J. Functional Programming **11** (5): 525–556, September 2001. Printed in the United Kingdom

Inductive benchmarking for purely functional data structures

GRAEME E. MOSS and COLIN RUNCIMAN
Department of Computer Science, University of York, York, UK

Abstract

Every designer of a new data structure wants to know how well it performs in comparison with others. But finding, coding and testing applications as benchmarks can be tedious and time-consuming. Besides, how a benchmark uses a data structure may considerably affect its apparent efficiency, so the choice of applications may bias the results. We address these problems by developing a tool for *inductive benchmarking*. This tool, *Auburn*, can generate benchmarks across a wide 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 then apply inductive classification techniques to obtain decision trees for the choice between competing data structures. 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, but they also reveal some limitations of the approach.

1 Introduction

In recent years, many papers have given details of new functional data structures (Chuang & Goldberg, 1993; Okasaki, 1995a; Okasaki, 1995b; Okasaki, 1995c; Brodal & Okasaki, 1996; Erwig, 1997; O'Neill & Burton, 1997). However, these papers give only limited attention to empirical performance. Okasaki writes in an open problems section of his thesis, 'The theory and practice of benchmarking [functional] data structures is still in its infancy' (Okasaki, 1996b). This paper develops the theory and practice of 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. What is wrong with this approach? First, if suitable applications are not available, writing them is a chore. Secondly, using the results of just a few benchmarks can be 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.

1.1 Overview of Auburn

We address these problems by developing a tool, called *Auburn*, for benchmarking functional data structures. Auburn works with the signature of an abstract datatype (ADT), such as the signature for lists in figure 1 of section 2, and with one or more Haskell implementations of the ADT.

Among other things, Auburn can automatically generate a range of pseudo-random benchmark tests making use of a given ADT. Auburn can run these tests using each implementation, apply an inductive classification procedure to the results, and derive a decision tree for the appropriate choice of implementation depending on the application. 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 structures are best for particular uses.

To apply a decision tree one needs a *profile* characterising how the ADT is to be used. Auburn can also extract such a profile from any application program making use of an ADT. Even if we have a single application of an ADT in mind, measuring its performance for each possible implementation could be tedious, and would still not give us any understanding of how the ADT is being used or why performance differs as it does between implementations. By applying a decision tree to the application's profile, we can predict the best choice of data structure and have a way to understand the choice.

The key technical idea on which Auburn is based is the *datatype usage graph* (DUG) – a detailed representation, using a labelled directed multi-graph, of how a program makes use of an ADT. All the benchmark tests Auburn constructs are represented as DUGs, and interpreted by specially-generated evaluators for each ADT. (Interpretive overheads are determined by generating *null* implementations, and subtracted appropriately from the measures of the actual implementations.) How an application uses an ADT is also recorded as a DUG.

A profile of ADT usage is actually a vector of attributes characterising a (large) family of DUGs. For example, attributes include the relative frequencies of different operations. Determining the profile of a DUG is easy; generating valid DUGs from profiles is much harder, and unless all the ADT operations are total, the programmer has to supply extra information with a signature and its implementations. However, the extra information can also be used to improve decision trees.

Auburn is implemented mainly in Haskell, with a supporting library in C. The implementation is freely available from http://www.cs.york.ac/fp/auburn/.

1.2 Overview of this paper

This paper is based on the first author's thesis (Moss, 1999), to which readers are referred for a more comprehensive account. For example, the thesis includes a full review of all the data structures we have investigated, and a full description of how Auburn is implemented. Here we are necessarily selective.

Section 2 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.

Section 3 describes the core algorithms of Auburn. 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.

Section 4 explains how Auburn induces decision trees for the choice of different implementations in different parts of DUG-space.

Section 5 reports the results of applying Auburn to various implementations of three different ADTs: heaps, queues and random-access sequences. We assess the accuracy of Auburn's results and investigate the sources of any inaccuracy.

Section 6 concludes with a summary and pointers to related and future work.

2 Datatype Usage Graphs

2.1 Abstract datatypes

An *abstract datatype* (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 restriction allows the *implementation* of the ADT to be isolated from its *use* – we may change implementations without changing how we use the ADT.

We shall restrict ourselves to *container types*, i.e. 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 1 (ADT)
For any type constructor T, and any set of functions F, the pair (T, F) is an ADT if the following conditions 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.

As a further simplification, we restrict ourselves to *simple* ADTs, according to the following definitions. 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*, are excluded.

Definition 2 (Simple Type)
For any type constructor T of arity one, we say that the type t is *simple* (or more

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

Fig. 1. The signature of a simple list ADT $\mathscr{A}_{List}$, expressed in Haskell. The exported type constructor is List, and the type of each operation is simple over List.

fully, *simple over* T) if t can be formed as *type* by the grammar

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

where a is a type variable, and t contains at least one occurrence of $T\ a$.

Example 2
The following types are simple over the type constructors *Queue*, *List* and *Set* respectively:

- $Queue\ a \rightarrow a \rightarrow Queue\ a$
- $List\ a \rightarrow Int \rightarrow a$
- $Set\ a$

Definition 3 (Simple ADT*)*
We define the ADT $\mathscr{A} = (T, \{f_1, \ldots, f_n\})$ to be *simple* if the type of each operation f_i is simple over T.

Example 3
The signature of a simple ADT $\mathscr{A}_{List}$ is given in figure 1, expressed in Haskell.

During the run of an application, many different instances of an ADT will exist. Each of these particular instances of the ADT is called a *version* (Okasaki, 1998).

Definition 4 (Generator, Mutator, Observer, Role, Version Arity)
The *role* of any operation f of type $t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_m$, simple over the type constructor T, can be classified as follows.

$$\begin{aligned} &Generator: && t_m = T\ a \text{ and } (\forall j, 1 \leqslant j < m)\ t_j \neq T\ a \\ &Mutator: && t_m = T\ a \text{ and } (\exists j, 1 \leqslant j < m)\ t_j = T\ a \\ &Observer: && t_m \neq T\ a \text{ and } (\exists j, 1 \leqslant j < m)\ t_j = T\ a \end{aligned}$$

The *version arity* of an operation is the number of version arguments it takes. Every generator has version arity 0, and every mutator and observer has version arity $\geqslant 1$.

Example 4
Looking at the signature of the simple ADT $\mathscr{A}_{List}$ in figure 1, empty is a generator; cons, tail and catenate are mutators; isEmpty, head and lookup are observers. Every mutator and observer has version arity 1, apart from catenate, which has version arity 2.

2.2 Usage graphs

To model how an ADT is used by an application we use a labelled directed multi-graph. The nodes are labelled with *partial applications* of the ADT operations to specific values for all the *non-version* arguments: for simplicity, these are restricted to atomic values. The version arguments are recorded as arcs from other nodes that compute them: 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 also numbered according to the order of evaluation. Such a graph is called a *Datatype Usage Graph* (DUG).

