhaseId$$

In general however, the function *phaser* needs more information than this.

Suppose we are generating DUGs over the list ADT $\mathcal{A}_{List}$. Suppose further that we want to split the lists into two phases: those below a given length, and those above. We wish to parameterise the phasing over this length. This is the *phase argument*. A function *phaseArgRead* is required to read in the argument from a string. A value *phaseArgDefault* is required to specify the phase argument to use if none is given.

Definition 3.24 (Phasing)

Let $\mathcal{S} = (s, F')$ be a shadowing of some ADT. The 4-tuple

$$\mathcal{P} = (r, phaseArgRead, phaseArgDefault, phaser)$$

```
data PhaseArg = MeanSize Int | NoMeanSize

phaseArgRead :: String -> PhaseArg
phaseArgRead s = MeanSize (read s)

phaseArgDefault :: PhaseArg
phaseArgDefault = NoMeanSize

phaser :: Shadow -> PhaseArg -> PhaseId
phaser _ NoMeanSize = 1
phaser s (MeanSize m)
   | s <= m    = 1
   | otherwise = 2
```

Figure 3.7: Functions implementing an $\mathcal{S}_{List}$-phasing assigning lists no longer than the phase argument to phase 1, and those longer to phase 2. See Example 3.19 for the definition of $\mathcal{S}_{List}$. If no phase argument is given, all nodes are placed in phase 1.

---

provides an $\mathcal{S}$-*phasing* when the following type signatures are correct:

$$\begin{aligned} \mathit{phaseArgRead} &:: \mathit{String} \to r \\ \mathit{phaseArgDefault} &:: r \\ \mathit{phaser} &:: s \to r \to \mathit{PhaseId} \end{aligned}$$

Note that the type *PhaseId* is a type synonym for *Int*.

Example 3.24

The Haskell code of Figure 3.7 defines an $\mathcal{S}_{List}$-phasing. This phasing places lists of length less than or equal to the phase argument (an integer) into phase 1, and the rest into phase 2.

Each part of the DUG profile defined in Section 3.3 can be parameterised over the phase of a version node, except for generation weights.

Definition 3.25 (Phased DUG Profile)

Let $\mathcal{D}$ be a DUG for ADT $\mathcal{A}$ with version graph $\mathcal{G}_V$. Let $\mathcal{S}$ be a shadowing of $\mathcal{A}$. Let $\mathcal{H}_1, \ldots, \mathcal{H}_p$ be the subgraphs of $\mathcal{G}_V$ partitioned by the $\mathcal{S}$-phasing $\mathcal{P}$. The *$\mathcal{P}$-phased* DUG *profile* of $\mathcal{D}$ can be calculated by replacing $\mathcal{G}_V$ with $\mathcal{H}_i$ in Definition 3.16 for every property bar generation weights. The phased profile of a DUG is therefore:

- A generation weights ratio
- A set of the following properties, one for each phase of the DUG: mutation-observation weights, mortality, PMF, and POF.

Example 3.25

Using the $\mathcal{S}_{List}$-phasing $\mathcal{P}_{List}$ of Example 3.24 with a phase argument of 1, partition the DUG shown in Figure 3.4 into two phases: (1) lists of length zero or one, and (2) lists of length two or more. Example 3.20 gives the length of each list. Let $\mathcal{H}_1$ contain nodes in phase (1), namely v_0, v_1, v_2, v_3, v_6, and v_8. Let $\mathcal{H}_2$ contain nodes in phase (2), namely v_5 and v_7. The $\mathcal{P}_{List}$-phased profile of this DUG is given below:

- Generation weights—as there is only one generator, *empty*, this property is redundant at: $empty = 1$.
- Set of profiles of each phase.
  - For $\mathcal{H}_1$, the lists of length zero or one, we have the following profile:
    - Mutation-observation weights:

      $$catenate : cons : tail : head : lookup : isEmpty = 4 : 2 : 0 : 1 : 0 : 1$$

    - Mortality—of the six version nodes in $\mathcal{H}_1$, only one (v_8) is not mutated, so the mortality is 1/6.
    - PMF—there are six mutations of nodes in $\mathcal{H}_1$, one of which ($v_1 \rightarrow v_7$) is persistent, giving a PMF of 1/6.
    - POF—there are two observations of nodes in $\mathcal{H}_1$, neither of which is persistent, giving a POF of 0.

- For $\mathcal{H}_2$, the lists of length two or more, we have the following profile:
  - Mutation-observation weights:

    $$catenate : cons : tail : head : lookup : isEmpty =$$
    $$0 : 0 : 2 : 0 : 1 : 0$$

  - Mortality—all version nodes are mutated, so the mortality is 0.
  - PMF—there are three mutations of nodes in $\mathcal{H}_2$, none of which are persistent, giving a PMF of 0.
  - POF—there is one observation of a node in $\mathcal{H}_2$ ($v_7 \rightarrow v_9$), which is persistent, giving a POF of 1.

3.4.4 Shadow Profiling

Shadow profiling allows information to be collected about every operation applied to a version, namely mutations and observations. The shadow of any version that is mutated or observed is the source of this information. For example, if the shadow of a list contained its length, we could sum the lengths of lists involved in mutations and observations and return the average. Note that this is not the same as summing the lengths of every mutated or observed list: if a list is mutated or observed more than once, its shadow is used more than once.

We will need to maintain a *shadow profile*. The initial value will be given by *shadowProfileZero*. Information will be collected using the function *shadowProfiler*. The final value will be shown using the function *shadowProfileShow*.

Definition 3.26 (Shadow Profiling)

Let $\mathcal{S} = (s, F')$ be a shadowing of some ADT. The 4-tuple

$$(p, shadowProfileZero, shadowProfiler, shadowProfileShow)$$

provides an $\mathcal{S}$-*profiling* when the following type signatures are correct:

$$\begin{aligned} shadowProfileZero &:: p \\ shadowProfiler &:: p \rightarrow s \rightarrow p \\ shadowProfileShow &:: p \rightarrow String \end{aligned}$$

```
data ShadowProfile = ShadowProfile Int Int

shadowProfileZero :: ShadowProfile
shadowProfileZero = ShadowProfile 0 0

shadowProfiler :: ShadowProfile -> Shadow -> ShadowProfile
shadowProfiler (ShadowProfile sum count) s =
   ShadowProfile (sum+s) (count+1)

shadowProfileShow :: ShadowProfile -> String
shadowProfileShow (ShadowProfile sum count) =
   "Average size = " ++ show (sum/count)
```

Figure 3.8: Functions implementing an $\mathcal{S}_{List}$-profiling. The average length over every mutation and observation of a list is calculated. See Figure 3.5 for the definition of $\mathcal{S}_{List}$.

Example 3.26

The Haskell code of Figure 3.8 defines an $\mathcal{S}_{List}$-profiling. This shadow profiling calculates the average length of a list over all mutations and observations. For the DUG of Figure 3.4, this reports an average length of 12/11. To verify this, here is a table of every mutation and observation, and the corresponding length of the mutated or observed list:

Mutation/Observation	$v_0 \to v_1$	$v_2 \to v_3$	$v_3 \to v_4$	$v_1 \to v_5$	$v_3 \to v_5$
Version Operated On	v_0	v_2	v_3	v_1	v_3
Length	0	0	1	1	1

$v_5 \to v_6$	$v_1 \to v_7$	$v_6 \to v_7$	$v_7 \to v_8$	$v_7 \to v_9$	$v_8 \to v_{10}$
v_5	v_1	v_6	v_7	v_7	v_8
2	1	1	2	2	1

Given a list l of shadows of versions that are mutated or observed, one might view the shadow profile with:

shadowProfileShow (*foldl shadowProfiler shadowProfileZero l*)

3.4.5 Definition

We are now in a position to give a formal definition of a shadow data structure, which includes shadowing, guarding, phasing, and shadow profiling.

Definition 3.27 (Shadow Data Structure)

For any simple ADT $\mathcal{A} = (T, F)$, any shadowing $\mathcal{S}$ of $\mathcal{A}$, any set $\mathcal{W}$ containing a single $\mathcal{S}$-guard of every operation in F, any $\mathcal{S}$-phasing $\mathcal{P}$, and any $\mathcal{S}$-profiling $\mathcal{O}$, the 4-tuple $(\mathcal{S}, \mathcal{W}, \mathcal{P}, \mathcal{O})$ is a *shadow data structure* for $\mathcal{A}$.

Example 3.27

Take $\mathcal{S}_{List}$ from Figure 3.5, $\mathcal{W}_{List}$ containing the $\mathcal{S}_{List}$-guards of Figure 3.6, $\mathcal{P}_{List}$ from Figure 3.7, and $\mathcal{O}_{List}$ from Figure 3.8. The 4-tuple

$$(\mathcal{S}_{List}, \mathcal{W}_{List}, \mathcal{P}_{List}, \mathcal{O}_{List})$$

is a shadow data structure for the ADT $\mathcal{A}_{List}$, whose signature is given in Figure 3.3.

3.5 Summary

We now have a formal model capturing how an application uses a data structure: a DUG. We also have a summary of the most important aspects of this use: a profile. By generating a DUG from a profile, and by defining the evaluation of a DUG, we can now create a benchmark from a profile. The shadow data structure plays an important role in the generation of DUGs by allowing us to avoid undefined applications of operations. By extracting a DUG from an application and by calculating its profile, we can also create a profile from an application. The shadow data structure helps here by adding useful information to the extracted profile.

However, all of this is defined only in theory. In the following chapter we shall turn this theory into practice by giving algorithms for the generation, evaluation, extraction, and profiling of DUGs.

Chapter 4

Implementing Datatype Usage Graphs

As stated at the start of Chapter 3, we want to be able (a) to create a benchmark from a description of use, and (b) to create a description of use from an application. In Chapter 3, we defined a DUG, describing how a data structure is used by an application. We then outlined in theory how we can (a) create a benchmark from a profile of a DUG, and (b) create a profile of a DUG from an application. In this chapter, Sections 4.1 and 4.2 show how these ideas can be implemented. Section 4.3 describes the technical details involved in these concrete implementations. Section 4.4 evaluates the accuracy and efficiency of these implementations.

4.1 From Profile to Benchmark

Recall from Section 3.3 that we create a benchmark from a profile as follows:

(1) Use a pseudo-random number generator to create a DUG that probabilistically has the given profile—that is, the *expected* profile is the one given.

(2) Use a DUG evaluator to evaluate this DUG using a given implementation of the ADT.

Section 4.1.1 describes (1), and Section 4.1.2 describes (2).

To generate a DUG:

while the DUG is too small **do**

choose an operation

choose version arguments for the operation

choose non-version arguments for the operation

add a node to the DUG

add arcs from the nodes used as arguments to the new node

label the node with the operation and the remaining arguments

Figure 4.1: Overview of the DUG generation algorithm.

4.1.1 DUG Generation

How shall we build a DUG? Figure 4.1 gives a reasonable starting point for an algorithm.

Problems with DUG Generation

Unfortunately, the simple algorithm of Figure 4.1 encounters some problems. These are listed below, together with the solutions we choose.

- *Creating undefined applications.* Some applications of operations may not be well defined. For example, the application *head empty* is usually not defined. We need to avoid these applications. We do this by maintaining extra information—a *shadow*—about each possible argument of an application. A *guard* protects us from creating an undefined application, by using the shadow of every argument. Shadows and guards make up part of a *shadow data structure*—see Section 3.4.

- *Allowing undefined arguments.* Lazy evaluation evaluates the operation before the arguments. Therefore, adding a node with (as yet) undefined arguments seems reasonable. However, without knowing the arguments, we cannot avoid undefined applications using a shadow data structure. Therefore we never add a node without knowing all the arguments.

- *Choosing the arguments from the whole graph.* We could pick the arguments from any part of the DUG already formed. However, in practice, because we must maintain a shadow of every possible argument, this can cost too much memory. Therefore we restrict choice of arguments to a subgraph, the *frontier*. We need only maintain shadows of nodes in the frontier. If the frontier becomes too large, we remove a node (though it stays in the DUG).

- *Choosing non-version arguments from the graph.* We could choose non-version arguments from the results of observers. However, this proves too restrictive—for example, whilst generating a DUG for the ADT of Figure 4.2, from where does the argument of type a for the first application of *cons* come? There can be no applications of *head* in the graph yet. But how else can we generate an argument of type a? As the role played by non-version arguments is a relatively minor one (for example, no profile properties depend on them), we restrict them to being integers—that is, we instantiate the type variable a to *Int*. For simplicity, now that every non-version argument has type *Int*, we then choose all non-version arguments independently of the graph.

  But what effect do arguments of type a have on the efficiency of ADT implementations? For those ADTs that do not examine the elements they carry (that is, arguments of type a), the only affect these elements can have is through their size—the larger the element and the more elements held onto by the ADT implementation, the larger the heap size which in turn affects efficiency. By restricting ourselves to elements of type *Int*, we have no means of measuring this effect.

  For those ADTs that *do* examine the elements they carry, for example, by comparing them under equality, or by ordering them, the values of these elements *can* affect efficiency. For the data structures considered in this thesis, only equality and ordering is used on elements. Under this use of elements, one of the main effects on efficiency is through the number of equal elements. This is controlled through the *pool size*.

The range of integer arguments given by the *Pool* data constructor of *IntSubset* (see Definition 3.21) are drawn from $\{1, \ldots, p\}$, where p is called the pool size. The smaller the pool size, the more equal elements will be inserted. Changing the pool size may effect the efficiency of ADT implementations. For example, one implementation of a set ADT may be more efficient than another at handling many insertions of equal elements.

Apart from this rather crude means of controlling the range of elements, we currently have no other control of the effect of elements on efficiency.

- *Choosing the operation before the arguments.* We could choose an operation for the new node before choosing its arguments. However, this proves rather awkward for generating a DUG to fit some of the profile properties. For example, persistence and mortality depend on whether the arguments have been previously mutated or not. Before choosing arguments, we would need to know which have been mutated and which have not, if we are to attempt to match these properties. Additionally, phasing the profiles increases our dependence on prior knowledge of the arguments. It is easier if we choose an argument first, and the operation second. Therefore, for each new node, we plan which operations each node should be involved in as an argument, and in which order. See Section 3.3 for a discussion of how the profile properties are used to plan a node's future.

  We choose an argument first, and let the first operation in its future determine the operation of the new node. However, we must cater for operations that take more than one node as an argument. Therefore, we place arguments in a buffer according to operation, and wait till it contains as many nodes as the operation takes arguments, before creating a new node with this operation. Unfortunately, this has the drawback that it is impossible to create an application where the same node appears as more than one argument, for example, *catenate v v*. However, there does not appear to be a simple solution to this problem.

- *Diverging.* If we allow the same operation and arguments to be chosen repeatedly, and if this application is rejected by the guard, we could diverge.

```
module List (List,empty,catenate,cons,tail,head,lookup,isEmpty)
   where
empty    :: List a
catenate :: List a -> List a -> List a
cons     :: a -> List a -> List a
tail     :: List a -> List a
head     :: List a -> a
lookup   :: List a -> Int -> a
isEmpty  :: List a -> Bool
```

Figure 4.2: Haskell code giving the signature of a simple list ADT providing normal list operations, catenation and indexing.

Therefore, once a guard rejects an application, we remove this operation from the node's future.

The DUG Generation Algorithm

We build a DUG one node at a time. Each node has a *future* and a *past*. The future records which operations we have planned to apply to the node, in order. The past records which operations we have already applied to the node. The nodes with a non-empty future together make up the *frontier*. The first operation in a future is called the *head operation*.

As we add a node to the DUG, we take arguments from the frontier. The frontier therefore is the subgraph on which we are building. We shall bound the size of the frontier above and below:

- Bounding above prevents the frontier from getting too large. If the PMF is non-zero, we shall need to mutate nodes more than once. This leads to exponential growth of the frontier, which may need to be capped to prevent running out of memory. When the frontier exceeds a given limit, we remove an arbitrary node from the frontier. This will affect the final profile, and

so should only be used when there is no alternative.

- Bounding below ensures there is at least one node to build on, and encourages diversity, especially in the presence of operations with large version arities.

When a new node is made, we record this as a *birth*. A list of births, in order, describes a DUG completely. When a node no longer has a future, we record its past as a *death*. A list of deaths also describe a DUG completely[1]. A list of births describes a DUG from a global perspective (how was a node added to the graph) whereas a list of deaths describes a DUG from a local perspective (what was applied to a node). A list of births is more convenient for evaluating a DUG. A list of deaths is more convenient for profiling a DUG. Hence, we shall produce both as we generate the DUG.

Our definition of a DUG restricts non-version nodes from being re-used, and so each non-version node always has an empty future and an empty past. To save time and space, we do not record the death of a non-version node—the node is assumed to die immediately after birth.

An overview of the algorithm is given in Figures 4.3 and 4.4. Fuller details of the algorithms are given below.

Generating the DUG. The main function generateDug takes an integer and returns a DUG with this many nodes, in the form of a list of births and deaths.

```
generateDug :: Int → [BirthOrDeath]
```

A birth records the identity of the node born, the operation used, the version arguments (identities of other nodes), and the non-version arguments (integers). A death records the identity of the dead node, the arcs from the dead node, and the shadow of the dead node.

```
data BirthOrDeath = Birth NodeId Operation [NodeId] [Int]
                  | Death NodeId [Arc] Shadow
```

[1] This is only true if we consider a generator as taking an imaginary void node as an argument (see Definition 3.14) and include the death of this node. However, in practice, it is easier to just use the birth of the generator, which is what we do for DUG profiling.

To generate a DUG:

while the DUG is too small **do**
  if the frontier is too small **then**
    try to make a new node using a generator (see part II)
  else-if the frontier is too large **then**
    remove a node from the frontier
    record the death of this node
  else
    remove a node from the frontier to act as a version argument
    place the node in the buffer corresponding to the node's head operation
    if this buffer is full **then**
      try to make a new node with the buffer's contents acting as the version
        arguments for their common head operation (see part II)
    fi
  fi
od
record the death of every node in the frontier and buffers

Figure 4.3: Overview of the DUG generation algorithm (part I).

To try to make a new node from an operation and some version arguments:

apply the guard of the operation to the shadow of every version argument
if the guard fails **then**
remove the head operation of each version argument
else
choose some non-version arguments from the result of the guard
make a new node by applying the operation to the arguments
record the birth of this node
add the new node to the DUG
if the operation is not an observer **then**
plan the future of the new node
else
leave the future of the new node empty
fi
if the new node has a non-empty future **then**
add the new node to the frontier
else
record the death of this node
fi
remove the head operation of each version argument
fi
record the death of every version argument with an empty future
add every other version argument to the frontier

Figure 4.4: Overview of the DUG generation algorithm (part II).

The shadow of a node is needed for DUG profiling but not for DUG evaluating, and so we only include it in a death. Recall that an arc from a node u to a node v represents the application of an operation at v to the result of the operation at u. The type Arc records the operation at v, the argument position of u, the non-version arguments, and the identity of v.

```
data Arc = Arc {targetNodeOp :: Operation, sourceNodeArgPosn :: Int,
                intArgs :: [Int], targetNodeId :: NodeId}
```

The function generateDug is defined using an auxiliary function—a function that performs the same task but maintains auxiliary arguments—called generateNodes, taking the following auxiliary arguments: the current frontier, the current buffers, the identity of the next node to be created, and the number of nodes left to create.

```
generateNodes :: {Node} → Buffers → NodeId → Int → [BirthOrDeath]
```

A node is identified by a value of type NodeId. The node also stores: the node's future, the node's past, and the node's shadow.

```
data Node = Node {nodeId :: NodeId, future :: [Operation],
                  past :: [Arc], shadow :: Shadow}
```

Each buffer holds the arguments waiting to be involved in the application of a particular operation. Therefore the type Buffers is a function taking an operation f to the buffer for f. A buffer is a list of arguments, in the order they were added.

```
type Buffers = Operation → [Node]
```

Initially, the frontier is empty, the buffers are empty, the next node is the first node, and every node still has to be made.

```
generateDug noOfNodes = generateNodes {} (λf · []) initialNodeId noOfNodes
```

Generating a node. At the core of the algorithm lies a loop. Each iteration of the loop is a call to generateNodes, and each call attempts to add a new node to the current DUG. If we have no more nodes to make, we record the deaths of the nodes left in the frontier and buffers. All other nodes had their deaths recorded as they left the frontier without entering into a buffer.

```
generateNodes frontier buffers newNodeId 0 =
    [Death (nodeId node) (past node) (shadow node) |
     node ← frontier ∪ range buffers]
```

If the frontier is too small, we attempt to make a new node using a generator chosen according to the generation weights of the profile.

```
generateNodes frontier buffers newNodeId nodesLeft
    | size frontier < frontierMin =
        tryApplication (chooseOperation† generationWeights) []
                       frontier buffers newNodeId nodesLeft
```

The function tryApplication attempts to make a node from an operation and a list of version arguments. It also carries through the arguments given to generateNodes.

```
tryApplication :: Operation → [Node] → {Node} → Buffers → NodeId → Int →
                  [BirthOrDeath]
```

The function chooseOperation takes some operation weights and returns an operation pseudo-randomly, biased according to the weights. This requires a random seed, but we omit that argument here, for the threading of seeds clutters the code. Therefore, for the purposes of this presentation of code, consider the function as using hidden state and hence being impure. All such functions are indicated by a † superscript. For details on the implementation of these functions, see Section 4.3.1.

```
chooseOperation† :: {(Operation,Weight)} → Operation
```

If the frontier is too large, a node is removed from the frontier. The death of this node is recorded, and we repeat the main loop with a call to generateNodes.

```
generateNodes frontier buffers newNodeId nodesLeft
    | size frontier > frontierMax =
        let (node, frontier') = removeNode† frontier
        in Death (nodeId node) (past node) (shadow node) :
           generateNodes frontier' buffers newNodeId nodesLeft
```

The function removeNode† removes a node at random from the frontier, and returns this node and the new frontier.

```
removeNode† :: Node → (Node,{Node})
```

Otherwise, we choose a node v from the frontier as an argument for a new node.

```
generateNodes frontier buffers newNodeId nodesLeft
  | otherwise =
    let (v, frontier') = removeNode† frontier
    in useArgument v frontier' buffers newNodeId nodesLeft
```

For each operation f, we keep a buffer of version nodes whose head operation is f. We add v to the appropriate buffer.

```
useArgument :: Node → {Node} → Buffers → NodeId → Int → [BirthOrDeath]
useArgument v frontier buffers newNodeId nodesLeft =
  let (f : rest) = future v
      buffers' g | g == f    = v : buffers g
                 | otherwise = buffers g
  in checkBuffer operation frontier buffers' newNodeId nodesLeft
```

If the buffer of f contains the same number of nodes as the version arity of f, we remove these nodes vs. We then try to make a new node from operation f and version arguments vs.

```
checkBuffer :: Operation → {Node} → Buffers → NodeId → Int →
               [BirthOrDeath]
checkBuffer f frontier buffers newNodeId nodesLeft
  | length (buffers f) == numberOfVersionArguments f =
    let vs = buffers f
        buffers' g | g == f    = []
                   | otherwise = buffers g
    in tryApplication f vs frontier buffers' newNodeId nodesLeft
  | otherwise = generateNodes frontier buffers newNodeId nodesLeft
```

Trying an application. Recall that the function tryApplication attempts to make a new node using an operation f and version arguments vs. We apply the guard of f to the shadow of every node in vs to find the ranges of possible non-version arguments. If these ranges are empty, we abandon this application using the function cleanUpFailure. Otherwise, we make a new node using makeNewNode.

```
tryApplication f vs frontier buffers newNodeId nodesLeft =
  case applyGuard f (map shadow vs) of
    Failure → cleanUpFailure vs frontier buffers newNodeId nodesLeft
    IntSubsets iss → makeNewNode f vs iss frontier buffers newNodeId nodesLeft
```

The function applyGuard applies the guard of an operation to a list of shadows, and returns the ranges of possible non-version arguments.

```
applyGuard :: Operation → [Shadow] → NonVersionArgs
```

If any of the ranges is empty, applyGuard returns Failure.

```
data NonVersionArgs = IntSubsets [IntSubset] | Failure
```

The type IntSubset is defined in Section 3.21.

Cleaning up after a failed application. If an application of the guard of an operation f to the shadows of the nodes vs fails, we change the nodes vs to reflect this using the function chronicleFail. We record the death of any node without a future, return the rest to the frontier, and repeat the main loop by calling generateNodes.

```
cleanUpFailure :: [Node] → {Node} → Buffers → NodeId → Int → [BirthOrDeath]
cleanUpFailure vs frontier buffers newNodeId nodesLeft =
  let (deadNodes,liveNodes) = splitWith (null ∘ future) (map chronicleFail vs)
      obituary = [Death (nodeId node) (past node) (shadow node) |
                  node ← deadNodes]
  in obituary ++
     generateNodes (frontier ∪ liveNodes) buffer newNodeId nodesLeft
```

The function chronicleFail removes the head operation of each node.

```
chronicleFail :: Node → Node
chronicleFail node = node {future = tail (future node)}
```

Making a new node from a successful application. If an application of the guard of an operation f to the shadows of the nodes vs succeeds with ranges of possible non-version arguments iss, we choose non-version arguments is, one from each set in iss using chooseInt. We change vs to reflect this successful application using chronicleSuccess, record the death of any node without a future, and return the rest to the frontier, as in cleanUpFailure. The birth of the new node is recorded. If the operation is not an observer, the new node is given a future using the operation bear, and placed in the frontier (if its future is not empty). Otherwise, it dies at birth (but we do not explicitly record the death). We repeat the loop, obtaining a new node identity, and decreasing the number of nodes left to generate by 1.

```
makeNewNode :: Operation → [Node] → [IntSubset] → {Node} → Buffers →
                    NodeId → Int → [BirthOrDeath]
makeNewNode f vs iss frontier buffers newNodeId nodesLeft =
  let is = map (chooseInt† poolSize) iss
      newNode = if role f == Observer then [] else [bear f vs is]
      vs' = zipWith (chronicleSuccess f is newNodeId) vs (versionArgs f)
      (deadNodes,liveNodes) = splitWith (null ∘ future) (vs' ++ newNode)
      obituary = [Death (nodeId node) (past node) (shadow node) |
                    node ← deadNodes]
  in Birth newNodeId f (map nodeId vs) is : obituary ++
     generateNodes (frontier ∪ liveNodes) buffer
                   (nextNodeId newNodeId) (nodesLeft–1)
```

The function chooseInt chooses an integer from an IntSubset using the given pool size.

```
chooseInt† :: Int → IntSubset → Int
```

To reflect the successful application of an operation f to a node v at argument position pos with non-version arguments is to create a new node with identity newNodeId, we remove the head operation, and record the application as an Arc in the node's past.

```
chronicleSuccess :: Operation → [Int] → NodeId → Node → Int → Node
```

```
chronicleSuccess f is newNodeId v pos =
    v {future = tail (future v), past = Arc f pos is newNodeId : past v}
```

The new node is given an identity tag, a future calculated by the function plan using information contained in the shadow of the new node, an empty past, and a shadow.

```
bear :: Operation → [Node] → [Int] → Node
bear f vs is newNodeId = let newShadow = applyShadow f (map shadow vs) is
                         in Node newNodeId (plan newShadow) [] newShadow
```

The function applyShadow applies the shadow of an operation to the shadows of the version arguments and to the non-version arguments.

```
applyShadow :: Operation → [Shadow] → [Int] → Shadow
```

The function versionArgs returns the positions of the version arguments of a given operation.

```
versionArgs :: Operation → [Int]
```

Planning the future of a new node. The function plan decides the future of a new node v using information contained in the shadow of v.

```
plan :: Shadow → [Operation]
```

The phase of v is given by the shadow of v and the phase arguments.

```
phase = phaser shadow phaseArgument
```

The profile of this phase determines the node's future. See Section 3.4.3 for further details.

```
phaser :: Shadow → PhaseArg → PhaseId
mutationObservationWeights :: PhaseId → {(Operation,Weight)}
mortality, pmf, pof :: PhaseId → Double
```

We first decide if we shall mutate v or not, using the mortality. If we are to mutate v, recall from Section 3.3 that the average number of extra mutations of mutated nodes is $p_m/(1-p_m)$, where p_m is the PMF. We use a Poisson distribution with this mean to determine how many extra mutations we shall apply to v.

```
noOfMutns | chance† (mortality phase) = 0
          | otherwise = 1 + poisson† (pmf phase / (1 – pmf phase))
```

The function chance makes a decision based on the given probability.

```
chance† :: Probability → Bool
```

The Poisson distribution was chosen because it is well-known, because it ranges over non-negative integers, and because it is simple. Another similar distribution would also be appropriate.

```
poisson† :: Mean → Int
```

The profile gives the mutation and observation weights together, to relate frequency of mutators to frequency of observers. We use the ratio of mutators to observers to calculate the number of observations we shall apply to v. Section 3.3 details how we reach the approximation given in the code below.

```
mutnObtnWgts = mutationObservationWeights phase
mutnWgts = [(f,w) | (f,w) ← mutnObtnWgts, role f == Mutator]
obtnWgts = [(f,w) | (f,w) ← mutnObtnWgts, role f == Observer]
noOfObtns = sum [w | (f,w) ← obtnWgts] /
            sum [w | (f,w) ← mutnWgts]
```

The number of ephemeral observations and the number of persistent observations are calculated directly from the POF.

```
noOfEphmObtns = poisson† (noOfObtns * (1 – pof phase))
noOfPersObtns = poisson† (noOfObtns * pof phase)
```

We use the mutation-observation weights to determine which operations to use for the planned mutations and observations. Note that these operations are not all the same, despite the use of replicate, because we have hidden the pseudo-random choice within the impurity of chooseOperation.

```
mutns = replicate noOfMutns (chooseOperation† mutnWgts)
ephmObtns = replicate noOfEphmObtns (chooseOperation† obtnWgts)
persObtns = replicate noOfPersObtns (chooseOperation† obtnWgts)
```

The future of v is therefore the ephemeral observations, followed by the first mutation (ephemeral, if it exists), followed by a mixture of the remaining mutations (persistent) and the persistent observations, mixed using the function $\mathsf{mix}^\dagger$.

```
plan shadow = let mixPersOps [] os = os
                  mixPersOps (m:ms) os = m : mix† ms os
                  ...definitions of phase through persObtns...
              in ephmObtns ++ mixPersOps ms postMutnObtns
```

Biased choices ensure that the function $\mathsf{mix}^\dagger$ combines the lists evenly (see Section 4.3.1).

$$\mathsf{mix}^\dagger :: [a] \to [a] \to [a]$$

4.1.2 DUG Evaluation

The process of DUG evaluation is quite straightforward. Unlike DUG generation, we encounter no theoretical problems. In practice however, efficiency is a problem. The DUG evaluator sometimes takes more time over input-output and maintaining a lookup table than it does over performing the ADT operations. Times taken for DUG evaluation therefore vary little between ADT implementations, preventing us from accurately measuring their relative efficiencies. In such cases, we can solve this problem by using a C program to perform the input-output and lookup table maintenance. This requires an interface to C that allows C to call Haskell. We use an extension to the *Green Card* package [43]. See Section 4.3.2 for further technical details.

Definition 3.9 defines how a DUG should be evaluated lazily. When a non-version node is born, its result must be demanded immediately. As the result of an observer is either of type *Int* or of type *Bool*, we demand this value by converting it to an integer, and adding it to the *checksum*. This checksum is the result of the DUG evaluation. Different implementations of the same *observationally-equivalent* ADT evaluating the same DUG should return the same checksum. This may be used to check the correctness of one implementation against the correctness of another. An ADT that is not observationally equivalent allows many values for a single evaluation of an observation. For example, a bag ADT may support an operation that returns an unspecified element in the bag.

```
while not at the end of the DUG file do
    read the next birth or death
    if we read a birth then
        apply an operation to integers and nodes in the frontier, as given by the birth
        if the operation is an observer then
            convert the result to an integer and add it to the checksum
        else
            add the resulting node to the frontier
        fi
    else
        we read a death, so remove the dead node from the frontier
    fi
od
report the checksum
```

Figure 4.5: Overview of the DUG evaluation algorithm.

An overview of the algorithm is given in Figure 4.5. Fuller details follow, using the types defined in Section 4.1.1.

The main function takes a list of births and deaths, and returns the checksum made from evaluating the observations.

```
evaluateDug :: [BirthOrDeath] → Int
```

We shall read one birth or death at a time. As with DUG generation, we shall maintain a *frontier*, containing the nodes awaiting further applications. To define evaluateDug, we use an auxiliary function evaluateNodes, taking the current frontier and the current checksum as auxiliary arguments. Each node is identified by a NodeId and contains just a version of type T Int, where T is the type constructor exported by the ADT implementation used to evaluate the DUG.

```
evaluateNodes :: (NodeId → T Int) → Int → [BirthOrDeath] → Int
```

Initially, the frontier is empty, and the checksum is 0.

```
evaluateDug dug = evaluateNodes (λn · undefined) 0 dug
```

If there are no more births or deaths to read, we return the checksum.

```
evaluateNodes frontier checksum [] = checksum
```

A birth of an observer node creates a value either of type Int or of type Bool. We convert this value to an integer using resultToInt and add it to the checksum. As we must demand this value immediately, we must explicitly demand its value using seq, which evaluates its first argument before returning its second argument.

```
evaluateNodes frontier checksum (Birth nodeId f vs is : dug)
    | role f == Observer =
        let result = resultToInt (applyOperation f (map frontier vs) is)
        in seq result (evaluateNodes frontier (checksum + result) dug)
```

If a version node is born, we add the node to the frontier.

```
evaluateNodes frontier checksum (Birth nodeId f vs is : dug)
    | otherwise =
        let frontier' n
            | n == nodeId = resultToNode (applyOperation f (map frontier vs) is)
            | otherwise   = frontier n
        in evaluateNodes frontier' checksum dug
```

A death of a version node removes the node from the frontier. Recall that we do not record the death of a non-version node.

```
evaluateNodes frontier checksum (Death nodeId arcs shadow : dug) =
    let frontier' n | n == nodeId = undefined
                    | otherwise   = frontier n
    in evaluateNodes frontier' checksum dug
```

The following functions allow the result of an application of any operation to be manipulated, whether of type T Int, Int, or Bool.

```
applyOperation :: Operation → [T Int] → [Int] → Result
```

```
data Result = Node (T Int) | Int Int | Bool Bool

resultToInt :: Result → Int
resultToInt (Int i) = i
resultToInt (Bool b) = fromEnum b

resultToNode :: Result → T Int
resultToNode (Node v) = v
```

4.2 From Application to Profile

We create a profile of a DUG from a run of an application as follows:

(1) Extract the DUG describing how the run of the application uses an implementation of the ADT.

(2) Calculate the profile of this DUG.

Section 4.2.1 describes (1), and Section 4.2.2 describes (2).

4.2.1 DUG Extraction

The task of extracting a DUG from the run of an application is quite tricky in a lazy language like Haskell. One approach is to modify the compiler. However, as this solution depends on the details of a specific compiler, it would not be portable. An alternative approach is to transform the original program into one that gives the same result, but also produces a DUG. We adopt this method.

Problems of DUG Extraction

Here are two key goals we must achieve by transforming the original program, the problems they pose, and the solutions we choose:

- *Lazy Evaluation.* Whilst recording the operations applied, we must be careful not to evaluate anything that was not evaluated by the original program, and to evaluate everything in the same order as the original program. Otherwise we may get a different DUG, or the resulting program may fail to

terminate. We only examine something we know has been evaluated to at least the same degree that we will force.

