Husky: A C++

Functional

Programming Library

Testing Report

Contents

1. Introduction ………………………………………………………………………………………… 3
2. Testing Methodology and Platform Specification……...…………………………….. 4
3. Evaluation: Husky vs FC++ …………………………………………………………………… 5
4. Evaluation: Husky vs FTL, Haskell and Python 3.4 ….…….……………………… 8
5. Conclusion …………………………………………………………………………………………… 11
6. Appendix: …………………………………………………………………………………………... 12

**Introduction**

In designing the tests for our library we attempted to be as complete and comprehensive in the testing to find the strengths and weaknesses of the different functional implementations across all the libraries. The libraries we used as our benchmarks are:

FC++ : A functional library introduced in April 2001 at SIGPLAN. It was  
 based on a research paper presented by the authors at the 2000  
 International Conference on Functional Programming (ICFP 2000).

FTL: A more recent functional library based on c++11 and updated as of   
 2013.

Python: Functional library in Python 3.4

Haskell: Haskell prelude

The functions from our library we tested are below in Table 1. The other C++ libraries did not have as many functions as our library so, those comparison could not be made. Python and Haskell had the full complement.

**Table 1 Husky higher-order functions tested**

|  |  |
| --- | --- |
| maps and concatenation | map, concatMap, concat1, concat |
| filters | filter, takeWhile, dropWhile, span\_container, break\_container |
| folds | foldl, foldr, scanl, scanr |
| container manipulation | cons, reverse, take, drop, splitAt |
| zips | zip, zip3, zipWith, zipWith3, unzip, unzip3 |
| container properties | tail, init, at |

**Testing Methodology and Platform Specification**

Our testing methodology was to generate a random vector of 3 different data types:

* integers
* std::string
* objects ( a struct that contained both a random integer and std::string).

The container used is a vector of 5 different sizes; 100000, 200000, 300000, 400000, 500000.

Each test was then run three times. The result used in our data is an average of those 3 results. (Please see the appendix for an example of data generated for 100,000 units for each library and every data type.)

The exception is with FCC++ which we will discuss more in depth in our report since it was supposed to be our main benchmark.

Testing platform:

Operating System: Mac OS X 10.10.3 (64 bit)

Hardware: 2.3 GHz Intel i7 quad-core,   
16gb RAM 1600Mhz DDR3, SSD drive

Compiler: Clang++

However due to problems running the FC++ library we had to test on a Windows 8.1. The library did not work on a Mac OS X. So we re-ran the test for both Husky and FC++ on:

Operating System: Windows 8.1 (64 bit)

Hardware: 2.5 GHz Intel i7 quad-core,   
16gb RAM 1600Mhz DDR3, SSD drive

Compiler: g++

**Evaluation: Husky vs FC++**

We will discuss our results of Husky versus FC++ separately from the rest of our results because the testing had to be done separately. It did not compile on our main testing platform, Mac OS X, and therefore we had to rerun Husky and FC++ on Windows to generate results.

In addition, FC++ was always going to be our main benchmark, as it was the only other extensive Functional Library in C++, as well as being widely published and presented at a conference. Therefore it is worthy to compare separately.

Evaluation criteria: Ease of use

One of the criteria for evaluating a library was ease of use. With Haskell and Python we encountered no difficulties. Nor did we with the FTL though it had very limited documentation and a limited number of functions. The library we encountered the most difficulties with was FC++.

The first difficulty encountered when writing tests for the FC++ library was getting the library to compile without using any documentation or tutorial, since it does not provide them. In order to get FC++ to compile on g++ -std=c++11, we had to look into the actual implementation of the library in order to fix some bugs. An example is the addition of a “typename” qualifier in the line 436 of list.h.

After getting the library to compile, the next difficulty was using its functions and its wrapper class (List<T>) without, again, any documentation. The only sources of help for this task were the sparsely distributed comments in each header file. After finally being able to run the tests, we noticed that stack overflows were occurring even when the only thing done in main() was constructing the fcpp::Lists. After a couple of hours trying to find the source of the error, we found a comment in the implementation of list.h that reads:

// Long lists create long recursions of destructors that blow the

// stack. So we have an iterative destructor. It is quite tricky to

// get right. The danger is that, when "bypassing" a node to be

// unlinked and destructed, that node's 'next' pointer is, in fact, a

// List object, whose destructor will be called. As a result, as you

// bypass a node, you need to see if its refC is down to 1, and if

// so, mutate its next pointer so that when its destructor is called,

// it won't cause a recursive cascade.

