Auburn: A Kit for Benchmarking Functional Data Structures

Graeme E. Moss and Colin Runciman

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

Abstract. Benchmarking competing implementations of a data structure can be both tricky and time consuming. The efficiency of an implementation may depend critically on how it is used. This problem is compounded by persistence. All purely functional data structures are persistent. We present a kit that can generate benchmarks for a given data structure. A benchmark is made from a description of how it should use an implementation of the data structure. The kit will improve the speed, ease and power of the process of benchmarking functional data structures.

1 Motivation

Purely functional data structures differ from their imperative counterparts as the original value of a data structure may continue to exist after an update. Data structures that allow this are termed *persistent*. A sequence of operations that never uses this property is called *single-threaded*: after a data structure has been updated, its previous value is never reused.

When analysing the performance of competing implementations of an abstract datatype (ADT), it is common to choose a handful of applications, or benchmarks, and time each application using each implementation of the ADT. However, the choice of application can significantly affect the performance of each implementation. Moreover, the reasons why an implementation suits one application better than another is often not clear. Persistence compounds this problem: the performance of implementations can differ substantially between applications that take advantage of persistence to different degrees.

To illustrate this, consider the queue ADT whose signature is given in Fig.1. Listed below are three simple implementations of this ADT (code is given in Appendix A):

Naïve queues are implemented as ordinary lists. This provides $O(1)$ access to the front of the queue and $O(n)$ access to the rear, where n is the length of the queue.

Batched queues use a pair of lists to represent the front and rear of the queue [HM81]. This provides $O(1)$ amortized access to the front and rear of the queue, provided that the queue is used in a single-threaded manner, otherwise the complexity degenerates to $O(n)$. The name is taken from [Oka96a].

```
module Queue (Queue,empty,snoc,tail,head,isEmpty) where
empty :: Queue a
snoc :: Queue a -> a -> Queue a
tail :: Queue a -> Queue a
head :: Queue a -> a
isEmpty :: Queue a -> Bool
```

Fig. 1. A signature for a queue ADT.

Banker's queues use the same implementation as batched queues but ensure that the rear list is never bigger than the front list [Oka95]. This provides $O(1)$ amortized access to the front and rear of the queue regardless of the way in which the queue is used.

To see how these implementations behave under different uses of queues, the artificial simple applications shown in Fig.2 are run using each of the three implementations. The results are shown in Table 1.

These three applications were chosen to reveal how differently the queue implementations can behave according to how they are used. The huge values for `app2` using naïve queues and `app3` using batched queues are extreme examples of this. We might be able to guess at why this is so, especially where the implementations and applications are simple, as is the case here. However, this is by no means an easy task, and we would need to write and test several more applications to confirm any hypothesis.

To illustrate that these artificial applications are not unusual in how they distinguish between implementations, we also considered an application that implemented Shell's sort [She59] using queues. Shell's sort depends on using insertion sort on successively larger sublists. The partitioning into sublists depends on a set of increments which may be adjusted to obtain greater efficiency. Using two different sets of increments and each implementation of queues, the same list of 1500 pseudo-random integers was sorted. The running times are given in Table 2. Code implementing Shell's sort using queues is given in Appendix A.

Again we see that a change in how the queue is used can change the efficiency of the queue drastically. Naïve queues perform well on the small set of increments, but very badly on the large set. But why? Given a knowledge of how this application uses the queues, one might guess that it is because the queues are much larger when using the larger set of increments. However, we cannot draw this conclusion from just looking at the times provided by Shell's sort.

To address this problem, we describe a kit named *Auburn* that generates applications which use an ADT in a specified manner. We may then measure the efficiency of competing implementations of the ADT when used by the generated applications. This will allow us to draw conclusions about the efficiency of an implementation of an ADT according to how it is used. For example, the results of this paper back up the suspicion that large naïve queues are rather inefficient.

```
-- 'apply n f q' applies 'f' to 'q' 'n' times,
-- ie. 'f (f .. (f q) ..)'
apply :: Int -> (a -> a) -> a -> a
apply 0 _ q = q
apply n f q = apply (n-1) f (f q)

-- 'snocTrue q' snoc's 'True' onto 'q'.
snocTrue :: Queue Bool -> Queue Bool
snocTrue q = snoc q True

-- 'app1 n' calculates the 'and' of 'n' _separate_ copies of:
-- '(head . tail . tail . snocTrue . snocTrue . snocTrue) empty'
app1 :: Int -> Bool
app1 n = (and . map getTrue . take n . repeat) ()
   where getTrue () =
            (head . apply 2 tail . apply 3 snocTrue) empty

-- 'app2 n' performs 'n' snoc's followed by 'n-1' tails,
-- finishing with a 'head'.
app2 :: Int -> Bool
app2 n = (head . apply (n-1) tail . apply n snocTrue) empty

-- 'app3 n' performs 'n' snoc's to give queue 'q'.
-- Then it performs tail then head on 'n' _shared_ copies of 'q'.
app3 :: Int -> Bool
app3 n = (and . map (head . tail) . take n . repeat) q
   where q = apply n snocTrue empty
```

Fig. 2. Three artificial simple applications of queues: `app1`, `app2` and `app3`.