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.*

We express these ideas more precisely in the following definitions.

Definition 5 (Partial Application, $Pap(\mathscr{A})$)
Given a simple ADT $\mathscr{A} = (T, \{f_1, \ldots, f_n\})$, a *partial application* of f_i is any function of the following form:

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

Here, m is the arity of f_i, each version argument x_j occurs exactly once in the sequence $[a_1, \ldots, a_m]$, and all the other elements of this sequence are atomic values such as integers. 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_i occurs before x_j for $i < j$. The set of all partial applications of any function of a simple ADT $\mathscr{A}$ is denoted by $Pap(\mathscr{A})$.

Example 5
For the list ADT $\mathscr{A}_{List}$, whose signature is given in figure 1, the following functions are in $Pap(\mathscr{A}_{List})$:

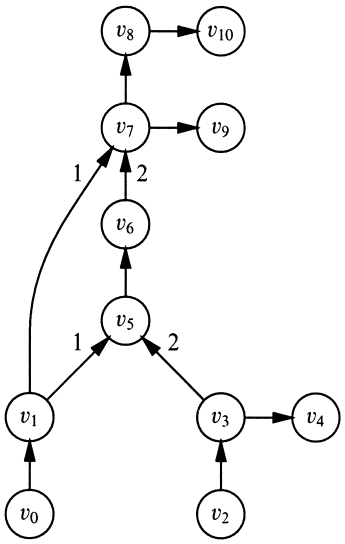
- $\lambda l \cdot cons$ 'a' l
- *empty*
- $\lambda l_1 \cdot \lambda l_2 \cdot catenate \; l_1 \; l_2$

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.

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 with an argument position.

Definition 6 (DUG)
Given a directed multi-graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, a simple ADT $\mathscr{A} = (T, \{f_1, \ldots, f_n\})$, a total mapping $\eta : \mathscr{V} \to Pap(\mathscr{A})$, a bijection $\sigma : \mathscr{V} \to \{1..|\mathscr{V}|\}$ and a total mapping $\tau : \mathscr{E} \to \mathbb{N}$, the 4-tuple $(\mathscr{G}, \eta, \sigma, \tau)$ is a DUG for $\mathscr{A}$, if for every $v \in \mathscr{V}$ the following properties are satisfied:

1. The arity of $\eta(v)$ equals the in-degree of v.
2. The mapping τ restricted to the incoming arcs of v is a bijection with the set $\{1..indegree(v)\}$.



	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

Fig. 2. A DUG for the list ADT $\mathscr{A}_{List}$.

3. If the incoming arcs to v, ordered by τ, have sources $v_1, \ldots, v_k$, then the types of arguments required by $\eta(v)$ match the types of results supplied by $\eta(v_1), \ldots, \eta(v_k)$.
4. If v has successor $w \in \mathscr{V}$, then $\sigma(v) < \sigma(w)$.
5. The type of every argument of $\eta(v)$ is $T\ a$, for some uniform instantiation of 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, forcing the graph to be acyclic – see the problem *Choosing the operation before the arguments* of section 3.1.1 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* of section 3.1.1 for justification.

Example 6
Figure 2 shows an example of a DUG. A table defines η. The ordering σ of the evaluation of the nodes is given by $\sigma(v_i) = i$. For nodes of in-degree > 1 the 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 instantiated to 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.

2.3 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 artificial slice of the application that uses an ADT implementation in the same way. We can then use this slice as a benchmark with a known pattern of use.

We define DUG evaluation more precisely by first defining how we may associate each node with a function application.

Definition 7 (Interpretation of Partial Applications)
Let $\mathscr{A}$ be any simple ADT. Let f be an operation of $\mathscr{A}$. Let $g \in Pap(\mathscr{A})$ be any partial application of f. Let $\mathscr{I}$ be an implementation of $\mathscr{A}$. The interpretation of g under $\mathscr{I}$, denoted by $[\![g]\!]_{\mathscr{I}}$, is the value of g using the implementation of f in $\mathscr{I}$.

Example 7
Let $\mathscr{L}$ be the ordinary Haskell implementation of lists, then

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

Definition 8 (Interpretation of Nodes)
Let $(\mathscr{G}, \eta, \sigma, \tau)$ be any DUG for the ADT $\mathscr{A}$, let v be any node of $\mathscr{G}$, and let $\mathscr{I}$ be an implementation of $\mathscr{A}$. Let the arcs incident to v, ordered by τ, be from the nodes $v_1, \ldots, v_k$ respectively. The interpretation of v under $\mathscr{I}$, denoted by $[\![v]\!]_{\mathscr{I}}$, is the following expression:

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

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

Example 8
Using the DUG shown in figure 2, and the ordinary Haskell implementation $\mathscr{L}$ of lists,

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

2.3.1 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 can accommodate either scheme by using the node ordering σ of a DUG $(\mathscr{G}, \eta, \sigma, \tau)$ in different ways, but here we consider only one.

Lazy Evaluation 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 may affect efficiency, but for simplicity we assume that the order of evaluation for observer nodes coincides with the order of formation for their associated closures.

Definition 9 (Lazy Evaluation of a DUG*)*
Given a DUG $(\mathscr{G}, \eta, \sigma, \tau)$ for an ADT $\mathscr{A}$, and an implementation $\mathscr{I}$ of $\mathscr{A}$, the *lazy evaluation* of the DUG with respect to $\mathscr{I}$ is the process of performing the following steps on each node $\sigma(i)$ in order:

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

Example 9
The lazy evaluation of the DUG of figure 2 would form the closures $[\![v_i]\!]$ for $0 \leqslant i \leqslant 10$ in order. When the closures for the observer nodes are formed, namely $[\![v_4]\!]$, $[\![v_9]\!]$, and $[\![v_{10}]\!]$, their values are demanded at the same time.

2.4 Profile

We want to create a benchmark from a DUG, and we want to extract a DUG from an application. However, a DUG may be too large and complex to serve as an intelligible pattern of use. So we next 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.

So what characteristics do we choose to record in a profile? A distinctive property of purely functional data structures is that their operations are non-destructive; but the extent to which applications depend on this property varies greatly. So one component we choose to include in a profile 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.

Definition 10 (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 10
Figure 2, 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 $v_7 \rightarrow v_9$ is ordered after the mutation $v_7 \rightarrow v_8$.

The ordering of mutations and observations from any node v can be made total by appealing to τ to resolve any ties between arcs to a common target.

Definition 11 (Persistent, Original)
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 (and τ) *after the earliest mutation* of v. A mutation or observation that is not persistent is called *original*.

