The Performance of the Haskell CONTAINERS Package

Milan Straka

Department of Applied Mathematics
Charles University in Prague, Czech Republic
fox@ucw.cz

Abstract

In this paper, we perform a thorough performance analysis of the CONTAINERS package, the de facto standard Haskell containers library, comparing it to the most of existing alternatives on HackageDB. We then significantly improve its performance, making it comparable to the best implementations available. Additionally, we describe a new persistent data structure based on hashing, which offers the best performance out of available data structures containing `String`s and `ByteString`s.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures; E.1 [*Data Structures*]: Trees; Lists, stacks, and queues

General Terms Algorithms, Measurement, Performance

Keywords Benchmarking, Containers, Haskell

1. Introduction

In almost every computer language there are libraries providing various data structures, an important tool of a programmer. Programmers benefit from well written libraries, because these libraries

- free the programmer from repeated data structure implementation and allow them to focus on the high level development,
- prevent bugs in the data structure implementation,
- can provide high performance.

For some languages, standardized data structure libraries exist (STL for C++ [Stepanov and Lee 1994], Java Collections Framework, .NET System.Collections, etc.), which provide common and effective options in many cases.

Being the only data structure package coming with GHC and the Haskell Platform (the standard Haskell development environment), the CONTAINERS package has become a “standard” data structure library for Haskell programmers. It is used by almost every third package on the HackageDB (674 out of 2083, 21st May 2010), which is a public collection of packages released by Haskell community.

The CONTAINERS package contains the implementations of

- *sets* of elements (the elements must be comparable),
- *maps* of key and value pairs (the keys must be comparable),
- ordered *sequences* of any elements,
- *trees* and *graphs*.

All data structures in this package work persistently, ie. they can be shared [Driscoll et al. 1989].

Our decision to compare and improve the CONTAINERS package was motivated not only by the wide accessibility of the package, but also by our intention to replace the GHC internal data structures with the CONTAINERS package. Therefore we wanted to confirm that the performance offered by the package is the best possible, both for small and big volumes of data stored in the structure, and possibly to improve it.

The contributions of this paper are as follows:

- We present the first comprehensive performance measurements of the widely-used CONTAINERS package, including head-to-head comparisons against half a dozen other popular container libraries (Section 3).
- We describe optimisations to containers that improve the performance of `IntSet` by up to 8% and the performance of `Set` by 30-50% in common cases (Section 4).
- We describe a new container data structure that uses hashing to improve performance in the situation where key comparison is expensive, such as the case of strings. Hash tables are usually thought of as mutable structures, but our new data structure is fully persistent. Compared to other optimised containers, performance is improved up to three times for string elements (Section 5).

2. The CONTAINERS package

In this section we describe the data structures available in the CONTAINERS package. We tried to cover the basic and most frequent usage, for the eventual performance boost to be worthwhile. Focusing on basic usage is beneficial for the sake of comparison too, as the basic functionality is offered by nearly all implementations.

2.1 Sets and maps

A *set* is any data structure providing operations `empty`, `member`, `insert`, `delete` and `union` as listed in Figure 1. Real implementations certainly offer richer interface, but for our purposes we will be interested only in these methods.

```
data Set e
empty   :: Set e
member  :: Ord e => e -> Set e -> Bool
insert  :: Ord e => e -> Set e -> Set e
delete  :: Ord e => e -> Set e -> Set e
union   :: Ord e => Set e -> Set e -> Set e
```

Figure 1. A *set* implementation provided by the CONTAINERS package

A *map* from keys to values is a set of pairs (key, value), which are compared using the key only. To prevent duplication we discuss only sets from now on, but everything applies to maps too[1].

2.2 Intsets

A set of `Int`s, or a map whose key type is `Int`, is used so frequently, that the CONTAINERS package offers a specialized implementation. By an *intset* we therefore mean a specialized implementation of a set of `Ints`[2]. It should of course be faster than a regular set of `Int`s, otherwise there would be no point in using it.

2.3 Sequences

The CONTAINERS package also provides an implementation of a *sequence* of elements called a `Seq` with operations listed in Figure 2. A `Seq` is similar to a list, but elements can be added

```
data Seq a
data ViewL a = EmptyL | a :< (Seq a)
data ViewR a = EmptyR | (Seq a) :> a
empty  :: Seq a
(<|)   :: a -> Seq a -> Seq a
(|>)   :: Seq a -> a -> Seq a
viewl  :: Seq a -> ViewL a
viewr  :: Seq a -> ViewR a
index  :: Seq a -> Int -> a
update :: Int -> a -> Seq a -> Seq a
```

Figure 2. An implementation of a *sequence* of elements provided by the CONTAINERS package

(`<|` and `|>`) and removed (`viewl` and `viewr`) to the front and also to the back in constant time, allowing to use this structure as a *double-ended queue*. Elements can be also indexed and updated in logarithmic time and two sequences can be concatenated also in logarithmic time.

2.4 The rest of the CONTAINERS package

The CONTAINERS package also contains a data type of a multi-way tree. Aside from the definition of this type, it contains only trivial methods (folds), so there is no point in benchmarking those.

The last data structure offered by the package is a graph, which is built on top of the ARRAY package, and some simple graph algorithms. We perform no graph benchmarks, as the most similar FGL package is very different in design. We only describe some simple performance improvements.

3. The benchmarks

Our first step is to benchmark the CONTAINERS package against other popular Haskell libraries with similar functionality.

3.1 Benchmarking methodology

To benchmark a program written in a language performing lazy evaluation is a tricky business. Luckily there are powerful benchmarking frameworks available. We used the CRITERION package for benchmarking and the PROGRESSION package for running the benchmarks of different implementations and grouping the results together.

[1] In reality it works the other way around – a set is a special case of map that has no associated value for a key. We could use a `Map e ()`, where `()` is a unit type with only one value, as a `Set e`. But the unit values would still take space, which is why a `Set e` is provided.

[2] When the GHC compiles one source file, it spends 5-15 times more performing intmap operations comparing to map operations (depending on the code generator used), which we measured with the GHC-head on 26th March 2010.

All benchmarks were performed on a dedicated machine with Intel Xeon processor and 4GB RAM, using 32-bit GHC 6.12.2. All Cabal packages were compiled using default compiler switches (except for the CONTAINERS package, where we adopted the switches of the precompiled GHC version). We tried to benchmark all available implementations on the HackageDB. The list of packages used, together with their versions, can be found in Appendix A.

The benchmarking process works by calling a benchmarked method on given input data and forcing the evaluation of the result. The evaluation forcing can be done conveniently using a DEEPSEQ package. But as the representation of the data structures is usually hidden from its users, we could not provide `NFData` instances directly and had to resort to a fold which performs an evaluation of all elements in the structure.

Because the benchmarked method can take only microseconds to execute, the benchmarking framework repeats the execution of the method until it takes reasonable time (imagine 50ms) and then divides the elapsed time by the number of iterations.

This process is repeated 100 times to get the whole distribution of the time needed, and the mean and confidence interval are produced.

The results are displayed as graphs, one for each benchmark (Figures 4 to 17). One implementation is chosen as a baseline and the execution times are normalized with respect to the selected baseline. For each implementation and input, the mean time of 100 iterations is displayed, together with 95% confidence interval (which is usually not visible on the graphs as it is nearly identical to the mean). For every implementation a geometric mean of all times is computed and displayed in the legend. The implementations except for the baseline are ordered according to this mean.

Each benchmark consists of several inputs. The size of input data is always measured in binary logarithms (so the input of size 10 contains 1024 elements). This size is always the first part of description of the input, which is displayed on the x axis. The input elements are of type `Int` unless stated otherwise (`String`s and `ByteString`s will be used with the `HashSet` in Section 5). Where any order or elements in the input data could be used, we tried ascending and random order (`asc` and `rnd` in the description of the input) to fully test the data structure behaviour. The random data are uniformly distributed, generated using standard Haskell random generator with fixed seed, and duplicates are allowed.

All graphs together with the numerical data are available on the author's website `http://fox.ucw.cz/papers/containers/`. For comparison, there are also graphs obtained by using only a `seq` instead of an all-element fold to evaluate the data structure.

3.2 Benchmarking Sets

The `Set` interface is polymorphic in the elements, provided the element type is an instance of `Ord`. Since the only element operation available is a comparison, nearly all implementations use some kind of a balanced search tree. We will not describe the algorithms used, but will provide references for interested readers.

We benchmarked the following set implementations:

- `Set` and `Map` from the CONTAINERS package, which uses bounded balance trees [Adams 1993],
- `FiniteMap` from the GHC 6.12.2 sources, which also uses bounded balance trees [Adams 1993],
- `AVL` from AVLTREE package, which uses well-known AVL trees [Adelson-Velskii and Landis 1962],
- `AVL` from TREESTRUCTURES package, which we denote as `AVL2` in the benchmarks, also using AVL trees,
- `RBSet` implemented by the author which uses well-known red-black trees [Guibas and Sedgewick 1978].