It is possible that some arguments may not be evaluated at all. In such circumstances, after the program has finished, we record any such unevaluated arguments explicitly in the DUG. The DUG evaluation and profiling algorithms must accommodate these special nodes. See Section 4.3.2 and Section 4.3.4.

- *Recording the* DUG. We must record the DUG somewhere. However, side-effects are only allowed within the IO monad in Haskell. It would be highly undesirable to transform every function to work within the IO monad. Neither do we wish to pass information about the DUG as a result from every function that calls an ADT operation, all the way up to the main function. This would involve changing a lot of code. We avoid this problem by cheating. We interface to a side-effecting C function that records the DUG in a file.

  We cannot however, record arguments of type a, as we do not know in general how to store these. The user could supply a function to convert any value of type a to, say an integer. However, extracting this value could evaluate the argument more than previously. Therefore we decide not to record such arguments.

The DUG Extraction Algorithm

We modify the application and ADT implementation to perform the same task, but produce a DUG as a side-effect. We do this by *wrapping* the main function and every ADT operation. The wrapped main function performs some initialization, calls the old main function, and then tidies up the results. Each wrapped ADT operation works with wrapped versions. A version is wrapped with an *identity tag*. A wrapped operation uses the identity tags to record which nodes were used in the creation of the new node using which operation. A wrapped operation also calls the old operation, and wraps the result into a node with a new identity tag.

For example, the list ADT of Figure 4.2 provides the type constructor List. The wrapped version datatype for this ADT is given by:

```
data WrappedList a = Node Int (List a)
```

The wrapped implementation of cons is given by:

```
wrappedCons :: a → WrappedList a → WrappedList a
wrappedCons i v = let nodeId = new_node Cons
                  in seq nodeId (Node nodeId (cons i (arc v nodeId 1)))
```

where new_node is a C function that returns a new identity tag for a node, after recording which operation labels this new node. The function arc unwraps and returns the version argument, after recording the arc from this version node to the newly created node:

```
arc :: WrappedList a → NodeId → Int → List a
arc (Node from v) to position = seq from (seq (new_arc from to position) v)
```

where new_arc is a C function that returns only unit, after recording the arc, including argument node identity, result node identity, and the position of the argument node.

The function wrappedCons is only evaluated when cons would have been evaluated in the original program. It forces the evaluation of the identity of the new node, and then returns the wrapped result.

However, we do not record any of the arguments yet, as we do not know that they will be evaluated. We wrap the version argument with a call to arc. When the version argument would have been evaluated by the original program, we can examine the identity of the argument. The function arc does this, and then records the arc.

We do not wrap the argument to cons of type a for reasons given in the problems of DUG extraction above, but we can wrap arguments of type Int. The wrapped implementation of lookup is given by:

```
wrappedLookup :: WrappedList a → Int → a
wrappedLookup v i = let nodeId = new_node Lookup
                    in seq nodeId (lookup (arc v nodeId 0) (intArg i nodeId 1))
```

The function intArg records the integer argument in the label of this node at the given argument position, and returns the integer argument:

```
intArg :: Int → NodeId → Int → Int
intArg int node position = seq int (seq (int_arg int node position) int)
```

where int_arg is a C function that returns only unit, after recording the relevant details.

A general definition of a wrapped ADT is given in Figure 4.6. Further details are somewhat technical. For example, interfacing to the C functions requires use of a package called *Green Card* [43]. We leave these details to Section 4.3.3.

4.2.2 DUG Profiling

As with DUG evaluation, we read one birth or death at a time. The algorithm is quite straightforward.

The type of a profile is consistent with Section 4.1.1, except that there was an implicit profile in Section 4.1.1, whereas here it is an explicit argument. For example, the code mortality phase in Section 4.1.1 becomes mortality (phases profile phase) in this section, and similarly with the other profile properties.

```
data Profile = Profile {generationWeights :: {(Operation,Weight)},
                        phases :: (PhaseId → Phase)}
data Phase = Phase {mutationObservationWeights :: {(Operation,Weight)},
                    mortality :: Double, pmf :: Double, pof :: Double}
```

To calculate the generation weights and the mutation-observation weights, we keep a note of the number of nodes made by each operation (qualified by phase in the case of mutations and observations). To calculate the mortality, we need to keep both the number of nodes not mutated, and the total number of nodes. From this we can calculate the proportion of nodes not mutated: that is, the mortality. Similarly, we need to keep a numerator and denominator for the PMF and the POF. All this information is kept in a value of type ProfileData.

```
data ProfileData =
  ProfileData {gWgts :: {(Operation,Weight)},
               phaseDatas :: (PhaseId → PhaseData)}
data PhaseData =
```

$$\mathsf{data\ T^w\ a\ =\ Node\ Int\ (T\ a)}$$

$$\mathsf{f_i^w\ ::\ } w_T(\mathsf{t_{i,1}}) \rightarrow \cdots \rightarrow w_T(\mathsf{t_{i,n_i}})$$
$$\mathsf{f_i^w\ a_1\ \ldots\ a_{n_i-1}} =$$
$$\quad \mathsf{let\ nodeId\ =\ new\_node\ } w_N(\mathsf{f_i})$$
$$\quad \mathsf{in\ seq\ nodeId\ } w_R(\mathsf{f_i\ } w_A(\mathsf{a_1})\ \ldots\ w_A(\mathsf{a_{n_i-1}}))$$

where

$$w_T(t) = \begin{cases} \mathsf{T^w\ a}, & \textit{if } t = \mathsf{T\ a} \\ t, & \textit{otherwise} \end{cases}$$

$w_N(\mathsf{f_i})$ *gives the data constructor that names* $\mathsf{f_i}$

$$w_R(e) = \begin{cases} \mathsf{Node\ nodeId}\ e, & \textit{if } e \textit{ has type } \mathsf{T\ a} \\ e, & \textit{otherwise} \end{cases}$$

$$w_A(\mathsf{a_j}) = \begin{cases} \mathsf{arc\ a_j\ nodeId\ j}, & \textit{if } \mathsf{a_j} \textit{ has type } \mathsf{T^w\ a} \\ \mathsf{intArg\ a_j\ nodeId\ j}, & \textit{if } \mathsf{a_j} \textit{ has type Int} \\ \mathsf{a_j}, & \textit{otherwise} \end{cases}$$

Figure 4.6: Definition of a wrapped ADT. For an ADT exporting type constructor T and operations $\mathsf{f_i\ ::\ t_{i,1}} \rightarrow \cdots \rightarrow \mathsf{t_{i,n_i}}$, the wrapped ADT exports type constructor $\mathsf{T^w}$ and operations $\mathsf{f_i^w}$.

```
PhaseData {moWgts :: {(Operation,Weight)}, unmutated :: Int, total :: Int,
           persMus :: Int, mus :: Int, persObs :: Int, obs :: Int}
```

Initially, the generation weights and mutation-observation weights are all zero, as are the remaining fields.

```
profile :: [BirthOrDeath] → Profile
profile = let emptyProfileData =
                ProfileData {(f,0) | f ← operations, role f == Generator}
                            (λp · emptyPhaseData)
              emptyPhaseData =
                PhaseData {(f,0) | f ← operations, role f /= Generator}
                          0 0 0 0 0 0
          in calculateProfile ∘ foldl gatherProfile emptyProfileData
```

The function gatherProfile is folded over the list of births and deaths to calculate the final profile data.

```
gatherProfile :: ProfileData → BirthOrDeath → ProfileData
```

The function calculateProfile converts the final profile data into a profile.

```
calculateProfile :: ProfileData → Profile
calculateProfile (ProfileData gWgts phaseDatas) =
    Profile gWgts (calculatePhase ∘ phaseDatas)


calculatePhase :: PhaseData → Phase
calculatePhase (PhaseData moWgts unmutated total persMus mus persObs obs) =
    Phase moWgts (fromIntegral unmutated/fromIntegral total)
                 (fromIntegral persMus/fromIntegral mus)
                 (fromIntegral persObs/fromIntegral obs)
```

Births of generators are used to calculate the generation weights. The other births are ignored, as the deaths are sufficient to calculate the rest of the profile.

```
gatherProfile (ProfileData gWgts phaseDatas) (Birth n op vs is)
    | role op == Generator = ProfileData (addWgt gWgts op) phaseDatas
    | otherwise = ProfileData gWgts phaseDatas
```

The function addWgt increases the weight of an operation by one.

```
addWgt :: {(Operation,Weight)} → Operation → {(Operation,Weight)}
addWgt wgts op = {if f == op then (f,w+1) else (f,w) | (f,w) ← wgts}
```

The death of a node v affects the profile of the phase to which v is assigned.

```
gatherProfile (ProfileData gWgts phaseDatas) (Death n past shadow) =
    let phase = phaser shadow phaseArgument
        oldPhaseData = phaseDatas phase
        newPhaseDatas p | p == phase = gatherPhase oldPhaseData past
                        | otherwise  = phaseDatas p
    in ProfileData gWgts newPhaseDatas
```

The function gatherPhase returns a new phase data using the past of the dead node v.

```
gatherPhase :: PhaseData → [Arc] → PhaseData
gatherPhase (PhaseData moWgts unmutated total persMus mus persObs obs)
            past =
    let ops = map targetNodeOp past
        ms = length [op | op ← ops, role op == Mutator]
        os = length [op | op ← ops, role op == Observer]
        postMutnObs = length [op | op ← dropWhile ((/= Mutator) ∘ role) ops,
                                   role op == Observer]
        newMoWgts = foldl addWgt moWgts ops
        newUnmutated = if ms == 0 then unmutated + 1 else unmutated
        newTotal = total + 1
        newPersMus = persMus + max (ms–1) 0
        newMus = mus + ms
        newPersObs = persObs + postMutnObs
        newObs = obs + os
    in PhaseData newMoWgts newUnmutated newTotal newPersMus newMus
                 newPersObs newObs
```

The calculation of the phase is quite straightforward: ops is the list of operations applied to v, ms is the number of mutations of v, os is the number of observations

of v, and postMutnObs is the number of observations occurring after the first mutation (ie. the persistent observations). The new mutation-observation weights ratio is calculated by adding every operation in ops. If the number of mutations is zero, then v is not mutated. The number of persistent mutations is one less than the number of mutations, if any. The number of persistent observations has already been calculated.

4.3 Technical Details

The algorithms presented in this chapter are implemented in Haskell to create the tool of Chapter 6. However, for both efficiency and practical reasons, some refinements of this code were necessary. That is, some of the code is too slow, and some is not primitive to Haskell (eg. sets). We shall now detail the key points of these refinements.

4.3.1 DUG Generation

Frontier

The frontier is presented as a set in Section 4.1.1, but sets are not primitive to Haskell. As we also need to remove a node pseudo-randomly (using removeNode), we need a set ADT with random retrieval. As we will never try to add the same node twice to the frontier, a bag ADT with random retrieval will suffice. A bag with random retrieval does not require any examination of the elements, and is therefore easier to implement than a set with random retrieval.

The implementation of this ADT is based on the random-access lists of Okasaki [33]. An element is added using *cons*. An element is randomly retrieved by randomly choosing a valid index into the list. The element at this index is then removed by updating it with the *head* of the list, and then taking the *tail* of the result.

Buffers

Section 4.1.1 represents the buffers as a function from operations to lists of nodes. In practice it is easier to implement the buffers as a list of lists of nodes. As the

number of operations is quite small, this is efficient.

Argument Position in Death

Section 4.1.1 uses a function versionArgs to allocate the correct argument position to the version arguments of an application, recorded in the past of each argument, and subsequently in their deaths. As we restrict every node argument to version nodes, we actually just record the position of the version argument with respect to other version arguments. So, for example, for the application $\mathsf{f}\ \mathsf{i}_0\ \mathsf{v}_0\ \mathsf{i}_1\ \mathsf{i}_2\ \mathsf{v}_1$, we record the argument position of v_i as i. This allows us to define versionArgs by:

```
versionArgs :: Operation → [Int]
versionArgs f = [1..]
```

and let the application of zipWith truncate this to the appropriate length.

Choice Functions

As indicated in Section 4.1.1, the pseudo-random functions must each take a seed as an additional argument, which was left out of the presentation of the algorithm for the sake of clarity. These seeds are threaded through every function calling a pseudo-random function. The pseudo-random number generator was taken from [9]: the "minimal standard random number generator", taken in turn from [42]. On recommendations of [41], the multiplier is changed as follows:

$$a = 48271,\ q = 44488,\ r = 3399$$

This random-number generator requires a Haskell implementation supporting integers in the range $[-2^{31}\,..\,2^{31}-1]$. All of the functions implementing some choice are based on a function rndRng that returns an integer between 0 and a given ceiling, inclusive of 0 and exclusive of the ceiling.

```
rndRng :: Int → Int → Int
rndRng ceiling seed = seed `mod` ceiling
```

A seed is simply an integer ranging over $[1\,..\,2^{31}-2]$. An integer between m and n inclusive can be chosen by m + rndRng (n–m+1).

The function chooseInt may choose an integer from: All, which resolves to choosing an integer between minBound and maxBound; a Pool, which resolves to choosing an integer between 1 and the pool size; a range m:..:n, which resolves to choosing an integer between m and n; or a set, which is implemented using a set ADT with random-retrieval, implemented simply as an ordered list.

The function mix is implemented by choosing one element from each list, with probability biased according to the length of each list, ensuring an even mixing—to mix a list xs of m elements with a list ys of n elements, elements are taken with $m/(m+n)$ probability from xs, and with $n/(m+n)$ probability from ys.

The functions poisson, chance, and chooseOperation use a discrete random variable with a particular distribution. The functions chance and poisson are combined in the choice of noOfMutns inside the definition of plan to create one random variable. The function chooseOperation is a random variable ranging over the operations, biased according to the given weights.

Such a random variable is implemented by creating a cumulative distribution, represented as a list of integers ranging between 0 and some large fixed upper limit scale. An integer n is chosen between 0 and scale, and the index i of the first integer in the list greater than n is the value of the discrete random variable. If the random variable has a range of values of some other type than integer, for example operation, then an enumeration of the range will allow i to index into this enumeration.

The choice of scale must reflect three points:

- The larger the value of scale, the more accurate the random variables are. The smallest change in probability that a scale of n can capture is 1/n.

- The larger the value of scale, the more chance of bias in values chosen between 0 and scale using rndRng scale seed. Recall that rndRng is implemented using mod. If a scale of 15×10^8 (approximately 2/3 of the largest possible seed) is used, we would expect more low values than usual, because values from 0 to approximately 7×10^8 can each be produced by two different seeds whereas values above this can each only be produced by one seed. In practice we observe this bias as producing values with an average of around 0.4 times the largest value. However, with a scale of 2^{30} (half the

largest possible seed), we should have no bias, and in practice this produces values with an average of 0.5 times the largest value, confirming a lack of bias.

- Ideally, the best value of scale would be the ceiling of the range of seeds, where rndRng scale seed becomes id. Unfortunately, we experience rounding problems with Int using this value of scale (as it is the largest possible value of type Int).

Therefore, a scale of 2^{30} was used.

Weights

A collection of weights is given in Section 4.1.1 as a set of pairs (Operation,Weight), but in practice is implemented as a list [Weight] with the operation given by the index, when operations are ordered first according to role and then alphabetically.

Format of DUG Files

Section 4.1.1 represents a DUG by a list of births and deaths. Within the Haskell world, this is indeed the representation of a DUG. However, if we wish to store a DUG in a file, without the use of a special library, we need to store the DUG as a sequence of characters. We also compress the DUG representation to minimise the input-output overhead of DUG evaluation.

A birth is represented as a sequence of integers: the operation identity tag, the identity tags of the nodes used as version arguments, and the integer arguments. The births are ordered in the file according to identity tag. Therefore, the identity tag of a new born node is given by its position in the file. A death is also represented as a sequence of integers: a zero, and the identity tag of the dead node. Operation identity tags start at 1 to distinguish a birth from a death. The number of integers making up a birth or death is determined by the first integer: for a birth it is the number of arguments of the operation plus one, and for a death it is two. The other fields of a death given in Section 4.1.1 (outgoing arcs and shadow) are not required for DUG evaluation, and can be reconstructed from the births for DUG profiling.

An integer i is stored as a sequence of characters: s, x_{s-1}, x_{s-2}, ..., x_0; where $0 \leq s \leq 4$, and $i = \sum_{n=0}^{s-1} x_n 2^{8n}$; that is, the non-zero 8-bit bytes representing i with most-significant byte first, preceded by the number of these bytes. Note that 0 is represented simply by 0. If a character is larger than 8 bits, this representation could be improved.

As the identity tags of the nodes start from 1, we use 0 to represent an undefined version argument, whose creation is possible through DUG extraction.

4.3.2 DUG Evaluation

Two versions of DUG evaluation were implemented: one wholly in Haskell, and one partly in C and partly in Haskell. The former suffers from a very large overhead of input-output and bookkeeping, leaving the work done by the ADT operations swamped, sometimes yielding unsatisfactory results. The latter cuts down the overhead to a consistently satisfactory level by implementing everything bar the ADT operations in C. This requires a version of the Green Card foreign language interface [43] that allows C to call Haskell. As such an interface is only currently available for one compiler (York nhc13 [53]), the pure Haskell version was kept. See Section 6.1.2 for an estimate of the overhead of DUG evaluation for each version.

Typically, a DUG evaluator made with Green Card, evaluating a reasonably large DUG file (around 100Kb), is around 20 times faster than the same DUG evaluator made without Green Card evaluating the same DUG file.

Without Green Card

The Haskell version requires two changes from the algorithm presented in Section 4.1.2.

Frontier. Section 4.1.2 represents the frontier as a function. We replace this with a finite map ADT, implemented by a data structure very similar to the Elevator implementation of random-access lists—see Section 2.2.7.

Inlining. The operation applyOperation is fused with each operation of the ADT implementation to remove a layer of interpretation. This creates one right-hand

side of evaluateNodes per operation. For example, for the operation lookup of Figure 4.2, the following code is used:

```
evaluateNodes frontier checksum (Birth nodeId Lookup [v] [i] : dug) =
    let result = fromEnum (lookup (frontier v) i)
    in seq result (evaluateNodes frontier (checksum + result) dug)
```

Note that resultToInt and resultToNode are now redundant, as is the test for the operation being an observer. Note also that the arguments for lookup are pattern matched out of the lists stored in a birth. The format of a DUG file is more like a list of integers (see Section 4.3.1) and so the pattern matching is more efficient than as presented here (the pattern matching is closer to (4:v:i:dug)).

Strictness. The strictness of different implementations of the same ADT vary in general. This could mean that some operations are forced by one implementation but not by another. In order to ensure that the DUG evaluator for each ADT implementation performs the same amount of bookkeeping, regardless of which operations are forced, the bookkeeping is made strict.

This means demanding the lookup of a version argument in the frontier, without demanding the argument value, and demanding the value of a non-version argument. This is achieved by wrapping up the version arguments in the frontier:

```
data Node = Node (T Int)
```

and by adding unwrapping of nodes and calls to seq in the definition of evaluateNodes. For example, the definition above becomes:

```
evaluateNodes frontier checksum (Birth nodeId Lookup [v] [i] : dug) =
    let v' = frontier v
        Node v'' = v'
        result = fromEnum (lookup v'' i)
    in seq v' (seq i (seq result (evaluateNodes frontier (checksum + result) dug)))
```

Undefined Arguments. DUG extraction makes undefined arguments a possibility. DUG evaluation gives the value undefined to such arguments.

With Green Card

The DUG evaluator built using Green Card is a small Haskell program containing information specific to the ADT implementation used. This Haskell program calls a larger, more general C library. Essentially the same algorithm is used in C to read in and evaluate the DUG operations, except that the C program must somehow call Haskell functions to perform the operations. Before the Haskell program calls the C program, it registers each Haskell ADT operation as a *stable pointer* with the C program. During evaluation, the C program uses these Haskell references to call the ADT operations. The frontier is implemented as a hash table, and input-output is buffered. Note that as the bookkeeping is now in C, it is strict (see the *Strictness* heading above).

4.3.3 DUG Extraction

Whilst the DUG extracting version of an application is running, a hash table of every node is maintained. The function new_node adds a node to the hash table, and the functions new_arc and int_arg update the relevant arguments of the target node. After the application has finished, we traverse the hash table for every observer node in the order they were created. For each observer node, we traverse the graph of its predecessors until we reach a previously written node. On the way back to the observer node, we write the birth of every node to the DUG file, in depth-first order to ensure all argument nodes are written before their operation nodes.

By maintaining a count of how many arcs exist from each node to currently unwritten nodes, when a node is no longer an argument of an unwritten node, we write the death of this node, as this node has left the implicit frontier. This check is made every time a node v is reached by a graph traversal from an observer node, whether v is previously written or not.

The order in which the nodes are written is maintained, as this defines the node identity tags used by anything reading the DUG file. These node identity tags must be used when writing version argument identity tags. The order in which the nodes were actually evaluated is lost (except for preserving the order of evaluation of observers). This is a direct result of the restriction of Definition 3.6

constraining the order of evaluation. The actual order of evaluation could be reported, as it may be of interest, but this is not currently implemented.

When a node is added to the hash table, every argument is recorded as undefined. If a version argument is still undefined after the application has finished, we write the argument to the DUG file as being undefined. Currently, we make no provision for recording undefined non-version arguments: to do so would be costly, without much benefit; undefined non-version arguments are given the value 0. Note that this includes the non-version arguments of type a, which we cannot record for reasons given in Section 4.2.1.

4.3.4 DUG Profiling

The only difference between Section 4.2.2 and the actual implementation of DUG profiling is the format of the profile. As already indicated in Section 4.3.1, a collection of weights is implemented as a list. Phases are given by a function of type PhaseId $\rightarrow$ Phase in Section 4.3.1, whereas in practice they are given by a list, as PhaseId is an integer, letting the index of the Phase give the PhaseId.

DUG extraction makes unevaluated arguments a possibility. The DUG profiling algorithm must assign a shadow to an unevaluated version argument, in order to record the effect of any operations on the argument in the correct phase. The shadow data structure is therefore extended to supply the shadow of any unevaluated version argument. Nothing more is known about the version argument, for example what other nodes operate on it, yet its effect on the profile must be defined somehow. We define its effect separately as follows. The mutations and observations are counted (in weights, and in the denominators of PMF and POF), because these reflect the evaluation of the operation applied to the unevaluated arguments. None of the mutations and observations are considered persistent, on the grounds that persistence reflects reuse of a data structure, whereas an unevaluated argument is not even used once. The version argument is not counted as a node, on the grounds that it was never evaluated, and therefore in a sense, it never existed.

4.4 Testing

How accurate are the implementations of the DUG algorithms? How efficient are they? From the point of view of implementation, we address these questions for each algorithm individually in this section. More general questions concerning the accuracy or usefulness of the benchmarking process as a whole are tackled in Chapter 7.

As the performance of the algorithms can vary between ADTs, we conduct tests across a few very different ADTs:

- Queue
- Random-Access Sequence
- Set with Random Retrieval

The queue ADT is the simplest of the three. The random-access sequence ADT adds the complexity of operations taking integers as arguments. The set ADT includes operations taking more than one version argument, and quite a complex shadow data structure (based on a set itself). We use the York nhc13 compiler [53] (release v0.9.4), running executables in a heap of 80Mb, on an SGI Indy running IRIX 5.3.

4.4.1 DUG Generation

Accuracy

The accuracy of DUG generation is important, though the benchmarking techniques introduced in Section 5.4 reduce this importance. To measure the accuracy, we compare the target profile with the actual profile of the DUG generated. We do this for 100 DUGs from each of the three ADTs listed above. Table 4.1 lists the mean and maximum difference for each profile attribute. Some inaccuracy is due to the probabilistic means of generating a DUG. For example, if we want half of the 100 mutations to belong to an operation f, we choose f with probability 0.5 for each mutation. We will not always get 50 mutations belonging to f, but this will be the mean.

Profile Attribute	Mean Difference (%)	Maximum Difference (%)
Weight	1.4	31.3
Mortality	4.4	70.4
PMF	0.3	7.5
POF	2.4	35.9

Table 4.1: The mean and maximum differences between target and actual profiles of 100 DUGs for each of three ADTs. Each DUG has 1000 nodes. We group the generation and mutation-observation weights together. Each difference is given as a percentage of the possible range. By normalising the weights ratios, the range of each weight is [0..1], as it is for the other three profile attributes.

A larger degree of inaccuracy results from the rejection of planned applications of operations by the shadow data structure. To take an extreme example, if we want a DUG for lists with no `cons` operations, then we will not get any `tail` operations either, regardless of the target profile. To take another example, the largest difference shown in Table 4.1—70.4% difference in mortality—is for the random-access sequence ADT. The target PMF for this DUG is 0, and so all nodes will have at most a single mutation planned in their future. The target mutation weights ratio is

$$\texttt{cons} : \texttt{tail} : \texttt{update} = 1 : 1 : 20$$

and so 91% of mutations will be applications of `update`. A list can only be generated by `empty`. However, `update` cannot be applied to `empty`. Therefore, 91% of the lists generated by `empty` will not be mutated, and therefore contribute to the mortality. This increases the actual mortality to a value much larger than the target mortality.

Mortality is also increased by the death of all nodes in the frontier when the DUG generation algorithm finishes. This will be high for large PMF values.

Efficiency

The efficiency of DUG generation is not crucial to the benchmarking process. By examining the heap profile of DUG generation, we find that evaluating the future

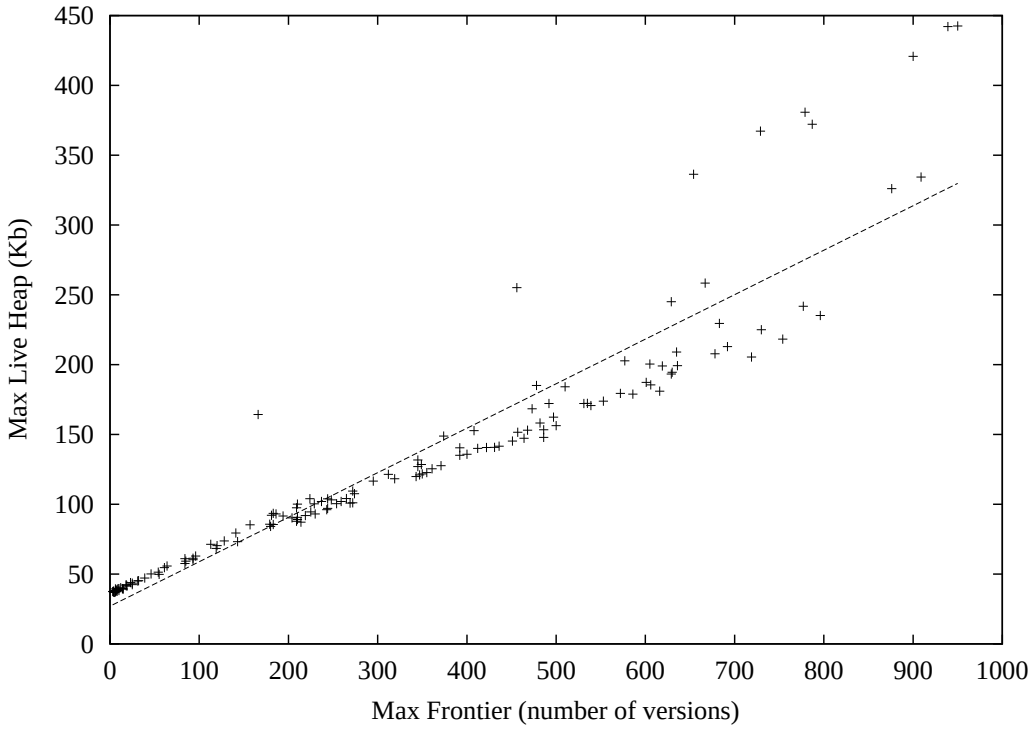


Figure 4.7: A plot of maximum live heap against maximum frontier for DUG generation on 50 randomly chosen profiles for each of three ADTs. Each DUG has 1000 nodes.

---

of a new node at the time of creation considerably improves space usage. The heap size is linear in the size of the frontier. To demonstrate this, Figure 4.7 plots the maximum frontier size against the maximum live heap size for the generation of several DUGs across three ADTs listed above. The plot confirms a general trend of linearity, though there are some surprisingly large heaps, especially for large frontiers. On closer examination we find that every point lying way above the interpolated line comes from the generation of a DUG with a target PMF of 0.95. The target PMF of every DUG was chosen from $[0, 0.05, \ldots, 0.95]$. The DUGs with target PMF of 0.9 have points that lie a little above the interpolated line, but still way below those with target PMF of 0.95.

We can explain this by considering the amount of space allocated to a node in the frontier. The *future* of the node accounts for the majority of this space, that is, the list of future operations to apply to the node. This list contains mutators and observers. The number of mutators has mean $\text{PMF}/(1 - \text{PMF})$. For a PMF of

0.9, this is 9; for a PMF of 0.95 however, this is 19, more than twice as many. So, in moving from a PMF of 0.9 to a PMF of 0.95, we double the amount of space allocated to each node in the frontier, and hence double the maximum live heap. This accounts for the sudden leap from points with a PMF of 0.9 to those with a PMF of 0.95.

4.4.2 DUG Evaluation

Accuracy

The only form of inaccuracy in DUG evaluation is that strictness issues may lead to only part of the DUG actually being evaluated—see Section 7.3.3.

Efficiency

The efficiency of DUG evaluation is very important in obtaining good benchmarking results. If the overhead of DUG evaluation is too great, the accuracy of estimating the ratio of work done by different ADT implementations is reduced. See Section 6.1.2 for a detailed discussion of this issue.

4.4.3 DUG Extraction

Accuracy

The DUG extraction algorithm accurately captures the DUG of an application, except for evaluation order, arguments of type a and the sharing of operations taking no arguments. The actual evaluation order has to be changed to suit the restriction given in the definition of a DUG, namely that an argument must be ordered before its operation. However, this change does preserve the order of evaluation of observer nodes, and only affects the POF attribute of the profile. Arguments of type a cannot be extracted for reasons given in Section 4.2.1.

Every result of an operation that takes no arguments and whose type does not have a class context will be shared. The application will only evaluate such an operation once, and will share the result. If however, the operation takes no arguments but has a type with a class context, like the *empty* of the heap ADT

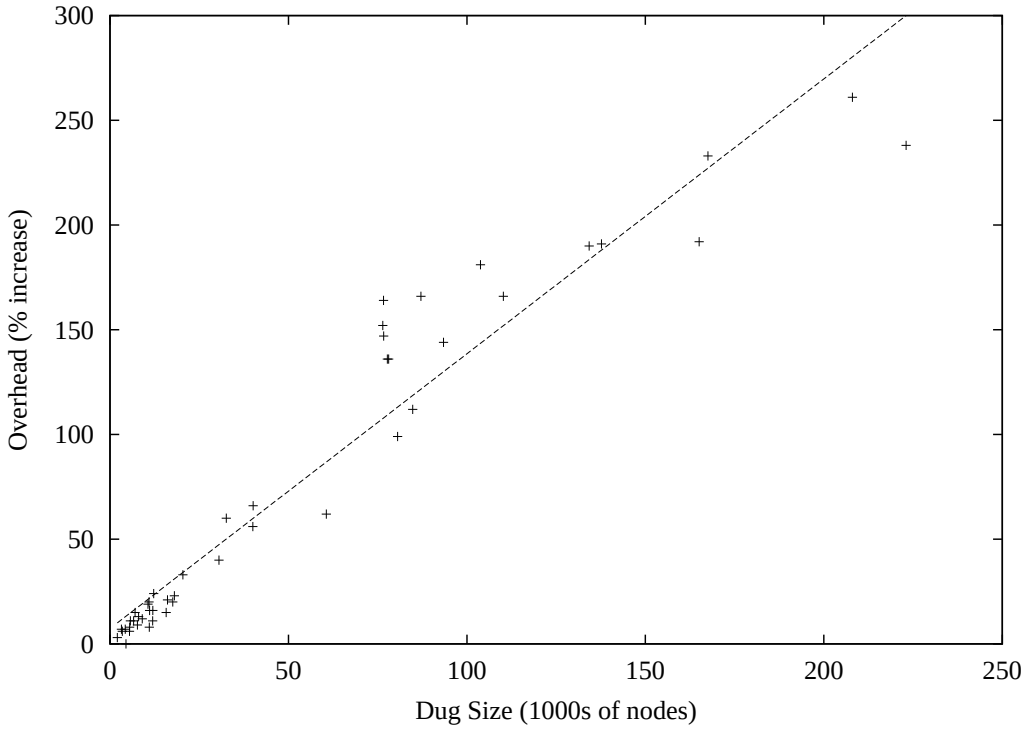


Figure 4.8: Overhead incurred by modifying an application for DUG extraction, plotted against size of the extracted DUG—12 different applications running on 4 different data sets each, over 3 different ADTs, making 48 points in all.

(see Table 2.3), then the application may re-evaluate the operation, as it restricts the operation to a particular instance of the class.

Efficiency

Modifying an application to extract a DUG as it runs introduces an overhead. To estimate this overhead, we time several applications both with and without the extraction modification. Figure 4.8 shows the overhead incurred by modifying an application for extraction. Over the 48 DUGs extracted, the average added overhead is 75%. The percentage overhead varies significantly according to how much work the application does that is not related to the ADT—most of the applications we examine use the ADT intensively, so the figure should be less for other applications.

4.4.4 DUG Profiling

Accuracy

There is no inaccuracy in DUG profiling, beyond the inaccuracy involved with using floating point numbers.

Efficiency

As with DUG generation, the efficiency of DUG profiling is not crucial to the benchmarking process. As the profile is only demanded at the end of analysing the DUG, care must again be taken to evaluate the information gathered as it arrives. A lazier approach would accumulate many suspended computations in the heap. The heap size is linear in the size of the frontier, as it is with DUG generation. To demonstrate this, Figure 4.9 plots the maximum frontier size against the maximum live heap size for the profiling of several DUGs across three ADTs listed above. As with DUG generation, the plot confirms a general trend of linearity.

4.5 Summary

We have defined algorithms for creating a benchmark from a profile, and calculating a profile of an application. The former comprises DUG generation and DUG evaluation, and the latter comprises DUG extraction and DUG profiling. These algorithms are bundled together to form the core of the benchmarking tool presented in Chapter 6.

As well as presenting the algorithms in an abstract manner, we have also tackled the issues surrounding a concrete implementation in Haskell. We have also tested the algorithms for accuracy and efficiency. We shall test the effectiveness of the benchmarking process as a whole in Chapter 7.