Example 11
In figure 2, 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 original.

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

Definition 12 (Weight)
For any DUG $\mathscr{D}$, the *weight* of a mutator f in $\mathscr{D}$ is the number of mutations that apply f to nodes in $\mathscr{D}$. 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.

Example 12
The weights of the operations in the DUG in figure 2 are:

Operation	:	*empty*	*catenate*	*cons*	*tail*	*head*	*lookup*	*isEmpty*
Weight	:	2	4	2	2	1	1	1

Generative power of profiles. 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.

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 original, 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 version 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. Under the simplifying assumption that a node was made by a mutator, the average number of observations of a node is $1/r$. As we have excluded nodes made by generators, this number 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).

We separate *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*. The ratio of generations to other operations is governed by mortality and by the persistence factors.

Definition 13 (DUG Profile)
The profile of a DUG $\mathscr{D}$ with version graph $\mathscr{G}_V$ 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 $\mathscr{G}_V$.
- *Mortality*: The fraction of nodes in $\mathscr{G}_V$ that are not mutated.
- PMF: The fraction of mutations of nodes in $\mathscr{G}_V$ that are persistent.
- POF: The fraction of observations of nodes in $\mathscr{G}_V$ that are persistent.

Example 13
The DUG shown in figure 2 has the following profile:

- *Generation weights*: As there is only one generator, *empty*, this property is redundant: $empty = 1$.
- *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 ($v_1 \rightarrow v_7$) is persistent, so the PMF is 1/8.
- POF: There are three observations, one of which ($v_7 \rightarrow v_9$) 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 14 (Single-Threaded)
An application using an implementation of a simple ADT $\mathscr{A}$ in a manner recorded by the DUG $\mathscr{D}$ is *single-threaded* (with respect to $\mathscr{A}$) if the PMF and POF of $\mathscr{D}$ are both zero. A single-threaded application does not require a persistent implementation of the ADT.

Example 14
The part of the DUG in figure 2 restricted to nodes $v_0, \ldots, v_6$ has PMF=POF=0 and is therefore single-threaded.

2.5 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 a specific implementation of the ADT; it is applicable to any implementation of the ADT. 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.

As a running example, for the ADT $\mathscr{A}_{List}$, whose signature is given in Figure 1, and for which each version is a list, let the shadow of a version contain the length of the list.

Guarding against undefined 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, the head of an empty list. 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 $\mathscr{A}_{List}$, if we maintain the length of a list in the shadow, we can restrict the use of an operation such as taking the head of a list, applying it only to lists of positive length.

Shadow Profiling. The shadow can also store any other useful information about what operations were performed. This *shadow profile* allows information specific to an ADT to be collected. For example, by maintaining the length of a list, we can calculate the average lengths of lists passed to each mutator or observer. This extra source of profiling information is important. Without it, for example, 'N insertions followed by N deletions' and 'N insertions each followed by a deletion' have indistinguishable profiles. If N is large, the ideal implementation is unlikely to be the same in each case.

To simplify the definitions that follow, we shall assume that each occurrence of the type variable a in the type of an ADT operation is instantiated to *Int*. This assumption also simplifies both DUG generation and DUG extraction,

Definition 15 (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 $shadow_s(t)$, derived from t by replacing all occurrences of $T\ Int$ by s. The shadows maintained by this shadow operation have type s. There are no shadows of observers as they do not return versions.

Example 15
For any type s, an s-shadow of the *update* operation of $\mathscr{A}_{List}$ (see figure 1) has the following type:

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

Definition 16 (Shadowing)
Let $\mathscr{A} = (T, \{f_1, \ldots, f_n\})$ be any simple ADT. Without loss of generality, let $\{f_1, \ldots, f_m\}$ be the generators and mutators of $\mathscr{A}$. For any set $F' = \{f'_1, \ldots, f'_m\}$ of operations, and any type s, the pair (s, F') is a *shadowing* of $\mathscr{A}$ if the following hold for all $i \in \{1, \ldots, m\}$:

- The operation f'_i is an s-shadow of f_i.
- There exists a homomorphism $\phi :: T\ Int \to s$; that is, for all $x_1, \ldots, x_k$, where k is the arity of f_i, if $f_i\ x_1\ \ldots\ x_k$ is well-defined, then

$$\phi\ (f_i\ x_1\ \ldots\ x_k) = f'_i\ (\phi'\ x_1)\ \ldots\ (\phi'\ x_k)$$

where for all x,

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

Example 16
In one possible shadowing $\mathscr{S}_{List}$ of $\mathscr{A}_{List}$ 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.

Definition 17 (Shadow Evaluation)
Let $\mathscr{D}$ be any DUG for ADT $\mathscr{A}$, and $\mathscr{S} = (s, F)$ be any shadowing of $\mathscr{A}$. The *shadow evaluation* of $\mathscr{D}$ is a mapping ζ that takes a version node v to the result of evaluating $[\![v]\!]_{\mathscr{S}}$, where an operation is interpreted by its shadow.

Example 17
Taking the DUG of Figure 2 with a shadowing $\mathscr{S}_{List}$ tracking list length, the shadow evaluation ζ of the DUG is:

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

2.5.1 Guarding

For each operation f we need to define a guard that indicates which applications of f are allowed. We could make a guard take the same arguments as f, but with shadows for versions, and return true or false, according to whether the application is allowed or not. However, this approach has the disadvantage of forcing a choice of all arguments before applying the guard. For an operation such as indexed lookup, we have to guess which indices are appropriate before testing the validity of the application – hardly efficient.

The definition of a DUG already restricts arguments supplied by the result of another operation to just version arguments. So non-version arguments can be chosen independently of the results of other operations. We pass the guard only the shadow version arguments of an operation, and the guard's result specifies the valid domains for the remaining arguments. For example, the guard for *lookup* could return a range of indices up to the length of the list. As we assume that every

non-version argument is of type *Int*, each domain of legitimate values for such an argument is a subset of the integers, for which the following definitions assume a suitable representation type *IntSubset*.

Definition 18 (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

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

where $[a]_m$ is the type of lists of m elements of type a, and where s represents the type of shadows. Every version argument is replaced by a shadow, and every non-version argument moves 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 $n - v$ elements each of type *IntSubset*.

Example 18
Consider the ADT $\mathscr{A}_{List}$, whose signature is in figure 1. For any type s, a guard of the operation *head* using shadows of type s must be of type

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

If we add the operation *update* of type

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

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

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

Definition 19 (Guard)
Let $\mathscr{S} = (s, F')$ be a shadowing of the ADT $\mathscr{A} = (T, F)$ defining a homomorphism $\phi :: T\ \ Int \to s$. For any operation $f \in F$ of type t, the function g is an $\mathscr{S}$-*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 $l \geqslant 1$, $f\ \ x_1\ \ \ldots\ \ x_n$ is well-defined if

$$g\ \ (\phi\ \ x_{i_1})\ \ \ldots\ \ (\phi\ \ x_{i_k}) = [xs_1, \ldots, xs_l]$$

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

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

```
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
```

Fig. 3. Initial outline of a simple DUG generation algorithm.

