Automated Benchmarking of Functional Data Structures

Graeme E. Moss and Colin Runciman

Department of Computer Science, University of York, UK
{gem,colin}@cs.york.ac.uk

Abstract. Despite a lot of recent interest in purely functional data structures, for example [Ada93, Oka95, BO96, Oka96, OB97, Erw97], few have been benchmarked. Of these, even fewer have their performance qualified by how they are used. But how a data structure is used can significantly affect performance. This paper makes three original contributions. (1) We present an algorithm for generating a benchmark according to a given use of data structure. (2) We compare use of an automated tool based on this algorithm, with the traditional technique of hand-picked benchmarks, by benchmarking six implementations of random-access list using both methods. (3) We use the results of this benchmarking to present a decision tree for the choice of random-access list implementation, according to how the list will be used.

1 Motivation

Recent years have seen renewed interest in purely functional data structures: sets [Ada93], random-access lists [Oka95], priority queues [BO96], arrays [OB97], graphs [Erw97], and so on. But, empirical performance receives little attention, and is usually based on a few hand-picked benchmarks. Furthermore, the performance of a data structure usually varies according to how it is used, yet this is mostly overlooked.

For example, Okasaki [Oka95] uses five simple benchmarks to measure the performance of different implementations of a list providing random access. He points out that three of the benchmarks use random access, and two do not. However, all the benchmarks are single-threaded. How do the data structures perform under non-single-threaded use? We simply do not know.

Okasaki presents many new data structures in his thesis [Oka96], but without measurements of practical performance. He writes in a section on future work: "The theory and practice of benchmarking [functional] data structures is still in its infancy."

How can we make benchmarking easier and more reliable? A major problem is finding a range of benchmarks that we know use the data structure in different ways. If we could generate a benchmark according to a well-defined use of the data structure, we could easily make a table listing performance against a range of uses.

To make precise "the use of a data structure" we need a model. Section 2 defines such a model: a *datatype usage graph*, or DUG. Section 2 also defines a *profile*, summarising the important characteristics of a DUG. Section 3 gives an algorithm for generating a benchmark from a profile. Section 4 introduces a benchmarking kit, called *Auburn*, that automates benchmarking using the algorithm of Sections 3. Section 4 then compares benchmarking six implementations of random-access lists manually against using Auburn. Section 5 discusses related work. Section 6 concludes and discusses future work.

Some of the details of this paper are only relevant to the language we use: Haskell, a pure functional language using lazy evaluation. Such details are clearly indicated.

2 Modelling Datatype Usage

How can we capture the way an application uses a data structure? Take the *Sum* benchmark of [Oka95] as an example of an application. Sum uses an implementation of random-access lists (see Fig. 1) to build a list of n integers using cons, and then sum this list using head and tail. Code for Sum is given in Fig. 2(a).

Let us use a graph to capture how Sum uses the list operations. Let a node represent the result of applying an operation, and let the incoming arcs indicate the arguments taken from the results of other operations. Let any other arguments be included in the label of the node. Figure 2(b) shows this graph.

Node 1 represents the result of empty, which is an empty list. Node 2 represents the result of applying cons to 1 and empty, which is a list containing just the integer 1. And so on, till node $n+1$ represents a list of n copies of the integer 1. This is how Sum builds a list of n integers.

Node $n+2$ represents the result of applying head to this list, which is the first element in the list. Node $n+3$ represents the result of applying tail to this list, which is all but the first element. Node $n+4$ represents the result of applying head to the list of node $n+3$, giving the second element. Every other element of the list is removed in the same way, till node $3n$ represents the last element, and node $3n+1$ represents the empty list. This is how Sum sums the list of n integers.

The authors introduced such a graph in [MR97], given the name *datatype usage graph*, or DUG. The definition was informal in [MR97] but we shall now give a brief formal definition. To abstract over many competing data structures providing similar operations, we insist on a DUG describing the use of an ADT. The same DUG can then describe the use of any implementation of that ADT. We restrict an ADT to being *simple*.

Definition 1 (Simple ADT)
A *simple* ADT provides a type constructor T of arity one, and only operations over types shown in Table 1.