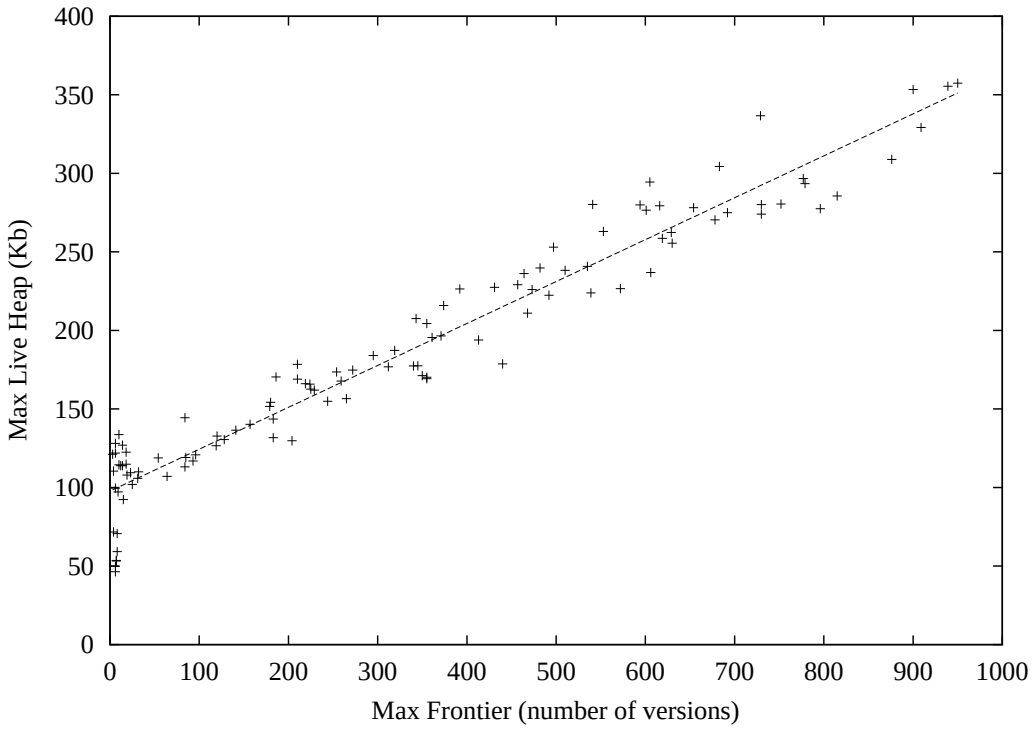


Figure 4.9: A plot of maximum live heap against maximum frontier for DUG profiling of 50 randomly generated DUGs for each of three ADTs. Each DUG has 1000 nodes.

Chapter 5

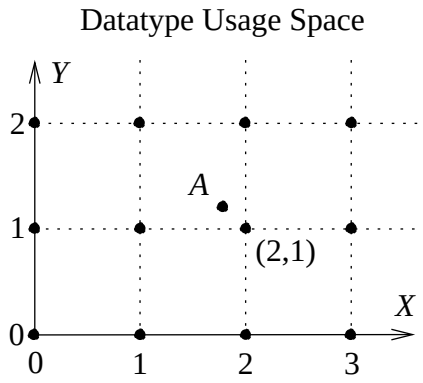
Exploring Datatype Usage Space

Chapter 1 motivated the need for benchmarking results qualified by the pattern of datatype usage. We proposed to provide these results by constructing a convenient means of obtaining benchmarks with known patterns of use. Chapter 3 showed (a) how to create a benchmark from a description of use, and (b) how to create a description of use from an application. Chapter 4 gave algorithms for (a) and (b). But how can we use (a) and (b) to generate and present benchmarking results qualified by use? The results must not take too long to gather and must be simple enough to be understood by the user. This chapter explores this problem by looking at several possible approaches to a solution.

5.1 Exhaustive Exploration

The most naive solution to providing benchmarking results is to create a benchmark with every possible pattern of use, and provide a lookup table of times of each implementation running each benchmark. The user simply obtains the pattern of use of their application, and looks up the quickest implementation in the appropriate row of the table.

We shall assume that a pattern of use consists of a list of n *attributes*. The profile we defined in Section 3.3 that captures the pattern of use has *continuous* attributes. Therefore the space covered by the profiles is continuous and hence contains an infinite number of points. Therefore we must divide each attribute using a suitable granularity, for example, by rounding the mortality to



Datatype Usage		Best Data Structure
X	Y	
0	0	Naive
2	1	AVL

Figure 5.1: Mapping datatype usage space with two attributes, X and Y. X and Y each capture some aspect of datatype usage (not given here). In general we may have many more dimensions to the coordinate system. The table lists points in the space against the best data structure for that use. In general we may list more about the efficiencies of data structures than which is best. An application may have datatype usage A, which is nearest to the coordinate (2,1). The table lists AVL as the best data structure for this datatype usage.

---

the nearest 0.01. Figure 5.1 shows an example of such a table of results, for patterns of use containing just two attributes X and Y, and listing just the quickest implementation.

Unfortunately, this approach is not practical. Such a table would cover a huge number of points, and the total time to collect the results for each point would be far too large. For example, consider an ADT with just 5 operations (1 generator, 2 mutators, and 2 observers). Using the profile defined in Section 3.3, the pattern of use consists of 8 attributes, two of which are redundant (the generation weight, and one of the mutation-observation weights), leaving just 6. Rounding each attribute very coarsely to give just three possible values gives a total of $3^6 = 729$ distinct profiles. Running even just one benchmark for each profile and each implementation would take a long time. The table would also be huge, and hence rather unreadable, especially if the user wants an overview of which implementation to use when.

This approach also relies on the accuracy of benchmark generation—that is, how well the profile of the generated benchmark matches the desired profile.

Although benchmark generation is reasonably accurate (see Section 4.4.1), it would be better to remove this dependency.

Summary. Exhaustive exploration is simple and straightforward, but not practical; it takes far too much time to run, generates verbose results, and relies on benchmark generation accuracy.

5.2 Selective Exploration

Exhaustive exploration is not practical primarily because the number of patterns of use is exponential in the number of attributes. Even just 6 attributes taking only 3 possible values each results in $3^6 = 729$ distinct patterns of use.

One way to reduce the number of attributes is to remove *insignificant* attributes—those attributes that have little or no effect on the performance of the ADT implementations. Removing such attributes should have little effect on the accuracy of the resulting *selective exploration* when considered as a summary of the *entire* space.

But how do we measure the effect of an attribute on the performance of ADT implementations? Suppose we measure their performance at a particular point p in the datatype usage space. Now let p' be another point obtained from p by altering the value of a single attribute A. Suppose we now measure the performance of the ADT implementations at p'. If the performance has not changed significantly from p to p', then we can conclude, for p and p' at least, that A has little effect on performance. By taking a sample of such points, we can conjecture which attributes are insignificant.

But how do we define a significant change in performance? We need some means of measuring the correlation between the two sets of performances. The standard statistical property *correlation coefficient* is defined over n pairs of values for x and y by:

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{[n\sum x^2 - (\sum x)^2][n\sum y^2 - (\sum y)^2]}}$$

and measures how well the two sets of data, if plotted, match a straight line.

What happens if we use the correlation coefficient over the two sets of performance times? Unfortunately, this measure can be heavily influenced by a very slow ADT implementation. For example, suppose the times (in seconds) for one pattern of use were $[1, 2, 4, 64]$, and for another pattern of use, $[4, 2, 1, 64]$ (listing the times in the same order for each). The correlation coefficient for these sets of times is 0.997, greater than the correlation between $[1, 2, 4, 7]$ and $[1, 2, 4, 9]$.

We are more interested in a small change in the performance of the best implementations than a large change in the performance of the worst implementations. Therefore it is reasonable to consider using the correlation of the reciprocals of the times:

Times		Correlation of Reciprocals
[1,2,4,64]	[4,2,1,64]	-0.055
[1,2,3,4]	[2,1,3,4]	0.262
[1,2,3,4]	[1,2,3,64]	0.971
[1,1.1,2,3]	[1.1,1,2,4]	0.972

This means of measuring a significant change in performance seems more reasonable. Those attributes with an average correlation above a given value could be removed.

However, both selective and exhaustive exploration assume that the pattern of use is captured entirely by the attributes of a profile. Unfortunately, one important pattern of use has proved very hard to capture adequately within a profile: size. The shadow profile *can* capture size, but selective and exhaustive exploration assume that a benchmark can be created with the given attributes. However, it is not possible in general to create a benchmark with a given shadow profile, only to calculate the shadow profile of a given benchmark.

The size of a data structure can significantly affect the efficiency of an operation applied to it. For example, performing the operation *snoc* on a naïve queue takes time proportional to the size of the queue (see Section 2.1.1).

Summary. Selective exploration improves on the impracticality of exhaustive exploration when there are sufficiently many insignificant attributes. However, in

common with exhaustive exploration, it does not explore the important attribute of size explicitly.

5.3 Capturing Size

Since neither selective nor exhaustive exploration capture the important attribute of size, we look at ways to remove or reduce this insufficiency.

The size of a data structure is determined by the quantity and order of applications of mutators (and by the choice of generator(s)). For example, the more applications of *cons*, the larger the list; and the more applications of *cons* in succession, the larger the list. The quantity aspect is captured by the weights ratio of a profile, but the order aspect is lost. For example, a sequence of n applications of *cons* followed by n applications of *tail* has the same profile as the applications *cons* then *tail* repeated n times. However, the former sequence of applications has average size of list $n/2$, whereas the latter has average size $3/2$.

We need to capture the order of mutations, but how? We present three attempts, with their advantages and disadvantages.

5.3.1 Growth and Decay

A simple way to capture order of mutations is to split the profile into *phases* (see Section 3.4.3). Phases partition a DUG, and the profile of each phase is recorded separately. The partitioning of the DUG is based on auxiliary *shadow* information stored about each version node (see Section 3.4.1). The shadow information is based on the history of the version node's creation—that is, which operations created it.

In particular, we could store the *age* of a data structure at each version node—the age of a version node being the number of mutators used to create it. We could partition the DUG into nodes of age A or less, and nodes of age greater than A, for some constant A. By setting the ratio of size-increasing operations higher in the former phase than in the latter, we can create a DUG with a *growth* phase and a *decay* phase. The size of data structures tends to increase more in the growth phase than in the decay phase.

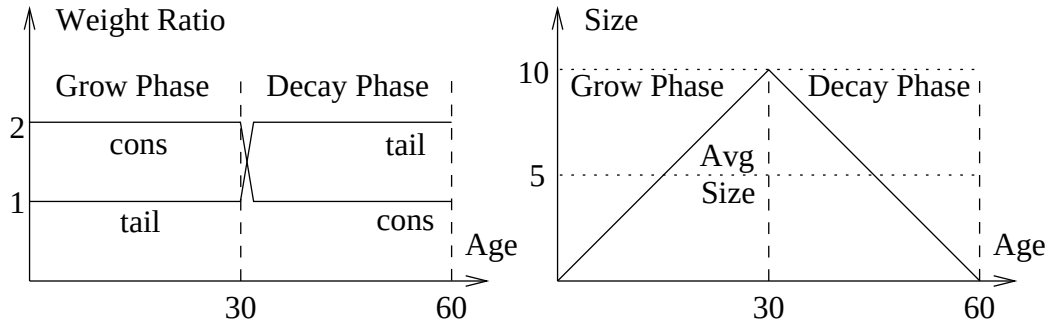


Figure 5.2: An example of growth and decay phasing on lists. The ratio of *cons* : *tail* is 2 : 1 for nodes aged under 30, and 1 : 2 for nodes aged 30 to 60. Assuming lists are generated by *empty*, and hence start at size 0, nodes aged 30 are on average lists of size 10. Nodes aged 60 are on average lists of size 0. Assuming an equal distribution of nodes over age, the average size of a list is 5.

For example, consider phasing a DUG over list operations into nodes aged 30 or less, and those older. Now set the profile of the former phase (the growth phase) to have a weights ratio of $cons : tail = 2 : 1$, and the latter phase to have $cons : tail = 1 : 2$. Also make sure that any nodes aged over 60 are not mutated (this can be done by adding a final phase for nodes aged over 60 with mortality 1). Generating a benchmark with these phased profiles will make the average size of a list about 5—see Figure 5.2.

Hence, for controlling the average size of a data structure when generating a benchmark from a profile, growth and decay is useful. Unfortunately, this is complicated by the possibility of the number of nodes varying over age. Both PMF and the weights of mutators taking more than one version argument affect the increase or decrease of the number of nodes over age. More importantly, imposing the structure of growth and decay phases is rather artificial: real applications may not fit this pattern at all.

Summary. Growth and decay phasing does control size better than exhaustive or selective exploration. However it is rather artificial, approximate, and does not apply very well to real applications.

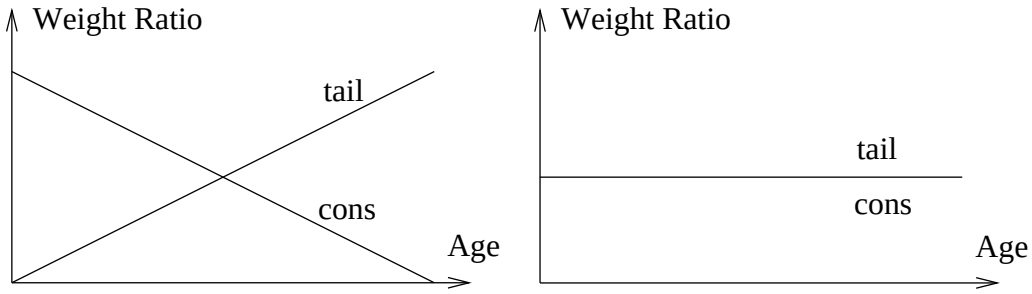


Figure 5.3: Two linear functions giving weight ratios for lists. Each produce an equal number of applications of *cons* and *tail*, but the left one produces applications on larger lists.

5.3.2 Linear Weights

The growth and decay method suffers from being rather artificial. Why *two* phases? Why split at a *particular* age? We can generalise away from these choices by approximating each mutator element of the weights ratio by a linear function over age. For example, consider making the *cons* component start high and decrease as age increases, whilst making the *tail* component start low and increase as age increases. Also consider making the *cons* and *tail* components equal and not vary over age. Each of these profiles will produce the same number of applications of *cons* as *tail* overall, assuming the number of nodes does not vary much over age, but the former will produce larger data structures. See Figure 5.3.

The profile of an application is amenable to this method too. By performing a linear regression (line of best fit) on the number of times a particular mutator is applied to a node against the age of that node, for each mutator, we will estimate the trend in the variance of mutator weights over age.

Unfortunately, this method has other disadvantages. What about a line of best fit that cuts the age axis? The portion of the line below the age axis indicates a negative weight ratio component. What does this mean? This method would need more formalisation and more examination.

Summary. The linear weights method looks promising, but needs further work.

Table 5.1: The effect of varying the likelihood of the next operation being the same as the last (odds of $n : 1$) for *cons* and *tail* on lists, whilst keeping the overall ratio of $cons : tail = 1 : 1$.

n	2^{-10}	2^{-9}	2^{-8}	2^{-7}	2^{-6}	2^{-5}	2^{-4}	2^{-3}	2^{-2}	2^{-1}	2^{0}
Avg. Size	1.9	3.4	4.2	5.7	11.8	21.4	37.5	81.8	122.5	142.4	190.6

2^{1}	2^{2}	2^{3}	2^{4}	2^{5}	2^{6}	2^{7}	2^{8}	2^{9}	2^{10}
279.2	424.9	530.8	523.1	540.9	657.7	778.4	844.7	1074.2	1221.5

5.3.3 Markov Chains

The ideas of growth and decay, and of linear weights, are both rather ad-hoc. A *Markov chain* [30] is a well-studied method for capturing patterns within sequences of states. The probability of what the next state in the sequence might be depends only on what the last state was. We use a Markov chain to parameterise the mutation weights ratio over the last operation used to create a node.

For example, instead of specifying $cons : tail = 1 : 1$ for a list profile, we might specify that

$$cons \to cons : cons \to tail = n : 1$$

$$tail \to tail : tail \to cons = n : 1$$

for some n. That is, the number of times a *cons* is followed by another *cons* is n times more than the number of times a *cons* is followed by a *tail*, etc. One can show that this ultimately yields an overall weights ratio of $cons : tail = 1 : 1$.

Varying n affects the average size of a list. The larger n is, the more likely a *cons* is followed by a *cons*, and hence the larger the list becomes. Generating DUGs with various values for n produces the results shown in Table 5.1.

We could replace the weights ratio by a list of weights ratios parameterised over the last operation, which we shall call the Markov weights ratios. However, the influence of size on the efficiency of a data structure is often separate from the influence of how often one operation is performed. Hence it would be useful to separate the Markov weights ratios into the overall weights ratio and other factors such as n in the example above. But how do we define these other factors in

general? Given that the Markov weights ratios are used to create the benchmarks, and are the result of profiling an application, we also need a way to convert between the Markov weights ratios, and the overall weights ratio with other factors like n, and back again.

We also need to decide whether to parameterise the weights ratio of mutators given the last operation was a generator. Without this we may lose some information, and perhaps even distort a profile, but with it we add more attributes, and we wish to keep the number of attributes down to a minimum.

A Markov chain is often represented by a transition matrix P. The probability of moving from state i to state j is given by the probability at row i, column j of P. The Markov weights ratios form the rows of P. If P is both *irreducible* and *recurrent* (see [30]), the average probability p_i of being in state i at any time is obtained by solving $\underline{p}P = \underline{p}$, where $\underline{p}$ is the row vector with p_i at column i. The vector $\underline{p}$ gives the overall weights ratio. However, in general, P may not be irreducible. This method would need more examination.

Summary. Using Markov chains is more theoretically sound than either growth and decay or linear weights, but it increases the number of attributes, which brings us back to the problems of exhaustive exploration. It also requires further work on translating between or unifying Markov weights ratios and ordinary weights ratios.

5.4 Inducing Decision Trees

Recall that we wish to derive, from a set of benchmarking trials, rules for determining the best data structure according to the datatype usage attributes. A common way to derive rules about a set of data is to *induce a decision tree* [44].

For our purposes, a decision tree is a binary tree with the following properties:

- Each branch node is labelled with a test of the form $A \leq v$, where A is a datatype usage attribute, and v is some constant.
- Each leaf node is labelled with the name of an ADT implementation.

An example of a decision tree is shown in Figure 5.4. To find the recommended

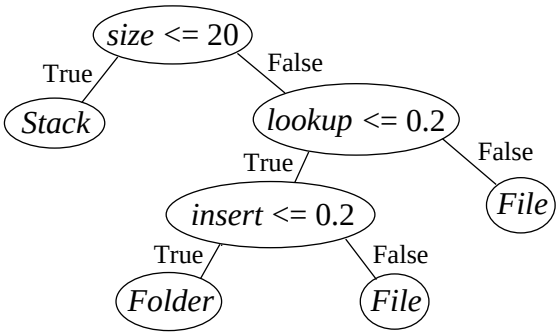


Figure 5.4: Decision tree for an (imaginary) ADT storing a collection of papers. Branch nodes are labelled with tests over datatype usage properties: *size*, *lookup*, and *insert*. Leaf nodes are labelled with ADT implementations: *Stack*, *Folder*, and *File*.

---

implementation for a particular datatype usage, start at the root and follow the appropriate branches till you reach a leaf. The implementation given by this leaf is the one recommended by this decision tree.

A decision tree is induced from a *training set* of the data it is to characterise. In our case, this training set is a sample of benchmarks. The sample is generated from a random selection of attribute values, but it is the attributes of the resulting benchmarks that are used, thereby including the attributes of both the profile *and* the shadow profile. Each benchmark in the sample is run, and the winning implementation is recorded. From these results, we induce a decision tree T. Given any benchmark B from the sample, using only the attributes of B, T will decide upon the winning implementation. Table 5.2 gives an example of results from which the decision tree of Figure 5.4 can be induced.

Given a sufficiently large and broad sample, the decision tree induced should be able to predict the winning implementation of any benchmark with good accuracy.

Summary. Inducing a decision tree solves all of the problems of exhaustive and selective exploration: size is captured in the shadow profile of the benchmark; the accuracy of benchmark generation has much less significance, since we use the actual profile rather than the desired profile; and every single benchmark is used

Datatype Usage Attributes			Best
insert	*lookup*	*size*	Implementation
0.3	0.5	10.0	*Stack*
0.1	0.1	40.0	*Folder*
0.4	0.1	45.0	*File*
0.3	0.1	36.0	*File*
0.3	0.3	30.0	*File*
0.1	0.4	42.0	*File*
0.1	0.5	33.0	*File*

Table 5.2: A training sample of results from which the decision tree of Figure 5.4 can be induced.

to influence the resulting decision tree, giving maximum use of the user's time. The only possible drawback concerns the accuracy of the resulting tree on unseen benchmarks. We choose to use this method, as it is by far the most promising one.

5.4.1 The Algorithm

We take an existing algorithm from the literature for constructing a decision tree from a sample. We use the algorithm C4.5 [46], which is a descendant of ID3 [44]. Both algorithms are widely known and respected in the machine learning community.

The basic idea underlying C4.5 is a simple divide and conquer algorithm due to Hunt [23]. Let S be the results of running a sample of benchmarks. Let $I_1, \ldots, I_k$ be the competing ADT implementations. There are two cases to consider:

- S contains only results reporting a single implementation I_j as the winner. The decision tree for S is a single leaf labelled with I_j.
- S contains results reporting a mixture of winners. By dividing S into S_1 and S_2 according to some test, we can recursively construct trees T_1 and T_2 from S_1 and S_2 respectively.

The key to a good implementation of Hunt's algorithm is the choice of test with which to split S.

The set of possible tests is limited by the range of attribute values for benchmarks in S. Let $[v_1, \ldots, v_n]$ be the distinct values, in order, of an attribute A for benchmarks in S. Consider two consecutive values, v_i and v_{i+1}. For any v satisfying $v_i \leq v < v_{i+1}$, splitting S with the test $A \leq v$ results in the same split. Therefore, there are at most $n-1$ distinct ways of splitting S using A. We consider only the tests $A \leq (v_i + v_{i+1})/2$.

For example, Table 5.2 gives a sample S which contains results reporting a mixture of winners. We could choose to split S with the test $size \leq 20$, as in the decision tree of Figure 5.4. Note that 20 is halfway between the next lowest and the next highest value of *size* in S. This test splits S into two samples, S_1 and S_2, from which we induce two decision trees T_1 and T_2 in the same manner. The sample S_1 contains just a single result reporting *Stack* as the winner. The decision tree for S_1 is a single leaf labelled with *Stack*. The sample S_2 contains results reporting a mixture of winners, and so we choose another test to split S_2, and so on.

But how do we choose which test to use at each stage? ID3 uses the *gain criterion* to measure the quality of a test, whereas C4.5 uses the *gain ratio criterion*. The latter is a modification of the former, so we shall describe both.

Gain Criterion

The gain criterion is based on the following principle of information theory: For a message that happens with probability p, the information conveyed by that message is $-\log_2 p$ bits. For example, the information conveyed by making any one of eight equally probable messages is $-\log_2(1/8)$ or 3 bits.

Suppose we choose a benchmark from a sample S and announce, correctly, that the winning implementation for that benchmark is I. The probability of this announcement is $|S^I|/|S|$, where S^I is the subset of S containing the benchmarks that give I as the winner. The information conveyed by that announcement is therefore $-\log_2(|S^I|/|S|)$ bits.

The expected value of a function f applied to a discrete random variable X

is

$$E(f(X)) = \sum f(x)P(X = x)$$

Let X be the winning implementation of a benchmark chosen from S. Let $f(I)$ be the information conveyed by an announcement of the value of I. For any implementation I

$$f(I) = -\log_2 \frac{|S^I|}{|S|} \quad \text{and} \quad P(X = I) = \frac{|S^I|}{|S|}$$

The expected information of an announcement of the winning implementation of a benchmark in S is therefore

$$info(S) = E(f(X)) = -\sum_{j=1}^{k} \left(\log_2 \frac{|S^{I_j}|}{|S|} \right) \cdot \frac{|S^{I_j}|}{|S|}$$

This expresses the average amount of information needed to identify the winner of a benchmark in S.

Suppose we split S into S_1 and S_2 using some test Z. Let $X = i$ if a benchmark chosen from S lies in S_i. Let $f(i)$ be the average amount of information needed to identify the winner of a benchmark in S_i. For $i = 1, 2$

$$f(i) = info(S_i) \quad \text{and} \quad P(X = i) = \frac{|S_i|}{|S|}$$

Therefore the expected information required to identify the winner of a benchmark in S split by Z into S_1 and S_2 is

$$info_Z(S) = E(f(i)) = \sum_{i=1}^{2} info(S_i) \cdot \frac{|S_i|}{|S|}$$

The difference between the expected information required before and after applying the test Z is therefore

$$gain(Z) = info(S) - info_Z(S)$$

Hence $gain(Z)$ measures the information gained by performing the test Z. The *gain criterion* chooses the test with the maximum gain.

For example, consider the sample S of Table 5.2, which contains one result reporting *Stack* as the winner, one result reporting *Folder* as the winner, and five results reporting *File* as the winner.

$$\begin{aligned} info(S) &= -\left(\tfrac{1}{7} \cdot \log_2 \tfrac{1}{7} + \tfrac{1}{7} \cdot \log_2 \tfrac{1}{7} + \tfrac{5}{7} \cdot \log_2 \tfrac{5}{7}\right) \\ &= 1.149 \text{ bits} \end{aligned}$$

The test Y, *lookup* ≤ 0.2, splits S into a sample S_1^Y containing one *Folder* result and two *File* results, and a sample S_2^Y containing one *Stack* result and three *File* results.

$$\begin{aligned} info_Y(S) &= \tfrac{3}{7} info(S_1^Y) + \tfrac{4}{7} info(S_2^Y) \\ &= -\tfrac{3}{7}\left(\tfrac{1}{3}\log_2\tfrac{1}{3} + \tfrac{2}{3}\log_2\tfrac{2}{3}\right) - \tfrac{4}{7}\left(\tfrac{1}{4}\log_2\tfrac{1}{4} + \tfrac{3}{4}\log_2\tfrac{3}{4}\right) \\ &= 0.857 \text{ bits} \\ gain(Y) &= info(S) - info_Y(S) \\ &= 0.292 \text{ bits} \end{aligned}$$

The test Z, *size* ≤ 20, splits S into a sample S_1^Z containing just one *Stack* result, and a sample S_2^Z containing one *Folder* result and five *File* results.

$$\begin{aligned} info_Z(S) &= \tfrac{1}{7} info(S_1^Z) + \tfrac{6}{7} info(S_2^Z) \\ &= -\tfrac{1}{7}\cdot\tfrac{1}{1}\log_2\tfrac{1}{1} - \tfrac{6}{7}\left(\tfrac{1}{6}\log_2\tfrac{1}{6} + \tfrac{5}{6}\log_2\tfrac{5}{6}\right) \\ &= 0.557 \text{ bits} \\ gain(Z) &= info(S) - info_Z(S) \\ &= 0.592 \text{ bits} \end{aligned}$$

Therefore, as the gain from using Z is larger than the gain from using Y, the gain criterion would prefer the test Z over the test Y.

Gain Ratio Criterion

The algorithm ID3 uses the gain criterion, giving quite good results. However, the gain criterion has a strong bias towards tests with many possible outcomes. The algorithm C4.5 attempts to remove this bias by modifying the gain criterion to produce the *gain ratio criterion*. Even though we only consider tests with two outcomes, Quinlan advises that the gain ratio criterion "even appears advantageous when all tests are binary" [46].

Consider the information content of an announcement of the result of a test Z applied to a benchmark in S. Let Z split S into the subsets $S_1, \ldots, S_n$. Let S_X be the subset into which Z places a benchmark chosen from S. Let $f(X)$ be the information conveyed by an announcement of the value of X. For $1 \leq j \leq n$

$$f(j) = -\log_2 \frac{|S_j|}{|S|} \quad \text{and} \quad P(X = j) = \frac{|S_j|}{|S|}$$

The expected information of such an announcement is therefore

$$splitInfo(Z) = E(f(X)) = -\sum_{j=1}^{n} \left(\log_2 \frac{|S_j|}{|S|} \right) \cdot \frac{|S_j|}{|S|}$$

This expresses the amount of information gained from dividing S into $S_1, \ldots, S_n$, irrespective of the winning implementations. Therefore the gain ratio defined by

$$gainRatio(Z) = \frac{gain(Z)}{splitInfo(Z)}$$

expresses what proportion of the information gained by splitting S using Z is relevant to the identification of a winning implementation. However, if the split is near-trivial—that is, some S_i is almost as large as S—the split information will be small, and the gain ratio unstable. Therefore, the *gain ratio criterion* chooses the test with the maximum gain ratio, subject to the constraint that the gain is large—at least as great as the average gain over all tests examined.

For example, consider again the sample S of Table 5.2. The test Y, $lookup \leq 0.2$, splits S into a sample containing three results and a sample containing four results.

$$\begin{aligned} splitInfo(Y) &= -\left(\tfrac{3}{7}\log_2\tfrac{3}{7} + \tfrac{4}{7}\log_2\tfrac{4}{7}\right) \\ &= 0.985 \text{ bits} \end{aligned}$$

From the previous section we know that

$$gain(Y) = 0.292 \text{ bits}$$

So

$$gainRatio(Y) = \frac{gain(Y)}{splitInfo(Y)} = 0.296$$

The test Z, $size \leq 20$, splits S into a sample containing one result and a sample containing six results.

$$\begin{aligned} splitInfo(Z) &= -\left(\tfrac{1}{7}\log_2\tfrac{1}{7} + \tfrac{6}{7}\log_2\tfrac{6}{7}\right) \\ &= 0.592 \text{ bits} \end{aligned}$$

From the previous section we know that

$$gain(Z) = 0.592 \text{ bits}$$

So

$$gainRatio(Z) = \frac{gain(Z)}{splitInfo(Z)} = 1$$

Therefore, assuming both tests have gain at least as large as the average gain over all tests examined, as the gain ratio from using Z is larger than the gain ratio from using Y, the gain ratio criterion would also prefer the test Z over the test Y.

5.4.2 Simplifying Decision Trees

The decision tree induced by the algorithm of Section 5.4.1 classifies the results of a sample *perfectly*. Unfortunately, this tree is not an ideal basis for choosing an implementation for the following reasons:

- The tree may be very large and complex.
- The tree is based on the chosen sample and may be over-specific.

Therefore we *prune* the induced tree to obtain a smaller and more accurate tree. There are several ways to prune a tree. We examine two, taken from existing literature, each based on the pruning scheme given in Figure 5.5. This scheme considers all subtrees bottom-up. If replacing a subtree with either one of its children or with a single leaf does not increase the *predicted error* of the subtree, it is pruned to this smaller tree. The two pruning techniques we consider differ in how they predict the error of a tree.

Reduced Error Pruning

Quinlan describes *reduced error pruning* in [45]. Two separate samples are required to perform reduced error pruning: a *training sample*, from which the original tree is induced; and a *test sample*, used to assess the accuracy of the induced tree.

Referring to the pruning scheme of Figure 5.5, we prune the induced tree using the test sample. The predicted error of a subtree on a subset of the test sample is simply the number of misclassifications made by the subtree when applied to the test sample subset.

If in addition to recording the winning implementation for a particular benchmark we also record the *ratio* of the time of *every* implementation to the time of the winning implementation, we may instead define the predicted error of a

To prune a tree T using the sample S:

if T is a branch node with children L and R, and labelled with test Z **then**

let Z split S into S_L and S_R

prune L using the sample S_L to give L_P

prune R using the sample S_R to give R_P

predict the errors of the following trees on the sample S:

a branch node with L_P and R_P as children, and labelled with test Z

L_P

R_P

every possible leaf

take the trees with the lowest predicted error

return the smallest such tree

else

T is a leaf, so return T untouched

Figure 5.5: Generic pruning scheme based on error prediction.

(a)

Datatype Usage Attributes			Best
insert	*lookup*	*size*	Implementation
0.3	0.5	10.0	*Stack*
0.1	0.1	40.0	*Folder*
0.4	0.1	45.0	*File*

(b)

Datatype Usage Attributes			Best
insert	*lookup*	*size*	Implementation
0.3	0.1	36.0	*File*
0.3	0.3	30.0	*File*
0.1	0.4	42.0	*File*
0.1	0.5	33.0	*File*

Table 5.3: To illustrate reduced error pruning, a split of the sample of Table 5.2 into (a) a training sample, and (b) a test sample.

subtree to be the average ratio of the implementation given by the subtree as the winner.

For example, consider the sample S of Table 5.2. We need two samples to perform reduced error pruning, so we split the sample into a training sample consisting of the first three results, and a test sample consisting of the remaining four results. Tables 5.3(a) and 5.3(b) give these samples. Figure 5.6 shows the tree we induce from the training sample, using either the gain criterion or the gain ratio criterion.

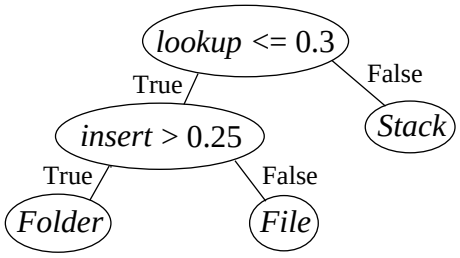


Figure 5.6: Decision tree induced from the training sample of Table 5.3(a).

To prune this tree, we first prune the left branch L, labelled with the test $insert > 0.25$. Following the pruning scheme of Figure 5.5, we then consider the predicted error of L and of each of the three possible leaves. Reduced error pruning calculates the predicted error of a replacement for a subtree by applying the replacement tree to the subset of the test sample covered by the original subtree. In the absence of any ratio information in the sample S, we use the number of misclassifications to measure the error of a tree in application.

The subset of the test sample covered by L contains the first and second results of Table 5.3(b), which are both *File* results. The subtree L misclassifies both of these results as *Folder*. The leaves *Folder* and *Stack* also misclassify both results. The leaf *File* however classifies both correctly. As this tree has the lowest predicted error, it replaces L.

Now we consider the predicted error of the original tree with L replaced by the leaf *File* (call this tree T'), and each of the three leaves, when applied to the whole test sample. The tree T' correctly classifies two results as *File*, but misclassifies the other two results as *Stack*. The leaves *Folder* and *Stack* misclassify every result of the test sample. The leaf *File* classifies every result in the test sample correctly, and so this replaces the original tree. Therefore, reduced error pruning simplifies the original tree to the leaf *File*.

Very Pessimistic Pruning

Quinlan describes *pessimistic pruning* in [45]. He also describes a "far more pessimistic" pruning technique in [46]. The latter technique we call *very pessimistic pruning*, in the absence of any name given by Quinlan. Whereas reduced error pruning predicts the error of a tree induced from a training sample by evaluating the tree on an additional test sample, very pessimistic pruning uses only a single training sample. This is useful when the data for a sample is scarce or expensive to collect.

Very pessimistic pruning estimates the error of a tree based on statistical reasoning that "should be taken with a large grain of salt" [46]. Consider the N cases classified by a leaf, E of which are classified incorrectly. We predict the error rate with confidence level CF to be $U_{CF}(E, N)$, the upper confidence limit

for the binomial distribution, defined for $X \sim B(N, p)$ by

$$U_{CF}(E, N) = p \Leftrightarrow P(X \leq E) = CF$$

See [46] for justification of this prediction.

We predict the number of errors produced by a leaf covering N cases to be $N \cdot U_{CF}(E, N)$. We predict the number of errors produced by a tree to be the sum of the errors produced by its children.

For example, consider the sample S of Table 5.2. Using either the gain criterion or the gain ratio criterion, we induce the tree of Figure 5.4. To apply very pessimistic pruning to this tree, we follow the pruning scheme of Figure 5.5 by first pruning the right branch R, labelled with the test *lookup* ≤ 0.2. To prune R, we must first prune its left child, R_L. However, using the default confidence level of 25%, very pessimistic pruning leaves R_L untouched. We shall not give details here, but instead we will give details of the more interesting case of pruning R.