Example 19
The guards for *head* and *tail* applied to shadow-length s return $s \neq 0$. The guard for *lookup* returns $[\{1 \ldots s\}]$.

3 Implementing DUGs

3.1 From profile to benchmark

We derive a benchmark from a profile in two stages:

(1) A DUG generator uses pseudo-random numbers to create a DUG that probabilistically has the given profile, i.e. the *expected* profile is the one given.
(2) A DUG evaluator executes this DUG using a given implementation of the ADT.

3.1.1 DUG *Generation*

How shall we build a DUG? Figure 3 gives a reasonable starting point for an algorithm, but proceeding along these lines one encounters various problems.

- *Creating undefined applications.* Some applications of operations may not be well defined. For example, the application *head empty* is usually not defined. We avoid these applications 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.
- *Allowing undefined arguments.* Lazy evaluation evaluates the operation, not 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. So we never add a node without knowing all the arguments.
- *Choosing the version arguments.* We could pick the arguments from any part of the DUG already formed. But as we must maintain a shadow of every possible argument, this unrestricted choice may cost too much. Therefore we restrict choice of arguments to a subgraph, the *frontier*, and maintain shadows only for nodes in the frontier. If the frontier becomes too large, we remove a node (though it stays in the DUG).

- *Choosing non-version arguments.* How can we generate an argument of type a? As no profile properties depend on non-version arguments, we restrict them to integers—that is, we instantiate the type variable a to *Int*. We then choose all non-version arguments independently of the graph.
- *Choosing the operation before the arguments.* If each operation is chosen before any arguments are selected, it is hard to make a generated DUG conform to some of the profile properties. It is easier if for each new node we plan a sequence of operations it should be involved in as an argument, and for each operation maintain a version-argument buffer of the appropriate arity. We place nodes in the buffer for their next planned operation. When a buffer is full, we create a new node accordingly, emptying the buffer.
- *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. Therefore, once a guard rejects an application, we revise the plans for argument nodes, skipping this operation.

A refined outline of the DUG generation algorithm is given in figures 4 and 5. 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. We 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, leading to continual growth of the frontier. So we cap the frontier size to prevent running out of memory. When the frontier exceeds a given limit, we remove an arbitrary node from the frontier.
- 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 created, we record this event 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 describes a DUG completely. A list of births is more convenient for evaluating a DUG, whereas a list of deaths is more convenient for profiling a DUG. So we produce both.

3.1.2 DUG *Evaluation*

The process of DUG evaluation is comparatively straightforward. Unlike DUG generation, we encounter no theoretical problems, only the practical one of efficiency. Our first DUG evaluator required more time for input-output and maintaining a lookup table than for ADT operations, preventing us from accurately measuring their relative efficiencies. We solve this problem by writing a driving program for the evaluator in C, with C routines to perform the input-output and handle the lookup table. We

```
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 (*)
    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 (*)
      fi
    fi
  od
  record the death of every node in the frontier and buffers
```

Fig. 4. Overview of the DUG generation algorithm (Part I). Further details of steps marked (*) are given in the next figure.

use an extension to the *Green Card* package to support calls from C to Haskell (Peyton Jones *et al*., 1997).

An overview of the evaluation algorithm is given in figure 6. 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; so by comparing checksums we can check the results of one implementation against those of another.

3.2 From application to profile

3.2.1 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:

- *Lazy Evaluation.* When we record 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.

```
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 succeeds then
      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
   fi
   remove the head operation of each version argument
   record the death of every version argument with an empty future
   add remaining version arguments to the frontier
```

Fig. 5. Overview of the DUG generation algorithm (Part II).

```
while not at the end of the DUG file do
   read the next birth or death
   if it is 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
      remove the dead node from the frontier
   fi
od
report the checksum
```

Fig. 6. Overview of the DUG evaluation algorithm.

Some arguments may not be evaluated at all; after the program has finished, we record any such unevaluated arguments explicitly in the DUG.

- *Recording the* DUG. We must record the DUG as output, but do not wish to transform every function to work within the IO monad. Neither do we wish to accumulate information about the DUG by extending the result from every function that calls an ADT operation. We avoid this problem by cheating. We interface to a side-effecting C function that records the DUG in a file.

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

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

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* $\mathsf{f_i}$

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

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

Fig. 7. Definition of a wrapped ADT. For an ADT exporting type constructor T and operations $\mathsf{f_i}\ ::\ \mathsf{t_{i,1}} \rightarrow \cdots \rightarrow \mathsf{t_{i,n_i}}$, the wrapped ADT exports type constructor $\mathsf{T^w}$ and operations $\mathsf{f_i^w}$.

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 do not record values of such arguments.

How DUG *Extraction is Done.* We modify the application and ADT implementation to perform the same task, but produce a DUG as a side-effect. The modification involves *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, each time it is called, which nodes supply its version arguments. A wrapped operation also calls the old operation, and wraps the result into a node with a new identity tag.

The rules for deriving a wrapped ADT are given in figure 7. For example, the wrapped version of a List datatype is

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

and the wrapped implementation of cons is

```
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)))
```

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)
```

The functions new_node and new_arc are implemented in C; new_node returns a new identity tag for a node, after recording which operation labels this new node, whereas new_arc returns only unit, after recording the arc, including the argument node identity, the result node identity, and the position of the argument node.

Evaluation of wrappedCons occurs only 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.

3.2.2 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

```
data Profile = Profile {generationWeights :: [(Operation,Weight)],
                        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. To calculate the mortality, we accumulate both the number of nodes not mutated, and the total number of nodes. From this information we can calculate the proportion of nodes not mutated: that is, the mortality. Similarly, we accumulate integer numerators and denominators to calculate the PMF and the POF.

4 Inductive classification of benchmarks

Our goal is a tool that gives benchmarking results qualified by the pattern of datatype usage. Naively we might hope to create a benchmark with *every possible* pattern of use – using some discrete scale for profile values – 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. 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, because the number of patterns of use is exponential in the number of usage attributes.

One way to reduce the number of attributes is to ignore the least significant attributes – those that have little or no effect on the relative performance – and concentrate on those that most influence the appropriate choice of implementation.

4.1 Decision trees

Of the many possible approaches to the problem of characterising the key attributes, our chosen method is to *induce a decision tree* (Quinlan, 1986). 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 $A \leqslant v$, where A is a datatype usage attribute, and v is some constant.
- Each leaf node is labelled with the name of an ADT implementation.

