Advanced Datastructures: Project 3
Functional Queues

Jan H. Knudsen (20092926) Kris V. Ebbesen (20094539)
Roland L. Pedersen (20092817)

January 5, 2014

Contents

1 Intro

Queues are an important type of structures, and we often see them used as the basic tools for different algorithms. In imperative languages, it is easy to implement queues with $O(1)$ worst-case guarantees for `pop` and `inject`, with common implementations being a Linked List or a Circular Buffer. When using a functional language, such as Haskell, we do not have access to side-effects, and as such, implementing a queue with the nice $O(1)$ guarantees becomes much harder, and we need to make certain trade-offs, such as relying on amortization or global rebuilding. In this project we look at several different queue implementations in the lazy functional language of Haskell, and try to show their different properties through experiments.

2 The Haskell Data Structures

2.1 A Note About Terminology

A queue generally supports 2 operations, one to insert an element at the end of the queue, and one to extract the front element. The operation that insert an element onto the end of the queue often goes by the names `push / inject /enqueue / snoc`, while the removal operation goes by the names `pop / dequeue / (head, tail)`. In this paper, we will generally use `pop` and `inject`, though we might at certain points also use `push`.

2.2 The 4 Queues

As specified in the problem description, 4 different queues were implemented in Haskell. The have various properties, described as follows:

1. **Haskell List:** A queue can be implemented using a single built-in list from Haskell. Performing a `pop` is simple, just remove the head of the list into the head and tail part. `Inject` on the other hand is slow, since we need to insert at the tail of the list. We do this, by reversing the list, adding the new final element as a head, and revering again. Note that this could probably be sped up by performing only a single run through the list, but it is, under any circumstance, an $O(n)$ operation. Depending on how Haskells strictness analysis performs, we might actually see that the heavy work of the `inject` function gets thunked, and evaluated at the next `pop` operation.

Alternative Single-List Options There are alternative ways to implement a queue in Haskell using only one list. We chose ours as being the conceptually simplest.

Another way to keep store the queue, would be to keep it in a single list, in reverse order. This would give $O(1)$ `injects`, but $O(n)$ `pop`.

A quite interesting approach is to switch between reversed and normal ordering of the queue as needed, making operations $O(n)$ only when switching between `inject` and `pop`.

2. **Paired O(1) Non-Reusable Queue:** This queue implementation is based on maintaining a pair of queues, with one being the front of the queue, and one being the end of the queue, in reverse order. When we `pop`, we simply return the head of the front list, or, if it is empty, reverse the end list into the front list, and perform the `pop`. `Inject` is done by adding the injected element as the head of the end list. If we assume that any operation is only done on the currently newest version of the list, we can have the `inject` pay an amortized $O(1)$, to `pop` be an $O(1)$ cost operation amortized in a strict execution context.

   Note that even with lazy evaluation, we will most likely perform the entire reverse operation each time we need to reverse, since we need to remove the head.

3. **Real-Time Strict Queues:** For the $O(1)$ worst case queues we use the queues of (Hood and Melville, 1980). Instead of using the implementation example from the slides of (Brodal, 2013), we adapted the implementation presented in (Okasaki, 1999). This queue maintains a front and a reversed end list, much like amortized $O(1)$ strict queue, but uses global rebuilding to ensure that we can always pop in $O(1)$ worst case time. The implementation we are working with explicitly keeps track of the state it is in when doing global rebuilding. At any point in time, the global rebuilding of the queue may be in one of the following states:

   Reversing The queue is in the process of reversing the front and end list, in preparation of concatenating them. This phase uses 4 lists.

   Appending After having reversed the lists, the queue will append the front list to the end list one element at a time. This phase uses 2 lists.

   Done The queue is done reversing and appending, and the new front list is ready. This phase uses 1 list.

   Idle The queue needs not do any more work right now.

Since we use at most 4 lists in any phase, and 2 list in the queue itself, we use at most 6 lists in total.

After doing a `inject` or `pop`, we perform 2 steps of work on the current state, which ensures that we will always by ready to perform `pop`.

Note that we also keep 3 counters in the queue; the front list length, the end list length and the amount of valid elements in the state. This ensures that we can easily determine whether or not to initiate a global rebuild, and which elements we can discard from the global rebuild.