The subtree R covers all bar the first result of the sample S, containing one *Folder* result, and five *File* results—call this subset S'. To prune R, we consider the predicted error of R, of R_L, and of each of the three possible leaves, when applied to the sample S'. The leaf *File* misclassifies one result out of six. Therefore, very pessimistic pruning predicts the error of this leaf as $6 \cdot U_{0.25}(1, 6) = 6 \cdot 0.389 = 2.337$. The leaf *Folder* misclassifies five results out of six, so the predicted error of this leaf is $6 \cdot U_{0.25}(5, 6) = 5.719$. The leaf *Stack* is even worse. The tree R_L misclassifies two out of three results on its left branch, and classifies correctly all three results on its right branch, so the predicted error of this tree is $3 \cdot U_{0.25}(1, 3) + 3 \cdot U_{0.25}(0, 3) = 3.131$. The tree R does not misclassify any of the results, and contains a leaf covering one result, a leaf covering two results, and a leaf covering three results, so the predicted error is $U_{0.25}(0, 1) + 2 \cdot U_{0.25}(0, 2) + 3 \cdot U_{0.25}(0, 3) = 2.860$. Therefore, very pessimistic pruning replaces R with the leaf *File*, as this has the lowest predicted error.

We next consider pruning the original tree with R replaced with the leaf *File*. We omit the details here, but very pessimistic pruning does not change this tree. Therefore, it is the final result of pruning.

5.5 Summary

In this chapter we have explored ways of using the DUG algorithms of Chapter 4. Exhaustive exploration is the most naïve solution, but takes too long to run. Selective exploration reduces this time, but does not capture the important attribute of size well enough. The growth and decay, linear weights, and Markov chains methods each capture size better, but introduce problems of their own. Finally, the induction of decision trees solves the problems of the previous methods, and looks promising. We evaluate the effectiveness of decision tree induction in Chapter 7. The implementation of decision tree induction is straightforward, and detailed in [46].

Chapter 6

Auburn: Benchmarking Tool

Chapter 4 gave algorithms for (a) creating a benchmark from a description of use, and (b) creating a description of use from an application. Chapter 5 illustrated how to use these algorithms to benchmark implementations of an ADT. This chapter describes the design decisions for a benchmarking kit called Auburn, built on the algorithms of Chapter 4, and the principles of Chapter 5. This chapter also details how to use Auburn.

Section 6.1 discusses the overall design of Auburn. Section 6.2 gives an overview of how the different parts of Auburn fit together. Sections 6.3–6.8 describe each part of Auburn, both the design decisions and the instructions for use by hand. Section 6.9 shows how Auburn can almost completely automate benchmarking.

Appendix C gives a reference for the Auburn executables.

6.1 Design Rationale

Auburn should provide the following functionality:

- DUG generation
- DUG evaluation
- DUG profiling
- DUG extraction

6.1.1 Dynamic Linking

Can we bundle each task above into one executable for all ADTs? Unfortunately not, for the following reasons. A DUG evaluator must link with the various implementations for the different ADTs Auburn encounters. Without some form of dynamic linking, which current Haskell implementations do not provide, we must re-compile a DUG evaluator for each new ADT and its implementations.

Can we bundle the remaining tasks into one executable for all ADTs? Unfortunately not, since DUG generation and DUG profiling must link with a user-defined shadow data structure, specific to an ADT. The process of DUG extraction, however, does not require any linking with code specific to an ADT, and can be compiled once for all ADTs.

Decision: Generate an executable *specific* to each ADT for the generation, evaluation, and profiling of DUGs, and define one executable for all ADTs for DUG extraction.

6.1.2 Overhead of DUG Evaluation

When an implementation of an ADT evaluates a DUG, there is some overhead: general bookkeeping, input, and output. The larger the overhead, the smaller the proportion of the whole time taken by the ADT operations, and hence the less accurate the estimation of the work done by them. Therefore we want to keep the overhead of DUG evaluation as small as possible.

We consider three alternative methods for DUG evaluation.

1. Generate the DUG, and translate each node directly into a Haskell call to an ADT operation. Output and compile the Haskell program. To evaluate the DUG, run the program. The only overhead of DUG evaluation comes from the mechanism for demanding the results of the observations.

2. Generate and evaluate the DUG within the same executable. The generation of the DUG forms most of the overhead of DUG evaluation.

3. Read the DUG from a previously generated file. Reading the file and general bookkeeping form most of the overhead of DUG evaluation.

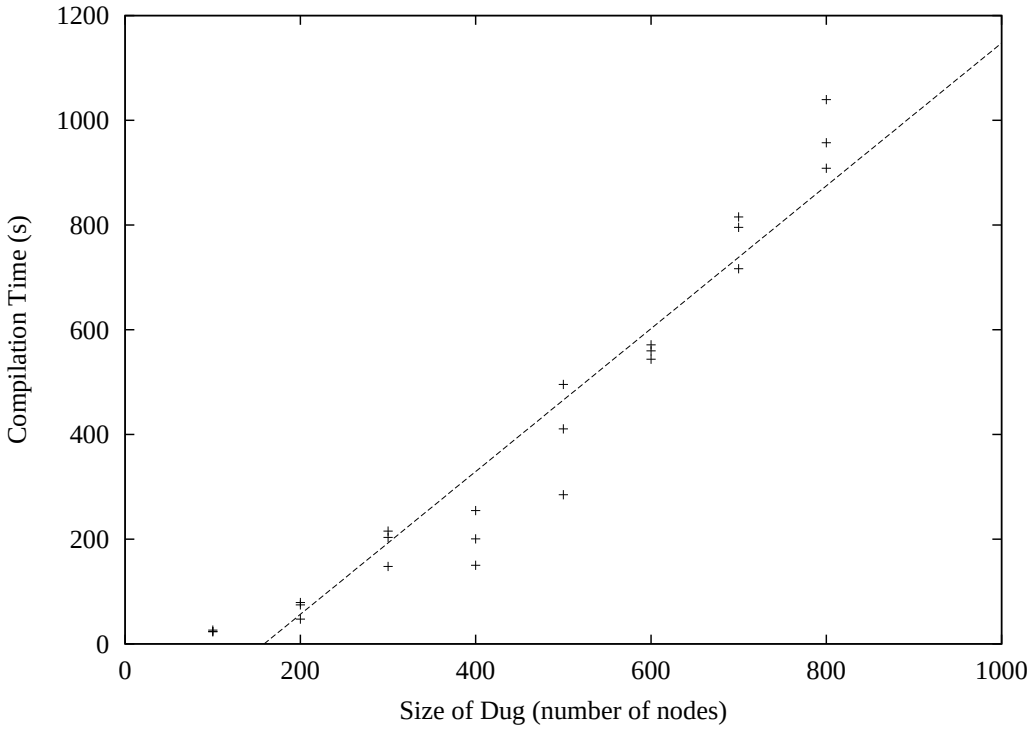


Figure 6.1: Times taken to compile different DUGs (as Haskell programs) of various sizes over three different ADTs.

In order to compare these three methods, we measure the overheads for a random selection of DUGs for three ADTs: queue, random-access sequence, and set with random retrieval (as in Section 4.4). We use the York nhc13 compiler [53] (release v0.9.4) running executables in a heap of 80Mb on an SGI Indy running IRIX 5.3.

Method 1 generates Haskell programs that take too much space and time to compile. Figure 6.1 shows the compilation times of DUGs of various sizes. The relationship is roughly linear. The largest DUG we can compile in a heap of 80Mb takes over 15 minutes to compile and has 800 nodes. This compares with taking about 1 second to generate the same DUG in a heap of 4Mb. The more DUGs we evaluate, the better conclusions we can form about the efficiency of the implementations. Since compiling a DUG is so slow, we reject Method 1.

For Method 2, the overhead of DUG evaluation is the cost of DUG generation. We cannot measure the cost of DUG generation directly, because we must process the DUG in some way in order for lazy evaluation to force its generation. Therefore, we make three different timings:

1. The time taken to generate and output the DUG as a binary file
2. The time taken to generate and output the DUG as a Haskell program
3. The time taken to generate and output the DUG as a binary file and also as a Haskell program

We can estimate the time taken to generate the DUG by adding Time 1 and Time 2 and then subtracting Time 3.

To estimate the cost of DUG evaluation, we read the DUG from a binary file, and evaluate the DUG with each implementation, including the *null implementation*. The null implementation performs very little work (see Section 6.7). By measuring the time taken to evaluate with the null implementation, we obtain an estimate of the overhead involved in evaluating with some real implementation. By subtracting this estimate of the overhead, we obtain an estimate of the actual cost of evaluating with some real implementation.

For some DUG D and some implementation I, let g be the time taken to generate D, let e_N be the time taken to evaluate D with the null implementation, and let e_I be the time taken to evaluate D with implementation I. For Method 2, the overhead of DUG evaluation is the time taken to generate the DUG divided by the total time to generate and evaluate the DUG, that is $g/(g + e_I - e_N)$. Note that we subtract the null implementation time, since this includes all of the overhead of reading the DUG from a file, which would not be done in Method 2. For Method 3, the overhead of DUG evaluation is e_N/e_I.

Additionally, for Method 3, we also time DUG evaluation using the C-Haskell hybrid described in Section 4.1.2.

Note that we are only estimating the overhead, as for instance, the overhead of the mechanism for extracting the results of the observations is present in both Method 2 and Method 3, but subtracting the null implementation time removes this overhead from our estimate of the overhead of Method 2. However, making a closer estimate is very hard, as lazy evaluation makes it very hard to separate tasks and measure them individually.

Table 6.1 gives the results. An overhead as large as 98% would make the benchmarking results rather inaccurate. Similarly, even 87% may be unacceptable. We therefore choose to evaluate DUGs by reading them from files, using the

ADT	Method 2	Method 3 (Haskell)	Method 3 (C/Haskell)
Queue	98.1	87.4	40.3
RASeq	96.5	82.4	25.6
Heap	99.0	90.3	29.0
Average	97.9	86.7	31.6

Table 6.1: Average percentage overhead for each method of DUG evaluation over every combination of 10 DUGs and 7 implementations for each of the three ADTs.

C-Haskell hybrid where possible (some compilers do not support the necessary language extension).

Decision: Separate DUG generation and DUG profiling from DUG evaluation. A DUG is generated, written to a file, and then read and evaluated.

6.1.3 Describing DUGs

Since we have decided to store DUGs in a compressed format in a file, we need another format for DUGs which the user can understand. Both a textual and a visual description serve this purpose well. Since we are compiling DUG generation and DUG profiling for each ADT, we decide to bundle these functions and the textual and visual description functions into one executable.

Decision: Generate a DUG *manager* for each ADT which performs the following tasks: DUG generation, DUG profiling, and DUG description (both textual and visual).

6.1.4 Re-compilation

DUG Manager

Generating and compiling a DUG manager for each ADT re-compiles a lot of similar functions. These common functions should be compiled just once, so we place them in a library.

Decision: Create a library of the functions common to every DUG manager. A DUG manager contains only the definitions of functions specific to the ADT—it imports the rest from the library.

DUG Evaluator

We could generate one DUG evaluator for all implementations of an ADT, using either Haskell's class system, or generate one copy of the DUG evaluating function for each implementation. However, this would have to be re-compiled if a new implementation were introduced, or if an implementation was changed. It is simpler to generate one DUG evaluator for each implementation.

Decision: Generate a DUG evaluator specific to each implementation of each ADT.

6.2 Overview of Auburn

Auburn uses a *signature* (Section 6.3) to identify an ADT. From the signature of an ADT, Auburn can provide a DUG *manager* (Section 6.4) specific to that ADT. The DUG manager can generate a DUG from a profile (Section 6.4.1), calculate a profile from a DUG (Section 6.4.2), and create a visual or textual description of a DUG (Section 6.4.3). In order to generate a DUG from a profile, the DUG manager requires a *shadow data structure* (Section 6.5) for the same ADT. From a signature, Auburn can provide a *trivial* shadow data structure (Section 6.5.1), or *guess* at a *size-based* shadow data structure (Section 6.5.2).

From the signature of an ADT and the name of an implementation of the ADT, Auburn can also provide a DUG *evaluator* (Section 6.6) specific to the ADT and the implementation. From the same signature, Auburn can provide a *null implementation* of the ADT (Section 6.7), performing as little work as possible. This is useful for estimating the overhead of DUG evaluation.

From the signature of an ADT, the name of an implementation of the ADT and the name of an application using that implementation, Auburn can provide a DUG *extracting* version of the application (Section 6.8). The application works

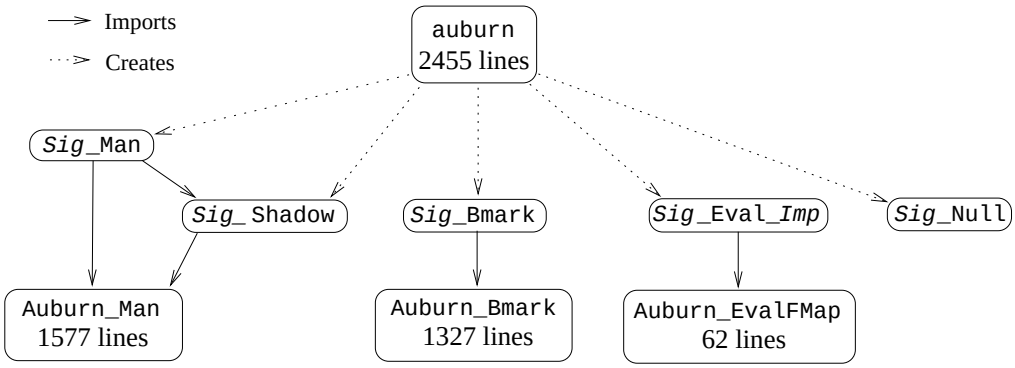


Figure 6.2: Structure of Auburn.

---

exactly as before, but also produces a DUG of how it uses the implementation of the ADT.

Auburn also provides automation tools (Section 6.9) for generating and using all of the above, saving a lot of user effort.

The Auburn package contains a main executable `auburn` (Sections 6.3–6.8) and other executables to automate the use of `auburn` (Section 6.9). Figure 6.2 shows the components of Auburn, and how they relate to each other. The figure gives the size of any component that is not generated; the size is the number of lines of Haskell.

Appendix C gives the help information provided with each Auburn executable.

6.3 ADT Signature

The whole process of benchmarking described in Chapters 3–5 is based on comparing different implementations of the same ADT. The definition and implementation of DUGs in Chapters 3 and 4 refers primarily to the ADT, and secondly to the implementations. Therefore, Auburn needs a description of the ADT to work with.

An ADT is identified by giving its *signature*. An ADT signature looks just like an implementation but contains no code—just an export declaration, and one type signature for each exported operation. Figure 6.3 gives an example of a signature. The ADT must be simple, as given by Definition 3.3.

```
module List (List,empty,catenate,cons,tail,head,lookup,isEmpty)
   where
empty    :: List a
catenate :: List a -> List a -> List a
cons     :: a -> List a -> List a
tail     :: List a -> List a
head     :: List a -> a
lookup   :: List a -> Int -> a
isEmpty  :: List a -> Bool
```

Figure 6.3: Haskell code giving the signature of a simple list ADT providing normal list operations, catenation and indexing.

Auburn can generate a signature of the simple operations common to any set of implementations with:

`auburn -c` {*Implementation Files*} {*Signature File*}

For example,

```
auburn -c NaiveList AVLList List
```

creates a signature file `List.sig` from the simple operations common to the implementations stored in the files `NaiveList.hs` and `AVLList.hs`. Operations that are not simple, or not exported by every implementation, are not included.

If an implementation exports every operation in a signature, but also exports an operation that is not included in the signature, the implementation can still evaluate a DUG made for that signature, though of course only the operations included in the signature will be used. An application importing the implementation may have its DUG extracted, so long as the application only imports operations found in the signature.

The signature file of an ADT is used by Auburn to perform every task specific to that ADT: DUG generation, DUG evaluation, DUG profiling, DUG extraction, and DUG description.

6.4 DUG Manager

A DUG *manager* processes DUGs; generating, profiling, and describing them. Auburn can generate a DUG manager specific to an ADT from the signature of the ADT:

```
auburn -m {Signature File}
```

For example,

```
auburn -m List
```

makes a DUG manager `List_Man.hs` from the signature file `List.sig`.

As discussed in Section 6.1.4, the generated file contains all of the code relevant to the ADT. The remaining code is imported from a library. The generation of a DUG manager is straightforward. The DUG manager may be compiled (linking with a shadow data structure, see Section 6.5) to produce an executable.

6.4.1 DUG Generating

The DUG manager can generate a DUG from a profile with:

```
Sig_Man -g {Profile} {Seed} -o {DUG File}
```

where the seed is used for pseudo-random number generation. The DUG is written to a file; using the flag `-oP` pipes the DUG to standard output. The profile is given using a Haskell data structure as follows:

```
Profile {Gen. Wgt. Ratio} {Phases}
```

where {*Phases*} is a list of phased profiles, starting from phase 1 in order, each given using the following Haskell data structure:

```
Phase {Mut-Obs. Wgt. Ratio} {Mortality} {PMF} {POF}
```

Each weight ratio is a list of numbers. For example, `[1,2,3]` represents the ratio 1 : 2 : 3. The order of the operations within the ratios is primarily by role (generator, mutator, and observer) and then alphabetically. Invoking help with the `-h` flag gives this order.

For example, using a DUG manager generated from the signature of Figure 6.3,

```
List_Man -g "Profile [1] [Phase [2,1.5,1,2.5,3,1] 0 0.2 0.3]" 123
         -o example.dug
```

generates a DUG in the file `example.dug`, using a single-phased profile: the generation weight ratio is redundant as there is only one generator; the mutation-observation weight ratio is

$$\texttt{catenate} : \texttt{cons} : \texttt{tail} : \texttt{head} : \texttt{lookup} : \texttt{isEmpty} = 2 : 1.5 : 1 : 2.5 : 3 : 1;$$

the mortality is 0; the PMF is 0.2; and the POF is 0.3. The DUG generator is given a seed `123` for pseudo-random number generation.

Other flags modify the behaviour of DUG generation:

`-a` {*Phase Argument*}

See Section 3.4.3. The default is no phase argument.

`-b` {*Pool Size*}

See Section 4.1.1, *Choosing non-version arguments from the graph.* The default is 10.

`-fL` {*Minimum Frontier Size*}

See Section 4.1.1, *The* DUG *Generation Algorithm.* The default is 1.

`-fU` {*Maximum Frontier Size*}

See Section 4.1.1, *The* DUG *Generation Algorithm.* The default is 10.

`-n` {*Number of Nodes*}

The number of nodes in the generated DUG. The default is 10000.

Sections 4.1.1 and 4.3.1 detail the implementation of DUG generation.

6.4.2 DUG Profiling

The DUG manager can calculate a profile of a DUG with:

Sig`_Man -p` {*Profile File*} {DUG *File*}

The profile may be piped to standard output using the `-pP` flag. The profile is written in the form given in Section 6.4.1, along with the shadow profile (Section 3.4.4), the maximum frontier size, and the mean frontier size. The initial frontier size is always zero.

For example,

```
List_Man -p example.profile example.dug
```

places the profile of `example.dug` in the file `example.profile`.

As with DUG generating, a phase argument can be given using the `-a` flag.

6.4.3 DUG Describing

As DUG files are compressed binary files—to reduce input and output overhead in DUG evaluation—Auburn also provides visual and textual descriptions of a DUG. The visual description of a DUG is suitable for the *GraphViz* package of AT&T [17], and produced by:

Sig`_Man -d {`*Graph File*`} {DUG` *File*`}`

The textual description of a DUG is very simple, and produced by:

Sig`_Man -t {`*Text File*`} {DUG` *File*`}`

As with DUG generation and DUG profiling, output can be piped to standard output using similar flags: `-dP` and `-tP` for visual and textual descriptions respectively.

For example, the DUG of Figure 3.4 can be converted to a file viewable through GraphViz (see Figure 6.4) or converted to a text file (see Figure 6.5). Note that the textual description resembles Haskell code. Indeed, adding the `-H` flag makes the textual description a Haskell program that evaluates the DUG—see Figure 6.6.

6.5 Shadow Data Structure

A shadow data structure aids the generation of DUGs, and adds information to profiles—see Section 3.4. The shadow data structure must export the following:

- The type of a shadow
- The shadow operations
- The shadow of an unevaluated version argument (see Section 4.3.4)

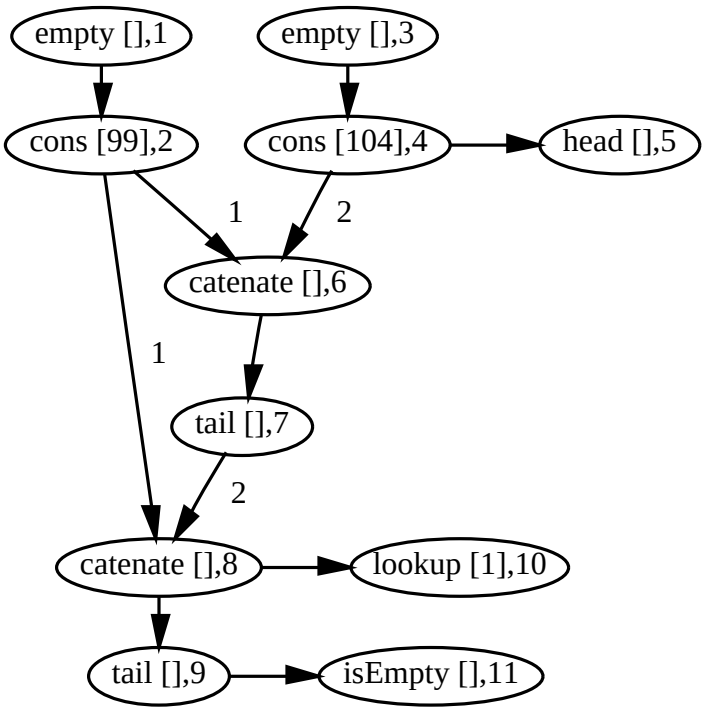


Figure 6.4: Output from the GraphViz package viewing the DUG of Figure 3.4 (the orientation, the spacing and the font size were altered so the output could fit on this page). The functions η, σ, and τ—see Definition 3.6—are indicated on the graph. Each node is labelled with the partial application given by η (the name of an operation and a list of non-version arguments), and the node's position in the order of evaluation, given by σ. The arc labels given by τ are placed next to the relevant arcs.

```
n1 = empty
n2 = cons 99 n1
n3 = empty
n4 = cons 104 n3
n5 = head n4
n6 = catenate n2 n4
n7 = tail n6
n8 = catenate n2 n7
n9 = tail n8
n10 = lookup n8 1
n11 = isEmpty n9
```

Figure 6.5: Textual description of the DUG of Figure 3.4. Each line describes the birth of a node.

- The guards
- The type of a shadow profile
- The shadow profile functions
- The type of a phase argument
- The phase functions

Given only the signature of an ADT, it is impossible to generate a suitable shadow data structure for an ADT in general. However, Auburn can generate a trivial shadow data structure, or guess at one based on size.

6.5.1 Trivial Shadow Data Structure

A trivial shadow data structure stores no information in the shadow, allows every operation application, gives an empty shadow profile, and puts every version in

```
import List

os0 = []
n1 :: List Int
n1 = empty
n2 :: List Int
n2 = cons 99 n1
n3 :: List Int
n3 = empty
n4 :: List Int
n4 = cons 104 n3
n5 = head n4
os1 = fromEnum n5 : os0
n6 :: List Int
n6 = catenate n2 n4
n7 :: List Int
n7 = tail n6
n8 :: List Int
n8 = catenate n2 n7
n9 :: List Int
n9 = tail n8
n10 = lookup n8 1
os2 = fromEnum n10 : os1
n11 = isEmpty n9
os3 = fromEnum n11 : os2

main = print (sum (reverse os3))
```

Figure 6.6: Textual description of the DUG of Figure 3.4 as a Haskell program. Running this program evaluates the DUG it describes.

Phase 1. It is useful for providing a base on which to build a non-trivial shadow data structure. For example, the types of every function required are present.

Auburn builds a trivial shadow data structure from a signature file with:

```
auburn -sT {Signature File}
```

For example,

```
auburn -sT List
```

generates a trivial shadow data structure in the file `List_Shadow.hs`.

The generation of a trivial shadow data structure is quite straightforward, so we do not give any implementation details here.

6.5.2 Size-Based Shadow Data Structure

A size-based shadow data structure stores the size of a version in its shadow. This size is then used: (1) to guard against undefined applications; (2) to phase versions into those no larger than a given size, and those larger; and (3) to calculate the average and standard deviation of the size of every version across all mutations and observations. Example 3.27 is an instance of such a shadow data structure.

Auburn can only guess at a size-based shadow data structure, using the types of the ADT operations, as given by the signature. For most of the common ADTs Auburn guesses correctly: queues, lists, random-access sequences, catenable sequences, and heaps. However, some ADTs require a more sophisticated shadow data structure, for example, sets and finite maps (the size of a set varies according to which element is added or removed, and this is not captured by the type of an operation).

Auburn sets the size of an unevaluated version argument to 0, on the basis that none of the elements of an unevaluated version are examined.

Auburn guesses at a size-based shadow data structure by using the signature file with:

```
auburn -sS {Signature File}
```

For example, using the signature `List.sig` of Figure 6.3,

```
auburn -sS List
```

guesses (correctly) at a size-based shadow data structure for lists, and places it in the file `List_Shadow.hs`.

Guessing Size-Based Shadow Data Structures

The method for guessing the definitions of the shadow operations and the guards of a size-based shadow data structure is tailored for the simple ADTs that can be shadowed by size. The phasing and the shadow profiling remain constant for every ADT—for further details of these, see Example 3.27.

Consider the following simple ADTs: sequences (with or without access to front or rear, random access, and catenation), heaps, sets, finite maps (with fixed key type to make the ADT simple), and bags. Of these, sets, finite maps, and bags cannot be shadowed by size. Of the rest, all have their size-based shadow data structure guessed correctly by Auburn[1]. Table 6.2 shows the desired definitions of the shadow operations of all these ADTs. Table 6.3 condenses these definitions into rules for Auburn to use. Table 6.4 shows the desired definitions of guards of the same ADTs. Table 6.5 condenses these definitions into rules for Auburn to use.

For example, for the signature of Figure 6.3, Auburn defines the type of shadows with

```
data Shadow = Shadow {size :: Int}
```

and defines the shadow of the `cons` operation as

```
cons_Shadow :: Int -> Shadow -> Shadow
cons_Shadow i0 (Shadow {size=s0}) = Shadow {size=s0+1}
```

and defines the guard of the `head` operation using

```
head_Guard :: Shadow -> Bool
head_Guard (Shadow {size=s0}) = s0>0
```

[1]It may be possible to form an ADT signature that models a sequence or a heap in a way that makes Auburn guess incorrectly, but that is not the case for the ADT signatures given in this thesis.

and defines the guard of the `cons` operation using

```
cons_Guard :: Shadow -> [IntSubset]
cons_Guard (Shadow {size=s0}) = [Pool]
```

Note that Auburn uses `Pool` instead of `All` to select an argument where there are no restrictions. This enables the user to control `Pool` arguments with the pool size (see Section 4.1.1).

6.6 DUG Evaluator

Auburn can generate a DUG *evaluator* specific to an ADT and an implementation of that ADT.

`auburn -e` {*Implementation Name*} {*Signature File*}

For example,

```
auburn -e NaiveList List
```

produces a DUG evaluator in the file `List_Eval_NaiveList.hs` importing the module `NaiveList` which should implement the ADT whose signature is given in `List.sig`.

The DUG evaluator takes two arguments: the name of the DUG file to evaluate, and the number of internal repetitions of this evaluation (useful for increasing the time of evaluation to a measurable size).

Sig`_Eval_`*Implementation* {DUG *file*} {*No. of Repetitions*}

For example,

```
List_Eval_NaiveList example.dug 10
```

evaluates the DUG `example.dug` 10 times using the implementation `NaiveList`.

As Section 4.3.2 and Section 6.1 mention, the overhead of a DUG evaluator implemented entirely in Haskell can sometimes be unacceptable. Moving the algorithm into C and calling the Haskell ADT operations from within C reduces this overhead significantly. The C routines are interfaced to Haskell using Green

Operation	No. of Arguments			Result of Shadow Operation
	$T\ a$	a	Int	
empty	0	0	0	0
singleton	0	0	1	1
tail/init/deleteMin	1	0	0	$s_0 - 1$
update	1	1	1	s_0
#/merge	2	0	0	$s_0 + s_1$
cons/snoc/insert	1	0	1	$s_0 + 1$

Table 6.2: Shadow operations of simple ADTs that can be shadowed by size. A shadow operation takes shadow arguments s_0, s_1, ..., s_k.

No. of Arguments			Condition	Result of Shadow Operation
$T\ a$	a	Int		
0	m	n		n
1	m	0		$s_0 - 1$
l	m	n	$n = 0 \vee m > 0$	$s_0 + \ldots + s_{l-1}$
l	m	n	$n > 0 \wedge m = 0$	$s_0 + \ldots + s_{l-1} + n$

Table 6.3: Rules for guessing the result of a size-based shadow operation. A shadow operation takes shadow arguments s_0, s_1, ..., s_k.

Operation	No. of Arguments			Type of Result	Result of Guard
	$T\ a$	a	*Int*		
tail/init/deleteMin	1	0	0	$T\ a$	$s_0 > 0$
head/findMin	1	0	0	a	$s_0 > 0$
size	1	0	0	*Int*	*True*
isEmpty	1	0	0	*Bool*	*True*
empty	0	0	0	$T\ a$	*True*
#/merge	2	0	0	$T\ a$	*True*
singleton	0	0	1	$T\ a$	$[Pool]$
cons/snoc/insert	1	0	1	$T\ a$	$[Pool]$
lookup	1	1	0	a	$[0..s_0 - 1]$
update	1	1	1	$T\ a$	$[Pool, 0..s_0 - 1]$

Table 6.4: Guards for simple ADTs that can be shadowed by size. A guard takes arguments s_0, s_1, ..., s_k.

No. of Arguments			Type of Result	Result of Guard
$T\ a$	a	*Int*		
1	0	0	$T\ a$ or a	$s_0 > 0$
l	0	0	Any	*True*
l	m	n	Any	Replace a with *Pool* and *Int* with $0..s_0 - 1$

Table 6.5: Rules for guessing the result of a guard using size-based shadows. A guard takes arguments s_0, s_1, ..., s_k.

Card [43], extended to allow C to call Haskell (included with the nhc13 compiler [53]). Supplying the flag `-G` informs Auburn to use Green Card in creating the DUG evaluator. For example,

```
auburn -G -e NaiveList List
```

produces a DUG evaluator in the file `List_Eval_NaiveList.gc` importing the Haskell module `NaiveList`, and the C library `Auburn_evaldug.c`.

Sections 4.1.2 and 4.3.2 detail the implementation of a DUG evaluator.

6.7 Null Implementation

Auburn can generate a *null implementation* of an ADT.

```
auburn -n {Signature File}
```

A null implementation performs very little work but provides operations of the correct type. Evaluating a DUG with the null implementation gives an estimate of the overhead of DUG evaluation, allowing a better estimate of the actual work done by the operations of other implementations.

For example,

```
auburn -n List
```

produces a null implementation in the file `List_Null.hs` of the ADT whose signature is in the file `List.sig`.

A null implementation defines the exported type constructor as a nullary data constructor `Null`. For example, for the type constructor `List` of Figure 6.3, the null implementation defines

```
data List a = Null
```

Each operation ignores its arguments but returns some value of the correct type. But what value do we return of type a? We avoid this problem by noting that as we only use the null implementation to evaluate DUGs with the type variable a instantiated to *Int*, we define operations over versions of type $T\ Int$.

For `List Int` the null implementation returns `Null`, for `Int` it returns `7`, and for `Bool` it returns `True`[2]. For example, the `lookup` operation is implemented by

```
lookup :: List Int → Int → Int
lookup _ _ = 7
```

As the bookkeeping in DUG evaluation is strict (see Section 4.3.2), evaluating a DUG with this very lazy implementation will force all of the bookkeeping without performing much more work, giving a good estimate of the work done by the bookkeeping.

6.8 DUG Extraction

Auburn can transform an application that imports an implementation of an ADT into a similar application that performs the same work whilst also producing a DUG of the way it uses the ADT implementation.

`auburn -x` {*Implementation File*} {*Main File*} {*Signature File*}

Auburn wraps the implementation module and the main module to produce the DUG as a side-effect (see Sections 4.2.1 and 4.3.3).

But how do we implement this? The application may consist of many modules, some of which will import the ADT. We do not want to change every such module, so we must keep the same module name for the wrapped implementation. As Haskell compilers use the convention that a module appears in a file of the same name, we must replace the existing implementation module with the new wrapped module. Instead of trying to insert the new definitions into the old implementation module, we rename the old implementation module, place it in a different file, and import it. The import is qualified to avoid name clashes.

Similarly, instead of trying to insert the new definition of `main` into the main file, we import the old definition into a new main file. In order to import the old main module into the wrapped main module, we must rename the main

[2]Returning `0` for `Int` may invoke an optimisation in the compiler, reducing the bookkeeping work for the null implementation. However, we wish to use the bookkeeping of the null implementation as an estimate of the bookkeeping of other implementations. Similarly, we do not return `False` for `Bool`, since `fromEnum` evaluates this to `0`.

module from `Main`. As Haskell also has the convention that the `Main` module may be implicitly defined in a file of any name, we may need to add a module declaration and an export declaration, exporting the old `main` function. The new main module imports the old main module, qualified to avoid name clashes.

The wrapped modules files use Green Card. They import C functions from an Auburn library `Auburn_extractdug.c`. Auburn creates backups of the renamed files to prevent accidental loss and to aid recovery. Auburn can revert the implementation and main files to the original versions with:

`auburn -u` {*Implementation File*} {*Main File*} {*Signature File*}

For example,

```
auburn -x NaiveList mean List
```

moves `NaiveList.hs` to `Old_NaiveList.hs`, and `mean.hs` to `Old_mean.hs`. The module `NaiveList` is an implementation of an ADT whose signature is in `List.sig`. The main module in `mean.hs` defines an application that imports this implementation. Auburn also creates the files `NaiveList.gc` and `mean.gc` and creates backups of the old files at `auburn-backup.NaiveList.hs` and `auburn-backup.mean.hs`. The new main file `mean.gc` defines an implementation that imports the new implementation `NaiveList.gc`. These compile and link with the C file `Auburn_extractdug.c`, the files `Old_NaiveList.hs` and `Old_mean.hs`, and with any other files the old main file imported, to produce a DUG-extracting executable `mean`. This runs as before, but also produces a DUG in the file `app.dug`. Also,

```
auburn -u NaiveList mean List
```

removes the files `NaiveList.gc` and `mean.gc`, and restores the files `NaiveList.hs` and `mean.hs` from their backups.

6.9 Automation

Auburn provides tools to automate most of the work involved in a benchmarking experiment.

1. Making the executables: `auburnExp`
2. Making DUGs: `makeDugs`
3. Timing the evaluations of DUGs: `evalDugs`
4. Gathering the times of evaluations: `processTimes`
5. Cleaning up after Tools 2, 3, and 4: `cleanDugs`
6. Tracing bugs in ADT implementations
7. Gathering benchmarking results

Tools 2 through 5 are used by Tools 6 and 7. The implementation of Tools 2 through 5 is straightforward. The user will probably not need to use them directly, but instead use Tools 1, 6, and 7. For further details of Tools 2 through 5, see Appendix C.