Example 1
The random-access list ADT, whose signature is given in Fig. 1, is a simple ADT,

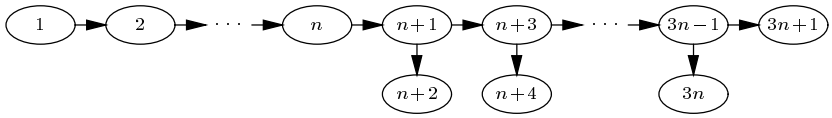
```
empty  :: List a
cons   :: a → List a → List a
tail   :: List a → List a
update :: List a → Int → a → List a
head   :: List a → a
lookup :: List a → Int → a
```

Fig. 1. The signature of a random-access list abstract datatype (ADT).

(a)

```
sum :: Int → Int
sum n = sumList n (buildList n)
        where buildList 0 = empty
              buildList k = cons 1 (buildList (k-1))
              sumList 0 xs = 0
              sumList k xs = head xs + sumList (k-1) (tail xs)
```

(b)



Node	Operation
1	empty
$2, \ldots, n+1$	λx · cons 1 x
$n+2, n+4, \ldots, 3n$	λx · head x
$n+3, n+5, \ldots, 3n+1$	λx · tail x

Fig. 2. The Sum benchmark of [Oka95] using the operations of the ADT in Fig. 1. (a) Code. (b) DUG.

Table 1. The types of operations allowed in a simple ADT providing a type constructor T, where a is a type variable. Any value of type T a is called a *version*. Each ADT operation is given a single *role*. A *generator* takes no versions, but returns a version. A *mutator* takes at least one version, and returns a version. An *observer* takes at least one version, but does not return a version.

Argument Types	Result Type	Role
a, Int	T a	Generator
T a (at least one), a, Int	T a	Mutator
T a (at least one), a, Int	a, Int, Bool	Observer

providing the following: a type constructor List; a generator empty; mutators cons, tail, and update; and observers head and lookup.

The nodes are labelled by a function η with partial applications of ADT operations.

Definition 2 (DUG PAP)
A DUG PAP (Partial APplication) is a function f obtained by supplying zero or more atomic values as arguments to any operation g of a simple ADT. We say that f is a PAP of g.

Example 2
The DUG PAPs that label nodes of Fig. 2(b) are: empty, λx $\cdot$ cons 1 x, λx $\cdot$ head x, and λx $\cdot$ tail x.

To indicate which argument corresponds to which arc, if more than one arc is incident to a node, we order the arcs with positive integers using a function τ. To indicate the order of evaluation, we label each node with a positive integer using a function σ.

Definition 3 (DUG)
Given a simple ADT $\mathcal{A}$, a DUG is a directed graph $\mathcal{G}$ with the following functions:
- A function η labelling every node with a PAP of an operation of $\mathcal{A}$
- A function τ, which for all nodes with two or more incoming arcs, when restricted to those arcs, is a bijection with the set $\{1, \ldots, j\}$ for some $j \geq 2$
- A bijection σ from the nodes to the set $\{1, \ldots, n\}$ for some $n \geq 0$

The following conditions must also be true:
C_1 If w is a successor of v, then $\sigma(w) > \sigma(v)$.
C_2 Every node labelled by η as an observer has no arcs from it.
The reasons for imposing these conditions are given in the problems list of Sect. 3.1.

Example 3
A DUG is shown in Fig. 2(b). The function η is given by the table, τ is redundant as no node has more than one incoming arc, and σ is given by the number labelling each node.

To help identify important characteristics of use, we compact important information from a DUG into a *profile*. One piece of important information is the degree of *persistence*, that is, the re-use of a previously mutated data structure.

Definition 4 (Persistent Arc)
Consider the arcs from some v to $v_1, \ldots, v_k$. Let v_i be the mutator node ordered earliest by σ, if it exists. Any arc from v to some v_j ordered *after* v_i represents an operation done *after* the first mutation, and is therefore *persistent*.

The properties making up a profile reflect important characteristics of datatype usage. They are fashioned to make benchmark generation easy. Other reasons for choosing these properties are not discussed here – see [Mos99].

Definition 5 (Profile)
The profile of a DUG comprises five properties:

- *Generation weights*: The ratio of the number of nodes labelled with each generator.
- *Mutation-observation weights*: The ratio of the number of arcs leading to each mutator or observer.
- *Mortality*: The fraction of generator and mutator nodes that have no arcs leading to a mutator.
- *Persistent mutation factor* (PMF): The PMF is the fraction of arcs to mutators that are persistent.
- *Persistent observation factor* (POF): The POF is the fraction of arcs to observers that are persistent.