Decision trees can be used to choose an implementation given knowledge of datatype usage: start at the root and follow the appropriate branches till you reach a leaf.

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 both of the profile *and of the shadow profile* – an important source of extra information. Each benchmark in the sample is run, and the performance of each 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. More generally, given a sufficiently large and varied sample, the decision tree induced should be able to predict the winning implementation of any benchmark with good accuracy.

4.2 Induction algorithms

We take an existing algorithm from the literature for inducing a decision tree from a sample. We use the algorithm C4.5 (Quinlan, 1993), which is a descendant of ID3 (Quinlan, 1986). 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 *et al.* (1966). 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 $v_i \leqslant v < v_{i+1}$, splitting S with the test $A \leqslant v$ results in the same split. Therefore, there are at most $n-1$ distinct ways of splitting S using A. We consider only the tests $A \leqslant (v_i + v_{i+1})/2$.

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* (Quinlan, 1993). We have tried both, but the results reported in section 5 are for the gain ratio criterion, as it proved more successful.

4.3 Simplifying decision trees

The decision tree induced by this method classifies the results of the given sample *perfectly*. Unfortunately, this tree is not an ideal basis for choosing an implementation for two reasons: (1) the tree may be very large and complex; (2) the tree is based on the chosen sample and may be over-specific. Therefore we *prune* the induced tree: wherever 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.

There is a choice of methods for error prediction (Quinlan, 1987; Quinlan, 1993). Some are based on further test samples: for example, if in addition to recording the winning implementation for the test we also record the *ratio* of the time of *every* implementation to the time of the winning implementation, one definition of the predicted error of a subtree is the *average ratio* of the implementation given by the subtree as the winner. However, we found that a statistical pruning method described by Quinlan as 'far more pessimistic' (Quinlan, 1993) works just as well without requiring further tests.

5 Results

In this section we present some results from the use of Auburn[1] to evaluate over twenty different data structures implementing queues, random-access sequences and heaps.

Queues Among the simplest of ADTs, queues have the following signature.

```
empty  :: Queue a
snoc   :: Queue a → a → Queue a
head   :: Queue a → a
tail   :: Queue a → Queue a
```

In addition to a *naive* implementation of queues as lists, we take the *batched* and *multihead* variants of a list-pair (Hood & Melville, 1981), two refinements of list-pairs justified by the *banker's* model (Okasaki, 1996c) and the *physicist's* model (Okasaki, 1998) of amortized complexity, a *real-time* variant of banker's queues (Okasaki, 1995c), and Okasaki's *bootstrapped* and *implicit* queues (Okasaki, 1998).

[1] All the benchmark tests reported here were compiled using the York nhc13 byte-code Haskell compiler, and run in a heap of 80Mb, on an SGI Indy under IRIX 5.3.

Random-access sequences We use the term random-access sequence to describe a list ADT that also supports indexed lookup and update, with the following signature.

```
empty   :: RASeq a
cons    :: a → RASeq a → RASeq a
head    :: RASeq a → a
tail    :: RASeq a → RASeq a
lookup  :: RASeq a → Int → a
update  :: RASeq a → Int → a → RASeq a
```

Besides a *naive* list implementation, we take *threaded skew binary* lists (Myers, 1983), Adams' *balanced trees* (Adams, 1993), *Braun trees* (Hoogerwoord, 1992), *slowdown deques* (Kaplan & Tarjan, 1995) restricted to the relevant operations, *skew binary* lists (Okasaki, 1995b) and *elevator* lists (Moss, 1999).

Heaps A heap is an ordered collection of elements, with the following signature.

```
empty    :: Ord a ⇒ Heap a
insert   :: Ord a ⇒ a → Heap a → Heap a
merge    :: Ord a ⇒ Heap a → Heap a → Heap a
findMin  :: Ord a ⇒ Heap a → a
deleteMin:: Ord a ⇒ Heap a → Heap a
```

Once again we include a *naive* implementation using (ordered) lists. The other implementations are *binomial* heaps (Okasaki, 1998), *skew binomial* and *bootstrapped skew binomial* heaps (Brodal & Okasaki, 1996), *pairing* heaps (Okasaki, 1996a), *leftist* heaps (Núñez *et al*., 1995) and *splay* heaps (Okasaki, 1998).

5.1 Performance measures

We apply the terms *score* and *cost* to implementations or decision trees with the following meanings.

Definition 20 (score,cost)
Given a set of implementations for an ADT $\mathscr{A}$ and a set of benchmark tests using $\mathscr{A}$, we define the *score* obtained by each implementation $\mathscr{I}$ to be the percentage of tests for which $\mathscr{I}$ is fastest, and the (normalised) *cost* of $\mathscr{I}$ to be the average across all tests of the ratios between the execution time for $\mathscr{I}$ and the execution time of the fastest implementation for each test.

If usage attributes are known for each test, a score and a cost can also be assigned to a decision tree, based on the percentage of tests for which the implementation selected by the tree is the fastest, and on the ratios between the execution times for the implementation selected by the tree and the fastest implementation for each test.

Example 20
If a single implementation is fastest for all tests, then it has a score of 100% and a cost of 1.0. If the implementation selected by a decision tree is never the fastest, but on average takes twice the time of the quickest implementation, then the decision tree has a score of 0% and a cost of 2.0.

5.2 Correctness checks and fine-tuning

Tracing Bugs. Before we benchmark the implementations, we ensure that we have coded them correctly. Although type checking catches most accidental errors, some may remain. It is also possible that an implementation presented in the literature contains a mistake. We can use Auburn to check that implementations of the same ADT produce the same results. Auburn can generate a series of pseudo-random DUGs, searching for the smallest DUG that causes an error: that is, either an implementation fails to evaluate the DUG – for example, because of a run-time error – or two implementations return different checksums. The benchmarker outputs any anomalous DUG as a Haskell program. In the 23 initial implementations, we detected 4 faults this way. The smaller the DUG, the easier it is to find the bug. So in each case we let the benchmarker run for a long time, trying to find the smallest failing DUG. For example, the queue benchmarker found a subtle bug in our bootstrapped queue implementation. The smallest failing DUG – discovered after a run of several hours – has 22 nodes.

Fine-Tuning. Coding an implementation involves many low-level design decisions, some of which can make a significant difference to performance. Auburn helps us to make such design decisions because it can compare the overall performance of an implementation with and without a minor modification. We use the benchmarker of each ADT to time each implementation and its variants over a sample of 100 benchmarks. For each variant $\mathscr{I}$ the benchmarker reports the score and cost of the 'decision tree' made from a single leaf $\mathscr{I}$ – the tree that always chooses $\mathscr{I}$. Guided by comparisons of these scores and costs, we made nine improvements to our initial implementations, each gaining between 4% and 63% in performance.