We performed these benchmarks:

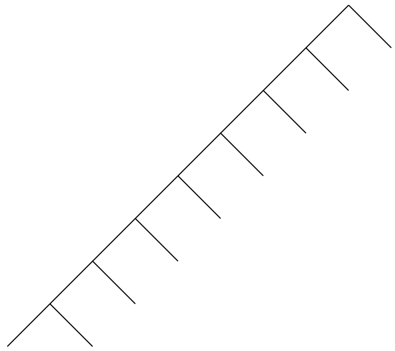

Figure 3. A tree called the *centipede*.

- *lookup benchmark*: perform a `member` operation on every element of the given set, either in ascending order (`asc` in the input description) or in random order of elements (`rnd` in the input description). For example the results for "`08/rnd`" are for a randomly-generated input of size 2^8.
- *insert benchmark*: build a set by sequentially calling `insert`, either in ascending (`asc` in the input description) or in random order of elements (`rnd` in the input description),
- *delete benchmark*: sequentially `delete` all elements of a given set, either in ascending (`asc` in the input description) or in random order of elements (`rnd` in the input description),
- *union benchmark*: perform a `union` of two sets of given sizes (the sizes are the first and second part of the input description). The input description `asc` means the elements in one set are all smaller than the elements in the other set. The description `e_o` stands for an input, where one set contains the even numbers and the other odd numbers. The last option `mix` represents an input, whose n elements are grouped in $\sqrt{n}$ continuous runs each of $\sqrt{n}$ elements, and there runs are split between the two sets.
- *tree union benchmark*: given a tree with elements in the leaves, perform union on all internal vertices to get one resulting set. The *tree union* benchmark models a particularly common case in which a set or map is generated by walking over a tree – for example, computing the free variables of a term. In these situations, most of the calls tu `union` are of very small sets, a very different test load to the *union benchmark*.

  The input description `asc` and `rnd` specify the order of the elements in the leaves. The shape of the tree is specified by the last letter of the input description. The letter `b` stands for perfectly balanced binary tree, `u` denotes unbalanced binary tree (one son is six times the size of the other son) and `p` stands for a centipede, see Figure 3.

The results of the benchmarks are plotted in Figures 4 and 5. The performance of the `Set` is comparable to the `FiniteMap`, but it is significantly worse than `AVL` and `RBSet`. This leaves a lot of space for improvements of the `Set` implementation to make it comparable to the `AVL` and `RBSet`. We describe such improvements in Section 4.

3.3 Benchmarking Intsets

The purpose of an intset implementations is to outperform a set of `Int`s. This can be achieved by allowing other operations on `Int`s in addition to a comparison. All mentioned implementations exploit the fact that an `Int` is a sequence of 32 or 64 bits.

We have benchmarked following intset implementations:

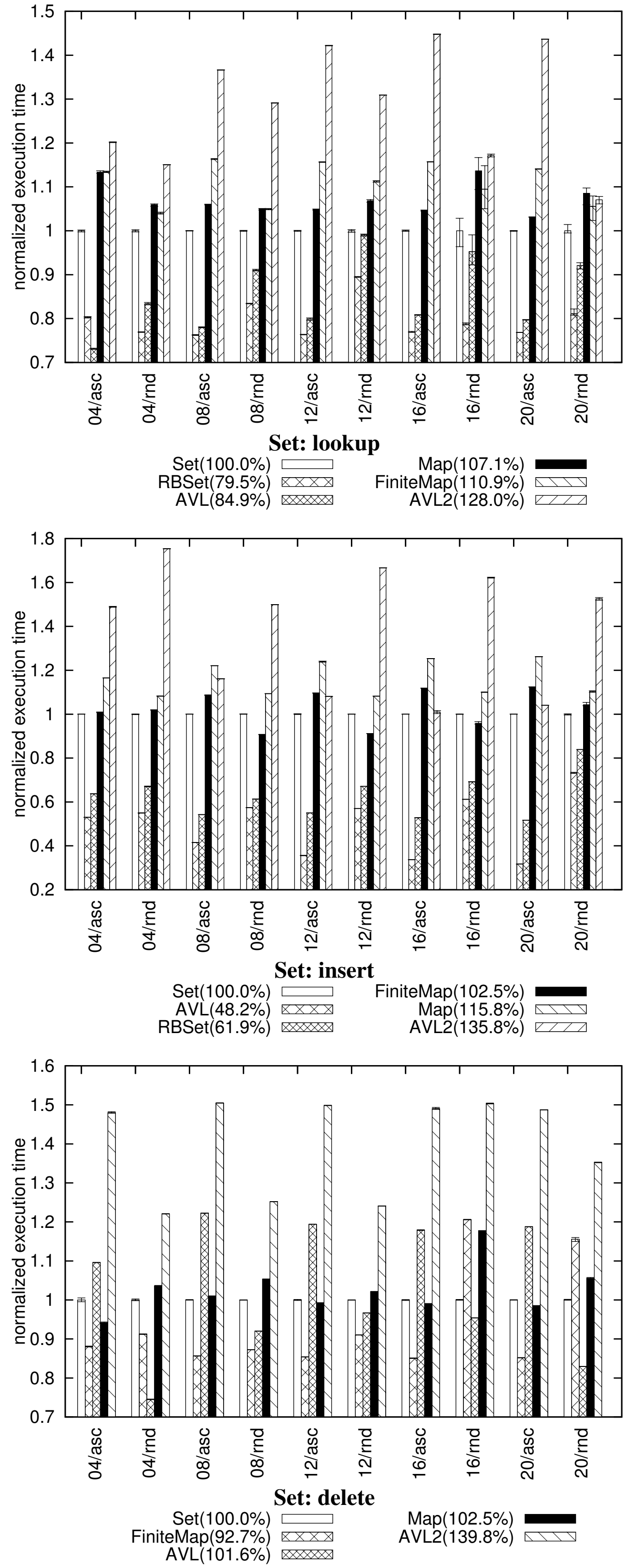

Figure 4. Benchmark of sets operations I

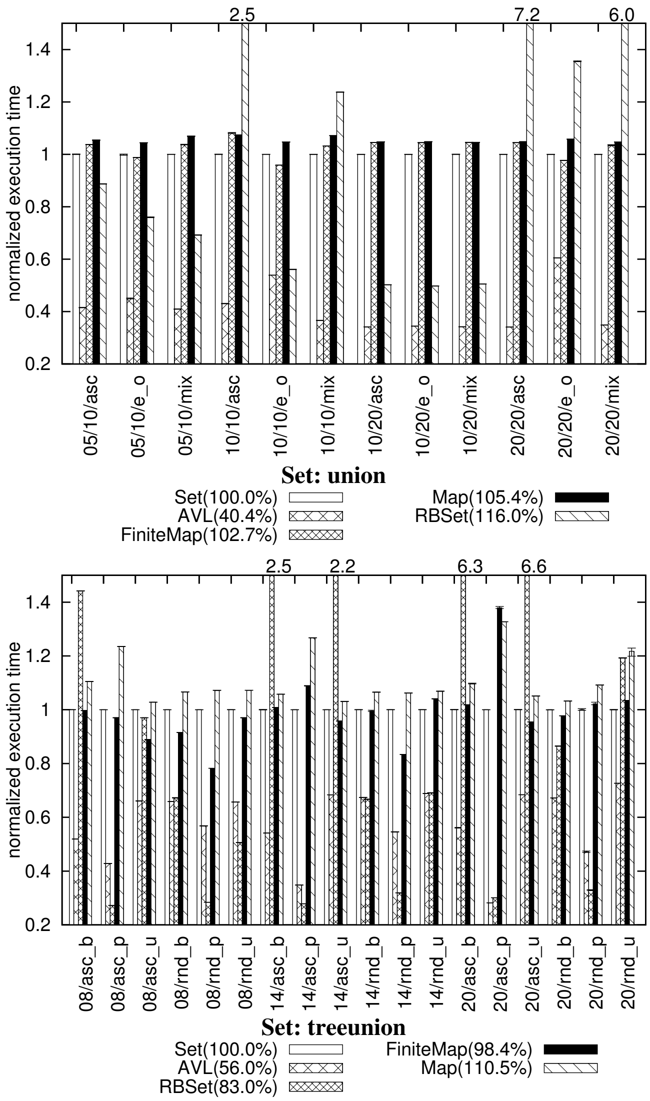

Figure 5. Benchmark of sets operations II

- `IntSet` from the CONTAINERS package which implements big-endian Patricia trees [Okasaki and Gill 1998],
- `UniqueFM` from GHC 6.12.2 sources which also implements big-endian Patricia trees,
- `PatriciaLoMap` from `EdisonCore` package, called `EdisonMap` in the benchmark, which implements little-endian Patricia trees [Okasaki and Gill 1998].