Table 1. Average times in seconds over three runs of each application using each implementation of a queue. Standard deviation was less than 1% of the mean in every case.

Application	Naïve	Batched	Banker's
`app1 1000000`	4.66	4.84	4.95
`app2 100000`	16375.87	0.73	1.32
`app3 100000`	1.51	11032.00	1.53

Table 2. Average times in seconds over three runs of Shell's sort using each set of increments and each implementation of a queue on the same list of 1500 pseudo-random integers. Standard deviation was less than 1% of the mean in every case.

Increments	Naïve	Batched	Banker's
1, 3, 7, 21, 48, 112, 336, 861	12.89	0.53	0.58
1, 7, 48	0.55	0.40	0.41

Section 2 describes the underlying framework of the kit: labelled graphs called *dugs* that capture how an ADT is used by an application. Section 3 imposes restrictions on ADTs. Section 4 imposes restrictions on dugs. Section 5 describes how a dug may be condensed into a *profile*. Section 6 describes a method for avoiding ill-defined applications of randomly-chosen operations. Section 7 gives an example of the kit in action on the queue implementations of this section: naïve, batched, and banker's. Section 8 concludes and outlines some future work. Appendix A gives code for the three implementations of queues.

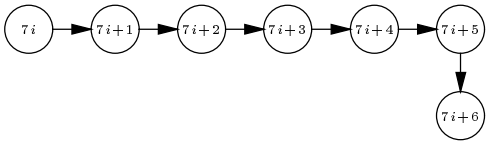
2 Datatype Usage Graph

A major obstacle to overcome is the ambiguity of the phrase "how an ADT is used". Without an exact definition of this property, we would find it hard to talk about the efficiency or otherwise of an implementation of an ADT according to how it is used, or indeed about how a particular application uses an ADT. Consider the three applications of queues in Fig.2. Inspecting the code for each application allows us to see what operations are being performed, in what order and how the result of one operation may rely on the result of another. But the task is by no means straightforward. With more complicated applications, the task would become extremely difficult. We need a simple record of how an ADT is used by an application.

We use a labelled directed graph. See Fig.3 for examples that describe how the queue ADT is used by the three applications of Fig.2. The nodes are labelled with partially applied operations of the ADT—each operation has all arguments that are *not* ADT values already supplied. 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. If v has more than one incoming arc, these arcs must be labelled with the order in which they are taken by the operation at v. The nodes are numbered according to the order of evaluation. Such a graph is a *datatype usage graph*, or *dug*. Dugs are very similar to *execution traces* [Oka96a] and *version graphs* [DSST89].

During the run of an application, many different instances of an ADT will exist. For example, whilst running queue application `app1` there will exist at some time an empty queue, a queue containing just `True`, a queue containing two copies of `True`, and so on. Each of these particular instances of the ADT is called a *version* [Oka96a]. A node of a dug is called a *version node* if it is labelled with an operation that results in a version. The subgraph of a dug containing just the version nodes is called the *version graph*. This is consistent with the definition of a version graph given by Driscoll et al. [DSST89]. As each operation returns only a single value, we may associate each node with the value it produces. The nodes of the version graph can be thought of as representing different versions of the ADT formed by either generating a fresh version or by mutating one or more previous versions. In the latter case, the arcs specify which versions to give to the mutating operation and represent the flow of data *within* the privacy of the ADT framework. The nodes outside of the version graph and

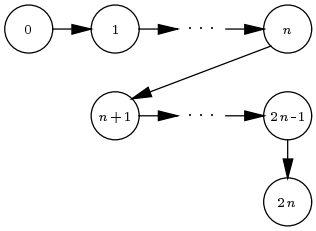
app1:



(One copy of this graph
for each i in $\{0, \ldots, n-1\}$.)

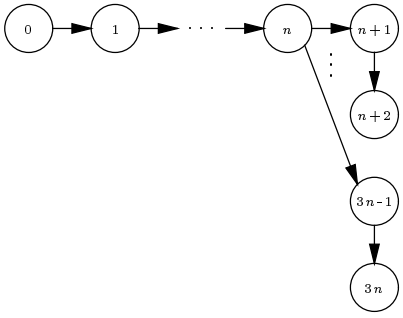
Node	Operation
$7i$	*empty*
$7i+1$	$\lambda l \cdot$ *snoc* l True
$7i+2$	$\lambda l \cdot$ *snoc* l True
$7i+3$	$\lambda l \cdot$ *snoc* l True
$7i+4$	$\lambda l \cdot$ *tail* l
$7i+5$	$\lambda l \cdot$ *tail* l
$7i+6$	$\lambda l \cdot$ *head* l

app2:



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

app3:



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

Fig. 3. Dugs showing how the queue ADT is used by the different applications given in Fig.2.

the arcs connecting them represent the flow of data *out* of the privacy of the ADT framework.