Example 5
The profile of the DUG of Fig. 2(b) is as follows: the generation weights are trivial as there is only one generator, the mutation-observation weights are

$$\mathsf{cons} : \mathsf{tail} : \mathsf{update} : \mathsf{head} : \mathsf{lookup} = n : n : 0 : n : 0,$$

the mortality is $1/(2n+1)$, and the PMF and the POF are both zero.

3 Benchmark Generation

If we can generate a DUG with a given profile, then a DUG *evaluator* can act as a benchmark with a well-defined use of data structure (the given profile). There are many DUGs with a given profile, but choosing just one might introduce bias. Therefore we shall use probabilistic means to generate a *family* of DUGs that have *on average* the profile requested.

3.1 DUG Generation

Here are some problems with DUG generation, and the solutions we chose:

- *Avoid ill-formed applications of operations.* For example, we need to avoid forming applications such as head empty. We introduce a *guard* for each operation, telling us which applications are well-defined. See the section *Shadows and Guards* below for more details.
- *Order the evaluation.* We cannot reject an ill-formed application of an operation till we know all the arguments. Therefore a node must be constructed after its arguments. Under the privacy imposed by ADTs and the restrictions imposed by lazy evaluation, we can only order the evaluation of the observers. For simplicity, we evaluate the observers as they are constructed. The function σ therefore gives the order of construction of all nodes, and the order of evaluation of observer nodes. This is condition C_1 of Defn. 3.

(a) **type** Shadow = Int

```
empty_S  :: Shadow                          empty_S       = 0
cons_S   :: Int → Shadow → Shadow           cons_S x s    = s+1
tail_S   :: Shadow → Shadow                 tail_S s      = s–1
update_S :: Shadow → Int → Int → Shadow     update_S s i x = s
```

(b)

```
empty_G  :: Bool                            empty_G       = True
cons_G   :: Shadow → [IntSubset]            cons_G s      = [All]
tail_G   :: Shadow → Bool                   tail_G s      = s>0
update_G :: Shadow → [IntSubset]            update_G s    = [0:..:(s–1),All]
head_G   :: Shadow → Bool                   head_G s      = s>0
index_G  :: Shadow → [IntSubset]            index_G s     = [0:..:(s–1)]
```

Fig. 3. (a) Shadow operations for random-access lists, maintaining the length of the list. (b) Guards for random-access lists. A guard returns a list of integer subsets, one for each non-version argument, in order. (Haskell does not support functions over tuples of arbitrary size – necessary for the application of an arbitrary guard – so we are forced to use lists.) If an operation does not take any non-version arguments, a boolean is returned. The type IntSubset covers subsets of the integers. The value m:..:n indicates the subset from m to n inclusive, and All indicates the maximum subset.

– *Choose non-version arguments.* Version arguments are of type T a, and can be chosen from the version results of other nodes. Choosing non-version arguments in the same way is too restrictive – for example, where does the argument of type a for the first application of cons come from? Within the type of each operation, we instantiate a to Int, so we need only choose values of type Int. For simplicity, we avoid choosing *any* non-version arguments from the results of other nodes. This is condition C_2 of Defn. 3.

Shadows and Guards. To avoid creating ill-defined applications whilst generating a DUG, we maintain a *shadow* of every version. The shadow contains the information needed to avoid an ill-defined application of any operation to this version. We create the shadows using shadow operations: the type of a shadow operation is the same except that every version is replaced by a shadow. A *guard* takes the shadow of every version argument, and returns the ranges of non-version arguments for which the application is well-defined. Any combination of these non-version arguments must provide a well-defined application.

For example, for random-access lists we maintain the length of the list associated with a version node. This allows us to decide whether we can apply head to this node: if the length is non-zero, then we can; otherwise, we cannot. Similarly, applications of other operations can be checked. Figure 3(a) gives shadow operations for random-access lists and Fig. 3(b) gives guards.

The Algorithm. We build a DUG one node at a time. Each node has a *future*, recording which operations we have planned to apply to the node, in order. The first operation in a node's future is called the *head operation*. The nodes with a non-empty future together make up the *frontier*. We specify a minimum and a maximum size of the frontier. The minimum size is usually 1, though a larger value encourages diversity. Limiting the maximum size of the frontier caps space usage of the algorithm.