We also include ordinary `Set Int` from the CONTAINERS package in the benchmarks. For comparison, we also manually specialised the `Set` implementation by replacing overloaded comparisons with direct calls to `Int` comparisons, a process that could be mechanised. By comparing with this implementation, called `SetInlined` we can see the effect of the *algorithmic* improvements (rather than mere specialisation) in other intset implementations.

The benchmarks performed are the same as in the case of generic set implementations. The results can be found in Figures 6 and 7.

The `IntSet` outperforms all the presented implementations, except for the lookup and delete benchmark, where the `UniqueFM` is

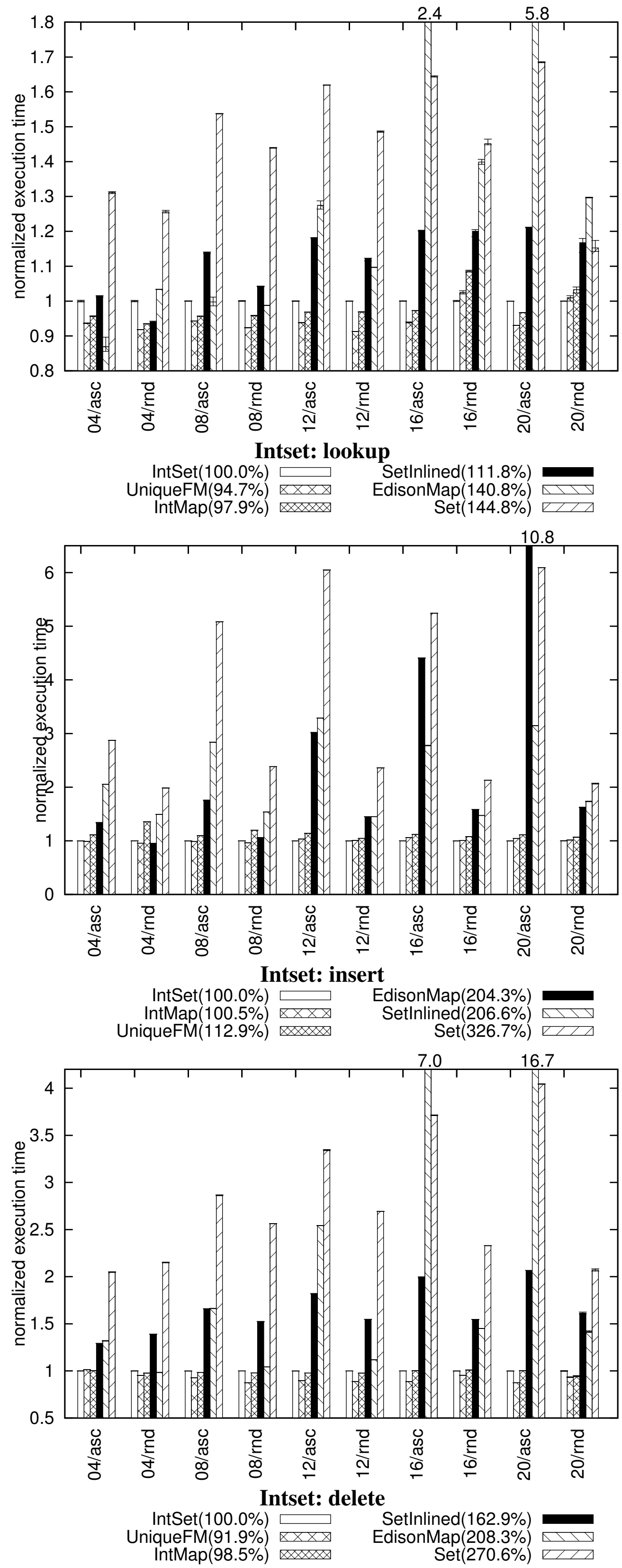

Figure 6. Benchmark of intsets operations I

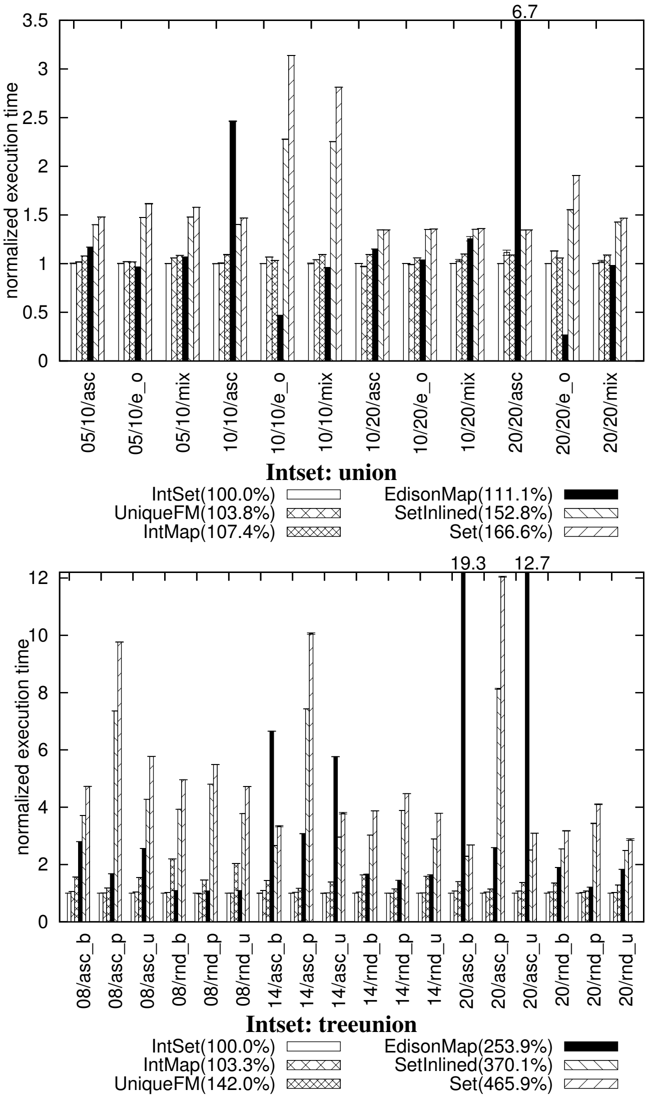

Figure 7. Benchmark of intsets operations II

faster. The `IntSet` is considerably faster than a `Set Int`, especially in the tree union benchmark, where it runs more than four times faster.

Although `IntSet` behaves very well, we describe some improvements in Section 4 that make it still faster.

3.4 Benchmarking Sequences

The `Seq` type in CONTAINERS supports beside others both (a) deque functionality (add and remove elements at beginning and end), and (b) persistent-array functionality (indexing and update). We compared it to several other libraries, most of which support only (a) or (b) but not both, and which might therefore be expected to outperform `Seq`.

3.4.1 Queue functionality

The queue functionality performance is significant, as there are no other implementations of queues and deques in standard Haskell libraries and so the `Seq` is the first choice when a queue is needed.

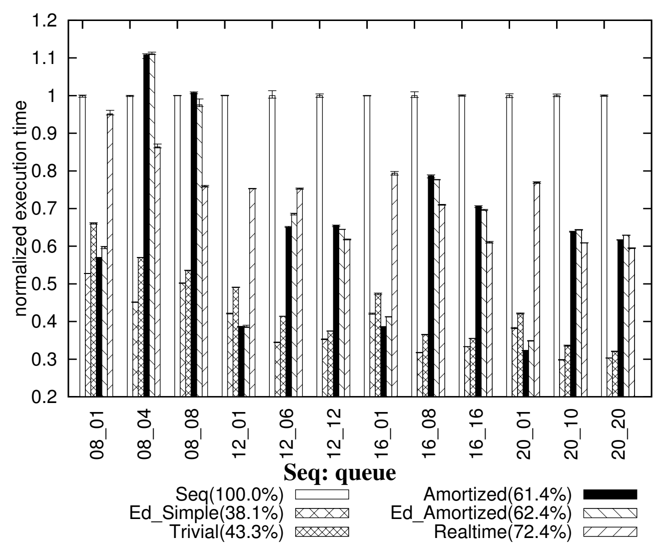

Figure 8. Benchmark of queue operations

The *queue* benchmark consists of two phases: first a certain number of elements is added to the queue (the number of the elements added is the first part of the input description) and then some of the previously added elements are removed from the queue (the second part of the input description). We also tried mixing the additions and deletions, but there were hardly any differences in performance, so we do not present these.