In short, the default implementation of the library’s List causes long recursions of destructor calls, therefore the stack overflows were actually caused by destructors. After some brute-force testing, we estimated a “long list” to be a ~130,000 element List in our testing suite. Since our testing began at 100,000 elements and increased by 100,000 thereafter, we were only able to test against FC++ for 100,000 elements.

In order to make the tests work for longer lists, we discovered that we had to manually set a flag (FCPP\_SAFE\_LIST) in order to prevent this recursive cascade of destructors. By doing that, we were able to run the tests in our Windows testing machine. However, even after setting the flag on, longer lists cause the system to terminate the program after a given amount of time has passed. This problem happened both on our Windows (which responded by reporting that the program was not working) and Mac OS X machines (which responded with a Killed: 9 signal). After some research, we hypothesized that this meant the program was consuming so much memory that the OS killed it.

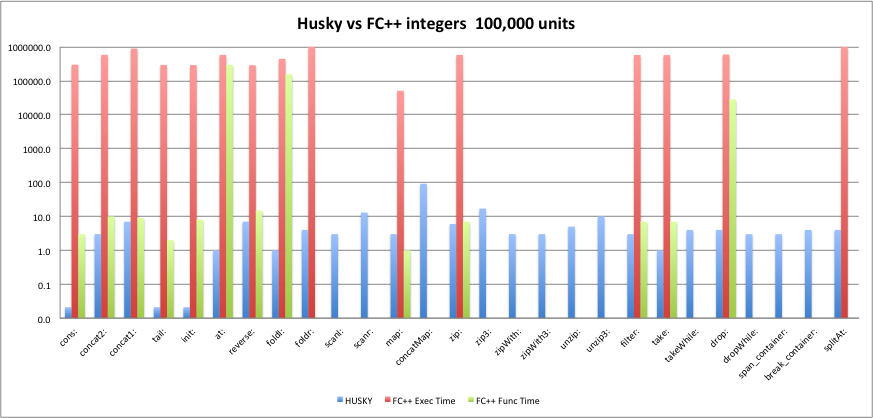
On ease of use our library far out performed FC++ on the following criteria:

* We provide extensive documentations of every function including a   
  tutorial and manual
* Our library can be used on containers larger than 130,000 elements
* Our library can be used on hardware platform that can compile C++11/14 and is not limited to just the Windows platform.