In terms of strictness, we use a `case` statement to ensure strict work step is strictly executed, but we fear that Haskell may still lazily thunk either some of the inner contents of the work operation, or during some of the functions leading to the work statement.

4. **Lazy Amortized $O(1)$ Queues:** The lazy queue keeps track of 2 lists, a front and a reversed end list, and their lengths. Whenever the reverse end list grows longer than the front list, we replace the front list by the concatenation of the front list and the end list reversed. Note that by requiring the end list to have grown significantly since each concat-reverse, we ensure that we have done enough `pop` or `inject` operations to have a $O(1)$ amortized cost.

Since these queues all follow the same interface, we also included wrapper functions that map generic `pop`, `inject` and `make` operations to the corresponding implementation-specific operation.

3 Haskell Performance Concerns

This is our first encounter with the development of performance-critical components i Haskell, and therefore we took our time looking into the different ways of making Haskell fast. Though we are sure that we have not done every trick in the book, and expect to have missed quite a few tricks that a Haskell veteran would have never let go, we were able to improve the performance of our initial data structures significantly through certain optimization patterns.

3.1 The GHC Compiler

The GHC compiler, short for *Glasgow Haskell Compiler* (ghc, 2013), was the compiler compiler used in this project. GHC was chosen, since it is the most widely used compiler supporting the entire feature set of Haskell, and has a wide array of optimization options.

The programs were compile used the following set of command line parameters `ghc -O2 -funbox-strict-fields -XBangPatterns -optc-O3`. Each parameter has the following function:

-O2 Perform heavy compile-time optimizations. This option should make the programs run a tiny bit faster than using [-O].

-funbox-strict-fields Automatically unbox strict fields. This is mostly a cosmetic parameter allowing us to omit `UNPAK` tags from our code.

-XBangPatterns Allow the use of the bang patterns extension, which is used for strict parameters.

-optc-O3 If using the C back-end, make GHC use the `-O3` option for GCC.

Used together, these flags allowed us to fine-tune the compilation of our Haskell code.

3.2 Haskell and Lazy Thunks

Haskell programs are lazily evaluated, using so-called *thunks*. This means that the execution of any expression gets replaced by a *thunk*, which acts as a placeholder for the value until it is actually needed. As soon as it becomes necessary to know the value of the expression, we evaluate the *thunk*, and store the associated value.

Note that a *thunked* expression is only evaluated once, a key feature of the Lazy Queue, and any expression that is never used will not be evaluated.

Lazy *thunks* do however have some performance concerns. Delayed execution may lead to a very heavy workload being assigned to a single operation, since long chains of unevaluated thunks might have developed. Also, the memory usage of storing these *thunks*, if they are not forced in time.

Since lazy evaluation is such a big part of Haskell performance, many of the optimization techniques used are based on forcing evaluation of thunks at certain critical points, and the GHC itself will try analyse when a value is imediately needed, and evaluate it strictly.

3.3 Using `case` to Force Strictness

In the Real-Time Strict Queues, we wish to enforce strictness on the evaluation of the global rebuilding. We do this using the `case` statement, which performs pattern matching on the result of a given expression. Since we need to know the structure of the result, we force the execution of the thunk representing the global rebuilding state, and maintain the O(1) worst-case guarantee for all operations.

3.4 Strict Data Types

Strictness can be enforced for type members, and unpacked into function parameters using the `-funbox-strict-fields` compiler flag. In certain parts of the program data structure, this can save us the overhead of thunking, and lead to a significant speedup.

Simply marking the general structure of the Lazy Queue as strict (note that inner values are still lazy, we do not force strict execution of the lists making up the queue), and unpacking it, lead to a speedup of $\sim 5\% - 30\%$, depending on the test suite.

Using strictness in the Real-Time Strict Queue structure was also used, but id did not in itself provide any speed-up.

3.5 Using Non-Generic Types

Haskell is statically typed, but uses type inference for most functions. If we however guarantee that we only use certain types, we allow the compiler to perform more in-depth optimization.

This is done by explicitly specifying the type of functions, and fully specifying what types will be used, which should provide a small performance gain.