In this benchmark we tested the following implementations:

- `Seq` from the CONTAINERS package, which implements 2-3 finger trees annotated with sizes [Hinze and Paterson 2006],
- `Trivial`, which is a non-persistent queue with amortized bounds, described in Section 5.2 of [Okasaki 1999],
- `Amortized`, which is a persistent queue with amortized bounds, described in Section 6.3.2 of [Okasaki 1999],
- `Realtime`, which is a persistent queue with worst-case bounds, described in Section 7.2 of [Okasaki 1999],
- `Ed_Simple`, `Ed_Amortized` and `Ed_Seq` from the EDISON-CORE package, which implement the same algorithms as `Trivial`, `Amortized` and `Seq`, respectively.

The results are displayed in Figure 8. The `Ed_Seq` is missing, as it was roughly 20 times slower than the `Seq` implementation. Because the `Trivial` queue implementation is not persistent (cannot be shared), we do not consider it to be a practical alternative. That means the `Seq` implementation is only 50% slower than the fastest queue implementation available. That is a solid result, considering the additional functionality it provides.

3.4.2 Persistent-array functionality

The *index* and *update* benchmark perform a sequence of `index` and `update` operations, respectively, one for each element in the structure (the size of this structure is in the input description). We benchmarked the following implementations:

- `Seq` from the CONTAINERS package,
- `Array` from the ARRAY package for the index benchmark only,
- `RandList` from the RANDOM-ACCESS-LIST package, which implements the skew binary random-access list from Section 9.3 of [Okasaki 1999],

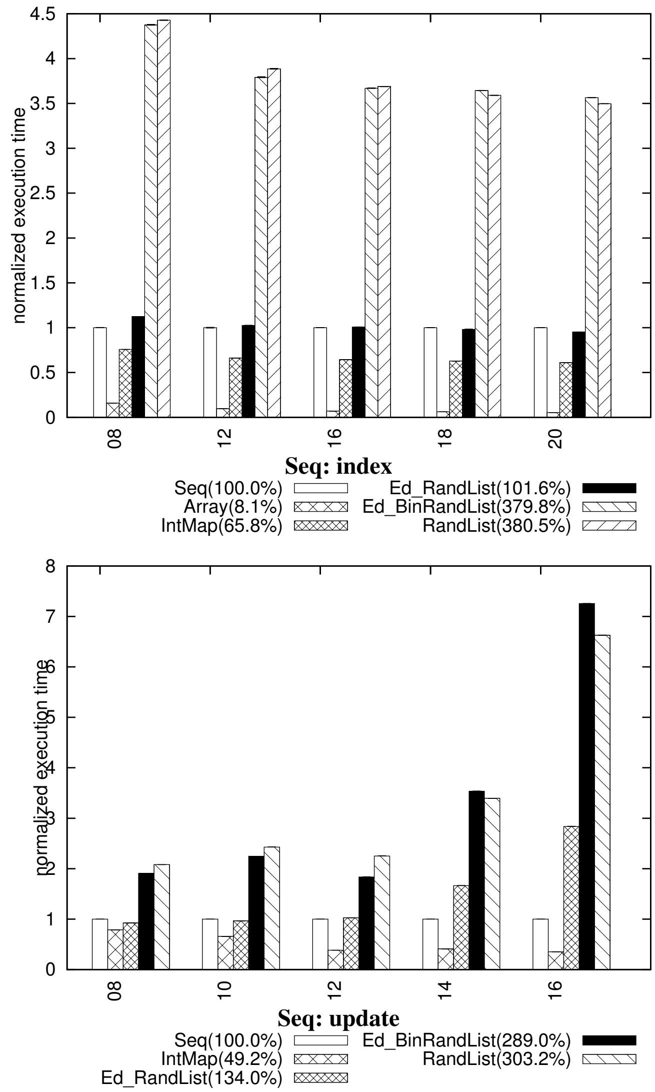

Figure 9. Benchmark of sequence operations

- `Ed_RandList` from the EDISONCORE package, which implements the same algorithm,
- `Ed_BinRandList` from the EDISONCORE package, which implements bootstrapped binary random-access list from Section 10.1.2 of [Okasaki 1999],
- `Ed_Seq` from the EDISONCORE package,
- `IntMap` from the CONTAINERS package.

The results are presented in Figure 9. Again we do not display `Ed_Seq`, because it was 10-20 times slower than `Seq`. The `IntMap` was used as a map from the `Int` indexes to the desired values. Despite the surplus indexes, it outperformed most of the other implementations. The `Array` is present only in the lookup benchmark, because the whole array has to be copied when modified and thus the `update` operation modifying only one element is very ineffective.

3.4.3 Summary

The `Seq` type is neither fastest queue nor the fastest persistent array, but it excels when both these qualities are required[3]. For comparison, when an `IntMap` is used in the queue benchmark, it is 2.5-times slower than `Seq`, and `Ed_RandList` and `Ed_BinRandList` are 5-times and 7-times slower, respectively.

4. Improving the CONTAINERS performance

There are several methods of improving an existing code.

The simplest is probably the "look and see" method – after carefully exploring the properties of the implementation (practically "staring at the source code for some time") some obvious deficiencies can be found.

As an example, consider the following definitions:

```
data Tree a = Node a (Forest a)
type Forest a = [Tree a]
```

In the `Data.Graph` module, function for pre-order and post-order `Tree` traversal are provided. The reader is welcome to consider what is wrong about *both* of these implementations:

```
preorder            :: Tree a -> [a]
preorder (Node a ts) = a : preorderF ts
preorderF           :: Forest a -> [a]
preorderF ts         = concat (map preorder ts)

postorder            :: Tree a -> [a]
postorder (Node a ts) = postorderF ts ++ [a]
postorderF           :: Forest a -> [a]
postorderF ts         = concat (map postorder ts)
```

The `postorder` case is straightforward – the list concatenation is linear in the length of the first list, so the time complexity of `postorder` performed on a path is quadratic.

The `preorder` is a bit more challenging – the `concat` takes the time of the length of all but the last list given. This also results in quadratic behaviour, for example when the `preorder` is executed on a centipede (Figure 3). The same mistake is also present in the `postorder` function.

It is trivial to reimplement both these functions to have linear time complexity.

However, potential performance improvements are usually not found merely by examining the source code. Another method is to use profiling to see which part of the code takes long to execute and which would be beneficial to improve.

Having two implementations, we can also examine why one is faster. In the simplest case it can be done at the level of Haskell sources. But if the reason for different performance is not apparent, we can inspect the differences at the level of Core Haskell [Tolmach 2001] using for example the `-ddump-stranal` GHC flag, which shows the results of strictness analysis. If this is not enough, we can examine the C-- code [Jones et al. 1999] using the `-ddump-cmm` GHC flag. We had to resort to analysis on all these levels when improving the performance of the CONTAINERS.

We now briefly describe the changes we made to improve the performance and present the benchmark results of the new implementations. The patches are available on the author's website `http://fox.ucw.cz/papers/containers/` and will soon be submitted for inclusion to the upstream. The correctness of these patches has been verified using tests from the CONTAINERS package.

[3] In addition, a `Seq` can also be split and concatenated in logarithmic time.

4.1 Improving Sets

Since the `Set` implementation already has good performance relative to its competitors, we did not change the algorithm, but instead focused in improving its implementation. We made the following improvements:

- As already mentioned, the methods of a `Set` works for any comparable type (i.e. an instance of `Ord`) and therefore use generic comparison method. That hurts performance in case the methods which spend a lot of time comparing the elements (like `member` or `insert`) are used non-polymophically. By supplying an `INLINE` pragma we allow these methods to be inlined to the call site and if the call is not polymorphic, to use the specialized comparison instead of the generic one. We inline only the code performing the tree navigation, the rebalancing code is not duplicated to keep the code growth at minimum.
- When balancing a node, the function `balance` checked the balancing condition and called one of the four rotating functions, which rebuilt the tree using smart constructors. This resulted in a repeated pattern matching, which was unnecessary. We rewrote the `balance` function to contain all the logic and to use as few pattern matches as possible. That resulted in significant performance improvements in all `Set` methods that modify a given set.
- When a recursive method accesses its parameter at different recursion levels, Haskell usually has to check that it is evaluated each time it is accessed. For a `member` or `insert`, that causes a measurable slowdown. We rewrote these methods so that they evaluate the parameter at most once. To illustrate, we changed the original `member` method

```
member :: Ord a => a -> Set a -> Bool
member x t = case t of Tip -> False
                       Bin _ y l r ->
                         case compare x y of
                           LT -> member x l
                           GT -> member x r
                           EQ -> True
```

  to the following:

```
member _ Tip = False
member x t = x `seq` member' t where
  member' Tip = False
  member' (Bin _ y l r) = case compare x y of
                            LT -> member' l
                            GT -> member' r
                            EQ -> True
```