3 Simple Abstract Datatype

To simplify early releases of Auburn, we restrict the ADTs considered. A *simple* ADT satisfies the following:

- The ADT defines one type constructor T of arity one.
- The ADT only defines *simple operations*. An operation is simple if
  - its type contains at most one type variable, say a,
  - every argument is of type $T\ a$, a or *Int*,
  - the result is of type $T\ a$, a, *Int* or *Bool*, and
  - at least one argument or the result is of type $T\ a$.

Every simple operation may be classified as a

generator if no argument is of type $T\ a$, or as a
mutator if at least one argument is of type $T\ a$ and so is the result, or as an
observer if at least one argument is of type $T\ a$ but the result is not.

This classification is called the *role* of an operation. Any simple operation has a single role.

For example, the queue ADT whose signature is given in Fig.1 is simple and defines one generator: *empty*, two mutators: *snoc* and *tail*, and two observers: *head* and *isEmpty*.

4 Simple Dug

The following requirements ensure we have a well-defined dug:

- The number of incoming arcs to a node must match the arity of the operation it is labelled with (where the arity of an operation is the number of arguments of type $T\ a$).
- The operations must be type consistent. There must be at least one type to which we are able to instantiate the type variable "a" in every operation label.

As with the restrictions placed on ADTs, we shall impose the following restrictions on dugs to simplify early releases of Auburn:

- The dug must be defined over a simple ADT.
- The ordering given to the nodes must not conflict with the ordering given by the arcs. If node u is the predecessor of node v, u must be ordered before v. Note that this also implies that the dug must be acyclic.
- There must be no outgoing arcs from a node labelled with an observer operation, that is, no dependency must be made on the results of observations.

We discuss in Sect. 8 how these simplifications may be removed.

5 Dug Profile

A dug captures information relevant to how an ADT is used by an application. However, it is rather hard to talk about a large graph in detail, so we shall condense the most important information into a *profile*.

How often one operation is used relative to another is one of the most obvious candidates for a profile. We define the *weight* of an operation f to be the number of arcs leading to nodes labelled with f. For the purpose of this definition, consider all generators as operating on a unique void version.

Another important characteristic to capture is the degree of persistence. We split this into persistent mutations and persistent observations. We define a mutation/observation of a version v to be an arc from v to a mutator/observer node. A persistent mutation/observation is any mutation/observation made on a version that has already been mutated (recall that the versions are ordered according to evaluation order). Therefore a persistent mutation is any mutation but the first. A mutation/observation that is not persistent is *ephemeral*.

It is useful to split a profile into *phases* covering different stages of an ADT's life. We define the *age* of a node to be the number of operations that have contributed to its creation. We define a *phase* to be a continuous age range. As the weights of generators are fixed at the start of an ADT's life, this property is separated out of a phased profile. Note that any given dug has many phased profiles, one for each choice of phases.

The following properties therefore make up a phased profile:

Generation Weight Ratio the ratio of the weights of each generator;
Phased Properties for a given phase P, the following four properties are recorded over the version nodes N of ages included by P:
- **Mutation/Observation Weight Ratio** the ratio of the weights of mutators and observers, considering only operations on nodes in N;
- **Mortality** the fraction of nodes in N that are not mutated;
- **Persistent Mutation Factor (PMF)** the fraction of mutations of nodes in N that are persistent;
- **Persistent Observation Factor (POF)** the fraction of observations of nodes in N that are persistent;

For example, consider the dugs shown in Fig.3. As there is only one generator in the queue ADT, the generation weight ratio is redundant.

Consider the profile of `app1` over only one phase covering all ages. The mutation/observation weight ratio is: $snoc : tail : head = 3 : 2 : 1$. The mortality is 1/6. The PMF and POF are both zero, indicating that the application is single-threaded.

Consider the profile of `app2` with respect to three phases: ages $\{1 \ldots n\}$, $\{n+1 \ldots 2n-1\}$ and $\{2n\}$. The age of node n is $n+1$ so nodes $0 \ldots n-1$ are in the first phase, nodes $n \ldots 2n-2$ are in the second and node $2n-1$ is in the third. The mutation/observation weight ratio $snoc : tail : head$ is $1 : 0 : 0$ in the first phase, $0 : 1 : 0$ in the second and $0 : 0 : 1$ in the third. Note that we count the application of an operation from the point of view of the node being

operated on, not the node that results. Using the latter point of view would make phased profiles and persistence measures untenable at worst and messy at best. The mortality is $1/2n$ and the PMF and POF are both zero, again illustrating the single-threaded nature of the application.

Finally consider the profile of `app3` with respect to a single phase covering all ages. The mutation/observation weight ratio is: $snoc : tail : head = 1 : 1 : 1$. The mortality is $1/2$. The PMF is $(n-1)/2n$ as all n applications of *snoc* are ephemeral, one application of *tail* is ephemeral and the remaining $n-1$ applications of *tail* are persistent. The POF is zero as no observations of a node v occur after a mutation of v.