5.3 Decision trees and their performance

For each ADT, after fine-tuning the implementations, we apply the inductive benchmarker to a training sample of 200 pseudo-random DUGs.

5.3.1 Analysis of the random-access sequence decision tree

The most accurate decision trees are too large to show and discuss here, but figure 8 shows a simplified tree for random-access sequences. 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. The significance of a leaf can be estimated from the number and proportion of winning implementations that it classifies correctly. Almost all of the leaves 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 $tail \leqslant 0.071$ 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

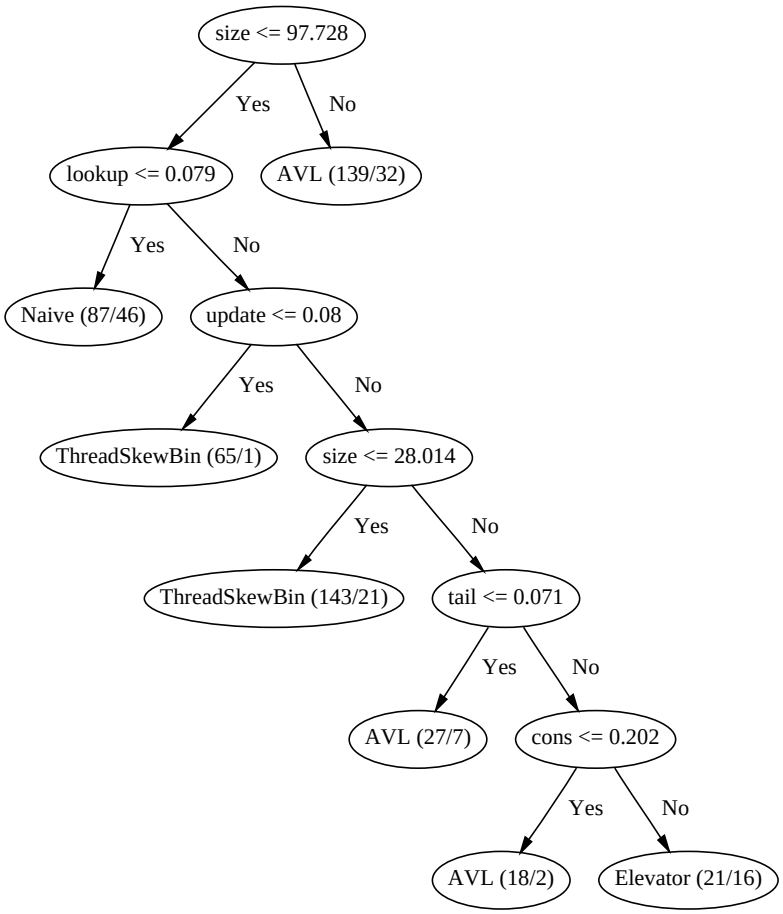


Fig. 8. A decision tree induced using the gain criterion on a training sample for the random-access sequence ADT, pruned using a reduced error method.

logarithmic complexity. The AVL implementation benefits from balancing specialised to adding or removing an element at the left, i.e. from *cons* or *tail*. It is not clear if the Adams implementation could use a similar improvement.

- *Fair size, small lookup weight (Naive)*. This decision 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 be fully evaluated only 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 force updates only on the first element. The other implementations are not as lazy, and so do not benefit as much.
- *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

Table 1. *The performance of decision trees induced by Auburn from a sample of 200* DUGs *and applied to a distinct test sample of 500* DUGs*. The size of a tree is the number of branch nodes*

ADT	Tree Size	Score (% wins)	Normalised cost
Queue	17	84	1.015
RASeq	22	78	1.093
Heap	17	85	1.045

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, it competes well with the log-time 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.

For a similar analysis of decision trees for Queues and Heaps see Moss (1999).

5.3.2 Usage-based decision vs. single implementation

One way to assess the quality of the decision trees induced by Auburn is to collect a further, larger sample of benchmarks for each ADT, and to examine the accuracy of the decision trees by applying them to these unseen test cases. For each of the three ADTs, Table 1 shows the scores and costs of the induced decision tree applied to a test sample of 500 DUGs.

Do we gain anything by choosing an implementation according to the datatype usage? How do the implementations chosen by a usage-based decision tree compare with the single implementation chosen simply because it has performed best overall in previous tests?

For queues, the Batched implementation wins for 72% of the tests and has a normalised cost of just 1.02. So it seems that for these implementations of this ADT there is little to gain by choosing the implementation according to datatype usage – one could just choose the Batched implementation regardless. However, with a score of 84% and a cost of 1.015, the decision tree does manage to improve on the fixed choice of a Batched implementation.

Similarly for heaps, the Pairing implementation wins for 80% of tests with a normalised cost of 1.08 and would make a good fixed choice. Still, the decision tree increases the score to 85%, and reduces the cost to 1.045.

For random-access sequences, decision trees succeed more convincingly. The highest scoring single implementations are AVL (36%) and ThreadSkewBin (46%), yet

the Elevator implementation has the lowest overall cost (2.12). By selecting implementations to match usage, the decision tree scores 78% and reduces the cost to 1.093 – far better than any uniform choice of a single implementation.

5.3.3 Real applications

So we can use Auburn to produce advice about the choice of implementation for at least three ADTs. But how good is this advice in practice? To answer this question, we construct several *real* benchmarks – real in that they produce useful results. We time each benchmark with each implementation and compare the results against Auburn's prediction, based on an extracted profile of the benchmark. We take four benchmarks for each ADT, and four data sets for each benchmark.

Random-Access Sequence Benchmarks. Again we give details only for random-access sequences. 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 Okasaki (1995b).

- *Bucketsort*. This sort uses random-access operations heavily, see Cormen *et al.* (1990, p. 180).
- *Quicksort*. This functional implementation of Quicksort (Hoare, 1962) does not use any random-access operations.
- *Depth-First Search (DFS)*. Implementing a graph as a random-access list of adjacent vertices (Cormen *et al.*, 1990, p. 465) allows any graph algorithm to use random-access lists. We choose one of the simplest graph algorithms, depth-first search (Cormen *et al.*, 1990, p. 477).
- *Kruskal's Minimum-Cost Spanning Tree (KMCST)*. Kruskal implements a minimum cost spanning tree algorithm (Cormen *et al.*, 1990, p. 504) using a disjoint-set data structure (Cormen *et al.*, 1990, p. 440), which we implement using a random-access list.

Table 2 gives the results. The costs of the implementations predicted by Auburn's decision trees are given alongside costs for the single implementation with the best overall performance (the least cost of any fixed choice of implementation regardless of datatype usage), and the average cost across all implementations (the expected cost of a randomly selected implementation).