3.6 Using Strict Parameters

Using the bangpatterns extension, we can mark certain parameters in pattern matching for strict execution. This can remove some of the overhead from thunking, and help the compiler perform strictness analysis.

When used in conjunction with strict typing, we achieved a $\sim 5\%$ performance gain on the Real-Time Strict Queues.

3.7 Numeric Types in Haskell

Haskell has a quite good arbitrary-precision integer type. However, for optimum performance, we switched to using machine-word integers, which gave us a $\sim 10\% - 30\%$ performance gain across the different tests.

4 Test Framework - Criterion

Criterion (cri, 2013) is a benchmarking framework for Haskell developed by (O'Sullivan, 2013) that gives accurate statistical information about the code segment that one is testing. This includes the mean and standard deviation of the running time for each sample. Timing values are useful, but having an indication of which times occurred most often is a lot more useful, because it can hint at outliers offsetting the mean running time. Criterion can output histograms in html showing this information.

When we test using Criterion the first thing it does is measure how long it takes for the system clock to tick and then run the code several times to ensure that the resolution of the clock does not introduce any significant errors. If the code takes 5μs to evaluate but the clock only ticks every 9μs, we cannot just run the code once and subtract start time from end time because then it would look like the code evaluated instantly sometimes and sometimes not. By running the code thousands of times the error is reduced to only a few instances out of the whole.

Criterion automatically calculates how many times it needs to evaluate the code and also reports an estimate of how long it will take. It uses a boxplot technique to get a quick sense of the quality of the data by calculating outliers, which are values that lies far from the mean of the sample. Once the system clock period has been found criterion figures out how expensive it is to use the clock and adjusts the time measurements to take the clock into account.

In the end criterion performs some statistical analysis on the results using a technique called bootstrapping to see if other processes on the computer might have influenced the results by looking at the number of outliers. If there are many outliers the results are essentially junk because the computer then would have varied compute capability during testing, which corrupts the results.

The graphs showing our test results was computed using the mean as execution time as it was close to the time that occurred most often in all test cases.

Due to all of these excellent features of Criterion, and its ease of use, we chose it as our the basis for all of our tests.

5 Experiments

5.1 Experiment Details

We ran the tests on a laptop using the Criterion benchmarking tool for haskell. The graphs were generated using Criterions ability to output data to csv. Each test was run for each queue type, n times. The values of n that was used went from 500 to 10000 with gaps of 500.

5.1.1 Test Types

Four types of tests was done:

1. `test_pop`
2. `test_push`
3. `test_pop_push`
4. `test_repeated_push_pop`

5.2 Expectations

When comparing the 4 different queue implementations, we expect them to perform as follow:

1. **Haskell List:** The Haskell List queue implementation performs `inject` in $O(n)$ time, and as such, we expect it to perform horribly on any test case that requires pushes. On pure `pop` operations however, it will be $O(1)$, with a very low overhead, and as such it will most likely be among the fastest queues at performing `pop`s.

2. **Paired O(1) Non-Reusable Queue:** The Paired Queue implementation should be fairly low-overhead, and as such perform quite well, except on the special test case, where we reuse queues.