6 Guarding Data Structure

Recall that the purpose of the kit is to generate an application that uses an implementation of an ADT in a specified manner. We shall do this by first generating a dug with a given profile and then by running an evaluator over the operations of the dug. This evaluator will therefore use the ADT in a manner specified by the dug profile.

The generated dugs only define operations on ADTs carrying elements of type *Int*. This simplifies the generation procedure (every argument to an operation is now either of type *Int* or *T Int*) and still allows the ordering or equality of elements to affect the efficiency of an implementation of an ADT. We discuss in Sect. 8 how this simplification may be removed.

Auburn generates a dug by making random choices that probabilistically result in a dug of the given profile. It builds a dug one node at a time. A major problem to overcome is the possibility of undefined results of partial operations. For example, the application *head empty* will not be defined in most specifications of a queue ADT and will probably result in an error if evaluated. We therefore need to guard against such applications. One way to do this is to provide *guarding operations*, or *guards*, that given an application will return true or false according to whether the application is well-defined or not.

For example, in the case of the *head* operation of queues, we may define an operation *head_Guard* by:

$$\begin{aligned} &\mathit{head\_Guard} \quad :: \mathit{Queue}\ a \to \mathit{Bool} \\ &\mathit{head\_Guard}\ q = \mathit{not}\ (\mathit{isEmpty}\ q) \end{aligned}$$

which defines the application *head* q to be well-defined if and only if q is not empty. However, this requires a dug generator to have access to an implementation of the ADT. It is simpler and more efficient to maintain a *shadow* of every ADT version. The shadow of a version v will contain all of the information required to determine if an application of an operation involving v is well-defined. We will shadow queues simply by their length. The definition of *head_Guard* becomes:

$$\begin{aligned} &\mathit{head\_Guard} \quad :: \mathit{Shadow} \to \mathit{Bool} \\ &\mathit{head\_Guard}\ s = (s > 0) \end{aligned}$$

```
type Shadow = Int

empty_Shadow :: Shadow
empty_Shadow = 0
snoc_Shadow :: Shadow -> Int -> Shadow
snoc_Shadow s _ = s+1
tail_Shadow :: Shadow -> Shadow
tail_Shadow s = s-1
```

Fig. 4. Haskell code defining shadow operations for queues.

where the type *Shadow* is defined to be *Int*.

The shadow of a data structure is maintained by providing a shadow operation for every generator and mutator (observers do not produce ADT versions). Given an operation of type t, the shadow operation is of type $shadow(t)$:

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

Shadow operations that shadow a queue by its length are given in Fig.4.

This approach has a drawback: every argument of an operation must be chosen *before* passing these arguments to the relevant guard. However, with an application such as *lookup* l i on lists which returns the i^{th} element of l, this means guessing which indices are available for *lookup* before testing the validity of the application. A better scheme passes the guard *only the version arguments* of an operation. The valid ranges of remaining arguments are returned as the result. One argument is then chosen from each range with the resulting application guaranteed to be valid. As we have ensured that every argument to an operation that is not a version is of type *Int*, a guard may return a range using the type *IntSubset*:

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

Given an operation of type t taking n arguments, v of which are versions, the type of its guard is given by $guard(v, n-v)$:

$$guard(v,i) = \begin{cases} Shadow \to guard(v-1,i) & \text{if } v > 0 \\ [IntSubset]_i & \text{if } v = 0 \wedge i > 0 \\ Bool & \text{if } v = 0 \wedge i = 0 \end{cases}$$

where $[a]_i$ is the type of lists of i elements of type a. [1]

[1] Haskell does not support functions over tuples of arbitrary size so we are forced to use lists, here annotated informally with their intended length.

```
empty_Guard :: Bool
empty_Guard = True
snoc_Guard :: Shadow -> [IntSubset]
snoc_Guard s = [All]
tail_Guard :: Shadow -> Bool
tail_Guard s = (s>0)
head_Guard :: Shadow -> Bool
head_Guard s = (s>0)
isEmpty_Guard :: Shadow -> Bool
isEmpty_Guard s = True
```

Fig. 5. Haskell code defining guards of queue operations.

For example, shadowing lists by their length, a suitable guard of *lookup* on lists could be given by:

$$\begin{aligned} &\mathit{lookup\_Guard} \quad :: \mathit{Shadow} \to [\mathit{IntSubset}]_1 \\ &\mathit{lookup\_Guard}\ s = [0 :..: (s-1)] \end{aligned}$$

which defines an application *lookup* l i to be well-defined if and only if i is greater than or equal to zero and less than the length of l. Guards of queue operations are given in Fig.5. Note that Haskell does not provide a type for lists of given length, and so general lists must be used.

A suitable definition of a shadow type, shadow operations and guard operations for an ADT $\mathcal{A}$ is called a *guarding data structure* of $\mathcal{A}$. Note that the guarding data structure is only used to aid the generation of dugs and is not involved in the evaluation of dugs or any other use of an implementation of the ADT.