Even the best uniform choice (of AVL trees) has a normalised cost of 1.837, indicating that the performance of implementations varies significantly across the benchmarks. Auburn's predictions perform significantly better on average than this uniform choice, and far better than a random choice.

Across all three ADTs, we found that Auburn gave good advice for all the real benchmarks we tested: performance of its selected implementation was within 10% of the best for queues and heaps, and within 30% of the best for random-access sequences.

Table 2. *Results from real applications of random-access sequences*

Benchmark algorithm	Data set	Winning implementation	Auburn cost	AVL cost	Average cost
Bucketsort	1	AVL	1.000	1.000	2.018
Bucketsort	2	AVL	1.000	1.000	2.405
Bucketsort	3	AVL	1.000	1.000	6.139
Bucketsort	4	AVL	1.000	1.000	3.186
DFS	1	AVL	1.000	1.000	1.748
DFS	2	Adams	1.002	1.002	2.316
DFS	3	AVL	1.000	1.000	3.075
DFS	4	AVL	1.000	1.000	5.992
KMC	1	ThreadSkewBin	1.000	1.181	1.404
KMC	2	ThreadSkewBin	1.000	2.063	1.932
KMC	3	ThreadSkewBin	1.000	1.699	1.672
KMC	4	ThreadSkewBin	1.557	1.954	1.599
Quicksort	1	Naive	1.000	4.856	3.193
Quicksort	2	Naive	1.000	3.069	2.310
Quicksort	3	Braun	1.889	1.826	1.828
Quicksort	4	Naive	1.000	4.740	3.088
		Column averages:	1.091	1.837	2.744

5.4 Locating inaccuracy in Auburn

So Auburn's advice is good, but not perfect. What can go wrong? We briefly examine the main sources of possible inaccuracy.

5.4.1 Insufficient DUG

Does the DUG model capture datatype usage sufficiently? To answer this question, we apply the following test. Take all the real application benchmarks, and run each using each ADT implementation, measuring the efficiency. Extract the DUG from each run. Evaluate each DUG using the corresponding ADT implementation. 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.

For each comparison of relative efficiencies of implementations, we calculate the *correlation coefficient*. The mean correlations are 0.924 for queues, 0.780 for heaps and 0.998 for random-access lists. The main reason why the correlation is poorest for heaps is that DUG extraction avoids problems involving polymorphism and strictness

Table 3. *Mean correlation coefficients when comparing real benchmarks with evaluation of* DUGs *generated from the benchmark profile*

ADT	DUG & Benchmark	DUG & DUG
Queue	0.859	0.923
RASeq	0.704	0.969
Heap	0.694	0.999

by not recording actual element values. Instead element values are generated pseudo-randomly, and the only constraint that can be specified is the range from which they are drawn. The impact is minor for ADTs whose operations do not depend heavily on comparisons of elements, but more marked for something like a heap in which order is of intrinsic importance.

5.4.2 Insufficient profile

Just as we design the DUG to capture datatype usage, we design the profile of a DUG to capture those aspects of datatype usage that most affect implementation efficiency. We base the whole of Auburn on this premise. We test its validity by generating several DUGs from the same profile and comparing the performance of implementations evaluating the different DUGs. If the profile of a DUG does capture datatype usage sufficiently, then the results should be similar. To avoid limiting the test to pseudo-random DUGs generated using Auburn, we extract the original profiles from real benchmarks. Table 3 shows the results. 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 significantly lower – though still quite high. This difference indicates that some important aspects of datatype usage in a benchmark are not being carried through a profile into a generated DUG. One important factor is the lack of size information: although size is captured in the profiling information, and figures prominently in decision trees, it does not influence DUG generation.

5.4.3 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, depending on the strictness of the ADT implementation evaluating the DUG.

To estimate the average proportion of a DUG not evaluated, we evaluate sample DUGs for each of the three ADTs, queue, random-access sequence, and heap, and all of their implementations. For each DUG D_0, we extract the DUG D_1 actually evaluated – by transforming a DUG evaluator for DUG extraction, as described in section 3.2.1. We then repeat this process, obtaining D_2, D_3, etc. till we obtain a fixed point, i.e. till $D_i = D_{i+1}$. In every case, we reach a fixed point on the second

iteration: $D_1 = D_2$. 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. It differs because neither DUG evaluation nor DUG extraction preserve the order of evaluation of mutations, only the order of evaluation of observations.

5.4.4 Inaccurate or over-specific trees

Some of Auburn's predictions of the best implementations for the real benchmarks are quite inaccurate. One reason for these inaccuracies is a constraint on induced decision trees. More accurate trees of similar size might employ tests on arithmetic *combinations* of attributes. However, as Quinlan (1993, Sect. 10.2) points out, introducing the possibility of such tests can slow down the process of induction by an order of magnitude.

Conversely, the very exactness of some binary decisions can be unhelpful if the expected range of values for some attribute of an application crosses a critical threshold. Recording normalised costs as well as simple scores is a help – both for programmers consulting a decision tree directly, and for pruning methods used to eliminate over-specific tests.

6 Conclusions, related and future work

Summary of contribution Previous approaches to benchmarking functional data structures have relied on hand-picked benchmarks, giving results biased towards an unknown datatype usage. This paper has described a way to automate the production of results qualified by a description of datatype usage, as implemented in the *Auburn* toolkit. The main contributions are

- A formally defined model, a DUG, of how an application uses an ADT.
- A method for extracting a DUG as a slice of an application.
- The definition of DUG profiles, summarising the most importance aspects of DUGs in vectors of numeric attributes.
- A method for creating a DUG, and hence an artificial benchmark application, from a profile of intended usage.
- The application of inductive classification to performance data for a pseudo-random sample of DUGs, deriving decision trees for the choice of data structures.
- Results of applying Auburn to over twenty data structures, implementing three different ADTs.

Despite various limitations in the way DUGs and their profiles are defined and implemented, decision trees induced by Auburn accurately predict the results of manual benchmarking using sample application programs. Auburn also has other uses. For example, it can be used to search for failing cases when testing the coding of implementations, or to asses the effect on performance of changes to the implementation of a specific data structure.

Related and future work Earlier versions of the DUG model and Auburn were more briefly reported in Moss & Runciman (1997; 1999). The fullest account of this work is in the first author's DPhil thesis (Moss, 1999).

We are not aware of any previous literature on how to benchmark functional data structures in a structured manner. Neither are we aware of any previous attempt to define a model for the pattern of use of an ADT in a lazy functional language. Some compilers do support profiling that includes counts of how often each function is called, but these counts ignore other aspects of datatype usage.