Evaluation criteria: Performance

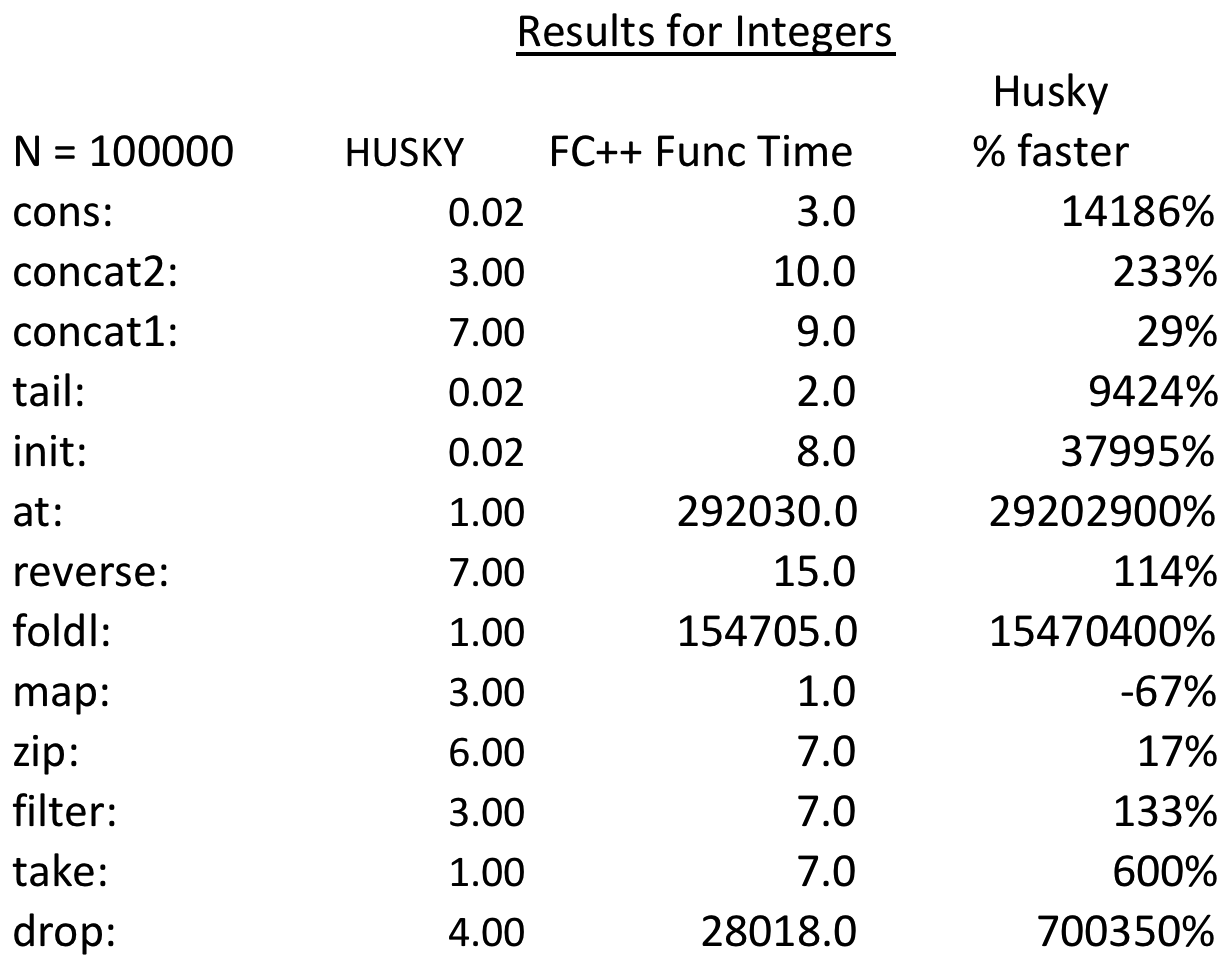
Since the only platform we could compile FC++ was Windows we re-ran our Husky library on Windows to get performance metrics. We also chose to measures for those tests in FC++: the time it took for the FC++ functions to return and the time it took for the whole program to return, with the FCPP\_SAFE\_LIST flag set on. We considered the foremost measure to be important because even if FC++ functions turned out to be faster than ours, the user could still need to go through the trouble of waiting for the excessive time the program takes to construct/destruct its data structures.

All of these difficulties implementing programs in FC++ are presented in our comparison graph, which can be seen below.



\*Scale is log 10. Time is in milliseconds

Due to the challenges we had with FC++, we chose to run our tests for a Lists of 100k integers only, and for less functions. In fact, there were some functions that caused the process to be killed by the OS (e.g. splitAt and foldr). The data is presented on a log scale to capture the inordinate amount of time it took to run each function for FC++. As illustrated in the chart above, if the time for the whole program to return is used for measurement the Husky library (blue bar) far outperformed FC++ (red bar) by orders of magnitude. If we used the time it took for just the functions to return for FC++ (green bar), the Husky library still outperformed FC++ and in some cases still by orders of magnitude. On the criteria of timed performance, Husky (and its implementation of extending the C++11/14 STL) shows it’s power against an older implementation in FC++.



All time in charts above are in milliseconds

**Evaluation: Husky vs FTL, Haskell and Python**

Evaluation criteria: Ease of use

Similar to FC++, the FTL library had very limited documentation. It also had a much more limited number of higher order functions as well. However, since it is a much more recent library and implements its functions through extending the C++11 STL similar to our approach we thought it was a worthy benchmark despite its very limited feature set. Another area where Husky outperforms FTL is in number of functions. Husky has about 5 times more functions than FTL making it a much more comprehensive functional library.

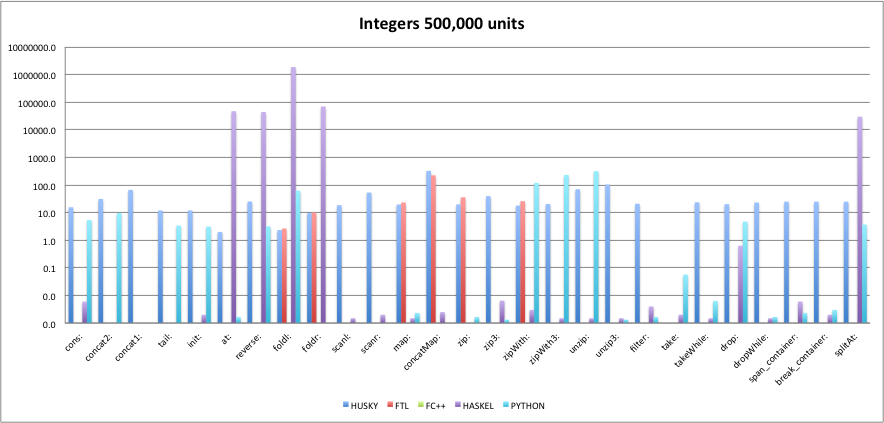
The documentation for Haskell and Python are both quite extensive. In terms of ease of use Haskell might be considered the gold standard because of its elegant and concise definitions of functions. As a point of departure, in looking up definitions of functions for our implementation, website such as Wiki and others would sometime include both the mathematical definition and Haskell definition as well.