7 An Example: Benchmarking Queues

There are five ingredients in any experiment using Auburn:

1. a signature of an ADT,
2. implementations of the ADT,
3. a guarding data structure,
4. a dug generator, and
5. dug evaluators, one for each ADT implementation.

The user must supply 1 and 2. Auburn provides a template for 3 which the user must complete, and provides 4 and 5 in full. A signature of an ADT is simply a Haskell module exporting the type constructor and operations of the ADT *with* type signatures but *without* code.

We illustrate use of the kit on the queue implementations given in Sect. 1: naïve, batched and banker's. The signature of queues that we shall use for our

experiment is given in Fig.1. The module is named `Queue` and is stored in the file `Queue.sig`. We start our experiment with the signature file and the three implementations:

```
$ ls
BankersQueue.hs  BatchedQueue.hs  NaiveQueue.hs  Queue.sig
```

7.1 Making the Dug Generator and the Dug Evaluators

Given a signature, Auburn produces type signatures of the required operations for a guarding data structure:

```
$ auburn -d Queue
Wrote 'Queue_GDS.hs'.
```

Auburn outputs these in a module called `Queue_GDS` based on the name of the signature module. The user must then add definitions of the required operations. Fig.4 and Fig.5 together define the guarding data structure of queues that we shall use.

Given a signature, Auburn creates a generator of dugs:

```
$ auburn -g Queue
Wrote 'Queue_Gen.hs'.
```

and one evaluator of dugs for each implementation to be tested:

```
$ auburn -e Queue NaiveQueue BatchedQueue BankersQueue
Wrote 'Queue_Eval_NaiveQueue.hs'.
Wrote 'Queue_Eval_BatchedQueue.hs'.
Wrote 'Queue_Eval_BankersQueue.hs'.
```

As we have three implementations of queues, we now have one generator of dugs and three evaluators of dugs. A dug is stored in a file as a sequence of operations. The dug generator produces such a file given a profile and a few other controlling parameters (such as a random seed). A dug evaluator reads and performs the operations listed, evaluating the dug using a particular implementation of the ADT. The evaluator returns a *checksum* made from the sum of the observations. This checksum may be used to check the correctness of a new implementation of an ADT by comparing it with the checksum produced by a trusted implementation.

Auburn can also produce a *null implementation* of the ADT:

```
$ auburn -n Queue
Wrote 'Queue_Null.hs'.
```

This performs almost no work and produces incorrect results, but it does provide a type constructor and operations of the required type. The purpose of the null implementation is to allow an evaluator to be built:

```
$ auburn -e Queue Queue_Null
Wrote 'Queue_Eval_Queue_Null.hs'.
```

which will spend almost all of its time doing the work required for reading a dug file such as input/output, summing the checksum, etc. Subtracting this time from the time spent by another evaluator yields an estimate of the actual time spent performing ADT operations.

7.2 Choosing the Profiles

For our experiment, we shall build dugs with two phases: a build phase and a fall phase. The phases shall be of equal length and provide a convenient framework for adjusting the following parameters:

1. the mean size of the queues involved in operations,
2. the speed of construction/de-construction, and
3. the degree of persistence used.

By varying each parameter across two values, a simple Haskell script produces eight profiles:

```
$ makeProfiles
```

in the files `dug-`n`.profile` for $1 \leq n \leq 8$, and a description of each profile in the file `dugs.profiles`. For example, setting the ratio of *snoc* : *tail* to 10 : 1 in a build phase lasting 122 operations, and to 1 : 10 in a fall phase lasting 122 operations, produces dugs with a mean queue size of 50 and a speed of construction/de-construction of 9/11. Together with zero persistence, this profile is stored in the file `dug-7.profile`.

Parameter 1 was chosen to indicate the expected problem naïve queues have in handling large queues. Parameter 2 is an attempt to capture the difference between the batched and banker's implementations. The batched implementation delays work that may be *repeated* later by persistent operations. This work may be forced by an application of *tail*. This is the root of the problem when the batched implementation performs poorly on application `app3` of Sect. 1. The banker's implementation avoids this by periodically packaging delayed work to be *shared* later. Therefore, building and taking apart a queue quickly with persistence may disadvantage the batched implementation. Parameter 3 is of interest primarily because little is known about the effect of persistence on the efficiency of data structures.

7.3 Generating the Dugs

As Auburn builds a dug, there is a part of the graph that is "live" and waiting to be attached to new nodes, and a part that is "dead" that will play no further role in the generation of the dug. The live section is called the *frontier* of the graph. The size of the frontier grows exponentially in the degree of persistence. To

counter this growth a maximum bound may be placed on the size of the frontier—this is an additional argument to the dug generator. For ADTs supporting non-unary mutators (operations that combine versions, for example catenation) it is also useful to be able to set the minimum size of the frontier. For our experiment, we set the frontier to have minimum size one and maximum size ten.