- We improved the `union` to handle small cases – merging a set of size one is the same as inserting that one element. We achieved that by adding the following cases to the definition of a `union`:

```
union (Bin _ x Tip Tip) t = insert x t
union t (Bin _ x Tip Tip) = insertR[4] x t
```

  That helped significantly in the tree union benchmark. We tried to use this rule also on sets of size 2 and 3, but the performance did not improve further.
- In the `union` method, a comparison with a possibly infinite element must be performed. That was originally done by supplying a comparison function, which was constant for the infinite bound. Supplying a value `Maybe elem` with infinity represented as `Nothing` improved the performance notably. To demonstrate the changes, consider the `filterGt` method, which keeps in the set only the elements greater than the given bound (which could be $-\infty$):

```
filterGt :: (a -> Ordering) -> Set a -> Set a
filterGt _   Tip           = Tip
filterGt cmp (Bin _ x l r) = case cmp x of
  LT -> join x (filterGt cmp l) r
  GT -> filterGt cmp r
  EQ -> r
```

  We altered it to:

```
filterGt Nothing t = t
filterGt (Just b) t = b `seq` filter' t where
  filter' Tip = Tip
  filter' (Bin _ x l r) = case compare b x of
    LT -> join x (filter' l) r
    GT -> filter' r
    EQ -> r
```

The results are displayed in Figures 10 and 11. The improved implementations are called `NewSet` and `NewMap`. We were able to reach the `AVL` implementation performance, except for the union benchmark. Yet we outperformed it on the tree union benchmark, which was our objective.

Note that using the existing `AVL` implementation as a `Map` is not trivial, because it does not allow to implement all the functionality of a `Map` efficiently (notably `elemAt`, `deleteAt` etc.).

4.2 Improving IntSets

The `IntSet` implementation was already extensively tuned and difficult to improve. We performed only minor optimizations:

- As with the `Set`s, some recursive functions checked whether the parameters were evaluated multiple times. We made sure it is done at most once. Because some functions were already strict in the key, it was enough to add the `seq` calls to appropriate places. This improved the `lookup` function significantly.
- The implementation uses a function `maskW`. When `m` contains exactly one bit set, the `maskW i m` should return only the values of bits of `i` than are higher than the bit set in `m`:

`m`	`0...010...0`
`i`	`a...abc...c`
`maskW i m`	`a...a00...0`

  This method is defined as

```
maskW i m = i .&. (complement (m-1) `xor` m)
```

  But there are other effective alternatives, for example:

```
maskW i m = i .&. (-m - m)
maskW i m = i .&. (m * complement 1)
```

  The last one is (unexpectedly for us) the best and caused the speedup in the insert, union and tree union benchmarks.

The results are presented in Figures 12 and 13, the improved implementations are called `NewIntSet` and `NewIntMap`. The `NewIntSet` implementation is faster especially in the lookup and the insert benchmark. The speedup of the `NewIntMap` is a bit smaller.

5. New set and map implementation based on hashing

When a comparison of two elements is expensive, using a tree representation for a set can be slow, because at least $log_2(N)$ comparisons must be made for each operation. In this section we investigate whether we can do better on average, by developing a new implementation for set/map optimised for the expensive-comparison case.

[4] The `insertR` method works just like an `insert`, but it does not insert the element if it is already present in the set.

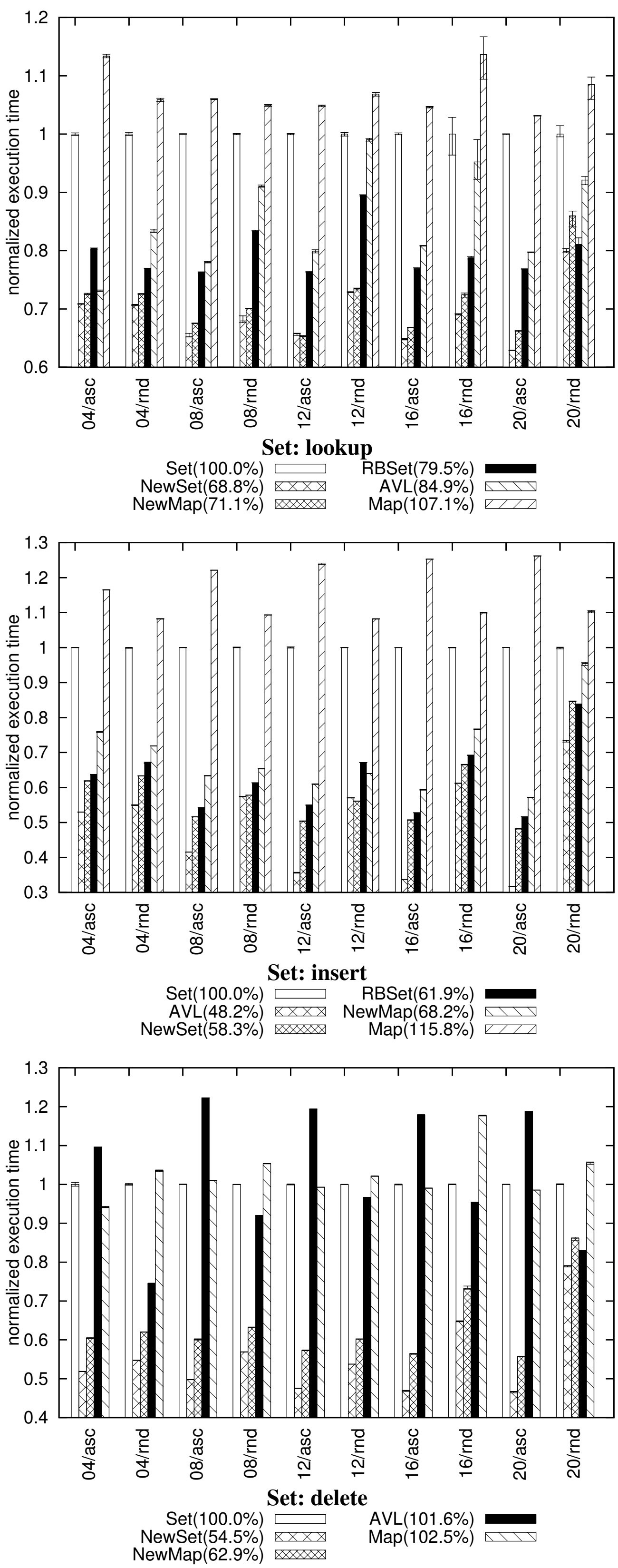

Figure 10. Benchmark of improved sets operations I

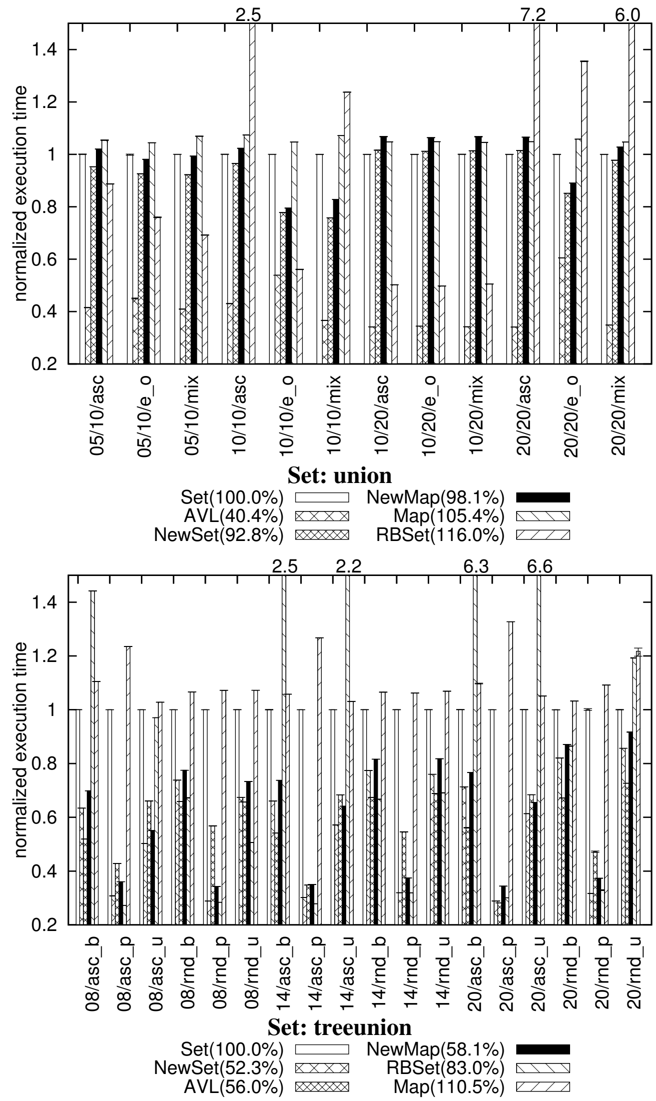

Figure 11. Benchmark of improved sets operations II

Two approaches suggest themselves. First, one could use a hash table (Section 6.4 of [Knuth 1998]) to guess the position of an element in the set and performs only one comparison if the guess was correct. Another alternative is a trie (Section 6.3 of [Knuth 1998]), which can also be implemented using a ternary search tree ([Bentley and Sedgewick 1998]), which compares only *subparts* of elements.