Evaluation criteria: Performance

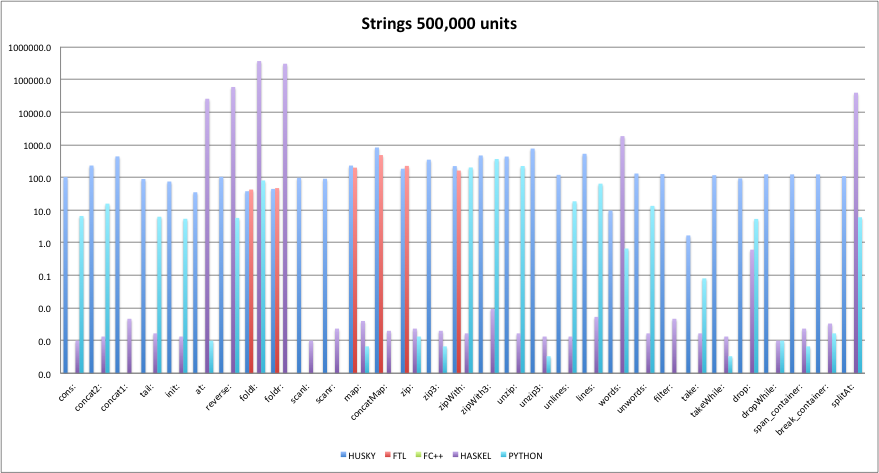
In discussing performance criteria the charts for vectors size 500,000 for integers, std::string and Records (objects) are used here. The charts for 100,000 to 400,000 elements for the different data types are included in the appendix. The observations for 500,000 elements generally holds true for all other sizes.

The graphs are log scale base 10 to capture the wide difference in times in Haskell. Due to this noteable underperformance by Haskell in the at, reverse, foldl, foldr and words functions it is worth pointing out here. Haskell is orders of magnitudes worse at these calculations than all the other libraries. So much so that we had to change the scale to logarithmic just to capture the full disparity. Another area where Haskell’s performance is abysmal is on vectors of objects. We just couldn’t get any compile times on these elements. The functions kept running for an inordinate amount of time, so we had to abort the tests.

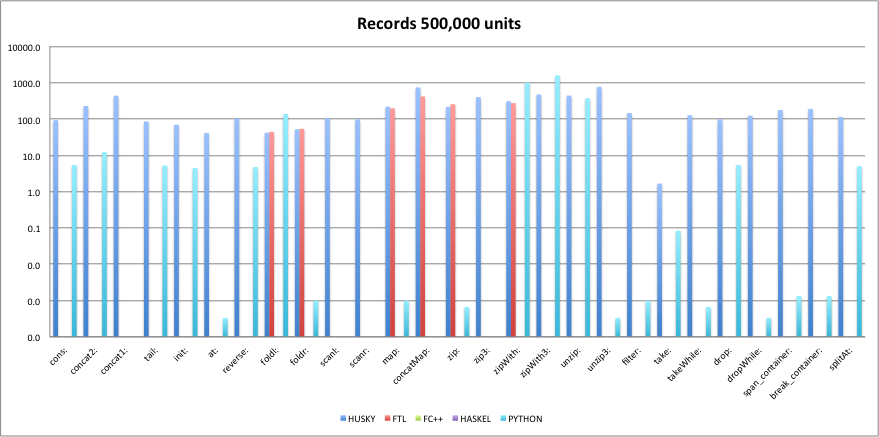
In comparing performance across these 4 libraries, one area that immediately stands out are functions where the library performs the function through lazy evaluation. Using the zip, map, filter functions as an example both Haskell and Python performs these function thru lazy evaluation almost instantaneously. Where as Husky and FTL though fast at around .1 milliseconds are still much slower. This performance feature is why implementing lazy evaluation would be an important 1.2 implementation for our Husky library.