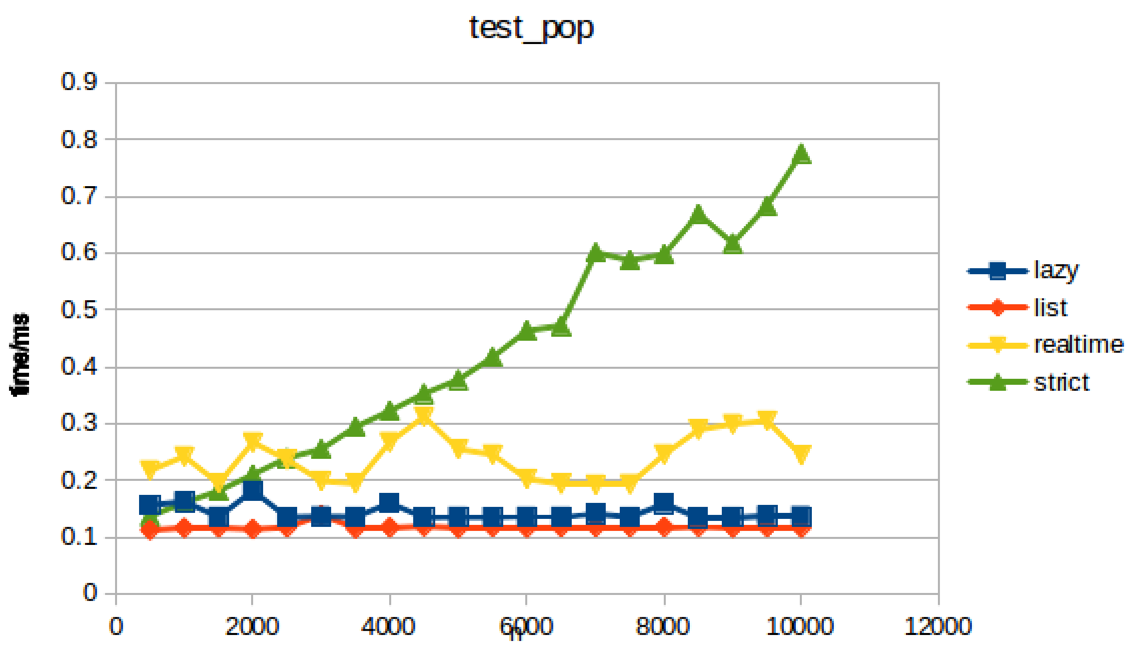
3. **Real-Time Strict Queues:** The Real-Time Strict Queue should generally perform well on all test cases, since we have a nice $O(1)$ worst-case guarentee for all operations. This queue however, has quite a large overhead, and will therefore most likely be outperformed by most other queues in tests cases where they exhibit $O(1)$ performance.

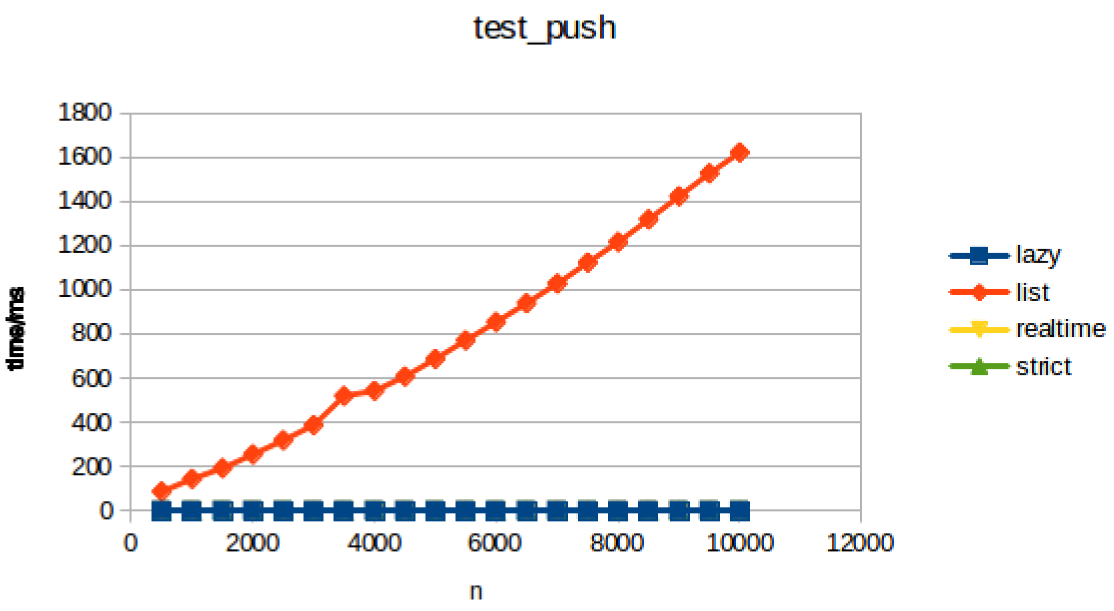
4. **Lazy Amortized $O(1)$ Queues:** The Lazy Queue should perform very well on all test cases. Not only does it have $O(1)$ amortized cost for all operation, it also has quite a low overhead.