Tool 1, `auburnExp`, is quite simple. It creates a *makefile*, for use with the GNU make utility [16]. This automates the building, compiling, and linking of all the executables needed by the other tools.

Tools 6 and 7 are implemented within the same executable, which we shall now describe in detail.

6.9.1 Benchmarker

Auburn can generate a *benchmarker* specific to an ADT and some of its implementations with the following:

`auburn -b` {*Implementation Modules*} {*Signature File*}

For example,

```
auburn -b NaiveList AVLList List
```

creates a benchmarker in the file `List_Bmark.hs`.

The benchmarker serves two purposes: (1) tracing bugs in ADT implementations, and (2) gathering benchmarking results.

Tracing Bugs

A benchmarker can search for the smallest DUG that causes an error when evaluated by the ADT implementations. A DUG causes an error when any implementation fails to evaluate the DUG—for example, because of a run-time error—or if any two implementations return different checksums.

```
Sig_Bmark -q {Seed}
```

The benchmarker uses the seed to direct the random search. The benchmarker generates a series of DUGs. If a DUG causes an error, the benchmarker reports the error and the DUG, and then generates a smaller DUG. It is possible that the benchmarker generates a DUG that is smaller than the smallest DUG that causes an error. Therefore, if a DUG does not cause an error, the benchmarker then generates a larger DUG.

The benchmarker displays a DUG as a Haskell program using the DUG manager with the flags `-t` and `-H`—see Section 6.4.3. This program does not require a DUG file to read, as the DUG is contained within the program. Hence it may be copied into a file, and compiled on its own, perhaps with the tracing facility of the compiler turned on.

If a DUG causes an error, and it is the smallest such DUG found so far, this fact is also reported. This allows the user to let the benchmarker run for as long as they like, scan the output for the last report of a smallest DUG, and hence find the smallest erroneous DUG found overall. A neater solution using some form of interrupt signal handling would be preferable, but Haskell does not support exception handling.

See Section 7.1.2 for an example of using a benchmarker to find bugs.

Gathering Benchmarking Results

A benchmarker can compute, gather, and analyse benchmarking results; that is, it can measure how well different ADT implementations perform across different datatype usages. Specifically, the benchmarker provides the following functionality:

- Generate a set of DUGs with randomly chosen profiles, measure the performance of each ADT implementation evaluating each DUG, and record the results as a *sample*.

  {*Sig*}_Bmark -g {*Seed*} -o {*Sample File*}

  The seed is used to direct the choice of profiles and the generation of DUGs.

- Induce a *decision tree* from a sample, perhaps using one of two pruning techniques.

  {*Sig*}_Bmark -s {*Sample File*} -i [-r | -P] -w {*Tree File*}

  The flag `-r` requests reduced-error pruning whereas the flag `-P` requests very pessimistic pruning—see Section 5.4.2.

- Report the accuracy of a given decision tree on a given sample.

  {*Sig*}_Bmark -s {*Sample File*} -t {*Tree File*} -c {*Report File*}

- Use a decision tree to decide which implementation suits a given profile.

  {*Sig*}_Bmark -t {*Tree File*} -d {*Profile File*}

These flags may be combined. An accuracy report may be written to standard output using the flag `-cP`. Similarly modified flags (using the postfix `P`) exist for reading or writing a tree or a sample from standard input or to standard output.

Random Sampling

Tracing bugs and gathering benchmarking results both require the benchmarker to create a DUG from a *randomly chosen* profile. Each profile attribute is chosen fairly from a list of about 20, with the list varying according to the attribute. Every weight ratio component is chosen from $[0, 0.05, \ldots, 1]$. The mortality is chosen from $[0, 10^{-4}, 10^{-3.75}, \ldots, 10^{-1}]$. The PMF is chosen from $[0, 2^{-20}, 2^{-19}, \ldots, 2^{-2}]$. The POF is chosen from $[0, 0.05, \ldots, 1]$. These lists may be changed by the user.

Auburn uses these lists by default to attempt a fair distribution of benchmarks over the datatype usage space. The quality of "fairness" must reflect the "typical"

application and therefore any such attempt is primarily guided by experience. On this basis, we shall now attempt to justify our choice of lists.

Using a uniform distribution for each weight ratio component treats each operation equally and in particular allows for no use of an operation. Making a zero weight even more likely than 1 in 20 may be justified on the grounds that applications often neglect an operation completely. The mortality and PMF attributes should be very low. For example, for a list ADT, a mortality of 0.5 implies that, on average, a list is mutated only once before being discarded. Similarly, a PMF of 0.5 implies that, on average, an empty list gives rise to over 1000 different lists after just 20 successive mutations. Given the need to keep these attributes generally low, with the occasional high value, it is natural to use an exponential scale. The POF however may take any value between 0 and 1 and so is given a uniform distribution.

The benchmarker excludes any impossible or unsuitable profiles. For example, a profile where the mutation weights are all 0 without mortality being 1 is impossible, and a profile where the observation weights are all 0 is possible but undesirable as it forces no work. Two other types of unsuitable profiles are excluded by default, both relating to operations that increase or decrease size. A profile with a greater sum of size-decreasing operation weights than size-increasing operation weights is often impossible without persistent mutation, and highly undesirable otherwise. A profile with all size-increasing operation weights 0 is also highly undesirable. The benchmarker excludes both of these types of profile by default. The user may add or remove other such exclusions of profiles.

Note that, as with generating a shadow data structure, it is impossible to tell the effect of an operation on size just from its type. Therefore, when generating a benchmarker, Auburn guesses which operations increase size and which decrease size, in the same manner as it does for generating a size-based shadow data structure (see Section 6.5.2).

6.10 Summary

Auburn can generate a benchmark from a description of use and a extract a description of use from an application, as motivated in Section 1.3. Moreover,

Auburn can automate calls to these functions to find small benchmarks revealing bugs in implementations and also to produce a summary, in the form of a decision tree, of which implementation is best according to the datatype usage.

Chapter 7 gives examples of using Auburn in this way, and evaluates Auburn's performance and accuracy.

Chapter 7

Results

Chapter 6 presented a benchmarking tool, Auburn, built on Chapters 3–5. This chapter uses Auburn and evaluates its accuracy at predicting the best data structure.

Section 7.1 uses Auburn on the data structures reviewed in Chapter 2 to produce a summary of which data structure is best when. Section 7.2 uses several real applications as benchmarks to test the advice produced by Auburn in Section 7.1. Section 7.3 examines the possible sources of inaccuracy in Auburn.

Technical Note. All benchmarks in this chapter, whether real or generated by Auburn, are compiled using the York nhc13 compiler [53] (release v0.9.4), and run in a heap of 80Mb, on an SGI Indy running IRIX 5.3. As with the remainder of this thesis, we use Auburn version 2.3. All benchmarks are run, repeating internally if necessary, till the total time is at least 1 second. Each benchmark is timed just once, to an accuracy of 0.01 seconds, given as the "user time" by the standard UNIX command `time`.

7.1 Benchmarking Three ADTs

In Chapter 2, we reviewed several implementations of three ADTs: queues, random-access sequences[1], and heaps. We shall now use Auburn to benchmark these implementations. There are five stages in our experiment:

[1] As some implementations of the random-access sequence ADT do not support the operations *snoc*, *last* and *init*, we remove these operations from the ADT for the purpose of benchmarking.

- Set up the Auburn executables.
- Check the correctness of the implementations.
- Fine-tune the implementations.
- Run and time the implementations.
- Induce decision trees from the times.

7.1.1 Setting Up

For each of the three ADTs, setting up the Auburn executables is straightforward.

- We make a directory for the ADT, say `Queue`. Into this directory, we place each implementation of the ADT. Auburn creates a makefile in this directory with

  ```
  auburnExp
  ```

- We make the Auburn executables with

  ```
  make SIG=Queue
  ```

  which instructs Auburn to create a common signature from all implementations with names ending in `Queue` (see Section 6.3). When prompted to check the guess at a size-based shadow data structure, we continue with

  ```
  make
  ```

  as Auburn guesses correctly for each of the three ADTs, and so we need not modify the shadow data structure.

All of the executables needed for our experiment are now available: the DUG manager, the DUG evaluators, and the benchmarker. The benchmarker uses the default profile space described in Section 6.9.1.

7.1.2 Tracing Bugs

Before we benchmark the implementations, we should ensure that we have coded them correctly. Although type checking may remove most accidental errors, some may remain. It is also possible that the implementation presented in the literature contained a mistake. We can use Auburn to check that the implementations do not produce any run-time errors and that they produce the same results as each other. Section 6.9.1 describes this in further detail. For example, we may enter a command such as

```
Queue_Bmark -q seed
```

where *seed* is an initial value for the pseudo-random number generator. The benchmarker may then output a report like the following:

```
*** Tracer: Potential bug found.  The following implementations:
*** PhysicistsQueue
*** either did not evaluate the dug correctly, or gave a different
*** checksum to the implementation 'BankersQueue'.
```

Given that `PhysicistsQueue` is the only implementation to differ in checksum from the implementation `BankersQueue`, we can be fairly sure that the error is in `PhysicistsQueue`. However, a report like the following:

```
*** Tracer: Potential bug found.  The following implementations:
*** Batched1Queue, BatchedQueue, Bootstrapped1Queue,
Bootstrapped2Queue, BootstrappedQueue, Implicit1Queue,
Implicit2Queue, ImplicitQueue, Multihead1Queue, MultiheadQueue,
NaiveQueue, PhysicistsQueue, RealTimeQueue
*** either did not evaluate the dug correctly, or gave a different
*** checksum to the implementation 'BankersQueue'.
```

tells us that the bug is probably in `BankersQueue`.

Along with the above report, the benchmarker outputs the DUG responsible as a Haskell program (see Section 6.9.1). To find the bug, we choose to compile the DUG with the York nhc13 compiler [53] with tracing enabled [50]. Note however that we may use any other tracer or debugger, or we may simply inspect the DUG.

```
import PhysicistsQueue

import Prelude hiding (head,tail)

os0 = []
n1 :: Queue Int
n1 = empty
n2 = isEmpty n1
os1 = fromEnum n2 : os0
n3 :: Queue Int
n3 = snoc n1 7

main = print (sum (reverse os1))
```

Figure 7.1: The smallest DUG found by the queue benchmarker that causes an error in the physicist's queues. The queue benchmarker searched for about an hour.

We let the benchmarker run for a long time, trying to find the smallest DUG that causes an error. The smaller the DUG, the easier it is to find the bug. Out of 23 implementations, we find 4 contain bugs. All of these bugs result from accidental errors. We shall now describe 2 of these bugs.

Physicist's Queues

The queue benchmarker finds that the DUG of Figure 7.1 causes our first implementation of physicist's queues (see Section 2.1.5) to evaluate with a checksum different to the other queue implementations. Using the tracer of nhc13, we quickly find that the physicist's queue is evaluating `isEmpty n1` to `False`. As `n1` is `empty`, we would instead expect `isEmpty n1` to evaluate to `True`. Examining the code for `isEmpty`

```
isEmpty (Queue (x:w) f lenF r lenR) = True
isEmpty _ = False
```

we find that the two cases are swapped, returning `True` when the answer is `False`, and vice versa. To fix the bug, we just swap `True` with `False`.

Bootstrapped Queues

The queue benchmarker finds a subtle bug in the bootstrapped queue implementation (see Section 2.1.7). It can only find DUGs of a reasonable size—above 20 nodes—that contain the bug. The smallest DUG that it finds on a fairly large run, taking several hours, has 22 nodes. We omit the DUG here as it is rather large. The DUG evaluator for bootstrapped queues reports the error `tail Empty`.

In order to understand the bug, it is necessary to understand part of the code implementing bootstrapped queues. A bootstrapped queue has a front list, a middle queue of lists, and a rear list. The code also stores the size of the front and middle combined, and the size of the rear.

```
data Queue a = Empty
             | Queue [a] (Queue [a]) Int [a] Int
```

So, `Queue f m fmN r rN` has front `f`, middle `m`, rear `r`, and the size of the front and middle combined is `fmN`, and the size of the rear is `rN`.

We compile the DUG with tracing, and look for the root of the problem. The error results from a call to `tail` on an empty queue. The source of the `tail` is in the DUG itself. The shadow data structure prevents such a call in a DUG, and so the error must lie in the empty queue. The tracer reveals that a call to `tail` on a queue with 1 element in the front and 3 elements in the rear produces the empty queue. However, the front-middle size field, `fmN`, is 5, where it should be 1. This error leads to the queue becoming empty.

We step back through the trace of the queue till `fmN` agrees with the size of `f` and `m` combined. At this point, a list is pulled out of `m`. Before the pull, `fmN` agrees with `f` and `m`; after the pull, it does not. Before the pull, `m` contains two lists, one of 2 elements, and one of 4 elements; after the pull removes the list of 2 elements, `m` is empty, whereas it should contain the list of 4 elements.

Therefore, we find that `fmN` is correct, but that the queue has lost some elements from its middle. Let $\mathtt{m}_1$ be the middle queue before the pull, and $\mathtt{m}_2$ be the empty middle queue after the pull. Examining the trace of $\mathtt{m}_2$, we find a check

on the size of the front and rear of m_2. The front and rear of m_1 each contain one list, but the pull leaves m_2 with an empty front. However, the `fmN` field for m_2 is 1. This error leads to the queue being discarded as empty.

But why is the `fmN` field of m_2 not 0? Further back in the trace of m_2, we find that the pull copies the `fmN` of m_2 from m_1. However, after a pull, the combined size of the front and middle of the middle should be one less. This is the bug: The implementation of pull on a queue

```
Queue f (Queue mf mm mfmN mr mrN) fmN r rN
```

does not reduce `mfmN`.

7.1.3 Fine-Tuning the Implementations

When coding an implementation, there are many design decisions to make. For example, we might have the option to use a strictness flag on an integer field. This may make a significant difference to the performance of the implementation. Auburn helps us to make such design decisions. Auburn can compare the overall performance of an implementation, with and without a minor modification, on a large sample of benchmarks.

We make several minor modifications to the implementations of the three ADTs. We use the benchmarker of each ADT to time each implementation and its modifications over a sample of 100 benchmarks. The benchmarker can report the overall performance of an implementation I by checking the accuracy of the tree with a single leaf I. A "decision tree" made from a single leaf I always chooses I. Therefore, the accuracy of this tree reports how many times this choice is correct—that is, how many times I *is* the winner—and the average ratio I to the actual winner.

For example, to find the overall efficiency of implementation `BankersQueue` on the sample `sample`, use

```
echo BankersQueue | Queue_Bmark -tP -s sample -cP
```

This gives the number of times `BankersQueue` was the best implementation, and more importantly, the mean ratio of the time for `BankersQueue` compared to the time of the best implementation. By comparing the mean ratio of an

Modification	Description	Effect on Performance	Use?
Bankers	Add strictness flags	−2%	×
Batched	Remove `reverse [x]` from `snoc`	+10%	✓
Bootstrapped	Merge calls to `head` and `tail` in `checkF`	+0%	×
Implicit–1	Use `TwoInTwo` instead of a pair in the inner queue.	+4%	✓
Implicit–2	Merge calls to `head` and `tail` in `tail`.	+12%	✓
Multihead	Change to Okasaki's implementation.	−6%	×
Physicists	Add strictness flags.	+1%	×

Table 7.1: The effect of modifications on performance of queue implementations over a sample of 100 benchmarks.

implementation with and without a modification, we have an estimate of the overall effect of the modification.

Each implementation may have several or no modifications. We choose the best combination of modifications for each implementation. Tables 7.1, 7.2 and 7.3 show the results of the fine-tuning. The effect on performance is calculated by

$$\frac{\textit{Average ratio after modification}}{\textit{Average ratio before modification}} \times 100\%$$

We decide to use the modification if the effect on performance is significant—above 3%. Note that the benchmarker uses the default profile space described in Section 6.9.1. Appendix B details each modification in full.

An interesting point to note from the results, is that adding strictness flags makes very little difference.

7.1.4 Inducing Decision Trees

For each ADT, we use the benchmarker to time the implementations chosen from the fine-tuning of Section 7.1.3. We have several options for inducing the decision

Modification	Description	Effect on Performance	Use?
AVL–1	Replace `<` and `==` with `compare` in `lookup` and `update`.	−21%	×
AVL–2	Replace `<` and `==` with `compare` in `lookup` and `update`, with `LT` first.	−21%	×
AVL–3	Place `<` first in `lookup` and `update`.	+1%	×
AVL–4	Split `case` on a pair into two, in `cons` and `tail`.	+21%	✓
Adams	Maintain the balance invariant perfectly.	−1%	×
Braun	Merge calls to `head` and `tail` in `tail`.	+4%	✓
Elevator–1	Change floor separation from 10 to 3.	+5%	×
Elevator–2	Change floor separation from 10 to 5.	+13%	✓
Elevator–3	Change floor separation from 10 to 25.	−32%	×
SkewBin	Add strictness flags.	+1%	×
ThreadSkewBin	Add separate constructor for rank 1 elements.	+63%	✓

Table 7.2: The effect of modifications on performance of random-access sequence implementations over a sample of 100 benchmarks.

Modification	Description	Effect on Performance	Use?
Binomial	Add strictness flags.	−2%	×
BootSkewBin	Add strictness flags.	+0%	×
Leftist	Specialise `insert`.	+19%	✓
Pairing–1	Replace `<=` with `<` in `merge`.	−10%	×
Pairing–2	Specialise `insert`.	+8%	✓
SkewBin	Add strictness flags.	−1%	×

Table 7.3: The effect of modifications on performance of heap implementations over a sample of 100 benchmarks.

tree. Do we prune the tree? If so, using which method? We want the tree that most accurately represents the efficiencies of the implementations according to datatype usage. But how do we know which tree is the best? We want to make a general recommendation, for any ADT.

Choosing the Best Decision Tree

One way to estimate the accuracy of a tree is through collecting an additional sample of benchmarking results, and examining the accuracy of each tree on the unseen results. How large a sample do we collect for the induction of decision trees, and how large an additional sample for testing these trees? We decide to take as large a sample as we can fit in an overnight batch for the induction of decision trees, on the basis that a user will not want to take much longer than this. We take a much larger sample for the purpose of testing these trees, on the basis that we want to test the trees as much as possible.

We take a training sample of 200 DUGs for each ADT from which to induce the decision trees. These samples take about 10 hours to collect in total. We take a further test sample of 500 DUGs for each ADT with which to test the trees. These samples take about 25 hours to collect in total. For example, the following command:

```
Queue_Bmark -g seed -n 200 -o final.sample
```

generates a sample of size 200, writing the sample to the file `final.sample`. We use a heap of 80Mb for the DUG evaluator compiled using nhc13, which takes heap flags within `+RTS` and `-RTS` flags, and we pass these flags as follows:

```
Queue_Bmark -g seed -n 200 -o final.sample
            -e "-r 1 -R 5 -o \"+RTS -H80M -RTS\""
```

The flags `-r 1 -R 5` are the default flags passed to the tool `evalDugs` describing how to run the DUG evaluator—for further details see Appendix C. All other settings are the default, including using the default profile space described in Section 6.9.1.

From each training sample, we induce two trees: one using the gain criterion and the other using the gain ratio criterion. As well as keeping these trees, we also prune each of them using both reduced error pruning and very pessimistic pruning. For example, the following command:

```
Queue_Bmark -s final.sample -i -r -G -w re.tree
```

induces a tree from the sample in `final.sample` using the gain criterion, prunes the tree using the reduced error method, and writes the tree to `re.tree`.

For each of the three ADTs, Table 7.4 shows the accuracy of each of the six resulting trees applied to the test sample.

Recommendation for the Most Accurate Tree. We want to make a general recommendation for which tree to use when we want the best prediction of the most efficient competing implementation.

For queues and heaps, the accuracy of the original tree is about the same as the accuracy of either of the pruned trees. However, for random-access sequences, the mean ratio of the trees pruned using the reduced error method is significantly higher than the original trees or the trees pruned using the very pessimistic method. Further, the mean ratio is lower when using the gain ratio criterion. There is little to choose between the accuracy of the original tree and the tree pruned using the very pessimistic method, but the latter is smaller. Therefore, based on this evidence, to produce an accurate tree, we recommend using the *gain ratio criterion*, followed by pruning using the *very pessimistic method*.

ADT	Pruning	Criterion	Size	Success Rate (%)	Mean Ratio
Queue	None	Gain	25	83	1.023
		Gain Ratio	29	79	1.026
	Reduced Error	Gain	5	86	1.011
		Gain Ratio	6	80	1.023
	Very Pessimistic	Gain	16	87	1.010
		Gain Ratio	17	84	1.015
RASeq	None	Gain	25	79	1.174
		Gain Ratio	28	77	1.099
	Reduced Error	Gain	6	75	1.506
		Gain Ratio	9	75	1.207
	Very Pessimistic	Gain	23	79	1.172
		Gain Ratio	22	78	1.093
Heap	None	Gain	19	83	1.054
		Gain Ratio	23	84	1.047
	Reduced Error	Gain	4	77	1.059
		Gain Ratio	5	84	1.035
	Very Pessimistic	Gain	17	83	1.054
		Gain Ratio	17	85	1.045

Table 7.4: The accuracy of various trees applied to the corresponding test sample. The size of a tree is the number of branch nodes. A success is a correct prediction of the winning implementation. The mean ratio is calculated from the ratios of the times taken by the predictions to the times taken by the winners.

Recommendation for the Smallest Accurate Tree. Although the accuracy of a tree is very important, it is also important for the tree to be small. The smaller the tree, the easier it is to analyse the tree, matching the design of the implementation to the resulting empirical performance. Therefore we also want to make a general recommendation for which tree to use when we want a prediction from a small but accurate tree.

In every case, pruning using the reduced error method produces the smallest trees. Of these trees, using the gain criterion produces a tree that is a little smaller. Therefore, to produce a small but fairly accurate tree, we recommend using the *gain criterion*, followed by pruning using the *reduced error method*.

Benefits of Using Trees

How much do we gain from choosing an implementation according to the datatype usage? How does using a tree compare with choosing the same implementation regardless of datatype usage? Tables 7.5, 7.6 and 7.7 show the average ratio of each implementation over the corresponding training samples.

For queues, the Batched implementation wins most often on the test sample with a very good mean ratio of 1.02. So in the case of queues, there is little to gain from choosing the implementation according to datatype usage—one should just choose the Batched implementation regardless. However, the most accurate tree still manages to improve on this uniform selection with a mean ratio of 1.01, as does the smallest tree with a mean ratio of 1.011.

Similarly, for heaps, the Pairing implementation wins most often on the test sample with a very good mean ratio of 1.08. So, as with queues, choosing the Pairing implementation regardless of datatype usage is close to the optimal choice. Still, the most accurate tree improves on this with a mean ratio of 1.035, as does the smallest tree with a mean ratio of 1.059.

However, for random-access sequences, the results are more mixed. The AVL and ThreadSkewBin implementations come first most often, but the AVL implementation has a better overall performance, and the Elevator implementation has the best overall performance with a mean ratio of 2.12. The most accurate tree manages a mean ratio of 1.093, and the smallest tree manages a mean ratio of

Implementation	Training Sample		Test Sample	
	Winner (%)	Mean Ratio	Winner (%)	Mean Ratio
Bankers	0	1.70	0	1.72
Batched	72	1.02	72	1.02
Bootstrapped	0	1.99	0	2.00
Implicit	16	1.16	17	1.18
Multihead	0	2.30	0	2.33
Naive	3	16.11	3	19.15
Physicists	0	2.12	0	2.14
RealTime	10	1.19	9	1.22

Table 7.5: Average ratio of the time taken by each queue implementation compared to the winner over the training sample of 200 benchmarks and the test sample of 500 benchmarks.

Implementation	Training Sample		Test Sample	
	Winner (%)	Mean Ratio	Winner (%)	Mean Ratio
AVL	38	1.69	36	2.21
Adams	0	4.18	0	6.05
Braun	0	5.24	0	5.67
Elevator	10	2.00	8	2.12
Naive	6	7.88	10	7.54
SkewBin	0	2.54	0	2.69
Slowdown	0	2.86	0	3.26
ThreadSkewBin	46	2.95	46	8.09

Table 7.6: Average ratio of the time taken by each random-access sequence implementation compared to the winner over training sample of 200 benchmarks and the test sample of 500 benchmarks.

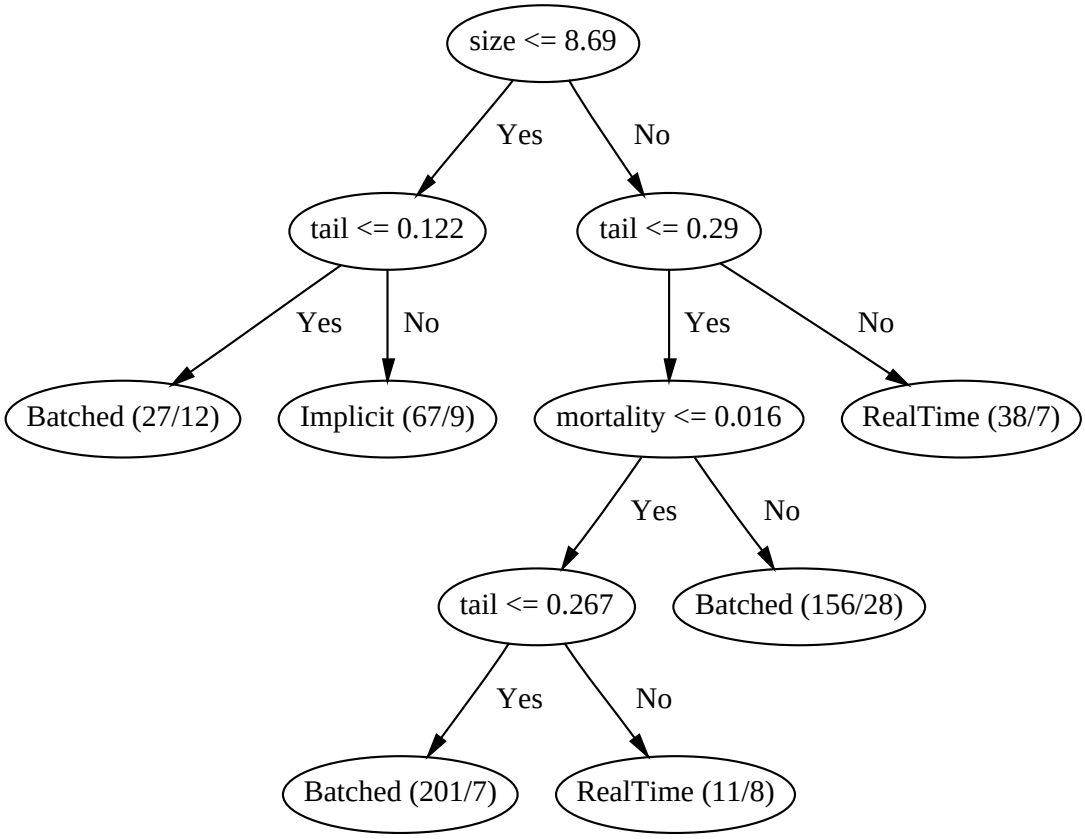


Figure 7.2: The tree induced using the gain criterion on the training sample for the queue ADT, pruned using the reduced error method.

1.506, each much better than the best uniform choice of a single implementation.

Therefore, based on these results, the best implementation of queues is Batched, and the best implementation of heaps is Pairing, regardless of how these data structures are used. However, for random-access sequences, the best implementation does vary according to how the data structure is used. These results are discussed below in greater detail.

Results

Using the recommendations above for accurate and small trees, the accurate trees are too large to show and discuss. However, Figures 7.2, 7.3 and 7.4 show the smallest trees. Each leaf is annotated with (N/E), where N is the number of benchmarks in the test sample covered by this leaf, and E is the number of misclassifications by this leaf. A benchmarker produces an accuracy report (see

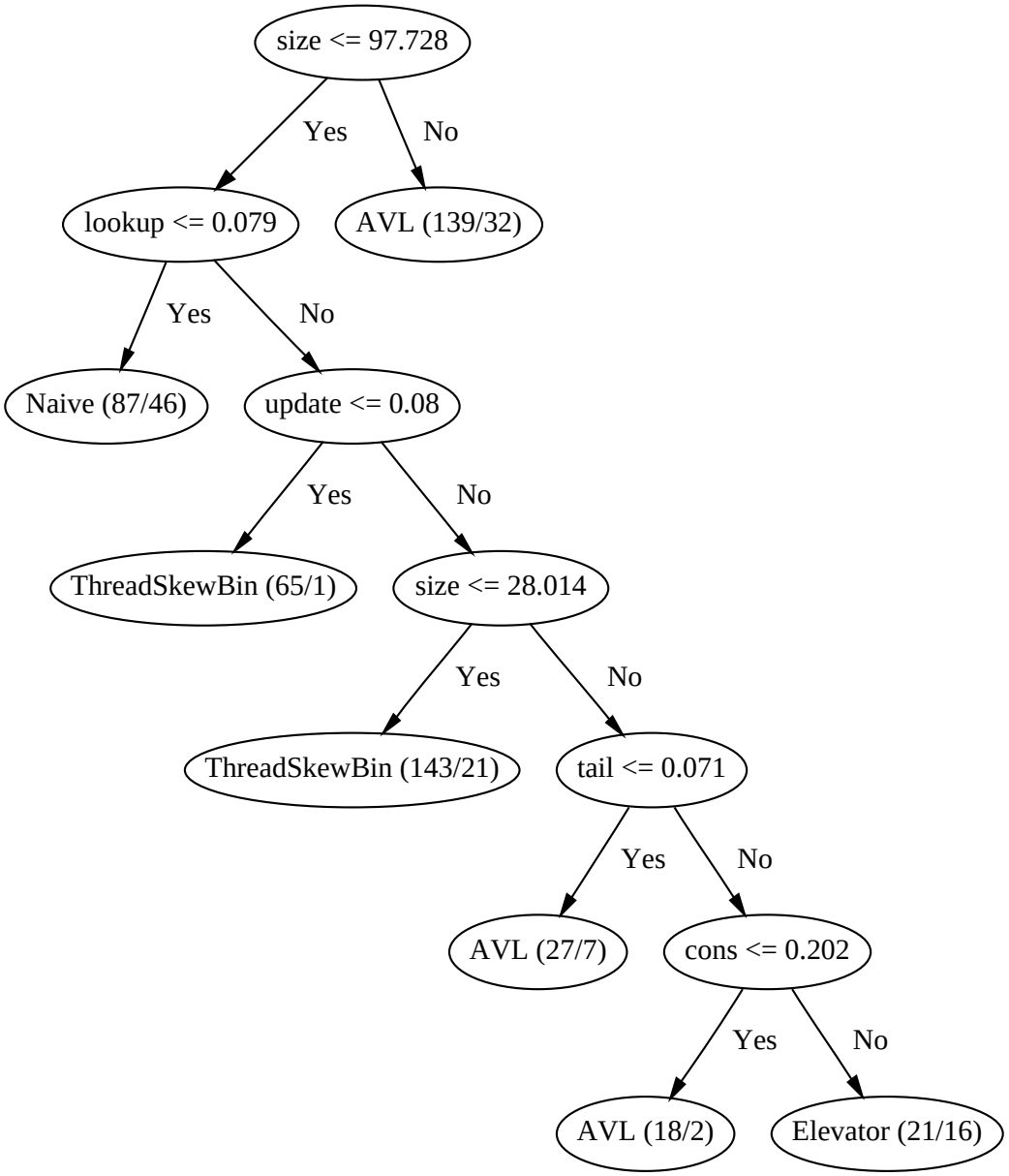


Figure 7.3: The tree induced using the gain criterion on the training sample for the random-access sequence ADT, pruned using the reduced error method.

Implementation	Training Sample		Test Sample	
	Winner (%)	Mean Ratio	Winner (%)	Mean Ratio
BinomialHeap	0	4.95	0	4.56
BootSkewBinHeap	0	4.64	0	4.52
LeftistHeap	0	1.99	0	1.98
NaiveHeap	26	1.30	20	1.28
PairingHeap	74	1.09	80	1.08
SkewBinHeap	0	5.40	0	5.01
SplayHeap	0	8.35	0	7.08

Table 7.7: Average ratio of the time taken by each heap implementation compared to the winner over training sample of 200 benchmarks and the test sample of 500 benchmarks.

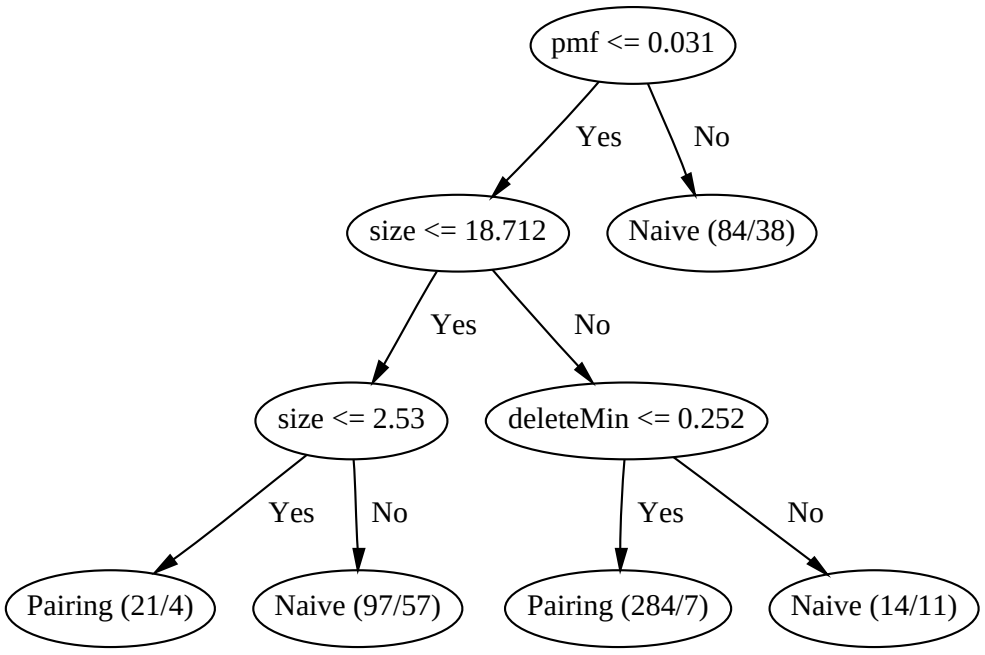


Figure 7.4: The tree induced using the gain criterion on the training sample for the heap ADT, pruned using the reduced error method.

Section 6.9.1) giving these annotations, which help to interpret the trees. The significance of a leaf can be estimated from the number and proportion of winning implementations that it classifies correctly.

For example, the deeper of the two leaves labelled with the RealTime queue implementation in Figure 7.2 only classifies 3 of the 11 winning implementations correctly. Hence this is not a very significant leaf. On the other hand, the top-most leaf labelled with the ThreadSkewBin random-access sequence implementation in Figure 7.3 classifies 64 out of 65 winning implementations correctly. Hence this is a very reliable leaf. Therefore it is a very significant leaf in the analysis of the tree. Recall that each test sample contains 500 different benchmarks.

Analysis of the Queue Decision Tree. Looking at Figure 7.2, there are only two significant cases where the Batched implementation consistently loses to another implementation:

- *Small size, fair tail weight (Implicit).* This may be the result of the Implicit implementation evaluating all operations on small queues without additional function calls. The Batched implementation on the other hand, must always make at least one extra function call in evaluating *tail* on a queue of any size.