\*Scale is log 10. Time is in milliseconds

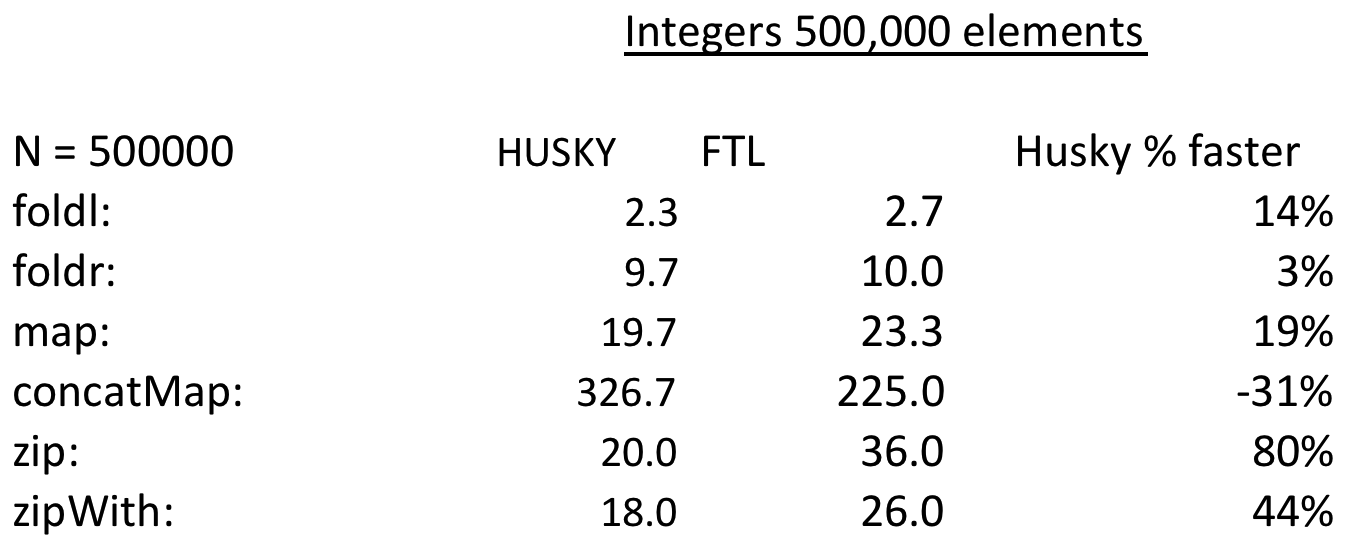


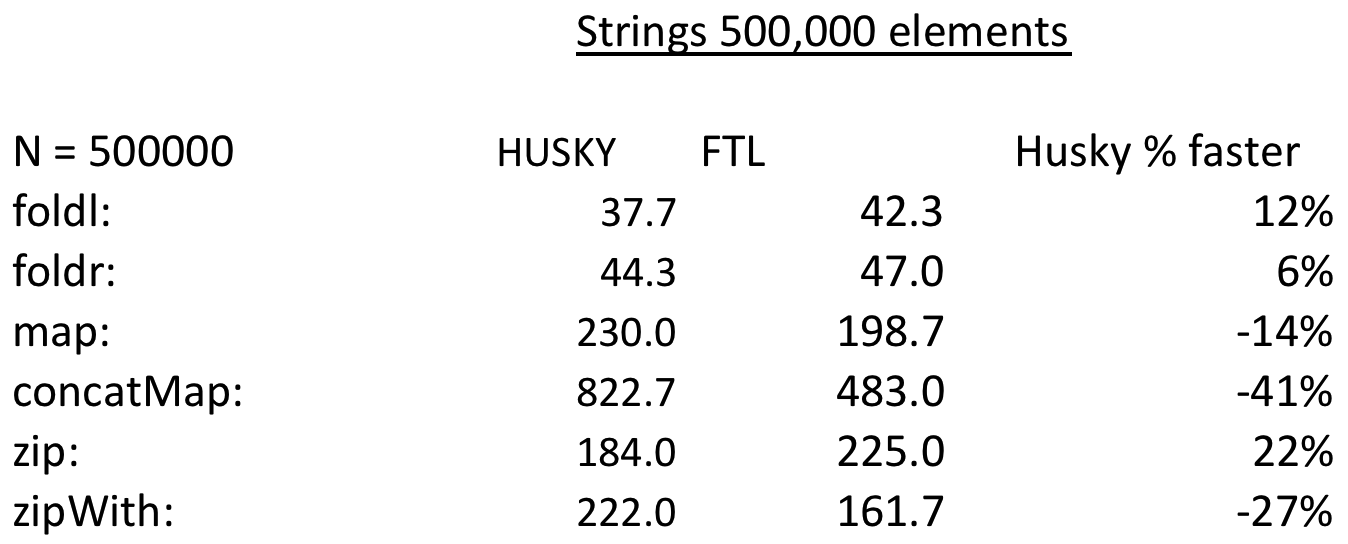
\*Scale is log 10. Time is in milliseconds