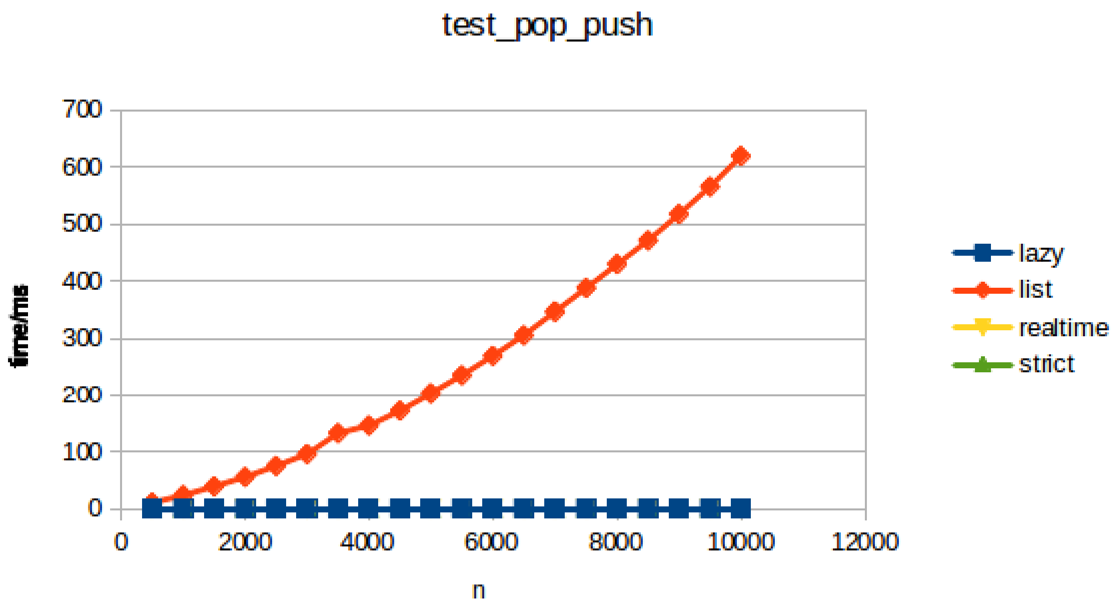
5.3 Computer Specs:

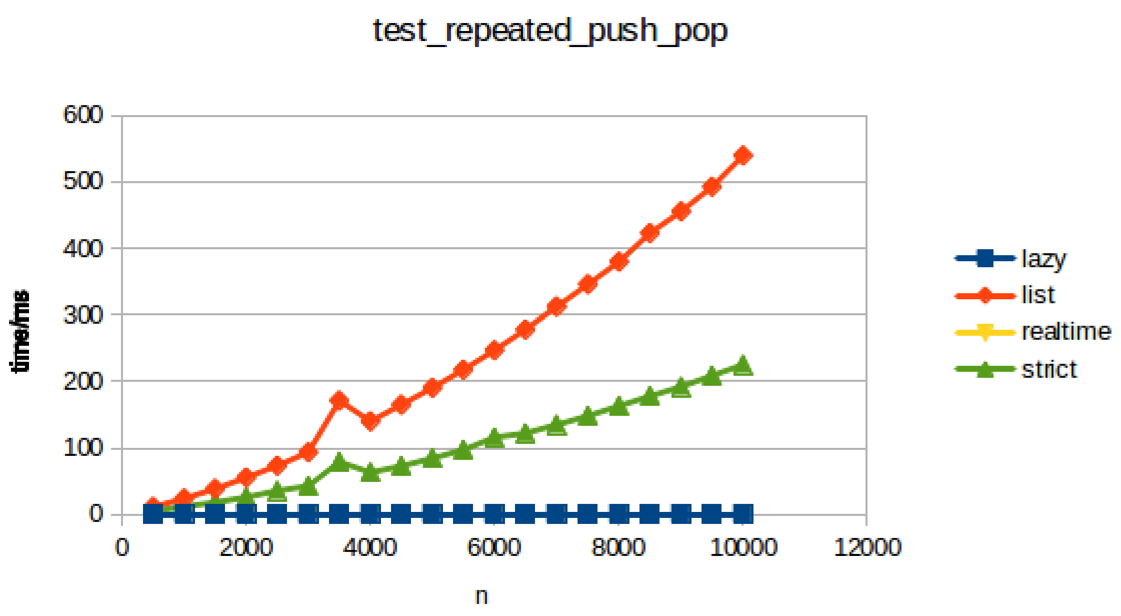
Model:	ASUS 1215N
OS:	Ubuntu 12.10
CPU:	Intel Atom Dual Core 1.8GHz
Memory:	2 GB