The problem with a hash table is that it is usually built using an array, but there is no available implementation of an array that could be shared, ie. be persistent. However, we have already seen that an `IntMap` can be used as a persistent array with reasonable performance. We used this fact and implemented a `HashSet elem` as

```
data HashSet elem = HS (IntMap (Set elem)).
```

The `HashSet` is therefore an `IntMap` indexed by the hash value of an element. In the `IntMap`, there is a `Set elem` containing elements with the same hash value (this set will be of size one if there are no hash collisions). A `HashMap` can be implemented in the same way as

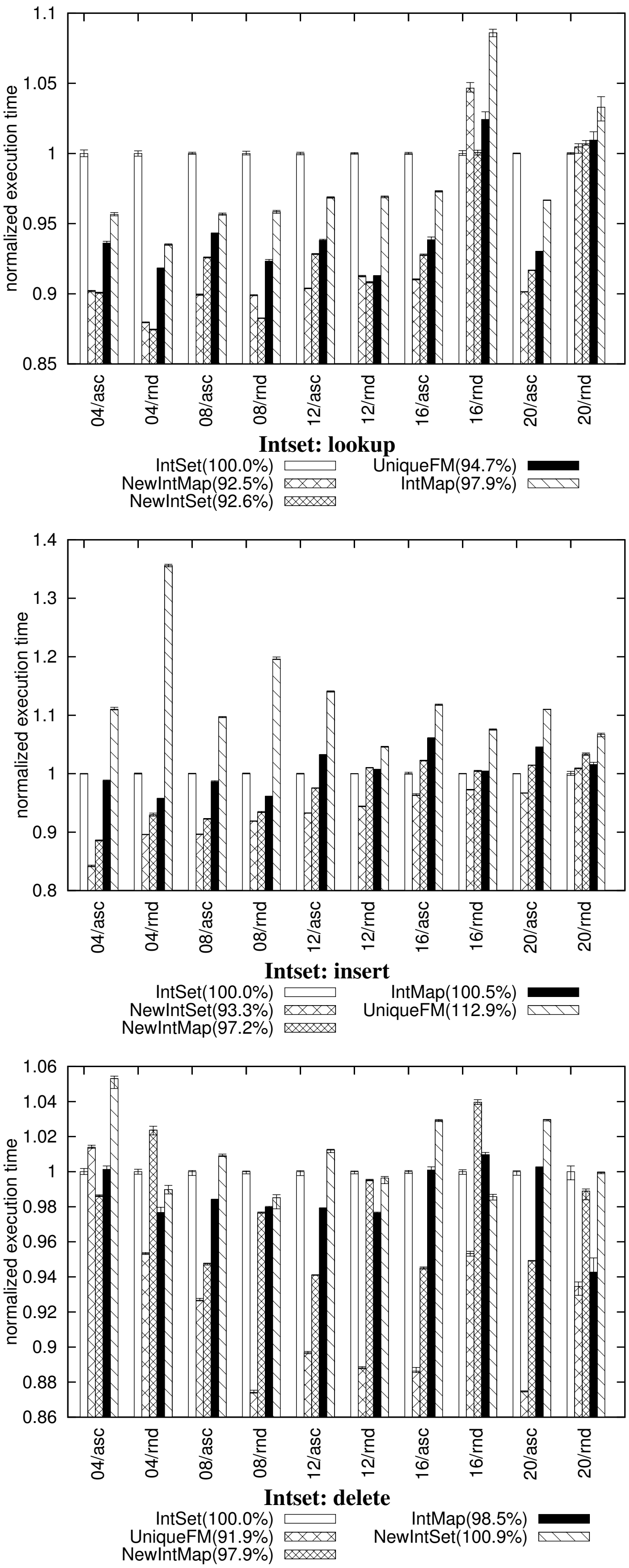

Figure 12. Benchmark of improved intsets operations I

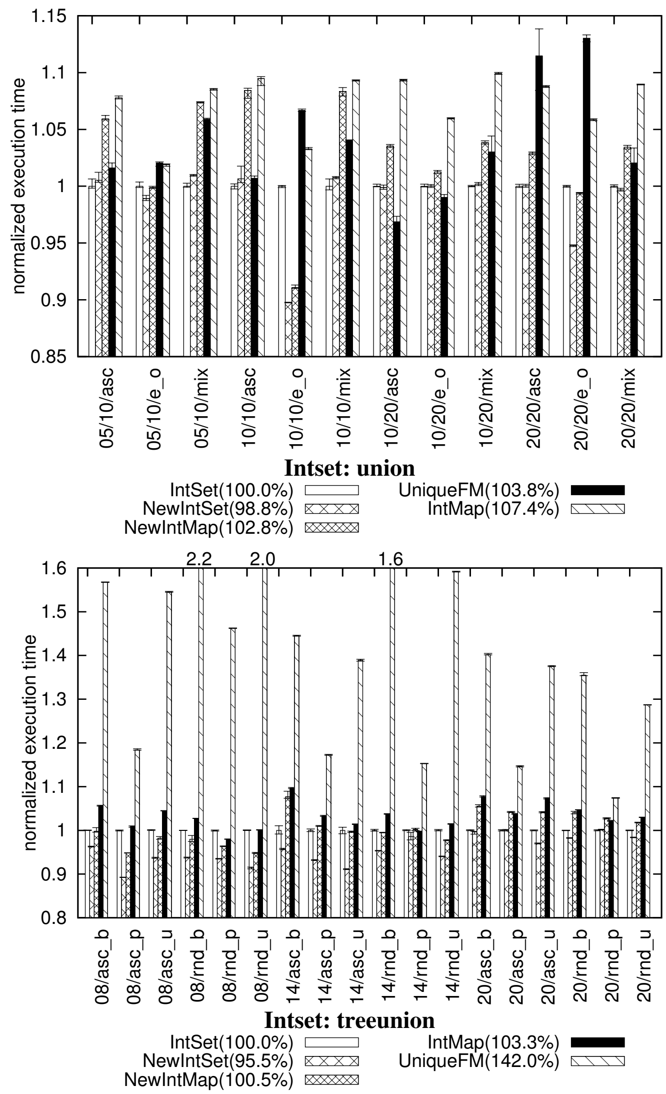

Figure 13. Benchmark of improved intsets operations II

data HashMap key val = HM (IntMap (Map key val)). Such a data structure is usually called a *hash trie* and described in [Goubault 1994] or in [Bagwell 2001].

This data structure is quite simple to implement, using the methods of an IntMap and a Set or a Map. It offers a subset of IntMap interface, which does not depend on the elements being stored in an IntMap in ascending order (the elements are stored in ascending order of the hash value only). Namely, we do not provide toAscList (users can use sort . toList), split, and the methods working with the minimum and maximum element (findMin, findMax and others). Moreover, the folds and maps are performed in unspecified element order.

We uploaded our implementation to the HackageDB as a package called HASHMAP.

We performed the same lookup, insert and delete benchmark on the HashSet as on the Set and IntSet. We used the original unimproved implementation of the CONTAINERS package – the performance of the HashSet will improve once the improvements from Section 4 are incorporated.