In all, a dug generator takes seven arguments:

1. the profile to aim for—the `makeProfiles` program will generate the profiles for our experiment;
2. the size of the pool to draw integer arguments from—this is not relevant in our experiment as queues do not look at the elements they carry, so we set it to ten to allow the checksum to vary reasonably;
3. the number of nodes in the dug—our experiment generates dugs of ten thousand nodes;
4. the minimum size of the frontier—set to one for our experiment;
5. the maximum size of the frontier—set to ten for our experiment;
6. a seed for pseudo-random number generation—for each profile, we shall generate five dugs from five different seeds; this allows variation of the dug within the given profile;
7. the mode to run in—to generate the dug file, `Compile` mode is used, and to analyse the actual profile of the resulting dug we use `Analyse` mode; other modes include `Draw` for producing output from which `dotty` [Gra] may draw the dug, and `Raw` for producing a raw text description of the dug.

Auburn is distributed with three simple Perl scripts that partially automate any benchmarking experiment. The first of these scripts generates dug files from the profiles already generated:

```
$ makeDugs
```

in files of the form `dug-`*n*`-`*seed*`.dc`. For example, the profile stored in `dug-7.profile` given above causes a dug to be built as follows:

```
Queue_Gen "Profile [1.0] [Phase 122 [1.81818, 0.181818, 2.0, 0.0]
           0.0 0.0 0.0, Phase 122 [0.181818, 1.81818, 2.0, 0.0]
           0.0 0.0 0.0]" 10 10000 1 10 1899102810 Compile
          >dug-7-1899102810.dc
```

and stored in the file `dug-7-1899102810.dc`.

The `Analyse` mode returns the actual profiles of the dugs generated. These were stored in files of the form `dug-`*n*`-`*seed*`.profile` and checked for being close to the profiles we aimed for. The mean size of a queue can be displayed if we supply additional operations in the module exporting the guarding data structure. These maintain information on the shadows of ADT versions passed to operations. As the shadow of a queue is its size, we simply sum the shadows and then calculate the mean.

7.4 Evaluating the Dugs

A dug evaluator takes two arguments:

1. the dug file, and
2. the number of times to evaluate the dug.

The latter argument is used when the evaluation time is too small and increasing the size of the dug file is impractical. Our experiment repeats each evaluation ten times. The call to the evaluator is also repeated three times to gain a better estimate of the evaluation time.

The second Perl script distributed with Auburn calls the dug evaluators:

```
$ evalDugs
```

and records the total times in files of the form `dug-`*n*`-`*implementation*`.time`. These times in seconds are given in Table 3. Here is how the dug stored in `dug-7-1899102810.dc` is evaluated using banker's queues:

```
Queue_Eval_BankersQueue dug-7-1899102810.dc 10
```

The third Perl script subtracts the null implementation times, and calculates the ratios of the resulting times:

```
$ processTimes
```

and stores the result in the file `dugs.times` using the descriptions of each profile given in `dugs.profiles`. These ratios are given in Table 4.

7.5 Analysis

The first thing to note is that the batched implementation is the clear winner in all categories considered by this experiment (see Table 4). The absolute times in Table 3 also show little variation according to the profile used. A large (de)construction speed has none of the intended ill effect on its performance. This is surprising. The following profile:

```
Profile [1.0] [Phase 50 [2.0, 0.0, 2.0, 0.0] 0.0 0.0 0.0,
               Phase 1 [0.0, 2.0, 0.0, 0.0] 0.0 0.9 0.9,
               Phase 1 [0.0, 0.0, 1.0, 0.0] 1.0 0.0 0.0]
```

simulates the key elements of the application `app3` of Sect. 1 that caused the batched implementation so much trouble. Evaluating dugs with this profile does indeed produce poor results for the batched implementation: it takes twice as long as the other two implementations. However, we were not able to find a less obscure profile that also produced poor performance for the batched implementation. This suggests that the batched implementation only performs poorly on a rather small isolated group of dugs.

The banker's implementation also performs evenly across all profiles considered, but lags behind the simple pair implementation by just over a factor of two.

Table 3. Total times in seconds of the evaluation of dugs of each profile using each implementation. These were obtained with the York release of the nhc compiler [Yor] using C interface extensions to Haskell to improve the accuracy of the results.

Profile Parameters			Implementation			
Mean Size	(De-)construction Speed	Persistence	Null	Naïve	Batched	Banker's
5	1/3	0	14.480	28.870	22.690	28.900
5	1/3	0.1	14.580	25.450	22.510	26.600
5	9/11	0	14.400	26.170	22.270	27.780
5	9/11	0.1	14.890	22.580	21.460	25.790
50	1/3	0	14.610	116.670	22.480	29.260
50	1/3	0.1	14.670	42.620	21.940	24.900
50	9/11	0	15.070	111.340	22.800	28.220
50	9/11	0.1	15.330	41.340	21.130	24.470

Table 4. Ratios of the total times given in Table 3 after subtracting the null implementation times.