Figure 4 shows the algorithm. We make a new node by choosing version arguments from the frontier. We place each argument in a buffer according to the argument's head operation. When a buffer for an operation *f* is full (when it contains as many version arguments as *f* needs), we empty the buffer of its contents *vs*, and attempt to apply *f* to *vs* using the function *try_application*.

Calling *try_application*(*f*,*vs*) uses the guard of operation *f* to see whether *f* can be applied to version arguments *vs*. If one of the integer subsets returned by the guard is empty, no choice of integer arguments will make the application well-formed, and so we must abandon this application. Otherwise, the guard returns a list of integer subsets, from which we choose one integer each, to give the integer arguments *is*. Applying *f* to *vs* and *is* gives a new node.

Planning for the Future. We plan the future of a new node v as follows. We use the five profile properties to make the DUG have the profile requested, on average. Mortality gives the probability that the future of v will contain no mutators. If the future will contain at least one mutator, then the fraction of persistent mutators should equal PMF. Every application of a mutator bar the first is persistent. Therefore the number m of persistent mutators is given by:

$$\frac{m}{m+1} = \text{PMF} \Rightarrow m = \frac{\text{PMF}}{1 - \text{PMF}}$$

Hence a random variable with mean $\text{PMF}/(1 - \text{PMF})$ gives the number of additional mutators. Note that a PMF of 0 guarantees each node is mutated at most once.

The mutation-observation weights ratio allows us to calculate the average number r of applications of observers per application of a mutator. We assume a mutator made v, and let a random variable with mean r decide the number of observers in the future of v. Typically the number of applications of generators is very small, and so this assumption is reasonable. The number of observers ordered after the first mutator is given by a random variable with mean POF. Finally, we choose which mutators and observers to use from probabilities given by the mutation-observation weights ratio.

Note that condition C_2 of Defn. 3 implies that every observer node has no future.

3.2 DUG Evaluation

A DUG evaluator uses an implementation of the ADT to evaluate the result of every observation. The nodes are constructed in the same order that they were

```
generate_dug() :=
    dug := {}
    frontier := {}
    ∀f·buffer(f) := {}
    while #dug < final_dug_size do
        if #frontier < min_frontier_size then
            g := choose a generator using generation weights ratio
            try_application(g,{})
        else-if #frontier > max_frontier_size then
            remove a node from frontier
        else
            v := remove a node from frontier
            f := remove head operation of v
            add v to buffer(f)
            if #buffer(f) = number of version arguments taken by f then
                vs := buffer(f)
                buffer(f) := {}
                try_application(f,vs)
            fi
        fi
    od

try_application(f,vs) :=
    int_subsets := apply guard of operation f to shadows of vs
    if each set in int_subsets is not empty then
        is := choose one integer from each set in int_subsets
        v := make node by applying f to version arguments vs and integers is
        shadow of v := apply shadow of f to shadows of vs
        if f is an observer then
            future of v := empty
        else
            plan future of v
        fi
        add v to dug
        if v has a non-empty future then
            add v to frontier
        fi
    fi
    add each node in vs with a non-empty future to frontier
```

Fig. 4. DUG generation algorithm.

generated. An observer node is evaluated immediately. This fits the intended behaviour (see the second problem listed for DUG generation).

4 An Example of a Benchmarking Experiment

Auburn is an automated benchmarking kit built around the benchmark generation of Sect. 3. We shall benchmark six implementations of random-access lists (1) using Auburn, and (2) using benchmarks constructed manually. We shall then compare these two methods.

4.1 Aim

We aim to measure the performance of six implementations of random-access lists: *Naive* (ordinary lists), *Threaded Skew Binary* (Myers' stacks [Mye83]), *AVL* (AVL trees [Mye84]), *Braun* (Braun trees [Hoo92]), *Slowdown* (Kaplan and Tarjan's deques [KT95]), and *Skew Binary* (Okasaki's lists [Oka95]). We will qualify this performance by two properties of use:

- *Lookup factor*. The number of applications of lookup divided by the number of applications of ordinary list operations. We use just two settings: 0 and 1.
- *Update factor*. This is as lookup factor but replacing lookup with update. Again, we use only two settings: 0 and 1.

There are 4 different combinations of settings of these properties.