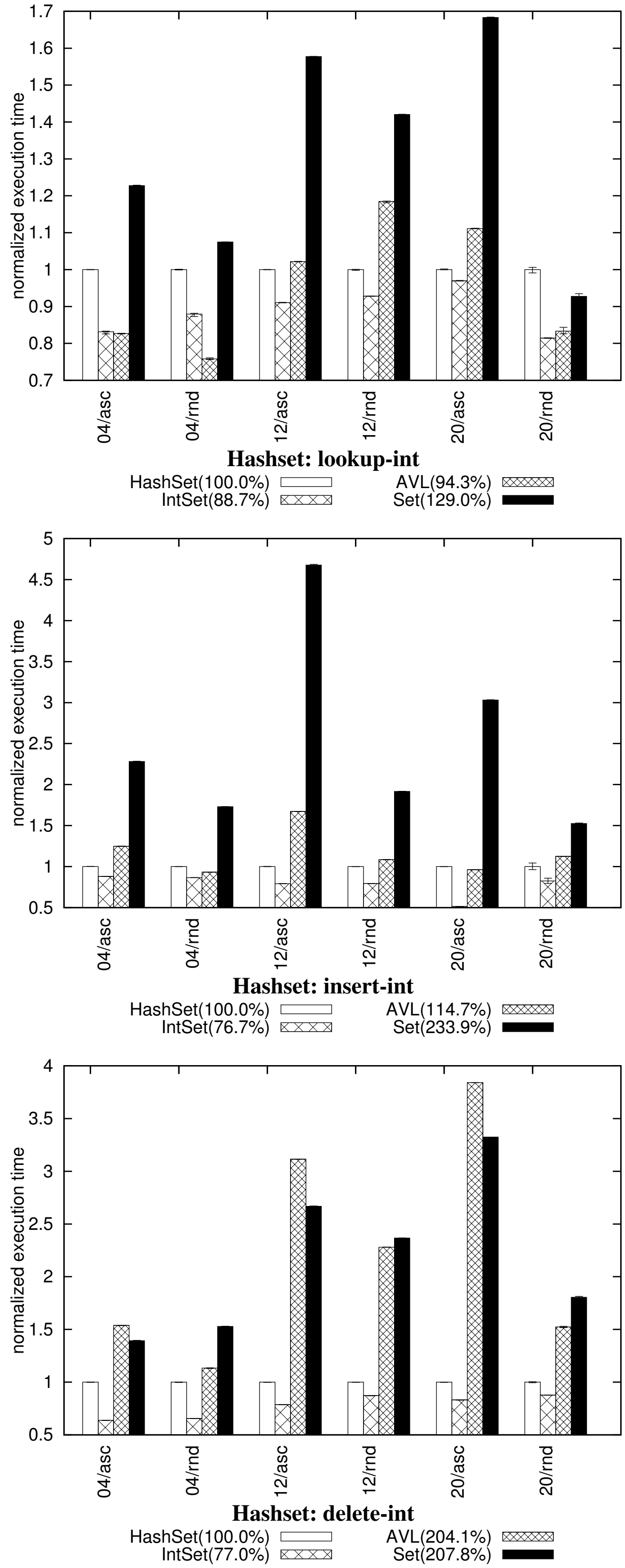

Figure 14. Benchmark of hashset operations on `Ints`

The performance of a `HashSet` when using elements of type `Int` is displayed in Figure 14. It is worse than the `IntSet`, because it uses an additional `Set` for elements with same hash values.

The `HashSet` should be beneficial when the comparison of the set elements is expensive. We therefore benchmarked it with `Strings` and `ByteStrings` elements. We compared the `HashSet` implementation to all alternatives present on the HackageDB (mostly trie-like data structures):

- `ListTrie` and `PatriciaTrie` from the LIST-TRIES package implementing a trie and a Patricia trie (Section 6.3 of [Knuth 1998]),
- `BStrTrie` from the BYTESTRING-TRIE package, which is specialized for `ByteStrings` and (like `IntSet`) implements a big-endian Patricia tree [Okasaki and Gill 1998],
- `StringSet` from the TERNARYTREES package, which implements a ternary search tree ([Bentley and Sedgewick 1998]) specialized for the elements of type `String`,
- `TernaryTrie` from `EdisonCore` also implementing a ternary search tree.

The results are presented in Figures 15 and 16. The length of the strings used in the benchmarks is the last number in the input description. We used uniformly distributed random strings of small letters (`rnd` in the input description) and also a consecutive ascending sequence of strings (`asc` in the input description). In the latter case the strings have a long common prefix of `a`'s. The `ListTrie` is not present in the benchmark results because it was 5-10 times slower than the `HashSet`.

The `HashSetNoC` is the same as the `HashSet`, only the computation of a hash value of a `ByteString` is done in Haskell and not in C. There is quite significant slowdown in the case Haskell generating the hashing code. We discussed this with the GHC developers and were informed that the problem should be solved using the new LLVM backend [Terei 2009].

We also performed the union benchmark. We generated a sequence of elements (its length is the first part of the input description) and created two sets of the same size, one from the elements on the positions and the other from the elements on odd positions. Then we performed a `union` of those sets. The results for `Int`, `String` and `ByteString` elements are presented in Figure 17.

The performance of a `HashSet` is superior to trie structures, even those specialised for the `String` or `ByteString` elements. As mentioned, the performance will improve even more with the enhancements of the CONTAINERS package.

6. Conclusions and further work

We have undertaken a thorough performance analysis of the CONTAINERS package, comparing it to the most of existing alternatives found on the HackageDB. These measurements are interesting of its own accord, because they allow existing data structure implementations to be compared.

Using the benchmark results and code profiling, we significantly improved the performance of the CONTAINERS package, making it comparable to the best implementations available. We will submit our patches for inclusion to the upstream shortly.

Inspired by the benchmark results we also implemented a new persistent data structure based on hashing, which offers the best performance out of available set implementations with `String` and `ByteString` elements, but should perform well for any element type whose comparison is expensive. This data structure is now available on the HackageDB.

Improving a library's performance is an unending process. Certainly the CONTAINERS package could be improved even further and more its methods could be benchmarked.