Profile Parameters			Implementation		
Mean Size	(De-)construction Speed	Persistence	Naïve	Batched	Banker's
5	1/3	0	1.753	1.000	1.756
5	1/3	0.1	1.371	1.000	1.516
5	9/11	0	1.496	1.000	1.700
5	9/11	0.1	1.170	1.000	1.659
50	1/3	0	12.968	1.000	1.861
50	1/3	0.1	3.845	1.000	1.407
50	9/11	0	12.454	1.000	1.701
50	9/11	0.1	4.484	1.000	1.576

The banker's implementation performs much of the same work as the batched implementation but uses more bookkeeping to ensure a constant complexity in all circumstances. In all profiles considered here, this extra work is not necessary.

The naïve implementation performs very poorly on the large queues, and reasonably well on the small queues, as expected. One notable feature is the increase in efficiency in moving from ephemeral to persistent use. This is true to some extent of all the implementations across all profiles. This can be explained by considering that the same number of operations will share more work with persistence than without. The degree of change is still surprising however.

It seems from this experiment that the batched implementation of queues performs the best amongst those considered in most circumstances. The banker's implementation only wins when queues are used in a particular, rather uncommon manner. It is worth replacing the naïve implementation with the batched implementation, even for reasonably small queues.

8 Conclusion, Present Status and Future Work

Auburn provides a convenient framework on which to build benchmarking experiments. Little is known about the empirical performance of functional data structures, but the kit should help us address this issue. Compared to building benchmarks by hand, the kit is very quick and easy to use. Compared to choosing a fixed set of benchmarks whose use of an ADT may be unclear, the kit allows a wide range of uses of an ADT to be defined easily and clearly. The kit may also be used to test the correctness of a new implementation of an ADT by comparing its results against a trusted implementation (see the discussion of checksums in Sect. 7.1). Auburn should enable functional programmers to make a more informed choice of efficient data structures.

The current version of Auburn may be downloaded from the Auburn web page [Aub]. Note that this paper is based on Auburn version 1.0, which is also available for downloading at the Auburn web page. A tutorial based on the example of Sect. 7 is included in any version of Auburn, and has been adapted to suit that version.

Listed below are the main areas for improvement.

Dug Extraction. A major addition to the kit would be a *dug extractor*. Given an application that uses an ADT, the extractor runs the application and produces a dug of how it uses the ADT. This allows a user to match their application to the most suitable implementation of an ADT.

Current work on Auburn version 2.0 provides an extractor via the Green Card language extension [PJNR97]. The extractor can also handle unevaluated portions of a dug, something not discussed in this paper but present in many applications. However, the extractor cannot currently record any non-version arguments.

Generalisation. The kit only handles simple ADTs covering most but not all operations. For example, higher-order operations such as *fold* and *map* are not simple. Operations involving more than one data structure such as *toList* and *fromList* are also not simple. Such operations must be excluded from the ADT signature.

To include higher-order operations would involve choosing a random function as an argument whilst generating a dug. It is not clear how this is best done, though one possibility might be to let the user supply a collection of functions from which to choose one at random.

To include operations over lists as well as over the ADT, we would also have to choose a random list. This is more straightforward than choosing a random function, but would still require a reasonable amount of input from the user; for example, how long should the list be?

Generating dugs with arguments other than versions or integers is problematic for similar reasons: how do we choose one at random?

Extending the dug model to include dependencies on observations is very tedious for dug generation, and very difficult for dug extraction. The value of observation dependencies does not appear to warrant this effort.

Evaluation order is a very confusing and difficult issue to deal with properly. In lazy functional languages, one cannot tell in general from a simple trace of events if the underlying algorithm is non-single-threaded or single-threaded, as lazy evaluation may re-order a single-threaded computation into an apparently non-single-threaded route. However, if any persistent mutation is observed, the algorithm must be non-single-threaded.

Automation. Much of the Auburn user's work could be automated, especially the running of dug evaluators and the collection and processing of times. Automating scripts and makefiles are distributed with the kit, but more could be done. Although it is reasonably straightforward to give the code for a guarding data structure, help in the form of suggestions or guidelines would be useful. Default shadows and guards for total operations would be useful too.

Current work on Auburn version 2.0 provides default shadows and guards for two standard shadow data structures: one that does nothing but provides a convenient template, and one based on the size of the ADT versions.

Acknowledgements

Thanks go to Martyn Pearce for his many comments on a draft of this paper, and to Nick Merriam for his implementation of Shell's sort.

References

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

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[Gra] Graphviz: Tools for viewing and interacting with graph diagrams. `http://www.research.att.com/sw/tools/graphviz/`.

[HM81] Robert Hood and Robert Melville. Real-time queue operations in pure Lisp. *Information Processing Letters*, 13(2):50–54, November 1981.

[Oka95] Chris Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(4):583–592, October 1995.

[Oka96a] Chris Okasaki. *Purely Functional Data Structures.* PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

[Oka96b] Chris Okasaki. The role of lazy evaluation in amortized data structures. In *Proceedings of the International Conference on Functional Programming*, pages 62–72. ACM Press, May 1996.