4.2 Method

Auburn. We use Auburn version 2.0a. For the latest version of Auburn, see the Auburn Home Page [Aub]. Perform the experiment using Auburn as follows:

- Copy the makefile for automating compilation of benchmarks with the command: `auburnExp`.
- With each random-access list implementation in the current directory, each file ending `List.hs`, make the benchmarks with: `make SIG=List`. This includes making shadows and guards for random-access lists (see Sect. 3.1). Auburn guesses at these from the type signatures of the operations. The makefile will stop to allow the user to check the accuracy of the guess. In the case of random-access lists, it is correct. Restart the makefile with: `make`.
- The makefile also makes a *null implementation*, which implements an ADT in a type-correct but value-incorrect manner. It does a minimum of work. It is used to estimate the overhead in DUG evaluation.
- Make a profile for each of the 4 combinations of properties. Auburn makes a Haskell script to do this. It requires a small change (one line) to reflect the properties and settings we chose.
- Make DUGs from these 4 profiles with: `makeDugs -S 3 -n 100000`. This uses 3 different random seeds for each, creating 12 DUGs, each DUG containing 100000 nodes.

- Run and time each of the 7 DUG evaluators on each of the 12 DUGs. Evaluate each DUG once – internal repetition of evaluation is sometimes useful for increasing the time taken, but we do not need it for this experiment. Take three timed runs of an evaluator to even out any glitches in recorded times. Use: `evalDugs -R 1 -r 3`.
- Process these times with: `processTimes`. This command sums the times for each implementation and profile combination, subtracts the null implementation time, and finally divides by the minimum time over all implementations. This gives an idea of the ratio of work across implementations per profile.

Manual. Perform the experiment without Auburn as follows:

- Construct benchmarks that use random-access lists in manners that cover the 4 properties of use. Take four of the five benchmarks of [Oka95], neglecting Quicksort. Alter them to match one of the 4 properties of use, to force work (as Haskell is lazy), and to construct lists before using them (as it is hard to separate the time of use from the time of construction). Here are the resulting benchmarks:
  - *Sum.* Construct a list of size n using cons, and sum every element using head and tail.
  - *Lookup.* Construct a list of size n using cons, and sum every element using lookup.
  - *Update.* Construct a list of size n using cons, update every element using update, update every element twice more, and sum every element using head and tail.
  - *Histogram.* Construct a list of n zeros using cons. Count the occurrence of each integer in an ordinary list of $3n$ pseudo-randomly generated integers over the range $0, \ldots, n-1$, using lookup and update to store these counts. Sum the counts with head and tail to force the counting.
- Work out what values of n to use in each of the above benchmarks to fix the number of operations done by each benchmark to approximately 100000, for consistency with the Auburn method. Use a loop within the benchmark to repeat the work as necessary.
- Run and time these benchmarks using each implementation (including the null implementation). As with the Auburn method, time three runs, sum these times, subtract the null implementation time, and divide by the minimum time.

4.3 Results

Tables 2 and 3 give the results. The tables agree on the winner of three of the four combinations of lookup factor and update factor. Naive is the best implementation when no random-access operations are used. Threaded Skew Binary is the best when only lookup operations are used. AVL is the best when both lookup and update operations are used.

Table 2. Ratios of times taken to evaluate benchmarks constructed by Auburn. Null implementation times were subtracted before calculating the ratios. An entry marked "–" indicates the benchmark took too long – the ratio would be larger than for any ratio given for another implementation.

Auburn Results								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
DUG Evaluator	0	0	1.0	5.8	127.8	36.3	15.0	18.1
	0	1	1.0	–	8.9	56.5	14.1	3.7
	1	0	–	1.0	1.4	7.2	4.3	3.8
	1	1	–	51.4	1.0	6.9	4.4	3.7

Table 3. As Table 2 but for hand-picked benchmarks.

Manual Results								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
Sum	0	0	1.0	2.5	171.6	24.9	18.6	2.8
Update	0	1	–	–	1.0	4.8	3.4	2.6
Lookup	1	0	–	1.0	3.3	9.4	5.8	4.7
Histogram	1	1	–	3.1	1.0	4.7	2.8	3.0

Table 4. As Table 2 but with PMF and POF set to 0.2.