- *Fair size, large tail weight (RealTime).* It is not clear why the RealTime implementation should beat the Batched implementation so consistently for this region of the profile space. Okasaki writes that the RealTime implementation is "the fastest known real-time implementation when used persistently". However, his comment concerns an implementation in a strict language (SML) where explicit laziness is costly, and it is not clear if the same applies to an implementation in Haskell. To examine the effect of persistence, we check the accuracy of a tree that splits up the test sample according to the PMF. Table 7.8 shows the results. It is clear from these results that the PMF does not have a significant role to play in deciding which implementation wins. Using POF instead of PMF produces similar results. Therefore, this case remains unexplained.

PMF	Winner (%)	
	RealTime	Batched
$0 \leq \text{PMF} < 0.0001$	10	63
$0.0001 \leq \text{PMF} < 0.001$	12	69
$0.001 \leq \text{PMF} < 0.01$	5	79
$0.01 \leq \text{PMF} < 1$	9	67

Table 7.8: The effect of persistence on the performance of the RealTime and Batched queue implementations on the test sample.

Analysis of the Random-Access Sequence Decision Tree. Almost all of the leaves in Figure 7.3 are significant—that is, almost all of them have a low proportion of errors. The Elevator leaf has a high proportion of errors, and the remaining leaves on the subtree from the test $tail \leq 0.071$ show AVL to win over half of the cases (36 out of 66). We consider the other leaves in turn.

- *Large size (AVL).* The AVL and Adams implementations are the most tree-like implementations, which gain strength as the size increases, because of their logarithmic complexity. The AVL implementation benefits from balancing specialised to adding or removing an element at the left—that is, from *cons* or *tail*. It is not clear if the Adams implementation could use a similar improvement.

- *Fair size, small lookup weight (Naive).* This is a little surprising. If few *update* operations are done, then we would expect the Naive implementation to win. But what if there are quite a lot of *update* operations? We might expect the Naive implementation to lose. The leaf's annotation does show quite a few errors, but there is another reason: An *update* will only be fully evaluated if it is forced. The only observations in the absence of *lookup* are *head* and *isEmpty*, and because the Naive implementation is so lazy, these observers will only force updates on the first element. The other implementations are not as lazy, and so do not benefit as much. The issue of strictness is examined in more detail in Section 7.3.3.

- *Fair size, fair lookup weight, small update weight (ThreadSkewBin).* The annotation shows this leaf is very reliable, with 64 out of the 65 cases correct. The ThreadSkewBin implementation deliberately implements an efficient *lookup* operation, at the expense of an inefficient *update* operation.

- *Small size, fair lookup weight, fair update weight (ThreadSkewBin).* Although ThreadSkewBin implements *update* to take $O(i)$ time, where i is the index of the element updated, for small lists, this is not very different from the logarithmic complexity of the AVL implementation. The simplicity of the ThreadSkewBin implementation makes it win on small lists, even with many *update* applications.

- *Fair size, fair lookup weight, fair update weight (AVL).* With enough *update* operations, and a reasonably sized sequence, the AVL implementation beats the ThreadSkewBin implementation.

Analysis of the Heap Decision Tree. As with queues, Figure 7.4 shows that a single implementation (Pairing) dominates the results. Once again, there are only two significant cases where another implementation beats the Pairing implementation.

- *Large* PMF *(Naive).* More than half of the benchmarks with a large PMF are won by the Naive implementation. Okasaki advises in [38] that Pairing heaps are not efficient under persistent use.

- *Small size, but not* very *small size (Naive).* The Naive implementation wins for small heaps, which is typical of a naive implementation of an ADT. Surprisingly though, the Pairing implementation wins significantly for very small heaps. This may be the result of the `newtype` constructor in Naive causing an extra function call, as compared with the Pairing implementation.

It is surprising that Splay heaps perform so poorly. We shall see in Section 7.2.1 that Splay heaps perform much better for some real benchmarks. Why do they perform so badly under Auburn-generated benchmarks? Perhaps there is an aspect of datatype usage that Auburn does not control but fixes in a region

where Splay heaps perform badly. Two possible candidates include the minimum frontier size, which affects the applications of non-unary mutators like *merge*, and the pool size, which affects the number of equal elements in a heap. The benchmarker cannot record either of these factors currently, only profile and shadow profile attributes.

7.1.5 Summary

Given these results from Auburn, which implementation should we use for queues, random-access sequences, and heaps? For queues, we recommend you always use Batched queues. For random-access sequences, we make the following recommendations: use AVL trees if your lists are quite large (an average length of above 100); use Naive lists if you are not doing many *lookup* or *update* operations; use *ThreadSkewBin* lists if you are doing quite a few *lookup* operations, but not doing many *update* operations, or if your lists are quite small (an average length of below 30); otherwise, use AVL trees. For heaps, we recommend that you always use Pairing heaps.

7.2 Evaluating Auburn

We use Auburn to produce advice about the choice of implementation of three ADTs in Section 7.1.4. But how good is this advice?

Ultimately, the value of Auburn's advice lies in how well it predicts which implementation *is* the best. To test this, we construct several *real* benchmarks—real in that they produce useful results. We time each benchmark with each implementation, to find which implementation *really is* the best. By comparing this with Auburn's prediction, based on the profile of the benchmark, we can estimate Auburn's accuracy in practice.

7.2.1 Real Benchmarks

All of the benchmarks are based on either sorting a list or processing a graph. There are four benchmarks for each ADT, and four data sets for each benchmark. This gives a total of 16 different uses of each ADT. We describe each benchmark

very briefly. References to literature give further details of the algorithms, and the source code is available from the Auburn web page [4].

Queue Benchmarks

The queue benchmarks are the hardest to find.

- *Shellsort.* It is possible to implement Shell's sort [48] using queues [26].
- *Breadth-First Search (BFS).* Breadth-first search of a graph is a common use of queues, see [11] (page 469).

Since we could find no more benchmarks, and since varying the *increments* used by Shellsort varies how the algorithm uses the queue dramatically, we let three sets of increments provide three of the four queue benchmarks.

Random-Access Sequence Benchmarks

An array is one of the most commonly used data structures, even in functional programs, so benchmarks are not hard to find. However, we also wish to include algorithms that use the sequences as lists, as in [33].

- *Bucketsort.* This sort uses random-access operations heavily, see [11] (page 180).
- *Quicksort.* Sorting a list using a functional implementation of Quicksort [19] does not use any random-access operations.
- *Depth-First Search (DFS).* Implementing a graph as a random-access list of adjacent vertices [11] (page 465) allows any graph algorithm to use random-access lists. We choose one of the simplest graph algorithms, depth-first search [11] (page 477).
- *Kruskal's Minimum-Cost Spanning Tree (KMCST).* Kruskal implements a minimum cost spanning tree algorithm [11] (page 504) using a disjoint-set data structure [11] (page 440) which we implement using a random-access list.

Heap Benchmarks

A few common algorithms use a priority queue, or a heap. Many of these however use additional operations like *decreaseKey*. This operation reduces the key of any element in the heap by a given amount. We replace this operation with an *insert* of the element with a lower key, and a guard against reading the same element more than once. This is not the most efficient implementation of *decreaseKey*, but it does give us real benchmarks using heaps in a variety of ways. Very few algorithms use the *merge* operation: We could find only one.

- *Heapsort.* This is a simple sorting algorithm [11] (page 147).
- *Mergeable Minimum-Cost Spanning Tree (MMC).* This is the only heap benchmark to use the operation *merge* [11] (page 418).
- *Dijkstra's Shortest Paths (DSP).* We replace *decreaseKey* with *insert* as explained above in the *modified* Dijkstra algorithm [11] (page 530).
- *Prim's Minimum-Cost Spanning Tree (PMC).* Similarly, we replace *decreaseKey* with *insert* in Prim's algorithm [11] (page 505).

Results

Tables 7.9, 7.10 and 7.11 give the results of running each benchmark, including: the winning implementation; the ratio of the implementation predicted to win by the recommended accurate tree; the ratio of the implementation predicted to win by the recommended small but accurate tree; the ratio of the implementation with the best overall performance in the training samples of Section 7.1.4 (see Tables 7.5, 7.6 and 7.7); and the average ratio of all implementations.

The ratios of the implementations predicted by the two trees that Auburn produced from the training samples in Section 7.1.4 indicate Auburn's accuracy. To aid the interpretation of this figure, the ratio of the implementation with the best overall performance in the training samples gives the difference between Auburn's prediction and a uniform choice made regardless of datatype usage. Further, the average ratio of all implementations gives the difference between Auburn's prediction and a random choice of implementation.

For queues, the uniform choice (of Batched queues) has a very good average ratio of 1.068, yet both of Auburn's trees predict a better implementation on average. All three are much better than a random choice.

For random-access sequences, the uniform choice (of AVL trees) has an average ratio of 1.837, indicating that the best implementation varies significantly across the benchmarks, as we would expect from the results of Section 7.1.4. Both of Auburn's predictions perform better on average than the uniform choice, and much better than the random choice.

For heaps, like queues, the uniform choice (of Pairing heaps) has a very good ratio of 1.022, and neither of Auburn's trees can improve on this. However, apart from one very bad prediction (PMC benchmark on data set 3), Auburn's predictions are still much better than a random choice.

For a discussion of the worst of Auburn's predictions, see Section 7.3.4.

Summary. The summary of Section 7.1.4 advised that we use always use Batched queues and Pairing heaps, regardless of datatype usage, and that we use a different random-access sequence implementation according to specific aspects of the datatype usage. This advice gives very good results for the real benchmarks of this section, making choices within 10% of the best implementation for queues and heaps, and within 30% of the best implementation for random-access sequences.

7.3 Locating Inaccuracy in Auburn

Section 7.2.1 showed that the advice of Section 7.1.4 is good, but not perfect. What is the source of any inaccuracy in Auburn's results? What can go wrong? Here are the main possibilities:

- The DUG does not capture datatype usage sufficiently.
- The profile of a DUG does not capture datatype usage sufficiently.
- Strictness issues cause the work that is *actually* done to be less than the work that is *reportedly* done.

Benchmark Name	Data Set	Winning Impn.	Acc. Tree Ratio	Small Tree Ratio	Unfm. Ratio	Avg. Ratio
BFS	1	Batched	1.046	1.046	1.000	1.054
BFS	2	Batched	1.050	1.000	1.000	1.052
BFS	3	Batched	1.000	1.000	1.000	1.175
BFS	4	Batched	1.000	1.000	1.000	1.060
Shellsort1	1	Implicit	1.110	1.054	1.110	1.525
Shellsort1	2	Implicit	1.098	1.040	1.098	1.805
Shellsort1	3	Implicit	1.079	1.026	1.079	4.632
Shellsort1	4	Implicit	1.080	1.025	1.080	4.961
Shellsort2	1	Implicit	1.087	1.054	1.087	1.454
Shellsort2	2	Implicit	1.081	1.052	1.081	1.414
Shellsort2	3	Implicit	1.068	1.038	1.068	2.546
Shellsort2	4	Implicit	1.065	1.037	1.065	2.431
Shellsort3	1	Implicit	1.000	1.000	1.126	1.388
Shellsort3	2	Implicit	1.000	1.000	1.116	1.357
Shellsort3	3	Implicit	1.093	1.042	1.093	1.719
Shellsort3	4	Implicit	1.093	1.040	1.093	1.713
		Average	1.059	1.028	1.068	1.955

Table 7.9: Results of running the queue benchmarks.

Benchmark Name	Data Set	Winning Impn.	Acc. Tree Ratio	Small Tree Ratio	Unfm. Ratio	Avg. Ratio
Bucketsort	1	AVL	1.000	1.000	1.000	2.018
Bucketsort	2	AVL	1.000	1.000	1.000	2.405
Bucketsort	3	AVL	1.000	1.000	1.000	6.139
Bucketsort	4	AVL	1.000	1.000	1.000	3.186
DFS	1	AVL	1.000	1.203	1.000	1.748
DFS	2	Adams	1.002	1.002	1.002	2.316
DFS	3	AVL	1.000	1.000	1.000	3.075
DFS	4	AVL	1.000	1.000	1.000	5.992
KMC	1	ThreadSkewBin	1.000	1.000	1.181	1.404
KMC	2	ThreadSkewBin	1.000	1.930	2.063	1.932
KMC	3	ThreadSkewBin	1.000	2.357	1.699	1.672
KMC	4	ThreadSkewBin	1.557	1.954	1.954	1.599
Quicksort	1	Naive	1.000	1.000	4.856	3.193
Quicksort	2	Naive	1.000	1.000	3.069	2.310
Quicksort	3	Braun	1.889	1.889	1.826	1.828
Quicksort	4	Naive	1.000	1.000	4.740	3.088
		Average	1.091	1.271	1.837	2.744

Table 7.10: Results of running the random-access sequence benchmarks.

Benchmark Name	Data Set	Winning Impn.	Acc. Tree Ratio	Small Tree Ratio	Unfm. Ratio	Avg. Ratio
DSP	1	Pairing	1.000	1.021	1.000	1.061
DSP	2	Splay	1.028	1.106	1.028	1.086
DSP	3	Splay	1.004	1.004	1.004	1.326
DSP	4	Splay	1.012	1.040	1.012	1.067
Heapsort	1	Naive	1.009	1.009	1.009	1.343
Heapsort	2	Splay	1.077	1.077	1.077	1.798
Heapsort	3	Naive	1.008	1.008	1.008	1.387
Heapsort	4	Splay	1.171	1.171	1.171	3.371
MMC	1	Leftist	1.027	1.005	1.027	1.106
MMC	2	Pairing	1.000	1.002	1.000	1.050
MMC	3	Pairing	1.000	1.000	1.000	1.144
MMC	4	Naive	1.006	1.000	1.006	1.009
PMC	1	Pairing	1.007	1.007	1.000	1.068
PMC	2	Pairing	1.019	1.019	1.000	1.075
PMC	3	Splay	3.363	1.018	1.018	1.446
PMC	4	Pairing	1.007	1.007	1.000	1.055
		Average	1.171	1.031	1.022	1.337

Table 7.11: Results of running the heap benchmarks.

- The induction and pruning processes produce inaccurate trees.

We shall now deal with these individually in detail.

7.3.1 Insufficient DUG

We define the DUG in Chapter 3 to capture the datatype usage of a data structure by an application. We base the whole of this thesis on this definition of a DUG. But does it capture datatype usage sufficiently? We can test this as follows.

We take a real application or benchmark, and run it using each ADT implementation, measuring the efficiency of each. We extract the DUG from each run. We then run a DUG evaluator on each DUG using the corresponding ADT implementation. We then compare the efficiencies of the implementations when used by the application with the efficiencies of the implementations when used by the DUG evaluators.

If the DUG captures all of the relevant information for influencing the efficiency of an ADT implementation, we would expect the relative efficiencies of the implementations to be the same. For example, the order of the implementations, most efficient first, should be the same for the application as for the DUG evaluator. Further, the efficiencies should correlate linearly. Note that the relative efficiencies need not be exactly the same, as the total amount of work done differs between the application and the DUG evaluator. However, this is only a constant difference, which should therefore produce a linear relationship.

We take the 12 benchmarks of Section 7.2.1 using all 4 data sets, giving 16 different uses of a data structure for each of the 3 ADTs. Ideally we would take one DUG for each implementation, because the DUG varies between implementations due to strictness (Section 7.3.3). However, the total number of DUGs for each ADT would be the number of implementations multiplied by 16, which is too many to handle. Hence we only take a DUG from one of the implementations, and let every implementation evaluate this representative DUG. This will not affect the results much, as the DUGs only vary by at most 2% across implementations, and usually not at all. Also, this simplification will more likely worsen our results than improve them.

For each comparison of relative efficiencies of implementations, we calculate

ADT	Correlation		
	Worst	Mean	Best
Queue	0.482	0.924	1.0
RASeq	0.983	0.998	1.0
Heap	-0.261	0.579	0.989

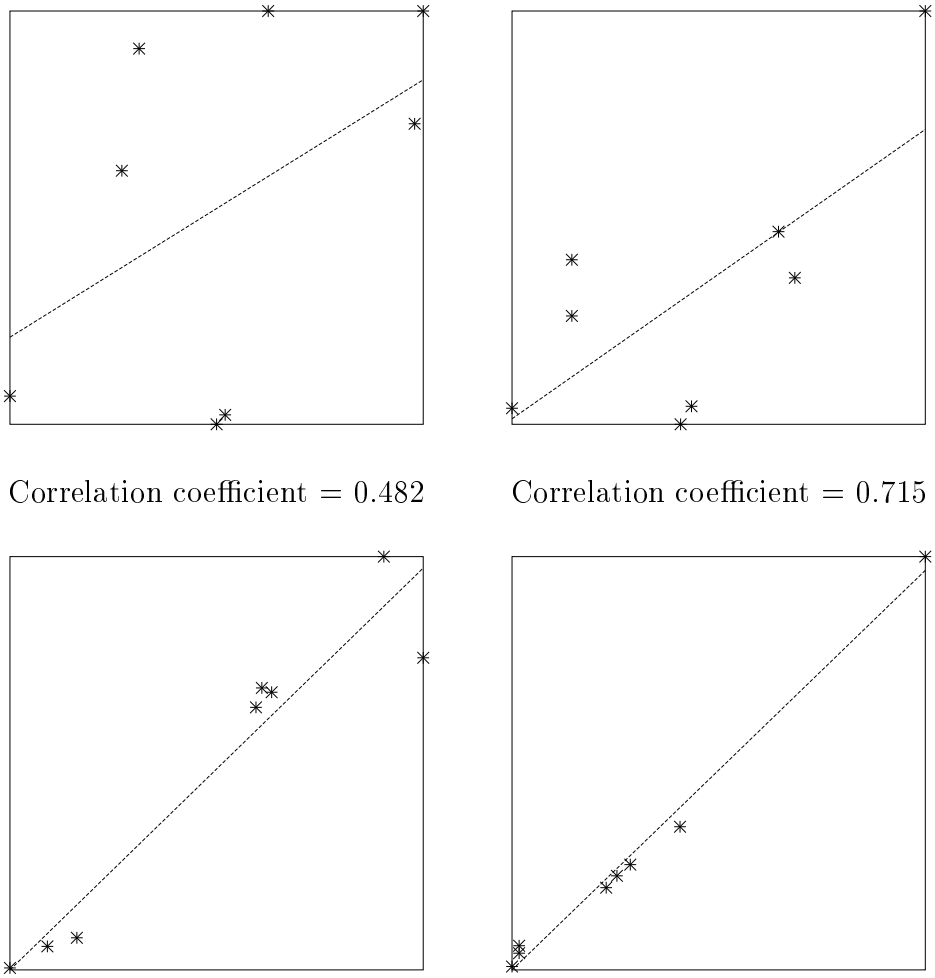
Table 7.12: Correlation coefficients for efficiencies of implementations, comparing a benchmark with a DUG evaluator. The DUG evaluator is evaluating the DUG extracted from the benchmark.

the *correlation coefficient* (as defined in Section 5.2). Table 7.12 gives these. To aid our understanding of how good or bad a correlation coefficient is, Figure 7.5 gives graphs for a range of examples—the better the graph looks like a line, the closer the relationship is to being linear.

From Table 7.12, we see that the queue ADT and the random-access sequence ADT show good correlations between the behaviour of implementations when used in an application and when used in a DUG evaluator. However, the heap ADT shows worse results.

The correlations for the queue ADT are mostly very good, with 70% being above 0.99. However, there were a few low correlations. What makes these correlations low? They all come from the same benchmark, breadth-first search. In fact, *every* correlation for this benchmark is less than 0.5, regardless of the data set used. It is not clear why the performance of the implementations differs so much between benchmark and DUG evaluation in this case. It is possible that there is some peculiar run-time behaviour due to garbage collection, as we find with the Quicksort benchmark in Section 7.3.4.

For the heap ADT, the main reason for the bad results comes from the inability of the DUG extraction to record the elements inserted into the heap (see Section 4.2.1). Therefore, all elements are recorded as being 0. This affects the efficiency of the different implementations greatly because every element in the heap has the same value. To test this suspicion, we replace the elements of the extracted DUGs with random values, and re-run the experiment to obtain new correlation coefficients. We find that the mean correlation coefficient increases



Correlation coefficient = 0.482 Correlation coefficient = 0.715

Correlation coefficient = 0.958 Correlation coefficient = 0.995

Figure 7.5: Examples of graphs plotting data with different correlation coefficients.

to 0.780, significantly improving on the previous mean of 0.579. If DUG extraction could record the elements' values, we suspect the correlation would rise even further.

7.3.2 Insufficient Profile

Just as we design the DUG to capture datatype usage, we design the profile of a DUG to capture the most important aspects of datatype usage, where we measure importance with regard to the effect on ADT implementation efficiency. We base the whole of Auburn on this premise. We can test its validity as follows.

We can generate several DUGs from the same profile, thereby having similar profiles, and compare the efficiencies of implementations evaluating the different DUGs. If the profile of a DUG does capture datatype usage sufficiently, then the results should be similar. However, all of the DUGs are generated using Auburn, and so this test is rather limited in scope.

Therefore, we take the profiles of DUGs extracted from real benchmarks, and generate a few DUGs from each profile. We then compare the efficiencies of the implementations at evaluating the DUGs and at running the benchmarks. We take the same 12 benchmarks across 4 different data sets each that we used in Section 7.3.1, giving the same 16 different uses of a data structure for each of the 3 ADTs. For each DUG extracted, we generate 3 more DUGs. Table 7.13 shows the mean correlation coefficients.

The correlation between DUGs generated from the same profile is very high for each ADT. However, the correlation between the benchmark and the generated DUGs is much lower, though still quite high. This indicates that some important aspects of datatype usage are not being carried through from a benchmark, through a profile, into a generated DUG.

The most probable reason for this is the lack of size information. This is captured in the *shadow* profile, but this does not influence the generated DUGs. To test this, let's look at some examples of low correlations between benchmark and generated DUG.

Take the Bucketsort benchmark for the random-access sequence ADT. Running on the third data set, the correlations between the benchmark and two of

ADT	Mean Correlation		
	between DUG & Benchmark	between DUG & DUG	Overall
Queue	0.859	0.923	0.891
RASeq	0.704	0.969	0.836
Heap	0.694	0.999	0.846

Table 7.13: Correlation coefficients for the efficiencies of implementations, comparing a benchmark with a DUG evaluator. The DUG evaluator is evaluating DUGs with similar profiles to the profile of the DUG extracted from the benchmark. The mean correlation between the benchmark and one run of the DUG evaluator is separated from the mean correlation between the different runs of the DUG evaluator.

the three DUGs are very low, at –0.118 and 0.0109. The shadow profile for the benchmark reports an average size of 667. The shadow profiles for the two DUGs report average sizes of 12 and 15. However, the shadow profile for the third DUG reports an average size of 88. The correlation between this DUG and the benchmark is much higher at 0.794. Therefore, for this example at least, a higher correlation coincides with a closer average size.

Take the Prim's minimum-cost spanning tree benchmark for the heap ADT. Running on the third data set, the correlations between the benchmark and all three DUGs are very low, at –0.250, –0.244 and –0.242. The average size for the benchmark is 239. The average size for the DUGs are 14, 16 and 33. Again, this example shows low correlations for distant average sizes.

In fact, almost every low correlation coincides with a large difference in average size. To show this, Figure 7.6 plots the correlation coefficient against the percentage size difference (calculated as the difference in size, expressed as a percentage of the larger size). Most of the low correlations have a high size difference, and most of the low size differences have high correlations. From this we deduce that an important datatype usage characteristic not caught in the profile is size. However, there are a lot of points with large size differences and high correlations, and from this we deduce that size is not always an important datatype

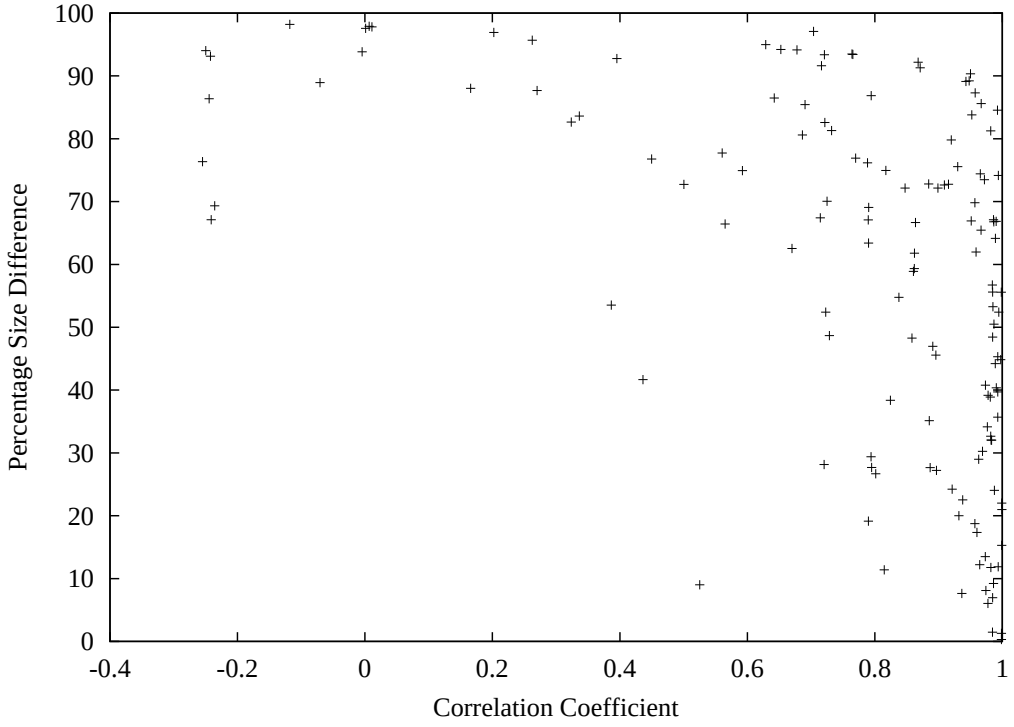


Figure 7.6: Correlation coefficient for implementation efficiency plotted against the percentage difference in size, as reported by the shadow profile.

usage characteristic.

7.3.3 Strictness Issues

When an implementation evaluates a DUG, only the observations are demanded. As a result, some of the generations and mutations may not be forced. This depends on the strictness of the ADT implementation evaluating the DUG. This discrepancy between what is reportedly evaluated (ie. the DUG) and what is actually evaluated can cause the following crucial problem: The profile of a DUG may no longer represent the important aspects of the *actual* datatype usage.

To estimate the average proportion of a DUG not evaluated, we evaluate 10 DUGs for each of the three ADTs, queue, random-access sequence, and heap, and all of their implementations. For any DUG D_0, we extract the DUG D_1 actually evaluated, by transforming a DUG evaluator for DUG extraction. We then repeat this process, obtaining D_2, D_3, etc. till we obtain a fixed point, that is, till $D_i = D_{i+1}$.

For each ADT, and for every combination of DUG and implementation, we reach a fixed point on the second iteration, that is, $D_1 = D_2$.

Every D_0 has 1000 nodes. For queues, the mean difference in size from D_0 to D_1 is 5 nodes, and the maximum difference is 43 nodes. For any DUG D_0, each queue implementation evaluates D_0 to the same degree, that is, D_1 is the same across all the implementations. We can account for the differences between the sizes of D_0 and D_1 entirely by the following two factors:

- Unless we apply an observation to the result of a mutation, the mutation is not evaluated.

- The *empty* generator takes no arguments, so the DUG extraction shares every application of *empty* (see Section 4.4.3).

For random-access sequences, the mean difference in size from D_0 to D_1 is 55 nodes, and the maximum difference is 694 nodes. Apart from the two factors given above for queues, these differences in sizes also result from an additional factor: Consider n successive applications of *cons* to an empty list; if we apply *head* to the result of these applications, a sufficiently lazy implementation of

lists will only evaluate the last application of *cons*. For the implementations of a random-access sequence that we consider, only the Naive implementation is lazy enough for this factor to cause any additional difference. Most of the large differences in size comes from the two factors listed above for queues.

For heaps, the mean difference in size from D_0 to D_1 is 37 nodes, and the maximum difference is 160 nodes. We can account for these differences in the same manner as random-access sequences, except that DUG extraction does not share every application of the *empty* generator. This results from the *Ord* context on *empty*. The context makes the DUG evaluator repeatedly evaluate *empty* applications.

Comparing the profile of D_0 with the profile of D_1, averaging across all of the DUGs of the three ADTs, each of the weights differ by less than 0.01, the mortality differs by about 0.05, the PMF differs by about 0.01, and the POF differs by about 0.35. So only the POF differs greatly. This is because neither DUG evaluation nor DUG extraction preserve the order of evaluation of mutations, only the order of evaluation of observations; DUG evaluation cannot enforce the order of mutations because of the privacy of the ADT framework combined with laziness (see Section 3.2.1); DUG extraction changes the order of mutations to fit the definition of a DUG (see Section 4.3.3).

What these experiments do not reveal, is how the *degree* of evaluation of *individual nodes* differs across implementations. For example, the Naive implementation of random-access sequences is lazy enough not to evaluate fully applications of *update*, unless an application of *lookup* or *head* demands it. This causes a surprising result in the analysis of the random-access sequence decision tree—see Section 7.1.4.

7.3.4 Inaccurate Trees

Some of Auburn's predictions of the best implementations for the real benchmarks of Section 7.2.1 are quite inaccurate. Is there any reason for these inaccuracies specific to the induction or pruning of trees?

Consider the predictions for the implementations of random-access sequences—see Table 7.10. The small tree predicts the winning implementa-

tion for 10 out of the 16 combinations of benchmark and data set. For the KMC benchmark however, it predicts the wrong implementation for three out of the four data sets. Looking at the profile of the KMC benchmark running on data set three, we find

$$update = 0,\ lookup = 0.046,\ size = 63$$

The small tree for random-access sequences in Figure 7.3 predicts the Naive implementation as the winner for this profile. This prediction would probably be correct for a smaller size, or a smaller *lookup* weight, but this detail has been pruned out of the tree. The most likely deciding factor between Naive and ThreadSkewBin is the *combination* of *lookup* and *size*. A more accurate tree of the same size might be obtained if the decision tree could employ tests on arithmetic combinations of attributes. For example, $lookup * size \leq 1$. However, as Quinlan points out in Section 10.2 of [46], introducing the possibility of such tests can slow down the process of induction by an order of magnitude.

For another example of the need for combinations of attributes, consider the DFS benchmark running on data set 1. The profile for this run shows

$$update = 0.469,\ lookup = 0.531,\ size = 9$$

The small tree for random-access sequences in Figure 7.3 predicts the ThreadSkewBin implementation as the winner for this profile. Again, the most likely deciding factor between the AVL and ThreadSkewBin implementations for this region of the profile space is the combination of *update* and *size*. For any of the other data sets, the *size* is above the 28.014 used in a test in the small tree, and the tree correctly predicts AVL as the winner. This test is accurate so long as *update* is not very high, as it is with the profile above. However, again, this detail has been pruned out of the tree. A more accurate test might be something like $update * size \leq 5$.

There are only two other bad predictions by Auburn: the PMC benchmark running on data set 3, and the Quicksort benchmark running on data set 3. The bad prediction for the PMC benchmark came from using the recommended accurate tree. This tree is too large to analyse (with 17 tests), and so we do not discuss this prediction. The bad prediction for the Quicksort benchmark reflects

Sampling	Time (s)	
Interval (s)	Naive	Braun
360	249	243
60	196	240
10	100	243

Table 7.14: Times taken to run the Quicksort benchmark using the Naive and Braun random-access sequence implementations. The benchmark was compiled for heap profiling, and run using different heap sampling intervals. The heap size is set at 80Mb, and a constructor profile is requested.

a very pathological result: For a benchmark with no random-access operations, the Braun tree is almost twice as fast as the Naive list! After compiling the benchmark with heap profiling, running the benchmark for each implementation with different sampling rates reveals some odd behaviour. From Table 7.14, we see that for a large sampling interval, the Braun tree is faster. As the sampling interval decreases, the Braun time remains fixed, but the Naive time reduces dramatically. When the run-time system takes a sample of the heap, it also performs a garbage collection. Therefore, as the heap sampling interval decreases, more garbage collections happen. In the original run of the benchmark without profiling, no garbage collections happened at all when using the Naive implementation. Given this, the most probable explanation is that without many garbage collections, the Naive implementation suffers from some space problem. This result is peculiar to the compiler nhc13. Using the compiler HBC [18], Naive is much faster than Braun, regardless of how many garbage collections happen, as expected.

7.4 Summary

For several competing implementations of three ADTs, we have used Auburn to check for their correctness, to fine-tune the code, and to give advice on when to use which, according to the datatype usage. The user of Auburn has to do very little to achieve all this, as most of it is automated.

Furthermore, we have found Auburn's advice to be quite good when applied to real benchmarks, making choices within 10% of the best implementation for queues and heaps, and within 30% of the best implementation for random-access sequences. We have also examined possible sources of inaccuracy in Auburn's advice, and identified the main problems: the inability of DUG extraction to record the values of type a (where the type of a version is $T\ a$), the lack of information on the degree of evaluation of individual applications of operations, and the lack of information about the space behaviour of a benchmark.

Chapter 8

Conclusions

In Chapter 1, we noted that the empirical performance of functional data structures has been neglected in the existing literature. From this we decided to the develop the theory and practice of benchmarking functional data structures. We shall now summarise the progress of this thesis towards this goal.

8.1 Benchmarking Theory

There is no previous literature on how to benchmark functional data structures in a structured manner. Neither is there any attempt to define "the use of a data structure", despite its importance in the efficiency of data structures.

In Chapter 3, we have presented a formally defined model, a DUG, to capture how an application uses a data structure. Chapter 3 also defined the profile of a DUG, summarising the most importance aspects of datatype usage. This allows us to talk about the efficiency of data structures with reference to a few important aspects of datatype usage.

Previously, anyone wanting to benchmark some data structures would have to create the benchmarks manually, mostly without knowing how these benchmarks used the data structures. In Chapter 4, we have presented a method for creating a benchmark from a profile of the datatype usage.

Some compilers support time profiling that records how often a function is called. However, there is no way to extract other aspects of datatype usage. In Chapter 4 we have presented a method for extracting a profile from an application.

In Chapter 5 we discussed how to use the algorithms of Chapter 4 to benchmark some competing data structures in a structured manner. After proposing a few alternatives, we chose to use the induction of a decision tree from the results of a random sample of generated benchmarks. The decision tree presents a summary of which data structure is best according to the datatype usage.

In summary, previous attempts to benchmark data structures relied on hand-picked benchmarks, giving results biased towards an unknown datatype usage. This thesis describes a way to automate the production of results qualified by a description of datatype usage.

8.2 Benchmarking Practice

As stated above, previously, the only way to measure the efficiencies of competing data structures was to find, code, and test benchmarks yourself. In Chapter 7 we applied this method to several implementations of three different ADTs. This proved to be very time-consuming and very tedious. Further, it is not clear how each benchmark uses a data structure. So the results of this manual benchmarking tell us little more than which implementation was best for those particular benchmarks.

To improve on this situation, we have built a tool, called Auburn, which takes much less time to use, and produces much more useful results. In Chapter 7, we have used Auburn on the same implementations of the same three ADTs. Using Auburn took much less effort than the manual creation of benchmarks. We have produced a decision tree for each ADT, and from these we gave advice on when to use which implementation. This advice accurately predicted the results of the manual benchmarking.