[PJNR97] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. Green Card: A foreign-language interface for Haskell. In *Haskell Workshop*, Amsterdam, June 1997. Published by Oregon Graduate Institute of Science & Technology.

[She59] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.

[Yor] York Functional Programming Group. `http://www.cs.york.ac.uk/fp/`.

A Code

Figures 6, 7 and 8 give Haskell code for the naïve, batched and bankers queue implementations respectively. Figure 9 gives Haskell code implementing Shell's sort using queues.

```
newtype Queue a = Queue [a]

empty = Queue []

snoc (Queue xs) x = Queue (xs ++ [x])

tail (Queue (_:xs)) = Queue xs

head (Queue (x:_)) = x

isEmpty (Queue []) = True
isEmpty _ = False
```

Fig. 6. Haskell code implementing naïve queues. Naïve queues use ordinary lists.

```
data Queue a = Queue [a] [a]

empty = Queue [] []

snoc (Queue f r) x = queue f (x:r)

tail (Queue (_:f) r) = queue f r

head (Queue (x:_) _) = x

isEmpty (Queue [] _) = True
isEmpty _ = False

queue :: [a] -> [a] -> Queue a
queue [] r = Queue (reverse r) []
queue f r  = Queue f r
```

Fig. 7. Haskell code implementing batched queues. Batched queues are based on an implementation due to Hood and Melville [HM81]. A queue is represented by a pair of lists (f, r)—f giving the front portion of the queue and r the rear portion of the queue but *reversed*. A batched queue (f, r) may be converted into a naïve queue by applying: *append f (reverse r)*. When the front list becomes empty we make a new front list from the reversal of the rear list. The new rear list is empty.

```
data Queue a = Queue !Int [a] !Int [a]

empty = Queue 0 [] 0 []

snoc (Queue lenf f lenr r) x = queue lenf f (lenr+1) (x:r)

tail (Queue lenf (_:f) lenr r) = queue (lenf-1) f lenr r

head (Queue _ (x:_) _ _) = x

isEmpty (Queue _ [] _ _) = True
isEmpty _ = False

queue :: Int -> [a] -> Int -> [a] -> Queue a
queue lenf f lenr r
   | lenr <= lenf = Queue lenf f lenr r
   | otherwise    = Queue (lenf+lenr) (f++reverse r) 0 []
```

Fig. 8. Haskell code implementing banker's queues. Banker's queues are given by Okasaki in [Oka95] and shown to have $O(1)$ amortized bounds in [Oka96b] and [Oka96a]. The implementation is similar to the batched queues except that the rear list is never allowed to grow larger than (a constant times) the front list—we use a constant of one here. When this invariant is about to be violated, we append the reversal of the rear list onto the back of the front list.

```
-- `fromList' and `mapQ' would probably be more efficient if
-- imported from an implementation (but Auburn does not support
-- these operations).
fromList :: [a] -> Queue a
fromList xs = foldl snoc empty xs
mapQ :: (a -> b) -> Queue a -> Queue b
mapQ = mapQ' empty
mapQ' q' f q | isEmpty q = q'
             | otherwise = mapQ' (snoc q' (f (head q))) f (tail q)

-- Insertion sort.
isort :: (a -> a -> Bool) -> [a] -> [a]
isort before xs = foldr insert [] xs
   where insert x [] = [x]
         insert x zs@(y:ys) | x `before` y = x : zs
                            | otherwise    = y : insert x ys

-- Merges from a queue of m lists to a queue of n lists.
mergeToN :: Int -> Queue [a] -> Queue [a]
mergeToN n ys = mergeToN' ys (fromList (take n (repeat [])))
mergeToN' ins outs | isEmpty ins = outs
                   | otherwise   =
   case head ins of
      [] -> mergeToN' (tail ins) outs
      (y:ys) -> mergeToN' (snoc (tail ins) ys)
                          (snoc (tail outs) (y : head outs))

-- We would normally generate the increments from the length of the
-- list to be sorted (all increments should be less than this length).
-- However, we know that the list to be sorted has length 1500.
incs :: Int -> Int -> [Int]
incs 0 len = [1, 3, 7, 21, 48, 112, 336, 861]
incs 1 len = [1, 7, 48]

-- As we merge from one queue of lists to another, each list is
-- reversed, and so the ordering we sort with must also be reversed.
incsOrders :: Ord b => Int -> Int -> [(Int, b -> b -> Bool)]
incsOrders incsSel len = zip (incs incsSel len) (cycle [(<=), (>=)])

-- Shell's sort is an alternating sequence of merges and
-- insertion sorts piped together.
shellSort :: Ord a => Int -> [a] -> [a]
shellSort incsSel [] = []
shellSort incsSel xs =
   head (foldr1 (.) (map makePipe (incsOrders incsSel (length xs)))
                    (snoc empty xs))
   where makePipe (inc,order) = mapQ (isort order) . mergeToN inc
```

Fig. 9. An implementation of Shell's sort using queues.