Auburn Results – PMF and POF = 0.2								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
DUG Evaluator	0	0	1.0	5.8	37.5	6.9	12.0	13.6
	0	1	1.0	6.2	4.9	10.3	4.7	4.4
	1	0	3.7	1.0	1.9	5.8	3.3	3.1
	1	1	1.7	1.2	1.0	4.0	2.2	2.1

But what about when only update operations are used? The Auburn results show Naive as the winner, whereas the manual results show AVL as the winner, with Naive as one of the worst! The probable cause of this difference is that when the relevant DUGs were evaluated, updates on elements towards the rear end of the list were not forced by the only observing operation: head. As Naive is very lazy, it benefits greatly. For the manual benchmarks, this does not happen because every element is forced. Adding a maximal sequence of applications of head and tail to the DUGs and re-timing evaluations produced the same results as the manual benchmarks. This adds weight to our suspicion of the cause of difference between times. As most applications will force most operations, we conclude that AVL is the best implementation when only update operations are used.

Although the Auburn and manual results differ in scale, the order in which they place the implementations are almost always the same. From the results of other experiments, we suspect that the differences are probably due in the main to differences in the sizes of the lists involved. The size of a data structure can significantly affect performance. Unfortunately, this characteristic has proved hard to capture completely. The mutation weights ratio goes someway to capturing size, but neglects the order of applications of mutators. The size of a version can be deduced from the operations used to produce it, but this does not help us to produce versions of a given size: We can measure size, but we cannot completely influence it.

4.4 Comparing Auburn with Manual

Although the description of how to perform this experiment was longer for Auburn than for manual, the user actually has to do less. Auburn automates most of the work, whereas manual benchmarks need to be designed, built, tested, compiled, run, and have the timings collected and analysed. The two most laborious steps for the Auburn user are:

- *Make the shadows and guards.* Auburn can guess at shadows and guards by inspecting the operation types. Auburn manages to guess correctly for many data structures: lists, queues, deques, priority queues, and so on, with or without random-access and catenation. For other data structures, minor corrections can be made to this guess. For more exotic data structures, Auburn can generate trivial shadows and guards, from which the user can build their own.
- *Make the profiles.* Auburn creates a simple script to do this, which needed a small change (one line) to suit our experiment. Further small changes yield other results easily. To illustrate this, let us consider two examples:
  - *Persistence.* How do the results change if we add persistence? For Auburn, we make another small change (one line) to the profiles script, setting PMF and POF to 0.2. We then re-run the experiment with three commands. The results are given in Table 4. Although a few marked changes occur (for example, for Naive with lookup factor non-zero), the winner

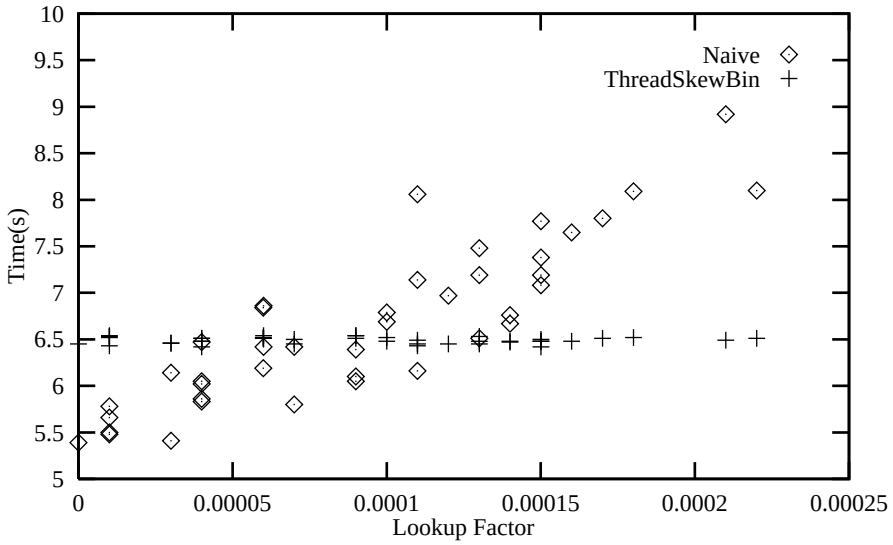


Fig. 5. Times taken to evaluate DUGs with lookup factors calculated from their profiles. Update factor is 0, and each DUG contains 100000 nodes. Each DUG was evaluated three times, and the total time recorded.

remains the same in each category. This shows that in this case, persistence does not change which implementation is best. With manual benchmarks, adding persistence would have been a lot harder.