We also showed in Chapter 7 that Auburn is very useful for finding bugs in the coding of implementations, and for testing the effect of minor modifications to this code.

8.3 Criticism

Taking a step back, we can ask the following question: Does this thesis achieve its goals? We list the main points both in favour of and against this thesis, starting with those in favour.

- Benchmarking functional data structures is a subject with very little coverage in the existing literature, and this thesis makes some key steps towards understanding the important issues, including how to define datatype usage, and how to use this definition to conduct a benchmarking experiment.

- Auburn is a useful tool for benchmarking new and existing data structures. We have demonstrated this for 23 different data structures across 3 different ADTs.

- Anyone wanting to use a queue, a random-access list, or a heap may use Section 7.1.5 to decide when to use which implementation. This will improve the efficiency of their application.

Here are the main points against this thesis:

- Auburn is not very user friendly and rather involved. For example, the user has to learn about and check the shadow data structure and profile space.

- Section 7.3 revealed some inaccuracies in Auburn, in particular its treatment of strictness, space behaviour, and values of type a (where a version has type $T\ a$).

- We do not consider the effect on the benchmarking results of changing language, operating system, or compiler. In particular, the advice of Section 7.1.5 may not apply to other systems.

- Neither do we consider the effect on the benchmarking results of changing the profile distribution. Is the distribution we use fair?

We consider these criticisms in the following section.

8.4 Future Work

Drawing on the previous section, here are the main areas for future work.

- Relax the restrictions on the operations that Auburn can benchmark. In particular, include higher-order operations, and operations over more than one type. For example, Auburn cannot currently benchmark the following operations:

  $$fold :: (a \to b \to b) \to b \to RASeq\ a \to b$$
  $$fromList :: [a] \to RASeq\ a$$

- Add tests on combinations of attributes to decision trees. This should improve the accuracy of the decision trees, but may slow down the induction process considerably.

- Examine the effect of changing language, operating system, and compiler on the benchmarking results Auburn produces. In particular, does the advice of Section 7.1.5 apply to other systems?

- Examine the fairness of the profile distribution.

- Incorporate space information into Auburn's benchmarking procedures. Currently Auburn only measures time.

- Examine the accuracy of Auburn in greater detail, explain the inaccuracies satisfactorily, and make appropriate improvements to Auburn to reduce these inaccuracies.

8.5 The Future

I have a dream that one day we will have a library of implementations of data structures, recommended according to datatype usage. This thesis is one step towards that dream.

Appendix A

Source Code of Implementations

Figures A.1 through A.28 give the implementations of the data structures in Chapter 2 used in the benchmarking of Section 7.1.4.

```
module BankersQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Queue [a] Int [a] Int

empty = Queue [] 0 [] 0

snoc (Queue f lenf r lenr) x = queue f lenf (x:r) (lenr+1)

tail (Queue (x:f) lenf r lenr) = queue f (lenf-1) r lenr

head (Queue (x:f) lenf r lenr) = x

queue f lenf r lenr
   | lenr <= lenf   = Queue f lenf r lenr
   | otherwise      = Queue (f++reverse r) (lenf+lenr) [] 0

isEmpty (Queue [] lenf r lenr) = True
isEmpty _ = False
```

Figure A.1: Bankers queue implementation.

```
module BatchedQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Queue [a] [a]

empty = Queue [] []

snoc (Queue [] _) x = Queue [x] []
snoc (Queue f r) x = Queue f (x:r)

tail (Queue (x:f) r) = queue f r

head (Queue (x:f) r) = x

isEmpty (Queue [] r) = True
isEmpty _ = False

queue [] r = Queue (reverse r) []
queue f  r = Queue f r
```

Figure A.2: Batched queue implementation.

```
module BootstrappedQueue (Queue,empty,snoc,head,tail,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Empty
             | Queue [a] (Queue [a]) Int [a] Int

empty = Empty

snoc Empty x = Queue [x] Empty 1 [] 0
snoc (Queue f m lenFM r lenR) x = queue f m lenFM (x:r) (lenR+1)

tail (Queue (x:f) m lenFM r lenR) = queue f m (lenFM-1) r lenR

head (Queue (x:f) m lenFM r lenR) = x

queue f m lenFM r lenR
   | lenR <= lenFM = checkF f m lenFM r lenR
   | otherwise     = checkF f (snoc m (reverse r)) (lenFM+lenR) [] 0

checkF [] Empty lenFM r lenR = Empty
checkF [] m lenFM r lenR = Queue (head m) (tail m) lenFM r lenR
checkF f m lenFM r lenR = Queue f m lenFM r lenR

isEmpty Empty = True
isEmpty _ = False
```

Figure A.3: Bootstrapped queue implementation.

```
module ImplicitQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data ZeroOrOne a = ZeroInOne | OneInOne a
data OneOrTwo a = OneInTwo a | TwoInTwo a a
data Queue a = Shallow (ZeroOrOne a)
             | Deep (OneOrTwo a) (Queue (OneOrTwo a)) (ZeroOrOne a)

empty = Shallow ZeroInOne

snoc (Shallow ZeroInOne) x = Shallow (OneInOne x)
snoc (Shallow (OneInOne x)) y =
   Deep (TwoInTwo x y) (Shallow ZeroInOne) ZeroInOne
snoc (Deep f m ZeroInOne) x = Deep f m (OneInOne x)
snoc (Deep f m (OneInOne x)) y =
   Deep f (snoc m (TwoInTwo x y)) ZeroInOne

tail (Shallow (OneInOne x)) = Shallow ZeroInOne
tail (Deep (TwoInTwo x y) m r) = Deep (OneInTwo y) m r
tail (Deep (OneInTwo x) (Shallow ZeroInOne) r) = Shallow r
tail (Deep (OneInTwo x) m r) = pull m r

pull (Shallow (OneInOne xy)) r = Deep xy (Shallow ZeroInOne) r
pull (Deep (TwoInTwo xy z) m iR) oR =
   Deep xy (Deep (OneInTwo z) m iR) oR
pull (Deep (OneInTwo xy) (Shallow ZeroInOne) iR) oR =
   Deep xy (Shallow iR) oR
pull (Deep (OneInTwo xy) m iR) oR = Deep xy (pull m iR) oR

head (Shallow (OneInOne x)) = x
head (Deep (OneInTwo x) m r) = x
head (Deep (TwoInTwo x y) m r) = x

isEmpty (Shallow ZeroInOne) = True
isEmpty _ = False
```

Figure A.4: Implicit queue implementation.

```
module MultiheadQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Queue Bool Int Int [a] [a] [a] [a] [a] [a]

empty = Queue False 0 0 [] [] [] [] [] []

snoc (Queue False 0 copy h t lh h' t' hr) x =
   onestep (onestep (Queue True 0 0 h (x:t) h [] [] []))
snoc (Queue False lendiff copy h t lh h' t' hr) x =
   Queue False (lendiff-1) 0 h (x:t) [] [] [] []
snoc (Queue recopy lendiff copy h t lh h' t' hr) x =
   onestep (onestep
      (Queue True (lendiff-1) copy h t lh h' (x:t') hr))

tail (Queue False 0 copy (x:h) t lh h' t' hr) =
   onestep (onestep (Queue True 0 0 h t h [] [] []))
tail (Queue False lendiff copy (x:h) t lh h' t' hr) =
   Queue False (lendiff-1) 0 h t [] [] [] []
tail (Queue recopy lendiff copy h t (x:lh) h' t' hr) =
   onestep (onestep (Queue True lendiff (copy-1) h t lh h' t' hr))

head (Queue False lendiff copy (x:h) t lh h' t' hr) = x
head (Queue recopy lendiff copy h t (x:lh) h' t' hr) = x

onestep q@(Queue False lendiff copy h t lh h' t' hr) = q
onestep (Queue recopy lendiff 0 [] [] lh h' t' hr) =
   Queue False lendiff 0 h' t' [] [] [] []
onestep (Queue recopy lendiff 1 [] [] lh h' t' (x:hr)) =
   Queue False (lendiff+1) 0 (x:h') t' [] [] [] []
onestep (Queue recopy lendiff copy [] [] lh h' t' (x:hr)) =
   Queue True (lendiff+1) (copy-1) [] [] lh (x:h') t' hr
onestep (Queue recopy lendiff copy [] (x:t) lh h' t' hr) =
   Queue True (lendiff+1) copy [] [] lh (x:h') t' hr
onestep (Queue recopy lendiff copy (x:h) (y:t) lh h' t' hr) =
   Queue True (lendiff+1) (copy+1) h t lh (y:h') t' (x:hr)

isEmpty (Queue False lendiff copy (x:h) t lh h' t' hr) = False
isEmpty (Queue recopy lendiff copy h t (x:lh) h' t' hr) = False
isEmpty _ = True
```

Figure A.5: Multihead queue implementation.

```
module PhysicistsQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Queue [a] [a] Int [a] Int

empty = Queue [] [] 0 [] 0

snoc (Queue w f lenF r lenR) x = queue w f lenF (x:r) (lenR+1)

tail (Queue (x:w) f lenF r lenR) = queue w f' (lenF-1) r lenR
   where (x':f') = f

head (Queue (x:w) f lenF r lenR) = x

queue w f lenF r lenR
   | lenR <= lenF = checkW w f lenF r lenR
   | otherwise    = checkW f (f++reverse r) (lenF+lenR) [] 0

checkW [] f lenF r lenR = Queue f f lenF r lenR
checkW w f lenF r lenR = Queue w f lenF r lenR

isEmpty (Queue [] f lenF r lenR) = True
isEmpty _ = False
```

Figure A.6: Physicists queue implementation.

```
module RealTimeQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data Queue a = Queue [a] [a] [a]

empty :: Queue a
empty = Queue [] [] []

snoc :: Queue a -> a -> Queue a
snoc (Queue f r s) x = queue f (x:r) s

tail :: Queue a -> Queue a
tail (Queue (x:f) r s) = queue f r s

head :: Queue a -> a
head (Queue (x:f) r s) = x

queue :: [a] -> [a] -> [a] -> Queue a
queue f r (x:s) = Queue f r s
queue f r [] = Queue f' [] f'
   where f' = rotate f r []

rotate :: [a] -> [a] -> [a] -> [a]
rotate [] (y:r) a = y : a
rotate (x:f) (y:r) a = x : rotate f r (y:a)

isEmpty :: Queue a -> Bool
isEmpty (Queue [] r s) = True
isEmpty _ = False
```

Figure A.7: RealTime queue implementation.

```
module AVLRASeq (RASeq,empty,cons,tail,update,head,isEmpty,lookup)
   where

import Prelude hiding (tail,head,lookup)

data Balance = L | B | R
data RASeq a = Empty
             | Node Balance Int (RASeq a) a (RASeq a)

empty = Empty

cons x xs = case ins xs of (b,t) -> t
   where
   ins Empty = (True,Node B 0 Empty x Empty)
   ins (Node b n l y r) =
      case ins l of
         (False,l') -> (False,Node b (n+1) l' y r)
         (_,l') ->
            case b of
               R -> (False,Node B (n+1) l' y r)
               B -> (True, Node L (n+1) l' y r)
               _ ->
                  case l' of
                     Node b m l' z r' ->
                        (False,Node B m l' z (Node B (n-m) r' y r))

tail xs = case del xs of (b,t) -> t
   where
   del (Node b 0 l x r) = (True,r)
   del (Node b n l x r) =
      case del l of
         (False,l') -> (False,Node b (n-1) l' x r)
         (_,l') ->
            case b of
               L -> (True, Node B (n-1) l' x r)
               B -> (False,Node R (n-1) l' x r)
               _ ->
                  case r of
                     Node R m l'' y r'' ->
                        (True, Node B (n+m) (Node B (n-1) l' x l'')
                                    y r'')
                     Node _ m l'' y r'' ->
                        (False,Node L (n+m) (Node R (n-1) l' x l'')
                                    y r'')
```

Figure A.8: AVL random-access sequence implementation (part I).

```
update (Node b n l x r) i y
   | i == n    = Node b n l y r
   | i < n     = Node b n (update l i y) x r
   | otherwise = Node b n l x (update r (i-n-1) y)

head (Node b 0 l x r) = x
head (Node b n l x r) = head l

isEmpty Empty = True
isEmpty _ = False

lookup (Node b n l x r) i
   | i == n    = x
   | i < n     = lookup l i
   | otherwise = lookup r (i-n-1)
```

Figure A.9: AVL random-access sequence implementation (part II).

```
module AdamsRASeq (RASeq,empty,cons,tail,update,head,isEmpty,lookup)
   where

import Prelude hiding (head,tail,lookup)

data RASeq a = Empty
             | Branch Int Int (RASeq a) a (RASeq a)

empty = Empty

isEmpty Empty = True
isEmpty _ = False

lookup (Branch n nl l x r) i
   | i < nl = lookup l i
   | i == nl = x
   | otherwise = lookup r (i-nl-1)

update (Branch n nl l x r) i y
   | i < nl = Branch n nl (update l i y) x r
   | i == nl = Branch n nl l y r
   | otherwise = Branch n nl l x (update r (i-nl-1) y)

cons x Empty = Branch 1 0 Empty x Empty
cons x (Branch _ _ l y r) = balBranch (cons x l) y r

tail (Branch _ _ Empty y r) = r
tail (Branch _ _ l y r) = balBranch (tail l) y r

head (Branch _ _ Empty y r) = y
head (Branch _ _ l y r) = head l
```

Figure A.10: Adams random-access sequence implementation (part I).

```
branch l x r = Branch (1 + sizeL + size r) sizeL l x r
   where sizeL = size l

singleL l x (Branch _ _ rl y rr) = branch (branch l x rl) y rr

doubleL l x (Branch _ _ (Branch _ _ rll y rlr) z rr) =
   branch (branch l x rll) y (branch rlr z rr)

singleR (Branch _ _ ll x lr) y r = branch ll x (branch lr y r)

doubleR (Branch _ _ ll x (Branch _ _ lrl y lrr)) z r =
   branch (branch ll x lrl) y (branch lrr z r)

sigma = 5

size Empty = 0
size (Branch n _ _ _ _) = n

balBranch l x r
   | sizeL + sizeR < 2 = branch l x r
   | sizeR > sigma * sizeL =
      let (Branch _ _ rl _ rr) = r
      in if size rl < size rr
         then singleL l x r
         else doubleL l x r
   | sizeL > sigma * sizeR =
      let (Branch _ _ ll _ lr) = l
      in if size lr < size ll
         then singleR l x r
         else doubleR l x r
   | otherwise = branch l x r
   where sizeL = size l
         sizeR = size r
```

Figure A.11: Adams random-access sequence implementation (part II).

```
module BraunRASeq (RASeq,empty,cons,tail,update,head,isEmpty,lookup)
   where

import Prelude hiding (tail,head,lookup)

data RASeq a = Empty
             | Node (RASeq a) a (RASeq a)

empty = Empty

cons x Empty = Node Empty x Empty
cons x (Node l y r) = Node (cons y r) x l

tail (Node l x r) = join l r
   where join Empty t = Empty
         join (Node l x r) t = Node t x (join l r)

update (Node l x r) 0 y = Node l y r
update (Node l x r) n y
   | n `mod` 2 == 0 = Node l x (update r ((n `div` 2)-1) y)
   | otherwise      = Node (update l ((n-1) `div` 2) y) x r

head (Node l x r) = x

isEmpty Empty = True
isEmpty t = False

lookup (Node l x r) 0 = x
lookup (Node l x r) n | n `mod` 2 == 0 = lookup r ((n `div` 2)-1)
                      | otherwise      = lookup l ((n-1) `div` 2)
```

Figure A.12: Braun random-access sequence implementation.

```
module ElevatorRASeq (RASeq,empty,cons,tail,update,head,isEmpty,lookup)
   where

import Prelude hiding (tail,head,lookup)

data RASeq a = Floor Int [a] (RASeq a)

floorSep = 5

empty = Floor 0 [] empty

cons x s@(Floor n xs yss)
   | n < floorSep = Floor (n+1) (x:xs) yss
   | otherwise = Floor 1 [x] s

tail (Floor n (x:xs) yss)
   | n > 1 = Floor (n-1) xs yss
   | otherwise = yss

update (Floor n xs yss) i y
   | n <= i = Floor n xs (update yss (i-n) y)
   | otherwise = Floor n (updateList xs i y) yss
updateList (x:xs) 0 y = y:xs
updateList (x:xs) n y = x:updateList xs (n-1) y

head (Floor n (x:xs) yss) = x

isEmpty (Floor n [] yss) = True
isEmpty _ = False

lookup (Floor n xs yss) i
   | n <= i = lookup yss (i-n)
   | otherwise = lookupList xs i
lookupList (x:xs) 0 = x
lookupList (x:xs) n = lookupList xs (n-1)
```

Figure A.13: Elevator random-access sequence implementation.

```
module NaiveRASeq (RASeq,empty,cons,tail,update,head,isEmpty,lookup)
   where

import Prelude hiding (tail,head,lookup)

newtype RASeq a = RASeq [a]

empty = RASeq []

cons x (RASeq xs) = RASeq (x:xs)

tail (RASeq (x:xs)) = RASeq xs

update (RASeq xs) n y = RASeq (updateList xs n y)
updateList (x:xs) 0 y = y : xs
updateList (x:xs) n y = x : updateList xs (n-1) y

head (RASeq (x:xs)) = x

isEmpty (RASeq []) = True
isEmpty _ = False

lookup (RASeq xs) i = xs !! i
```

Figure A.14: Naive random-access sequence implementation.

```
module SkewBinRASeq (RASeq,empty,cons,tail,update,head,
                     isEmpty,lookup) where

import Prelude hiding (tail,head,lookup)

data RATree a = Leaf a | Node (RATree a) a (RATree a)
data RASeq a = Nil
             | Root Int (RATree a) (RASeq a)

empty = Nil

cons x (Root size1 l (Root size2 r rest))
   | size1 == size2 = Root (1+size1+size2) (Node l x r) rest
cons x xs = Root 1 (Leaf x) xs

tail (Root size (Leaf x) rest) = rest
tail (Root size (Node l x r) rest) =
   Root size' l (Root size' r rest)
   where size' = size `div` 2

update (Root size t rest) i y
   | i < size  = Root size (treeUpdate size t i y) rest
   | otherwise = Root size t (update rest (i-size) y)

treeUpdate size (Leaf x) 0 y = Leaf y
treeUpdate size (Node l x r) 0 y = Node l y r
treeUpdate size (Node l x r) i y
    | i <= size' = Node (treeUpdate size' l (i-1) y) x r
    | otherwise  = Node l x (treeUpdate size' r (i-1-size') y)
    where size' = size `div` 2

head (Root size (Leaf x) rest) = x
head (Root size (Node l x r) rest) = x

isEmpty Nil = True
isEmpty _ = False

lookup (Root size t rest) i
   | i < size  = treeLookup size t i
   | otherwise = lookup rest (i-size)

treeLookup size (Leaf x) 0 = x
treeLookup size (Node l x r) 0 = x
treeLookup size (Node l x r) i
    | i <= size' = treeLookup size' l (i-1)
    | otherwise  = treeLookup size' r (i-1-size')
    where size' = size `div` 2
```

Figure A.15: SkewBin random-access sequence implementation.

```
module SlowdownRASeq (RASeq,empty,cons,tail,update,head,
                      isEmpty,lookup) where

import Prelude hiding (tail,head,lookup)

data RASeq a = RedOrGreen (Prefix (Pairs a)) (RASeq a)
             | Yellows [Prefix (Pairs a)] (RASeq a)
             | Deepest (Prefix (Pairs a))
data Pairs a = Elem a | Pair (Pairs a) (Pairs a)
data Prefix a = Zero | One a | Two a a | Three a a a | Four a a a a

pcons a Zero = One a
pcons a (One b) = Two a b
pcons a (Two b c) = Three a b c
pcons a (Three b c d) = Four a b c d

phead (One a) = a
phead (Two a b) = a
phead (Three a b c) = a
phead (Four a b c d) = a

ptail (One a) = Zero
ptail (Two a b) = One b
ptail (Three a b c) = Two b c
ptail (Four a b c d) = Three b c d

inPrefix size p i = i < plength size p

primcons x (Deepest p) = Deepest (pcons x p)
primcons x (RedOrGreen p (Yellows ps rest)) =
   Yellows (pcons x p : ps) rest
primcons x (RedOrGreen p rest) = Yellows [pcons x p] rest
primcons x (Yellows [p] rest) = RedOrGreen (pcons x p) rest
primcons x (Yellows (p:ps) rest) =
   RedOrGreen (pcons x p) (Yellows ps rest)

primhead (Deepest p) = phead p
primhead (RedOrGreen p rest) = phead p
primhead (Yellows (p:ps) rest) = phead p

primtail (Deepest p) = Deepest (ptail p)
primtail (RedOrGreen p (Yellows ps rest)) =
   Yellows (ptail p : ps) rest
primtail (RedOrGreen p rest) = Yellows [ptail p] rest
primtail (Yellows [p] rest) = RedOrGreen (ptail p) rest
primtail (Yellows (p:ps) rest) =
   RedOrGreen (ptail p) (Yellows ps rest)
```

Figure A.16: Slowdown random-access sequence implementation (part I).

```
fix (Deepest (Four a b c d)) =
   RedOrGreen (Two a b) (Deepest (One (Pair c d)))
fix (Yellows ps rest) = Yellows ps (fix rest)
fix (RedOrGreen Zero (Deepest Zero)) = Deepest Zero
fix (RedOrGreen Zero rest) = RedOrGreen (Two a b) (primtail rest)
   where Pair a b = primhead rest
fix (RedOrGreen (Four a b c d) rest) =
   RedOrGreen (Two a b) (primcons (Pair c d) rest)
fix xs = xs

empty = Deepest Zero

update xs i x = update' 1 xs i x

update' size (Deepest p) i x = Deepest (pupdate size p i x)
update' size (RedOrGreen p rest) i x
   | inPrefix size p i =
      RedOrGreen (pupdate size p i x) rest
   | otherwise =
      RedOrGreen p (update' (size*2) rest (i - plength size p) x)
update' size (Yellows [] rest) i x =
   Yellows [] (update' size rest i x)
update' size (Yellows (p:ps) rest) i x
   | inPrefix size p i = Yellows (pupdate size p i x:ps) rest
   | otherwise         = Yellows (p:ps') rest'
   where (Yellows ps' rest') = update' (size*2) (Yellows ps rest)
                                       (i - plength size p) x

pupdate size (One a) i x = One (pupdate' a i x (size `div` 2))
pupdate size (Two a b) i x
   | i < size  = Two (pupdate' a i x (size `div` 2)) b
   | otherwise = Two a (pupdate' b (i - size) x (size `div` 2))
pupdate size (Three a b c) i x
   | i < size   =
      Three (pupdate' a i x (size `div` 2)) b c
   | i < size*2 =
      Three a (pupdate' b (i - size) x (size `div` 2)) c
   | otherwise  =
      Three a b (pupdate' c (i - size*2) x (size `div` 2))
pupdate size (Four a b c d) i x
   | i < size   =
      Four (pupdate' a i x (size `div` 2)) b c d
   | i < size*2 =
      Four a (pupdate' b (i - size) x (size `div` 2)) c d
   | i < size*3 =
      Four a b (pupdate' c (i - size*2) x (size `div` 2)) d
   | otherwise  =
      Four a b c (pupdate' d (i - size*3) x (size `div` 2))
```

Figure A.17: Slowdown random-access sequence implementation (part II).

```
pupdate' (Elem a) 0 x mid = Elem x
pupdate' (Pair xs ys) i x mid
   | i < mid   = Pair (pupdate' xs i x (mid `div` 2)) ys
   | otherwise = Pair xs (pupdate' ys (i-mid) x (mid `div` 2))

cons x xs = fix (primcons (Elem x) xs)
head xs = case primhead xs of Elem x -> x
tail xs = fix (primtail xs)

isEmpty (Deepest Zero) = True
isEmpty _ = False

lookup xs i = lookup' 1 xs i

lookup' size (Deepest p) i = plookup size p i
lookup' size (RedOrGreen p rest) i
   | inPrefix size p i = plookup size p i
   | otherwise         = lookup' (size*2) rest (i - plength size p)
lookup' size (Yellows [] rest) i = lookup' size rest i
lookup' size (Yellows (p:ps) rest) i
   | inPrefix size p i = plookup size p i
   | otherwise         =
      lookup' (size*2) (Yellows ps rest) (i-plength size p)

plength size Zero = 0
plength size (One a) = size
plength size (Two a b) = size*2
plength size (Three a b c) = size*3
plength size (Four a b c d) = size*4

plookup size (One a) i = plookup' a i (size `div` 2)
plookup size (Two a b) i
   | i < size  = plookup' a i (size `div` 2)
   | otherwise = plookup' b (i - size) (size `div` 2)
plookup size (Three a b c) i
   | i < size   = plookup' a i (size `div` 2)
   | i < size*2 = plookup' b (i - size) (size `div` 2)
   | otherwise  = plookup' c (i - size*2) (size `div` 2)
plookup size (Four a b c d) i
   | i < size   = plookup' a i (size `div` 2)
   | i < size*2 = plookup' b (i - size) (size `div` 2)
   | i < size*3 = plookup' c (i - size*2) (size `div` 2)
   | otherwise  = plookup' d (i - size*3) (size `div` 2)

plookup' (Elem a) 0 mid = a
plookup' (Pair xs ys) i mid
   | i < mid   = plookup' xs i (mid `div` 2)
   | otherwise = plookup' ys (i-mid) (mid `div` 2)
```

Figure A.18: Slowdown random-access sequence implementation (part III).

```
module ThreadSkewBinRASeq (RASeq,empty,cons,tail,update,head,
                           isEmpty,lookup) where

import Prelude hiding (tail,head,lookup)

data RASeq a = Empty
             | Cons a (RASeq a)
             | Node a (RASeq a) Int (RASeq a)

empty = Empty

lookup (Cons x xs) 0 = x
lookup (Cons x xs) i = lookup xs (i-1)
lookup (Node x xs r xs1) 0 = x
lookup (Node x xs r xs1) i
   | i < r     = lookup xs (i-1)
   | otherwise = lookup xs1 (i-r)

update (Cons x xs) 0 y = Cons y xs
update (Cons x xs) i y = Cons x (update xs (i-1) y)
update (Node x xs r xs1) 0 y = Node y xs r xs1
update (Node x xs r xs1) i y =
   case update xs (i-1) y of
      xs@(Cons _ (Cons _ xs')) -> Node x xs 3 xs'
      xs@(Node _ _ _ (Node _ _ _ xs')) -> Node x xs r xs'

cons x xs@(Node x1 xs1 r1 (Node x2 xs2 r2 xs3))
   | r1 == r2 = Node x xs (1+r1+r2) xs3
cons x xs@(Cons _ (Cons _ xs')) = Node x xs 3 xs'
cons x xs = Cons x xs

head (Cons x xs) = x
head (Node x xs r xs1) = x

isEmpty Empty = True
isEmpty xs = False

tail (Cons x xs) = xs
tail (Node x xs r xs1) = xs
```

Figure A.19: ThreadSkewBin random-access sequence implementation.

```
module BinomialHeap (Heap,empty,isEmpty,insert,merge,findMin,
                     deleteMin) where

data Ord a => Tree a = Node Int a [Tree a]
newtype Ord a => Heap a = Heap [Tree a]

rank (Node r x c) = r

root (Node r x c) = x

link t1@(Node r x1 c1) t2@(Node _ x2 c2) =
  if x1 <= x2 then Node (r+1) x1 (t2:c1)
  else Node (r+1) x2 (t1:c2)

insTree t [] = [t]
insTree t ts@(t':ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'

mrg ts1 [] = ts1
mrg [] ts2 = ts2
mrg ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1 : mrg ts1' ts2
    | rank t2 < rank t1 = t2 : mrg ts1 ts2'
    | otherwise = insTree (link t1 t2) (mrg ts1' ts2')

removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
    if root t < root t' then (t, ts) else (t', t : ts')
    where (t', ts') = removeMinTree ts

empty = Heap []

isEmpty (Heap ts) = null ts

insert x (Heap ts) = Heap (insTree (Node 0 x []) ts)

merge (Heap ts1) (Heap ts2) = Heap (mrg ts1 ts2)

findMin (Heap ts) = root t
    where (t, _) = removeMinTree ts

deleteMin (Heap ts) = Heap (mrg (reverse ts1) ts2)
    where (Node _ x ts1, ts2) = removeMinTree ts
```

Figure A.20: Binomial heap implementation.

```
module BootSkewBinHeap (Heap,empty,isEmpty,insert,merge,findMin,
                        deleteMin) where

data Ord a => Heap a = Empty
                     | Root a (OldHeap (Heap a))

instance Ord a => Eq (Heap a) where
   Empty == Empty = True
   (Root x _) == (Root y _) = x == y
   _ == _ = False

instance Ord a => Ord (Heap a) where
   compare (Root x _) (Root y _) = compare x y

empty = Empty

isEmpty Empty = True
isEmpty _ = False

merge p Empty = p
merge Empty q = q
merge (Root x p) (Root y q)
   | x <= y    = Root x (oldInsert (Root y q) p)
   | otherwise = Root y (oldInsert (Root x p) q)

insert x Empty = Root x oldEmpty
insert x p = merge (Root x oldEmpty) p

findMin (Root x _) = x

deleteMin (Root x p)
   | oldIsEmpty p = Empty
   | otherwise       = Root y (oldMerge q1 q2)
   where Root y q1 = oldFindMin p
         q2        = oldDeleteMin p
```

Figure A.21: BootSkewBin heap implementation (part I).

```
newtype Ord a => OldHeap a = OldHeap [Tree a]
data Ord a => Tree a = Node Int a [a] [Tree a]

rank (Node r x xs c) = r
root (Node r x xs c) = x

link t1@(Node r x1 xs1 c1) t2@(Node _ x2 xs2 c2) =
    if x1 <= x2 then Node (r+1) x1 xs1 (t2:c1)
    else Node (r+1) x2 xs2 (t1:c2)
skewLink x t1 t2 =
    let Node r y ys c = link t1 t2
    in if x <= y then Node r x (y:ys) c else Node r y (x:ys) c

insTree t [] = [t]
insTree t ts@(t':ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'

mrg ts1 [] = ts1
mrg [] ts2 = ts2
mrg ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1 : mrg ts1' ts2
    | rank t2 < rank t1 = t2 : mrg ts1 ts2'
    | otherwise = insTree (link t1 t2) (mrg ts1' ts2')

normalize [] = []
normalize (t:ts) = insTree t ts

removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
    if root t < root t' then (t, ts) else (t', t : ts')
    where (t', ts') = removeMinTree ts

oldEmpty = OldHeap []

oldIsEmpty (OldHeap ts) = null ts

oldInsert x (OldHeap (t1:t2:ts)) | rank t1 == rank t2 =
   OldHeap (skewLink x t1 t2 : ts)
oldInsert x (OldHeap ts) = OldHeap (Node 0 x [] [] : ts)

oldMerge (OldHeap ts1) (OldHeap ts2) =
   OldHeap (mrg (normalize ts1) (normalize ts2))

oldFindMin (OldHeap ts) = root t
   where (t, _) = removeMinTree ts

oldDeleteMin (OldHeap ts) = foldr oldInsert (OldHeap ts') xs
   where (Node _ x xs ts1, ts2) = removeMinTree ts
         ts' = mrg (reverse ts1) (normalize ts2)
```

Figure A.22: BootSkewBin heap implementation (part II).

```
module LeftistHeap (Heap,empty,isEmpty,insert,merge,findMin,
                    deleteMin) where

data Heap a = Empty
            | Node Int (Heap a) a (Heap a)

empty = Empty

isEmpty Empty = True
isEmpty _ = False

insert x Empty = Node 1 Empty x Empty
insert x h@(Node s l y r)
   | x <= y    = Node 1 h x Empty
   | otherwise = node l y (insert x r)

findMin (Node _ _ x _) = x

deleteMin (Node s l x r) = merge l r

merge h Empty = h
merge Empty h = h
merge h1@(Node s1 l1 x1 r1) h2@(Node s2 l2 x2 r2)
   | x1 <= x2  = node l1 x1 (merge r1 h2)
   | otherwise = node l2 x2 (merge r2 h1)

node h x Empty = Node 1 h x Empty
node Empty x h = Node 1 h x Empty
node h1@(Node s1 _ _ _) x h2@(Node s2 _ _ _)
   | s1 <= s2  = Node (s1+1) h2 x h1
   | otherwise = Node (s2+1) h1 x h2

fromList = foldr insert empty
```

Figure A.23: Leftist heap implementation.

```
module NaiveHeap (Heap,empty,isEmpty,insert,merge,findMin,
                  deleteMin) where

newtype Ord a => Heap a = Heap [a]

empty = Heap []

isEmpty (Heap []) = True
isEmpty _ = False

insert w (Heap h) = Heap (insert' w h)
insert' w [] = [w]
insert' w vl@(v:vs) | w <= v    = w : vl
                    | otherwise = v : insert' w vs

findMin (Heap (v:vs)) = v

deleteMin (Heap (v:vs)) = Heap vs

merge (Heap ws) (Heap vs) = Heap (merge' ws vs)
merge' [] vs = vs
merge' ws [] = ws
merge' wl@(w:ws) vl@(v:vs)
   | w <= v    = w : merge' ws vl
   | otherwise = v : merge' wl vs
```

Figure A.24: Naive heap implementation.

```
module PairingHeap (Heap,empty,isEmpty,insert,merge,findMin,
                    deleteMin) where

data Heap a = Empty
            | Node a [Heap a]

empty = Empty

isEmpty Empty = True
isEmpty _ = False

insert x Empty = Node x []
insert x h2@(Node x2 hs2)
   | x <= x2   = Node x [h2]
   | otherwise = Node x2 (Node x []:hs2)

findMin (Node x _) = x

deleteMin (Node _ hs) = mergePairs hs

merge h Empty = h
merge Empty h = h
merge h1@(Node x1 hs1) h2@(Node x2 hs2)
   | x1 <= x2  = Node x1 (h2:hs1)
   | otherwise = Node x2 (h1:hs2)

mergePairs []  = Empty
mergePairs [a] = a
mergePairs (a:b:hs) = merge (merge a b) (mergePairs hs)
```

Figure A.25: Pairing heap implementation.

```
module SkewBinHeap (Heap,empty,isEmpty,insert,merge,findMin,
                    deleteMin) where

newtype Ord a => Heap a = Heap [Tree a]
data Ord a => Tree a = Node Int a [a] [Tree a]

rank (Node r x xs c) = r

root (Node r x xs c) = x

link t1@(Node r x1 xs1 c1) t2@(Node _ x2 xs2 c2) =
    if x1 <= x2 then Node (r+1) x1 xs1 (t2:c1)
    else Node (r+1) x2 xs2 (t1:c2)

skewLink x t1 t2 =
    let Node r y ys c = link t1 t2
    in if x <= y then Node r x (y:ys) c else Node r y (x:ys) c

insTree t [] = [t]
insTree t ts@(t':ts') =
  if rank t < rank t' then t:ts else insTree (link t t') ts'

mrg ts1 [] = ts1
mrg [] ts2 = ts2
mrg ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1 : mrg ts1' ts2
    | rank t2 < rank t1 = t2 : mrg ts1 ts2'
    | otherwise = insTree (link t1 t2) (mrg ts1' ts2')

normalize [] = []
normalize (t:ts) = insTree t ts

removeMinTree [t] = (t, [])
removeMinTree (t:ts) =
    if root t < root t' then (t, ts) else (t', t : ts')
    where (t', ts') = removeMinTree ts
```

Figure A.26: SkewBin heap implementation (part I).

```
empty = Heap []

isEmpty (Heap ts) = null ts

insert x (Heap (t1:t2:ts)) | rank t1 == rank t2 =
   Heap (skewLink x t1 t2 : ts)
insert x (Heap ts) = Heap (Node 0 x [] [] : ts)

merge (Heap ts1) (Heap ts2) =
   Heap (mrg (normalize ts1) (normalize ts2))

findMin (Heap ts) = root t
   where (t, _) = removeMinTree ts

deleteMin (Heap ts) = foldr insert (Heap ts') xs
   where (Node _ x xs ts1, ts2) = removeMinTree ts
         ts' = mrg (reverse ts1) (normalize ts2)
```

Figure A.27: SkewBin heap implementation (part II).

```
module SplayHeap (Heap,empty,isEmpty,insert,merge,findMin,
                  deleteMin) where

data Heap a = Empty
            | Node (Heap a) a (Heap a)

empty = Empty

isEmpty Empty = True
isEmpty _ = False

insert x h = Node l x r
   where (l,r) = partition x h

partition pivot Empty = (Empty,Empty)
partition pivot h@(Node l x r)
   | x <= pivot =
      case r of
         Empty -> (h,Empty)
         Node rl y rr ->
            if y <= pivot
            then let (small,big) = partition pivot rr
                 in (Node (Node l x rl) y small,big)
            else let (small,big) = partition pivot rl
                 in (Node l x small,Node big y rr)
   | otherwise =
      case l of
         Empty -> (Empty,h)
         Node ll y lr ->
            if y <= pivot
            then let (small,big) = partition pivot lr
                 in (Node ll y small,Node big x r)
            else let (small,big) = partition pivot ll
                 in (small,Node big y (Node lr x r))

findMin (Node Empty x r) = x
findMin (Node l x r) = findMin l

deleteMin (Node Empty x r) = r
deleteMin (Node (Node Empty x lr) y r) = Node lr y r
deleteMin (Node (Node ll x lr) y r) =
   Node (deleteMin ll) x (Node lr y r)

merge Empty h = h
merge (Node l x r) h = Node (merge small l) x (merge big r)
   where (small,big) = partition x h
```

Figure A.28: Splay heap implementation.