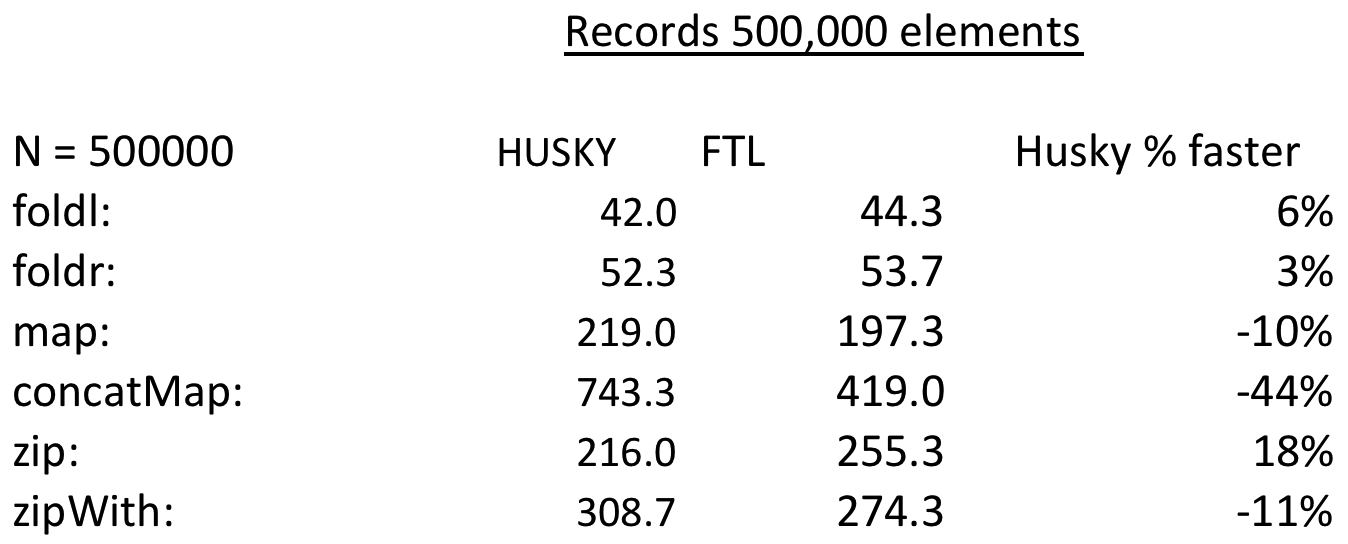


\*Scale is log 10. Time is in milliseconds

In comparing Husky’s performance directly to FTL, the other library that implements C++11 STL, the performance is very similar. While comparing function performance on vectors of integers, Husky is faster, however it gives up a lot of those gains when comparing function performance on vectors of string and it is a push when comparing vectors of objects. However, Husky is a much more comprehensive library with about 5 times more functions. Therefore with no hit on computation performance you get a more comprehensive library.







All time in charts above are in milliseconds

**Conclusion**

In evaluating the performance of Husky, our team was quite happy with the results we were able to achieve and still feel there is room for improvement. We were pleased to see that on many metrics from ease of use to timed performance we were able to extend the C++11/14 STL to beat, FC++, a widely published and well regarded though dated implementation of higher order functional programming in C++.

In addition, we were more than pleased to see we held our own in performance against a “pro” library in STL and give a more comprehensive feature set.

Against Haskell, we are far from approaching its elegance however, we feel that once we implement lazy evaluation we could possibly beat its performance because it is really slow on certain functions and certain data types.

In terms of all round performance Python 3.4 seems to be the clear winner. While it doesn’t have quite the elegant syntax of Haskell it is much better than the imperative languages. It also never had any problems with any size vector, or any data type. Though it was never actually the fastest in any particular function computation it always seemed to be the second fastest, so overall was a very strong performer.

**Appendix**