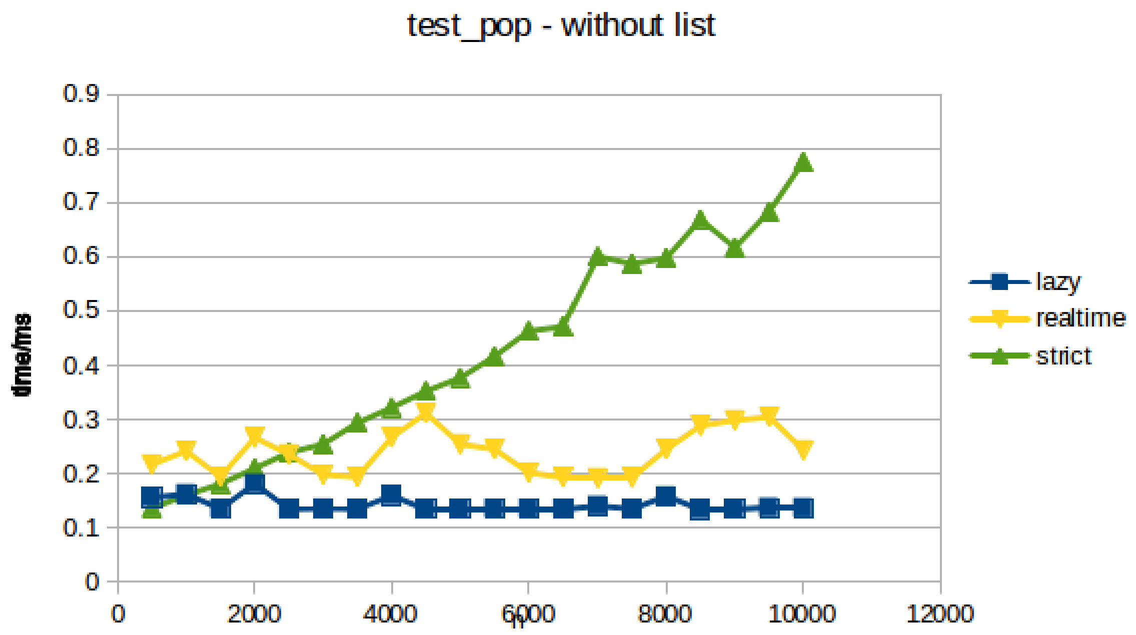
5.4 Result

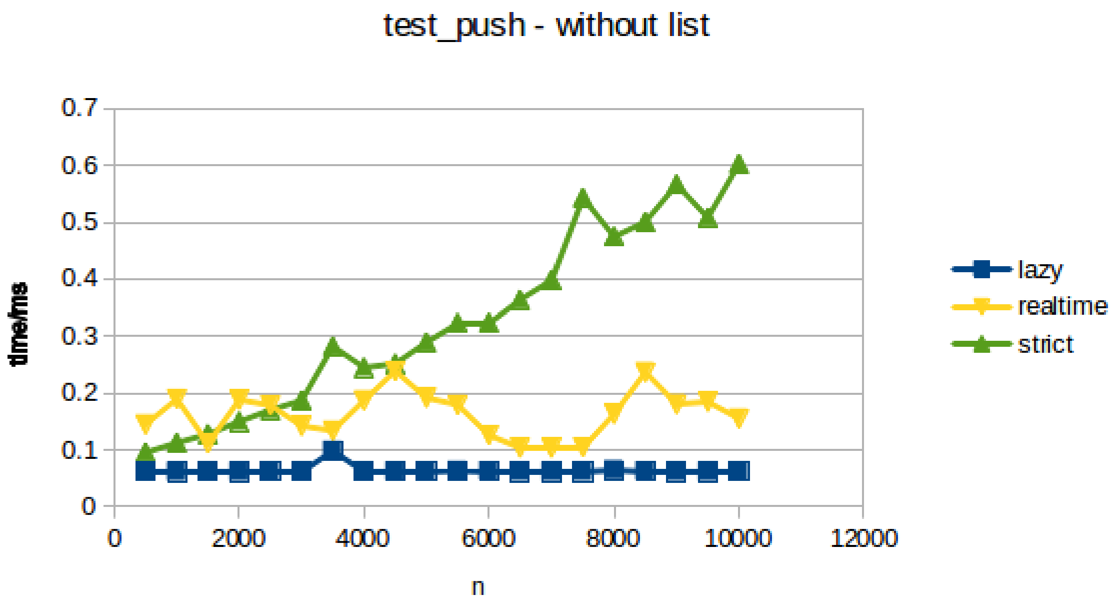


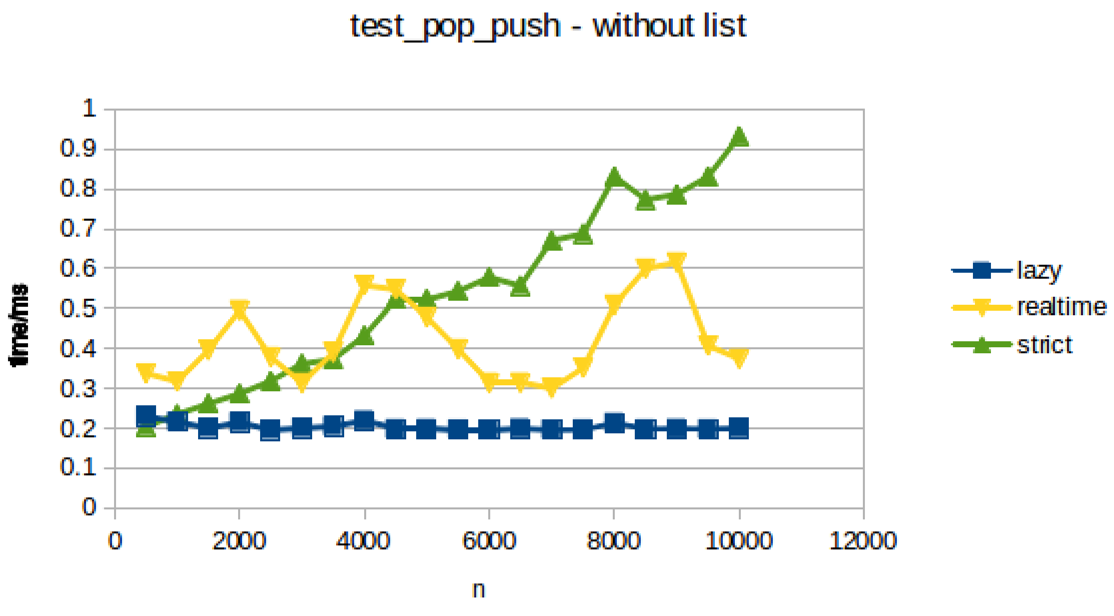

test_push
time/ms
n
lazy
list
realtime
strict
0
200
400
600
800
1000
1200
1400
1600
1800
0
2000
4000
6000
8000
10000
12000