Appendix B

Modifications to Implementations

Figures B.1 through B.25 give the modifications of Tables 7.1, 7.2 and 7.3, by showing the output of the UNIX `diff` command. Figure B.7 gives the modification Multihead in the form of the modified implementation, as almost all of the code is modified.

```
5c5
< data Queue a = Queue [a] Int [a] Int
---
> data Queue a = Queue [a] !Int [a] !Int
```

Figure B.1: Bankers queue modification.

```
11c11,12
< snoc (Queue f r) x = queue f (x:r)
---
> snoc (Queue [] _) x = Queue [x] []
> snoc (Queue f r) x = Queue f (x:r)
```

Figure B.2: Batched queue modification.

```
27c27,28
< checkF [] m lenFM r lenR = Queue (head m) (tail m) lenFM r lenR
---
> checkF [] (Queue (iX:iF) iM iLenFM iR iLenR) lenFM r lenR =
>     Queue iX (queue iF iM (iLenFM-1) iR iLenR) lenFM r lenR
```

Figure B.3: Bootstrapped–1 queue modification.

```
27c27
< checkF [] m lenFM r lenR = Queue (head m) (tail m) lenFM r lenR
---
> checkF [] m lenFM r lenR = pull m lenFM r lenR
28a29,32
>
> pull (Queue (iX:iF) iM iLenFM iR iLenR) lenFM r lenR =
>    Queue iX (queue iF iM (iLenFM-1) iR iLenR) lenFM r lenR
```

Figure B.4: Bootstrapped–2 queue modification.

```
8c8
<            | Deep (OneOrTwo a) (Queue (a,a)) (ZeroOrOne a)
---
>            | Deep (OneOrTwo a) (Queue (OneOrTwo a)) (ZeroOrOne a)
18c18
< snoc (Deep f m (OneInOne x)) y =
<    Deep f (snoc m (x,y)) ZeroInOne
---
> snoc (Deep f m (OneInOne x)) y =
>    Deep f (snoc m (TwoInTwo x y)) ZeroInOne
24,25c24
< tail (Deep (OneInTwo x) m r) = Deep (TwoInTwo y z) (tail m) r
<    where (y,z) = head m
---
> tail (Deep (OneInTwo x) m r) = Deep (head m) (tail m) r
```

Figure B.5: Implicit–1 queue modification.

```
24,25c24,32
< tail (Deep (OneInTwo x) m r) = Deep (TwoInTwo y z) (tail m) r
<    where (y,z) = head m
---
> tail (Deep (OneInTwo x) m r) = pull m r
>
> pull (Shallow (OneInOne (x,y))) r =
>    Deep (TwoInTwo x y) (Shallow ZeroInOne) r
> pull (Deep (TwoInTwo (x,y) z) m iR) oR =
>    Deep (TwoInTwo x y) (Deep (OneInTwo z) m iR) oR
> pull (Deep (OneInTwo (x,y)) (Shallow ZeroInOne) iR) oR =
>    Deep (TwoInTwo x y) (Shallow iR) oR
> pull (Deep (OneInTwo (x,y)) m iR) oR =
>    Deep (TwoInTwo x y) (pull m iR) oR
```

Figure B.6: Implicit–2 queue modification.

```
module MultiheadQueue (Queue,empty,snoc,tail,head,isEmpty) where

import Prelude hiding (head,tail)

data RotationState a =
      Idle
    | Reversing Int [a] [a] [a] [a]
    | Appending Int [a] [a]
    | Done [a]

data Queue a = Queue Int [a] (RotationState a) Int [a]

exec (Reversing ok (x:f) f' (y:r) r') =
   Reversing (ok+1) f (x:f') r (y:r')
exec (Reversing ok [] f' [y] r') = Appending ok f' (y:r')
exec (Appending 0 f' r') = Done r'
exec (Appending ok (x:f') r') = Appending (ok-1) f' (x:r')
exec state = state

invalidate (Reversing ok f f' r r') = Reversing (ok-1) f f' r r'
invalidate (Appending 0 f' (x:r')) = Done r'
invalidate (Appending ok f' r') = Appending (ok-1) f' r'
invalidate state = state

exec2 lenf f state lenr r =
    case exec (exec state) of
      Done newf -> Queue lenf newf Idle lenr r
      newstate -> Queue lenf f newstate lenr r

check lenf f state lenr r =
    if lenr <= lenf then exec2 lenf f state lenr r
    else let newstate = Reversing 0 f [] r []
         in exec2 (lenf+lenr) f newstate 0 []

empty = Queue 0 [] Idle 0 []

isEmpty (Queue lenf f state lenr r) = (lenf == 0)

snoc (Queue lenf f state lenr r) x =
   check lenf f state (lenr+1) (x:r)

head (Queue _ (x:f') _ _ _) = x

tail (Queue lenf (x:f') state lenr r) =
    check (lenf-1) f' (invalidate state) lenr r
```

Figure B.7: Multihead queue modification.

```
5c5
< data Queue a = Queue [a] [a] Int [a] Int
---
> data Queue a = Queue [a] [a] !Int [a] !Int
```

Figure B.8: Physicists queue modification.

```
41,44c41,45
< update (Node b n l x r) i y
<    | i == n    = Node b n l y r
<    | i < n     = Node b n (update l i y) x r
<    | otherwise = Node b n l x (update r (i-n-1) y)
---
> update (Node b n l x r) i y =
>    case compare i n of
>       EQ -> Node b n l y r
>       LT -> Node b n (update l i y) x r
>       _  -> Node b n l x (update r (i-n-1) y)
55,58c56,60
< lookup (Node b n l x r) i
<    | i == n    = x
<    | i < n     = lookup l i
<    | otherwise = lookup r (i-n-1)
---
> lookup (Node b n l x r) i =
>    case compare i n of
>       EQ -> x
>       LT -> lookup l i
>       _  -> lookup r (i-n-1)
```

Figure B.9: AVL–1 random-access sequence modification.

```
41,44c41,45
< update (Node b n l x r) i y
<     | i == n    = Node b n l y r
<     | i < n     = Node b n (update l i y) x r
<     | otherwise = Node b n l x (update r (i-n-1) y)
---
> update (Node b n l x r) i y =
>     case compare i n of
>       LT -> Node b n (update l i y) x r
>       EQ -> Node b n l y r
>       _  -> Node b n l x (update r (i-n-1) y)
55,58c56,60
< lookup (Node b n l x r) i
<     | i == n    = x
<     | i < n     = lookup l i
<     | otherwise = lookup r (i-n-1)
---
> lookup (Node b n l x r) i =
>     case compare i n of
>       LT -> lookup l i
>       EQ -> x
>       _  -> lookup r (i-n-1)
```

Figure B.10: AVL–2 random-access sequence modification.

```
42d41
<     | i == n    = Node b n l y r
43a43
>     | i == n    = Node b n l y r
56d55
<     | i == n    = x
57a57
>     | i == n    = x
```

Figure B.11: AVL–3 random-access sequence modification.

```
<       case (ins l,b) of
<          ((False,l'),b) -> (False,Node b (n+1) l' y r)
<          ((True,l'),R) ->  (False,Node B (n+1) l' y r)
<          ((True,l'),B) ->  (True, Node L (n+1) l' y r)
<          ((True,Node b m l' z r'),L) ->
<             (False,Node B m l' z (Node B (n-m) r' y r))
---
>       case ins l of
>          (False,l') -> (False,Node b (n+1) l' y r)
>          (_,l') ->
>             case b of
>                R -> (False,Node B (n+1) l' y r)
>                B -> (True, Node L (n+1) l' y r)
>                _ ->
>                   case l' of
>                      Node b m l' z r' ->
>                         (False,Node B m l' z
>                                    (Node B (n-m) r' y r))
29,38c33,44
<       case (del l,b) of
<          ((False,l'),b) -> (False,Node b (n-1) l' x r)
<          ((True, l'),L) -> (True, Node B (n-1) l' x r)
<          ((True, l'),B) -> (False,Node R (n-1) l' x r)
<          ((True, l'),R) ->
<             case r of
<                Node R m l'' y r'' ->
<                   (True, Node B (n+m) (Node B (n-1) l' x l'')
<                               y r'')
<                Node B m l'' y r'' ->
<                   (False,Node L (n+m) (Node R (n-1) l' x l'')
<                               y r'')
---
>       case del l of
>          (False,l') -> (False,Node b (n-1) l' x r)
>          (_,l') ->
>             case b of
>                L -> (True, Node B (n-1) l' x r)
>                B -> (False,Node R (n-1) l' x r)
>                _ ->
>                   case r of
>                      Node R m l'' y r'' ->
>                         (True, Node B (n+m)
>                                    (Node B (n-1) l' x l'')
>                                    y r'')
>                      Node _ m l'' y r'' ->
>                         (False,Node L (n+m)
>                                    (Node R (n-1) l' x l'')
>                                    y r'')
```

Figure B.12: AVL–4 random-access sequence modification.

```
47a48,50
> alpha :: Int
> alpha = 2
>
57c60
<       in if size rl < size rr
---
>       in if size rl < size rr * alpha
62c65
<       in if size lr < size ll
---
>       in if size lr < size ll * alpha
```

Figure B.13: Adams random-access sequence modification.

```
16,17c16,18
< tail (Node Empty x t) = Empty
< tail (Node l x r) = Node r (head l) (tail l)
---
> tail (Node l x r) = join l r
>    where join Empty t = Empty
>          join (Node l x r) t = Node t x (join l r)
```

Figure B.14: Braun random-access sequence modification.

```
8c8
< floorSep = 10
---
> floorSep = 3
```

Figure B.15: Elevator–1 random-access sequence modification.

```
8c8
< floorSep = 10
---
> floorSep = 5
```

Figure B.16: Elevator–2 random-access sequence modification.

```
8c8
< floorSep = 10
---
> floorSep = 25
```

Figure B.17: Elevator–3 random-access sequence modification.

```
7c7
<               | Root Int (RATree a) (RASeq a)
---
>               | Root !Int (RATree a) (RASeq a)
```

Figure B.18: SkewBin random-access sequence modification.

```
6a7
>               | Cons a (RASeq a)
12a14,15
> lookup (Cons x xs) 0 = x
> lookup (Cons x xs) i = lookup xs (i-1)
18a22,23
> update (Cons x xs) 0 y = Cons y xs
> update (Cons x xs) i y = Cons x (update xs (i-1) y)
20c25,28
< update (Node x xs r xs1) i y = cons x (update xs (i-1) y)
---
> update (Node x xs r xs1) i y =
>     case update xs (i-1) y of
>       xs@(Cons _ (Cons _ xs')) -> Node x xs 3 xs'
>       xs@(Node _ _ _ (Node _ _ _ xs')) -> Node x xs r xs'
25c33,34
< cons x xs = Node x xs 1 xs
---
> cons x xs@(Cons _ (Cons _ xs')) = Node x xs 3 xs'
> cons x xs = Cons x xs
27a37
> head (Cons x xs) = x
34a45
> tail (Cons x xs) = xs
```

Figure B.19: ThreadSkewBin random-access sequence modification.

```
3c3
< data Ord a => Tree a = Node Int a [Tree a]
---
> data Ord a => Tree a = Node !Int a [Tree a]
```

Figure B.20: Binomial heap modification.

```
45c45
< data Ord a => Tree a = Node Int a [a] [Tree a]
---
> data Ord a => Tree a = Node !Int a [a] [Tree a]
```

Figure B.21: BootSkewBin heap modification.

```
14c14,17
< insert x h = merge (Node 1 Empty x Empty) h
---
> insert x Empty = Node 1 Empty x Empty
> insert x h@(Node s l y r)
>     | x <= y    = Node 1 h x Empty
>     | otherwise = node l y (insert x r)
```

Figure B.22: Leftist heap modification.

```
26c26
<     | x1 <= x2  = Node x1 (h2:hs1)
---
>     | x1 < x2   = Node x1 (h2:hs1)
```

Figure B.23: Pairing–1 heap modification.

```
14c14,17
< insert x h = merge (Node x []) h
---
> insert x Empty = Node x []
> insert x h2@(Node x2 hs2)
>     | x <= x2   = Node x [h2]
>     | otherwise = Node x2 (Node x []:hs2)
```

Figure B.24: Pairing–2 heap modification.

```
4c4
< data Ord a => Tree a = Node Int a [a] [Tree a]
---
> data Ord a => Tree a = Node !Int a [a] [Tree a]
```

Figure B.25: SkewBin heap modification.

Appendix C

Auburn Reference

There are various executables produced by Auburn, with various flags for modifying their behaviour. Rather than give a lengthy explanation of these, we just quote the help information for each executable, that is, the output they produce when supplied with the flag `-h`. Here is a list of the help pages in order: `auburn`, a DUG manager, a benchmarker, `auburnExp`, `makeDugs`, `evalDugs`, `processTimes`, `cleanDugs`.

```
Usage: auburn [options] sigfile[.sig]
Options:

   -c IMP-MOD1[.hs] IMP-MOD2[.hs] ... IMP-MODn[.hs]
       Write a signature of the common operations exported by the
       the implementation modules IMP-MOD1, IMP-MOD2, ... IMP-MODn.
   -sT Write a trivial shadow data structure.
   -sS Write a best guess at size-based shadow data structure.
   -m  Write a dug manager.
   -e IMP-MOD1 IMP-MOD2 ... IMP-MODn
       Write a dug evaluator for each implementation module in
       IMP-MOD1, IMP-MOD2, ..., IMP-MODn.
   -n  Write a null implementation.
   -x IMP-MOD[.hs] MAIN[.hs]
   -xI IMP-MOD[.hs]
   -xM MAIN[.hs]
       Write wrapped, dug-extracting versions of the implementation
       module IMP-MOD and/or the main module stored in file MAIN.
       Warning: The files they wrap will be backed up before being
       overwritten, but they may be restored using '-u'.  The wrapped
       program will behave as before but will also extract and write a
       dug as it is run.  The wrapped files use Green Card.
   -u IMP-MOD[.gc] MAIN[.gc]
   -uI IMP-MOD[.gc]
   -uM MAIN[.gc]
       Unwrap the implementation module IMP-MOD and/or the main module
       stored in file MAIN which were previously wrapped with '-x'.
   -pT Write a script 'makeProfiles.hs' to make profiles (base version).
   -pS Write a best guess at a version of 'makeProfiles.hs' based on
       a size-based shadow data structure.
   -b IMP-MOD1 IMP-MOD2 ... IMP-MODn
       Write a benchmarker covering implementation modules
       IMP-MOD1, IMP-MOD2, ..., IMP-MODn.

(General.)
 -h   Show this help.
 -v   Show version info.
 -G   Use Green Card to construct dug evaluator.
```

Figure C.1: Help information for `auburn`.

```
Usage: Queue_Man [options] [dug-file | -]
Options:
   -g PROFILE SEED
       Generate a dug using the given profile, and the given seed for
       pseudo-random number generation.  Any dug file given on the
       command line is ignored.  The seed should lie between 1 and
       2147483646 inclusive.

       PROFILE is of the form:
          Profile GWGTS PHASES
       where GWGTS is the generator weights, and PHASES is a Haskell list
       with each element of the form:
          Phase MOWGTS MORTALITY PMF POF
       where MOWGTS is the mutator and observer weights, with the remaining
       arguments giving the mortality, the persistent mutation factor and
       persistent observation factor.

       Operator weights are given as a Haskell list of decimals and are
       ordered within the list firstly by role and then lexically, ie.
          empty, snoc, tail, head, isEmpty.

       Note that you will probably need to enclose arguments containing
       spaces or parantheses in quotes to avoid confusing the shell.

   -a PHASEARG
       When using a profile to generate a dug with `-g', or when
       producing a profile of a dug with `-p' or `-pP', use the phase
       argument PHASEARG.  PHASEARG is read in by `phaseArgRead'
       defined in the shadow data structure and is used by `phaser' to
       determine the phasing of nodes.

   -r FILE
   -rP
       Read a textual dug file, as outputted by `-t' or `-tP', from FILE
       or from standard input.

   -p FILE
   -pP
       Write a profile of the dug to FILE or to standard output.

   -N
       Normalise the profile written with `-p' or `-pP' with the profile
       given with `-g' (the averages of the weights are made equal for
       easier comparison).  If the dug is read rather than generated, make
       the averages of the weights equal to one.
```

Figure C.2: Help information for a typical dug manager (part I).

```
   -o FILE
   -oP
       Write the dug to FILE or to standard output.

   -d FILE
   -dP
       Write a visual depiction of the dug suitable for the ‘dotty’
       package of AT&T to FILE or to standard output.

   -t FILE
   -tP
       Write a text description of the dug to FILE or to standard output.

   -H
       When used with ‘-t’ or ‘-tP’, make the text description of the dug
       a valid Haskell program.

   -h
       This help.

The following options are only applicable when used with ‘-g’:
   -b POOLSIZE
       The size of the pool from which to draw integer arguments.
       Default: 10

   -fL MINFS
       The minimum size of the frontier.
       Default: 1

   -fU MAXFS
       The maximum size of the frontier.  A value of 0 indicates no maximum.
       Default: 0

   -n NODES
       The number of nodes to generate.
       Default: 10000

Note that outputting a large amount of data to a file is significantly
slower than to standard output, eg. we recommend writing a sizeable
dug to standard output and re-directing this to a file if necessary.
```

Figure C.3: Help information for a typical dug manager (part II).

```
Usage: Queue_Bmark [options]
Options:

   -h
      Print this help.

Decision Tree Inducer
---------------------

A sample of benchmarking results may be obtained via at most one of the
following flags:
   -g SEED
      Generate a random sample, using 'makeDugs', 'evalDugs', and
      'processTimes'.  The seed should lie between 1 and 2147483646
      inclusive.
   -s FILE
   -sP
      Read in a sample from FILE or standard input.

A decision tree may be obtained via at most one of the following flags:
   -i
      Induce a decision tree from the sample.
   -t FILE
   -tP
      Read in a tree from FILE or standard input.

At least one of the following flags must be supplied to request output:
   -c FILE
   -cP
      Check the accuracy of the decision tree on the sample.  Output the
      report to FILE or to standard output.
   -o FILE
   -oP
      Write the sample to FILE or to standard output.
   -w FILE
   -wP
      Write the decision tree to FILE or to standard output.
   -d FILE
   -dP
      Using the profile taken from FILE or standard input, use the decision
      tree to decide which implementation suits the profile.  Write the
      decision to standard output.

The following flags can be used to modify the behaviour of the '-i' flag:
   -G
      Use the gain criterion, rather than the default gain ratio criterion.
   -p SIZE
      Prune any leaves no larger than SIZE on the induced tree.
      Default: 0.
```

Figure C.4: Help information for a typical benchmarker (part I).

```
   -a
      Induce a decision tree on one half of the sample, prune this
      tree to different maximum leaf sizes, and choose the pruned tree
      with least error when applied to the other half of the sample.
   -r
      Perform reduced error pruning on the induced tree, by using half
      of the sample for induction and half for testing.
   -x
      When used with `-a' or `-r', use the number of misclassifications as
      the measure of error.  Without `-x', the mean ratio of the predicted
      winner is used (the larger the mean, the worse the prediction).
   -P
      Perform very pessimistic pruning on the induced tree.
   -C CF
      When pruning with `-P', use confidence level CF (0 < CF < 1).  The
      smaller CF is, the more pruning is done.
      Default: 0.25.

The following flags modify the behaviour of the flags above:
   -v
      Verbose.  Show some of the output of generating a sample with `-g'.
   -V
      Very verbose.  Show all of the output of generating a sample with `-g'.
   -n SIZE
      Specify the SIZE of a sample generated with `-g' (number of profiles
      chosen).
      Default: 100.
   -m OPTIONS
      Pass OPTIONS to `makeDugs' when generating a sample with `-g'.
      Default: "".
   -e OPTIONS
      Pass OPTIONS to `evalDugs' when generating a sample with `-g'.
      Default: "-r 1 -R 5".
   -I IMP1 IMP2 ... IMPn
      when generating a sample, use the ADT implementations named
      IMP1, IMP2, ..., IMPn.  When reading a sample, restrict the ratios
      read to these implementations.
   -A ATT1 ATT2 ... ATTn
      When reading a sample, restrict the profile attributes read to
      those named ATT1, ATT2, ..., ATTn.

ADT Implementation Tracer
-------------------------

   -q SEED
      Run the tracer.  The seed should lie between 1 and 2147483646
      inclusive, and is used to generate random dugs, printing any dug
      that causes an error.

The flags `-v', `-V', `-m', and `-I' also modify the behaviour of `-q'.
```

Figure C.5: Help information for a typical benchmarker (part II).

```
Creates a GNU makefile in the current directory to manage an
experiment using Auburn.

Flags:

-l LIBRARY
  Use Auburn library held in directory LIBRARY,
  eg. '-l /usr/local/lib/auburn'.
  Default: /usr/gem/lib/auburn

-q
  Quiet running: do nothing but print everything.

-h
  Show this help.
```

Figure C.6: Help information for `auburnExp`.

```
Makes dugs from the profiles given in files of the form
‘dug-${profile}.profile’ in the current directory.  The dugs are
stored in dug code files of the form ‘dug-${profile}-${seed}.dug’
with their _actual_ profiles stored in files of the form
‘dug-${profile}-${seed}.profile’.

Flags:

(Where more than one value is passed, eg. with ‘-p’, the string
 passed should be a perl expression that evaluates to an array,
 eg. "(1,2,4)", or "(1..4)" or even "(1..3,5..7)".  The following
 also seem to work fine: "4", "1,3", "3..5".)

-s SIG
  Name of signature which the dug manager uses, eg. ‘-s Queue’.
  Default: signature of first manager in current directory.

-p PROFILES
  Names of profiles, eg. ‘-p "(1..8)"’.
  Default: all profiles in current directory.

-S N
  Number of different seeds per profile, eg. ‘-S 3’.
  Default: 3.

-o OPTIONS
  Options to pass to the dug manager.  The options will immediately
  follow the dug manager and precede its arguments, so flags for the
  Haskell run-time system can be included either using ‘+RTS’ and
  ‘-RTS’ (GHC and nhc do this) or directly (HBC does this).
  Eg. ‘-o "+RTS -p -RTS"’ for GHC with profiling, and
      ‘-o "-"’ for HBC (as a minus must precede flags passed to an
         executable), and
      ‘-o "-m -"’ for HBC with profiling.
  Default: "-".

-O OPTIONS
  Additional options to pass to the dug manager.  Multiple ‘-O’s
  accumulate options.  The options will follow the base options
  given by ‘-o’.
  Eg. ‘-O "-n 1000"’ and ‘-O "-b 100"’ together with ‘-o "-"’ pass
  the options ‘- -n 1000 -b 100’ to the dug manager, telling it to
  generate dugs of size 1000 nodes using a pool size of 100.

-z SEED
  Initial seed (between 1 and 2147483646 inclusive).
  Default: Obtained from the system clock.

-q
  Quiet running: do nothing but print everything.

-h
  Show this help.
```

Figure C.7: Help information for `makeDugs`.

```
Runs and times the dug evaluators on dug code files (of the form
‘dug-${profile}-${seed}.dug’ as outputted by ‘makeDugs’) in the
current directory.  Writes total times (over all seeds) to files of
the form ‘dug-${profile}-${implementation}.time’.

Flags:

(Where more than one value is passed (with ‘-i’, ‘-p’ and ‘-d’), the
 string passed should be a perl expression that evaluates to an
 array, eg. "(1,2,4)", or "(1..4)" or even "(1..3,5..7)".  The
 following also seem to work fine: "4", "1,3", "3..5".  Note that some
 characters need to be quoted, eg. ".", so ‘-d test.dug’ becomes
 ‘-d ’"test.dug"’’.)

-s SIG
  Name of signature which the dug evaluators use, eg. ‘-s Queue’.
  Default: signature of first evaluator in current directory.

-i IMPS
  Names of implementations,
  eg. ‘-i "(NaiveQueue,SimpleQueue,BankersQueue,Queue_Null)"’.
  Default: all implementations for chosen signature in current directory.

-p PROFILES
  Names of profiles, eg. ‘-p "(1..8)"’.
  Default: all profiles in current directory.

-d DUGS
  Dugs to be evaluated.
  Default: all dugs in current directory matching ‘dug-${profile}-*.dug’.

-r N
  Number of separate timed runs per dug, eg. ‘-r 3’.
  Default: 3.

-R N
  Number of internal repeated evaluations per timed run, eg. ‘-R 10’.
  Default: 10.
```

Figure C.8: Help information for `evalDugs` (part I).

```
-t COMMAND
  Time command to produce time information in POSIX standard 1003.2,
  specifically:
         real %e
         user %U
         sys %S
  (Actually, only requirement is that the output contains the string
   "user %U" where `%U' is the user time.)
  Most UNIX time commands use this form of output.  GNU time does if
  passed the flag `-p'.  The user time may contain a colon `:'
  separating minutes from seconds, eg. `12:32.54'.
  Eg. `-t "gnutime -p"'.
  Default: "time".

-T TIME
  Timeout dug evaluators after TIME seconds.  Useful for preventing
  excessively slow runs of a dug evaluator.  Using a TIME of 0
  prevents any timeouts.
  Default: 600.

-o OPTIONS
  Options to pass to each dug evaluator.  The options will
  immediately follow the dug evaluator and precede its arguments, so
  flags for the Haskell run-time system can be included either using
  `+RTS' and `-RTS' (GHC and nhc do this) or directly (HBC does
  this).
  Eg. `-o "+RTS -p -RTS"' for GHC with profiling, and
      `-o "-m"' for HBC with profiling.
  Default: "".

-c
  Ignore checksum errors.

-q
  Quiet running: do nothing but print everything.

-h
  Show this help.
```

Figure C.9: Help information for `evalDugs` (part II).

```
Processes times outputted by ‘evalDugs’ (files of the form
‘dug-${profile}-${implementation}.time’ in the current directory).
Outputs resulting processed times in file ‘dugs.times’ using summary
information found in ‘dugs.profiles’.

Flags:

(Where more than one value is passed, eg. with ‘-i’ and ‘-p’, the
string passed should be a perl expression that evaluates to an
array, eg. "(1,2,4)", or "(1..4)" or even "(1..3,5..7)".)

-i IMPS
  Names of implementations,
  eg. ‘-i "(NaiveQueue,SimpleQueue,BankersQueue,Queue_Null)"’.
  Default: all implementations for chosen signature in current directory.

-p PROFILES
  Names of profiles, eg. ‘-p "(1..8)"’
  Default: all profiles in current directory.

-f FORMAT
  Format string used by ‘printf’ to output the times, eg. ‘-f 8.3f’.
  Default: "8.3f".

-F
  Use brief format.  Useful for automatic processing of results.
  One number per line.  First line contains number of
  implementations used.  Remaining lines contain ratios, in the
  expected order.

-S
  Sort profiles by string comparison, rather than by the default
  numerical comparison.

-q
  Quiet running, do nothing but print everything.

-h
  Show this help.
```

Figure C.10: Help information for `processTimes`.

```
Cleans up all dug and profile files in current directory, that is,
all files of the form ‘dug-*.profile’ ‘dug-*.dug’ ‘dug-*.time’, and
‘dugs.times’ and ‘dugs.profiles’.

Flags:

-q
  Quiet running: do nothing but print everything.

-h
  Show this help.
```

Figure C.11: Help information for `cleanDugs`.

Bibliography

[1] Stephen R. Adams. Implementing sets efficiently in a functional language. Technical Report CSTR 92-10, Department of Electronics and Computer Science, University of Southampton, 1992. (p 24)

[2] Stephen R. Adams. Efficient sets—a balancing act. *Journal of Functional Programming*, 3(4):553–562, 1993. (pp 5, 18, 21, 24)

[3] G. M. Adel'son-Velskii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademia Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262. (pp 1, 21)

[4] The Auburn Home Page. `http://www.cs.york.ac.uk/fp/auburn/`. (p 187)

[5] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. (p 1)

[6] W. Braun and M. Rem. A logarithmic implementation of flexible arrays. Memorandum MR83/4. Eindhoven University of Technology, 1983. (p 26)

[7] Gerth S. Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, December 1996. (pp 5, 19, 34, 37, 38)

[8] Adam L. Buchsbaum. *Data-structural bootstrapping and catenable deques.* PhD thesis, Department of Computer Science, Princeton University, June 1993. Technical Report TR-423-93. (p 14)

[9] F. Warren Burton and Rex L. Page. Distributed random number generation. *Journal of Functional Programming*, 2(2):203–212, April 1992. (p 103)

[10] Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, Copenhagen, June 1993. ACM Press. (p 5)

[11] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. (pp 1, 187, 188)

[12] Paul F. Dietz. Fully persistent arrays. In *Proceedings of the First Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 67–74. Springer-Verlag, August 1989. (p 44)

[13] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989. (pp 44, 46)

[14] Martin Erwig. Functional programming with graphs. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 52–65. ACM Press, June 1997. (pp 1, 5)

[15] Michael L. Fredman, Robert Sedgewick, Daniel D. K. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986. (p 39)

[16] Gnu make utility. `http://www.gnu.org/software/make/make.html`. (p 161)

[17] GraphViz: Tools for viewing and interacting with graph diagrams. `http://www.research.att.com/sw/tools/graphviz/`. (p 149)

[18] The HBC compiler. `http://www.cs.chalmers.se/~augustss/hbc/hbc.html`. (p 202)

[19] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962. (p 187)

[20] Robert Hood and Robert Melville. Real-time queue operations in pure LISP. *Information Processing Letters*, 13(2):50–54, November 1981. (pp 10, 11, 12)

[21] Rob R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, October 1992. (pp 18, 26)

[22] John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989. (p 1)

[23] E. B. Hunt, J. Martin, and P. J. Stone. *Experiments in Induction*. Academic Press, New York, 1966. (p 127)

[24] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93–102, May 1995. (pp 15, 18, 28)

[25] Donald E. Knuth. *Searching and Sorting*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973. (p 40)

[26] Graeme E. Moss and Colin Runciman. Auburn: A kit for benchmarking functional data structures. In *Proceedings of IFL'97*, volume 1467 of *LNCS*, pages 141–160, September 1997. (pp xvii, 187)

[27] Graeme E. Moss and Colin Runciman. Exploring datatype usage space. In *Draft Proceedings of IFL'98*, University College London, UK, September 1998. (p xvii)

[28] Graeme E. Moss and Colin Runciman. Automatic benchmarking of functional data structures. In *Proceedings of PADL'99*, LNCS, 1999. To be published. (p xvii)

[29] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983. (pp 18, 19)

[30] J. R. Norris. *Markov Chains*. Cambridge University Press, 1997. (pp 124, 125)

[31] Manuel Núñez, Pedro Palao, and Ricardo Peña. A second year course on data structures based on functional programming. In *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 65–84. Springer-Verlag, December 1995. (pp 5, 18, 21, 34, 40)

[32] Chris Okasaki. Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking. In *IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995. (pp 5, 12, 19, 21, 29)

[33] Chris Okasaki. Purely functional random-access lists. In *Conference Record of FPCA '95*, pages 86–95. ACM Press, June 1995. (pp 5, 18, 21, 102, 187)

[34] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995. (pp 5, 10, 14)

[35] Chris Okasaki. Functional data structures. In *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 131–158. Springer-Verlag, August 1996. (pp 34, 39, 40)

[36] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996. (p 5)

[37] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *Proceedings of the International Conference on Functional Programming*, pages 62–72. ACM Press, May 1996. (pp 10, 12, 13)

[38] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998. (pp 1, 10, 11, 12, 13, 14, 15, 19, 21, 34, 35, 41, 44, 46, 185)

[39] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 1999. To appear. (p 2)

[40] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997. (pp 1, 5, 44)

[41] Steve Park. Private communication, April 1996. (p 103)

[42] Steve Park and Keith Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1193–1201, October 1988. (p 103)

[43] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green Card: A foreign-language interface for Haskell. In *Haskell Workshop*, Amsterdam, June 1997. Published by Oregon Graduate Institute of Science & Technology. (pp 92, 98, 106, 158)

[44] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. (pp 125, 127)

[45] J. R. Quinlan. Simplifying decision trees. *International Journal of Man-Machine Studies*, 27:221–234, 1987. (pp 132, 135)

[46] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. (pp 127, 130, 135, 136, 137, 201)

[47] C. M. P. Reade. Balanced trees with removals: An exercise in rewriting and proof. *Science of Computer Programming*, 18(2):181–204, 1992. (p 5)

[48] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959. (p 187)

[49] Daniel D. K. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985. (p 41)

[50] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proceedings of PLILP'97*, volume 1292 of *Lecture Notes in Computer Science*, pages 291–308, 1997. (p 169)

[51] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315, April 1978. (pp 1, 35)

[52] Raymond T. Yeh, editor. *Data Structuring*, volume IV of *Current Trends in Programming Methodology*. Prentice-Hall, 1978. (p 43)

[53] York Functional Programming Group. `http://www.cs.york.ac.uk/fp/`. (pp 106, 110, 141, 158, 167, 169)