- *Boundaries.* For update factor equal to 0, we know that Naive is the best for lookup factor equal to 0, and Threaded Skew Binary for lookup factor equal to 1. But where is the boundary at which they perform the same? Again, using Auburn, this requires a small change (one line) to the profiles script, and re-entering three commands. The results are given in Fig. 5. We see that the boundary is approximately 0.0001, which is surprisingly small. With manual benchmarks, the lack of automation would have imposed more work on the user.

5 Related Work

The authors published a paper at IFL'97 [MR97] benchmarking queues using Auburn. Auburn was built around queues before being generalised, and this paper shows that Auburn can cope with other data structures. We also extend [MR97] by formalising the definition of a DUG. We also discuss the problems of benchmark generation, and the solutions we chose, including an algorithm for DUG generation. We make a direct comparison of benchmarking using Auburn with benchmarking using the traditional technique of manual benchmarks. We discuss the advantages and disadvantages of Auburn over manual benchmarking,

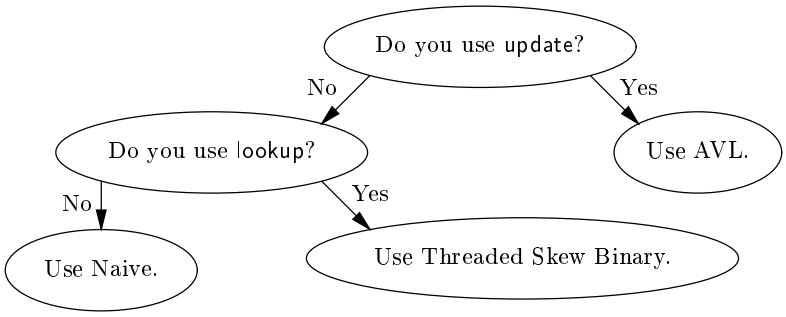


Fig. 6. Decision tree for choosing an implementation of random-access lists.

and provide some examples of where Auburn is more useful. Finally we present some advice on which random-access list implementation to use in Haskell, based on the benchmarking results. We know of no other attempt to automate benchmarking of functional data structures.

Okasaki [Oka95] and O'Neill [OB97] benchmark functional random-access lists and arrays, respectively. Okasaki uses five benchmarks, three of which use random-access, and two do not. However, none of the benchmarks use persistence. O'Neill uses two persistent benchmarks, one of which is randomly generated. However, there is no mention of the relative frequency of lookup and update.

6 Conclusions and Future Work

We have formalised a model of how an application uses a data structure. We have described the problems of benchmark generation, the solutions we chose, and a resulting algorithm. We have also used an automated tool (Auburn) based on this algorithm to benchmark six implementations of random-access list. We compared using Auburn to using hand-picked benchmarks, and provided some examples of where Auburn is more useful.

From the results of Sect. 4, we provide a decision tree in Fig. 6 to guide users choosing an implementation of random-access lists. Further experiments using more profiles and more implementations would refine this decision tree.

Future work would involve: lifting the restrictions on ADTs by allowing higher-order operations, and operations between two or more data structures (eg. to and from ordinary lists); lifting the restrictions on DUGs by separating order of construction from order of evaluation, and allowing dependencies on observations; understanding the concept of persistent observations in a lazy language; capturing size of data structure more adequately in the profile; setting down some guidelines on benchmarking.

We dream of a time when a library of functional data structures provides detailed decision trees for *every* ADT implemented.

References

[Ada93] Stephen R. Adams. Efficient sets – a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.

[Aub] The Auburn Home Page. `http://www.cs.york.ac.uk/~gem/auburn/`.

[BO96] Gerth S. Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, November 1996.

[Erw97] 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.

[Hoo92] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Proceedings of the Second International Conference on the Mathematics of Program Construction*, volume 669 of *LNCS*, pages 191–207, July 1992.

[KT95] 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.

[Mos99] Graeme E. Moss. *Benchmarking Functional Data Structures.* DPhil thesis, University of York, 1999. To be submitted.

[MR97] 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.

[Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.

[Mye84] Eugene W. Myers. Efficient applicative data types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.

[OB97] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997.

[Oka95] Chris Okasaki. Purely functional random-access lists. In *Conference Record of FPCA '95*, pages 86–95. ACM Press, June 1995.

[Oka96] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.