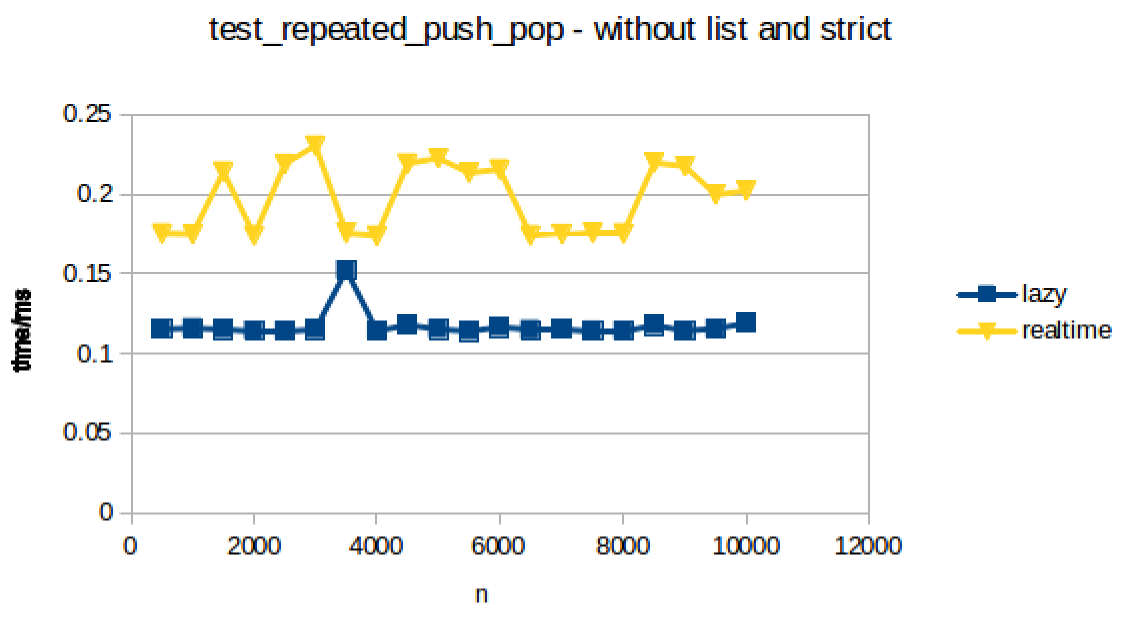

test_pop_push
time/ms
n
lazy
list
realtime
strict
0
100
200
300
400
500
600
700
0
2000
4000
6000
8000
10000
12000


test_repeated_push_pop
time/ms
n
lazy
list
realtime
strict
0
100
200
300
400
500
600
0
2000
4000
6000
8000
10000
12000


test_pop - without list
time/ms
n
lazy
realtime
strict
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
2000
4000
6000
8000
10000
12000


test_push - without list
time/ms
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2000
4000
6000
8000
10000
12000
n
lazy
realtime
strict


test_pop_push - without list
time/ms
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
2000
4000
6000
8000
10000
12000
n
lazy
realtime
strict



5.5 Test Conclusions

From the test results, we can conclude the following about the 4 different queues;

1. **Haskell List:** We see that the simple list-based data structure performs very well on `pop`, outperforming all the other data structures. However, on any test involving `inject`, the queue quickly slows down, due to the $O(n)$ cost of adding new items to the queue.

2. **Paired O(1) Non-Reusable Queue:** The Paired Queue shows some interesting behaviour. First of all, we note that the data structure seems to be slightly linear in the size of the queue on all operations. This is most due to the fact that a very large reversed end list will build up as the data structure is constructed, leading to an $O(n)$ reversal on the first `pop`. Not that this also happens on the `inject` test case, as a single `pop` is performed to force thunks.

   We can also clearly see the problems of the queue when reusing the data structure, which makes the queue perform and $O(n)$ operation for each pop.

3. **Real-Time Strict Queues:** The Real-Time Strict Queues show $O(1)$ performance on every operation, though we note that they are considerably slower than their lazy counterpart.

4. **Lazy Amortized $O(1)$ Queues:** The Lazy Queues seem to not only be quite fast, but also maintain an almost constant speed, as opposed to the Real-Time queues, whose performance seem to vary somewhat.

6 Conclusion

In working with this project, we have managed to explore the difficulty of implementing data structures in purely functional languages. We have looked at Haskell, and how to make this language in particular perform well, and worked with a very nice performance testing framework. Finally, we have implemented several queue data structures for functional languages, and experimentally tested their performance, to show that they performed as expected.

References

Robert T. Hood and Robert C. Melville. Real time queue operations in pure lisp. Technical report, Ithaca, NY, USA, 1980.

Gerth Stølting Brodal. `http://www.cs.au.dk/~gerth/aa13/slides/functional.pdf`, 2013.

Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 6 1999. ISBN 9780521663502.

`http://www.haskell.org/ghc/`, 2013.

`https://github.com/bos/criterion`, 2013.

Bryan O'Sullivan. `https://github.com/bos`, 2013.