A DUG is closely related to both an *execution trace* (Okasaki, 1998) and a *version graph* (Driscoll *et al.*, 1989). 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 (Okasaki, 1998). Version graphs have been used to explain the design of persistent data structures (Dietz, 1989; Driscoll *et al.*, 1989; O'Neill & Burton, 1997).

Some published studies of imperative data structures have given detailed comparisons of performance for alternative implementations of an ADT (Arnow & Tanenbaum, 1984; Jones, 1986). However, we are not aware of any DUG-like framework that has been systematically applied in connection with such empirical studies, nor do we know of any tools like Auburn for imperative languages. It is important for us that the DUG model can handle laziness, sharing and persistence in a purely functional world, but we see no reason why a similar model should not be applicable in an imperative setting. In a call-by-value language, extraction would be more straightforward. Of course the ADT discipline must be strictly adhered to, and side-effects must be curtailed to avoid any breakdown in the correspondence with shadow computations.

Of many possible lines of further work, here are some of the main ones:

- Relax the restriction to simple ADTs as defined in section 2. In particular, include higher-order operations and operations over more than one type. For example, Auburn cannot currently benchmark the following operations:

  $$fold :: (a \to b \to b) \to b \to RASeq\ a \to b$$
  $$fromList :: [a] \to RASeq\ a$$

  One problem is the need to record structures and functions without compromising laziness; it might be possible to adapt techniques used by Gill for observing such values (Gill, 2000).
- Incorporate space information into Auburn's benchmarking procedures. Currently the only measure of performance is time.
- Improve the precision of the DUG model and its implementation – for example, an improved model of evaluation order for observations, better use of shadow information in DUG generation, and more complete information about non-version arguments in extracted DUGs.
- Allow tests on combinations of attributes in decision trees. This generalisation should increase the accuracy of the decision trees, but may slow down the induction process considerably.

One day we hope there will be a library of implementations of data structures, each recommended according to datatype usage. The work reported here is a first step in that direction.

Acknowledgements

We are pleased to thank Chris Okasaki and the anonymous referees for their helpful comments and suggestions.

References

Adams, S. R. (1993) Efficient sets – a balancing act. *J. Functional Programming*, **3**(4), 553–562.

Arnow, D. M. and Tanenbaum, A. M. (1984) An empirical comparison of B-trees, compact B-trees and multiway trees. *ACM SIGMOD Record*, **14**(2), 33–46.

Brodal, G. S. and Okasaki, C. (1996) Optimal purely functional priority queues. *J. Functional Programming*, **6**(6), 839–857.

Chuang, T.-R. and Goldberg, B. (1993) Real-time deques, multihead Turing machines, and purely functional programming. *Proc. Conference on Functional Programming Languages and Computer Architecture*, pp. 289–298. ACM Press.

Cormen, T. H., Leiserson, C. E. and Rivest, R. L. (1990) *Introduction to Algorithms*. MIT Press.

Dietz, P. F. (1989) Fully persistent arrays. *Proc. 1st Workshop on Algorithms and Data Structures: Lecture Notes in Computer Science 382*, pp. 67–74. Springer-Verlag.

Driscoll, J. R., Sarnak, N., Sleator, D. D. and Tarjan, R. E. (1989) Making data structures persistent. *J. Computer and System Sciences*, **38**(1), 86–124.

Erwig, M. (1997) Functional programming with graphs. *Proc. ACM SIGPLAN International Conference on Functional Programming*, pp. 52–65. ACM Press.

Gill, A. (2000) Debugging haskell by observing intermediate data structures. *Proc. ACM Workshop on Haskell*.

Hoare, C. A. R. (1962) Quicksort. *Computer J.* **5**(1), 10–15.

Hood, R. and Melville, R. (1981) Real-time queue operations in pure LISP. *Infor. Process. Lett*. **13**(2), 50–54.

Hoogerwoord, R. R. (1992) A symmetric set of efficient list operations. *J. Functional Programming*, **2**(4), 505–513.

Hunt, E. B., Martin, J. and Stone, P. J. (1966) *Experiments in Induction*. Academic Press.

Jones, D. W. (1986) An empirical comparison of priority-queue and event-set implementations. *Comm. ACM*, **29**(4), 300–311.

Kaplan, H. and Tarjan, R. E. (1995) Persistent lists with catenation via recursive slow-down. *Proc. 27th Annual ACM Symposium on Theory of Computing*, pp. 93–102.

Moss, G. E. (1999) *Benchmarking Purely Functional Data Structures*. PhD thesis, Department of Computer Science, University of York, UK.

Moss, G. E. and Runciman, C. (1997) Auburn: A kit for benchmarking functional data structures. *Proceedings of IFL'97: Lecture Notes in Computer Science 1467*, pp. 141–160. Springer-Verlag.

Moss, G. E. and Runciman, C. (1999) Automated benchmarking of functional data structures. *Proceedings of PADL '99: Lecture Notes in Computer Science 1551*, pp. 1–15. Springer-Verlag.

Myers, E. W. (1983) An applicative random-access stack. *Infor. Process. Lett*. **17**(5), 241–248.

Núñez, M., Palao, P. and Peña, R. (1995) A second year course on data structures based on functional programming. *Functional Programming Languages in Education: Lecture Notes in Computer Science 1022*, pp. 65–84. Springer-Verlag.

Okasaki, C. (1995a) Amortization, lazy evaluation, and persistence: lists with catenation via lazy linking. *IEEE Symposium on Foundations of Computer Science*, pp. 646–654.

Okasaki, C. (1995b) Purely functional random-access lists. *Conference Record of FPCA '95*, pp. 86–95. ACM Press.

Okasaki, C. (1995c) Simple and efficient purely functional queues and deques. *J. Functional Programming*, **5**(4), 583–592.

Okasaki, C. (1996a) Functional data structures. *Advanced Functional Programming: Lecture Notes in Computer Science 1129*, pp. 131–158. Springer-Verlag.

Okasaki, C. (1996b) *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University.

Okasaki, C. (1996c) The role of lazy evaluation in amortized data structures. *Proc. International Conference on Functional Programming*, pp. 62–72. ACM Press.

Okasaki, C. (1998) *Purely Functional Data Structures*. Cambridge University Press.

O'Neill, M. E. and Burton, F. W. (1997) A new method for functional arrays. *J. Functional Programming*, **7**(5), 487–513.

Peyton Jones, S., Nordin, T. and Reid, A. (1997) Green Card: A foreign-language interface for Haskell. *Haskell Workshop*. Oregon Graduate Institute of Science & Technology.

Quinlan, J. R. (1986) Induction of decision trees. *Machine Learning*, **1**(1), 81–106.

Quinlan, J. R. (1987) Simplifying decision trees. *Int. J. Man-Machine Studies*, **27**, 221–234.

Quinlan, J. R. (1993) *C4.5: Programs for Machine Learning*. Morgan Kaufmann.