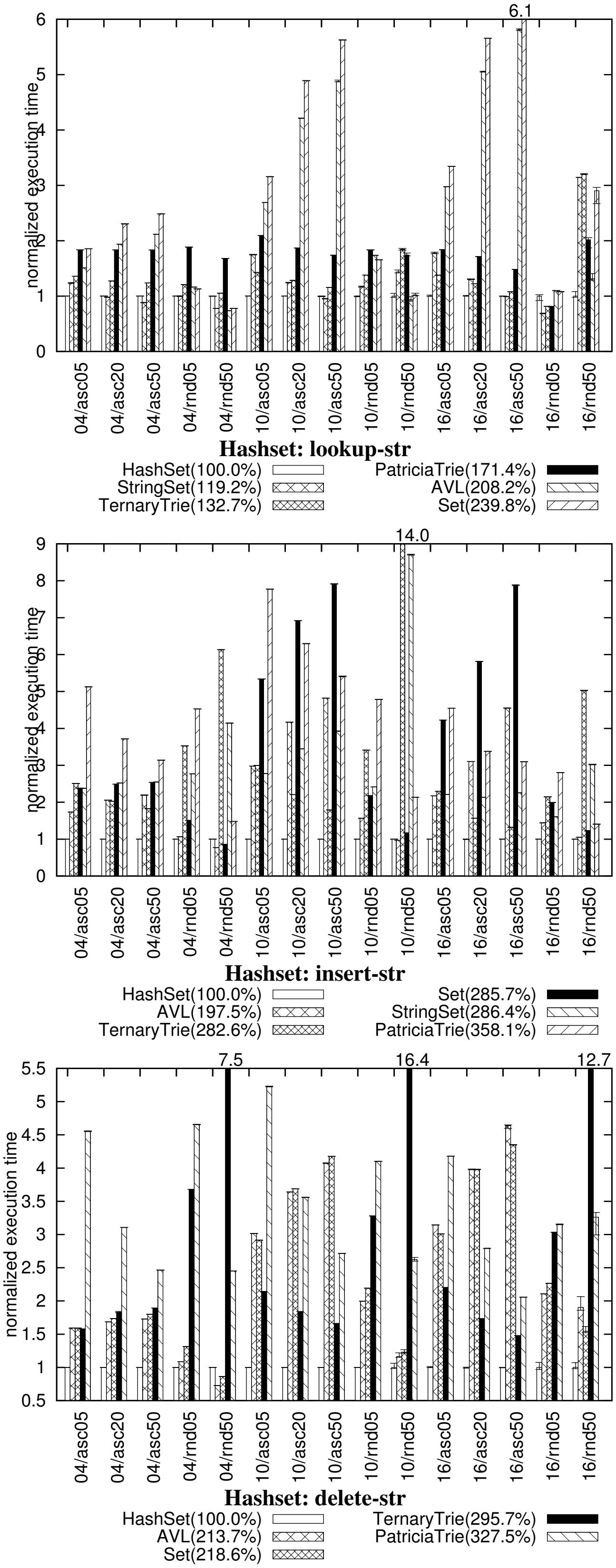

Figure 15. Benchmark of hashset operations on `Strings`

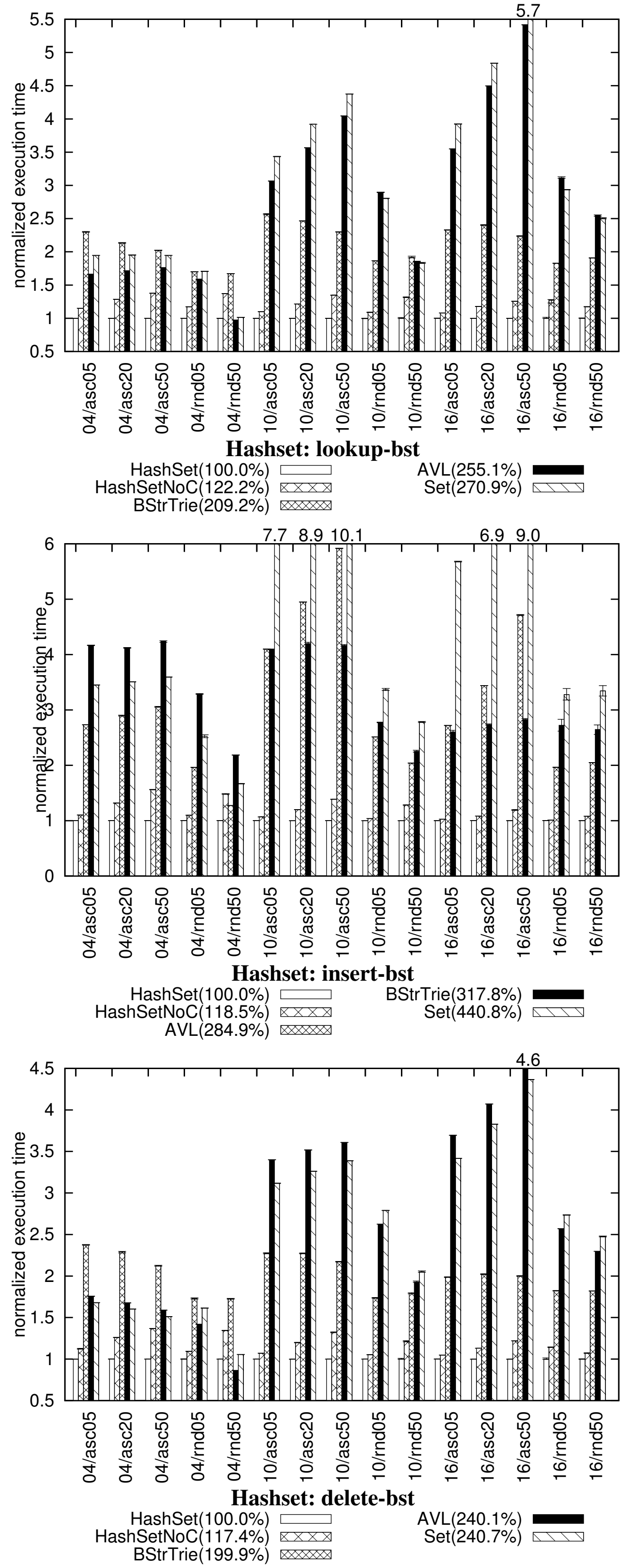

Figure 16. Benchmark of hashset operations on `ByteStrings`

Acknowledgments

I would like to express my sincere gratitude to Simon Peyton Jones for his supervision and guidance during my internship in Microsoft Research Labs, and also for the help with this paper. Our discussions were always very intriguing and motivating.

A. The list of referenced HackageDB packages

All packages mentioned in this paper can be found on the HackageDB, which is a public collection of packages released by the Haskell community. The list of HackageDB packages currently resides at `http://hackage.haskell.org/`.

We used the following packages in the benchmarks:

Packages used

ARRAY – 0.3.0.0	HASHMAP – 1.0.0.3
AVLTREE – 4.2	LIST-TRIES – 0.2
BYTESTRING-TRIE – 0.1.4	PROGRESSION – 0.3
CONTAINERS – 0.3.0.0	RANDOM-ACCESS-LIST – 0.2
CRITERION – 0.5.0.0	TERNARYTREES – 0.1.3.4
DEEPSEQ – 1.1.0.0	TREESTRUCTURES – 0.0.2
EDISONCORE – 1.2.1.3	

We also benchmarked internal data structures of the GHC compiler. Their implementation can be found in the sources of GHC 6.12.2, namely as files `FiniteMap.hs` and `UniqFM.hs` in the `compiler/utils` directory.

References

S. Adams. Efficient sets – a balancing act. *J. Funct. Program.*, 3(4): 553–561, 1993.

G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, (146), 1962.

P. Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

J. Bentley and R. Sedgewick. Ternary search trees. *Dr. Dobb's Journal*, April 1998.

J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1): 86–124, 1989. ISSN 0022-0000.

J. Goubault. Himml: Standard ml with fast sets and maps. In *In 5th ACM SIGPLAN Workshop on ML and its Applications*. ACM Press, 1994.

L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:8–21, 1978. ISSN 0272-5428.

R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006. ISSN 0956-7968.

S. L. P. Jones, N. Ramsey, and F. Reig. `C--`: A portable assembly language that supports garbage collection. In *PPDP*, pages 1–28, 1999.

D. E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, July 1999. ISBN 0521663504.

C. Okasaki and A. Gill. Fast mergeable integer maps. In *In Workshop on ML*, pages 77–86, 1998.

A. Stepanov and M. Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

D. A. Terei. Low level virtual machine for glasgow haskell compiler, 2009. URL `http://www.cse.unsw.edu.au/ pls/thesis/davidt-thesis.pdf`.

A. Tolmach. An external representation for the ghc core language, 2001. URL `http://www.haskell.org/ghc/docs/papers/core.ps.gz`.

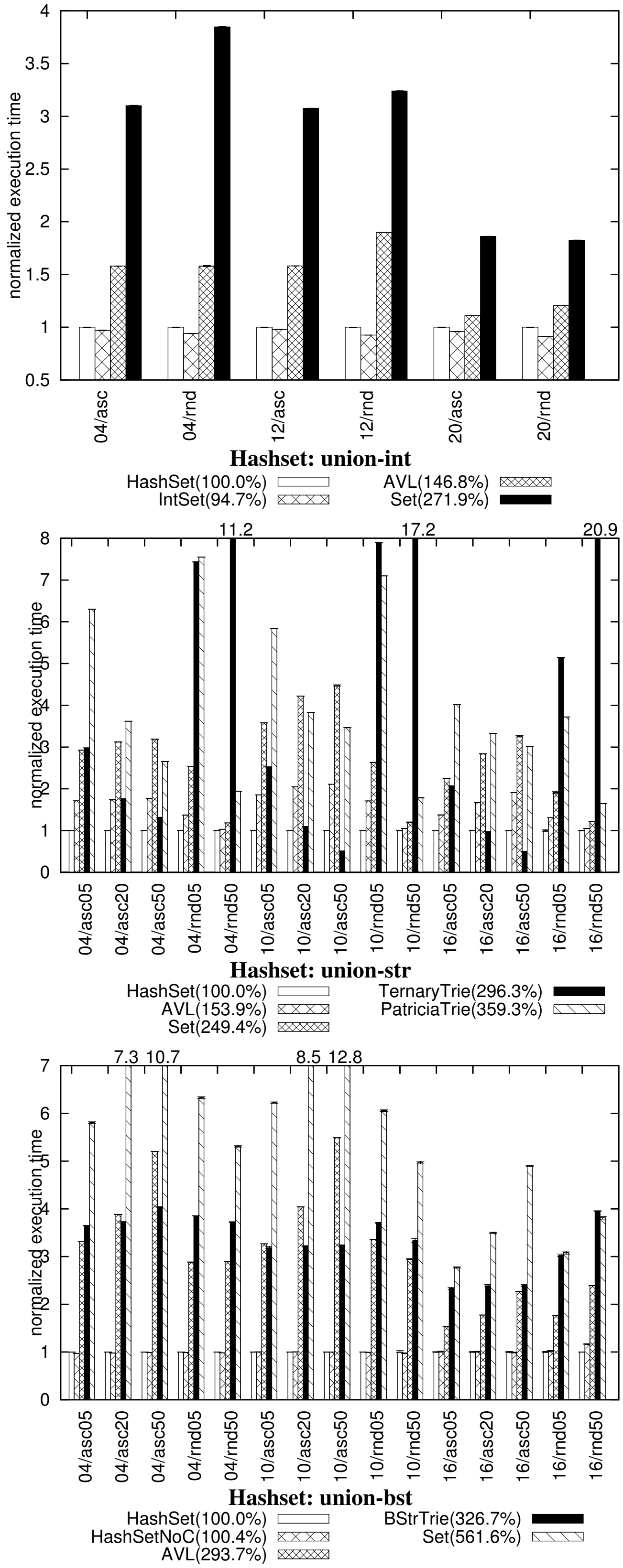

Figure 17. Benchmark of `union` operation on hashset