4 Results

In this section I will present the results obtained from the compiler optimisation. The metodology and the nature of the benchmarks is explained in the “Benchmarks & methodology” section.

Results1: Fibonacci

Out first test suite runs the benchmark on our 3 fibonacci variants. As a refresher they are as follow:

- The first one is implemented traditionally, carrying at all times 2 Ints representing the last 2 fibonacci numbers and computing the next one
- Second one boxes those Ints into a datatype that will be allocated every time it is changed
- The Third one will make use of our optimisation and mutate the boxes values instead of discarding the old one and allocating a new one.

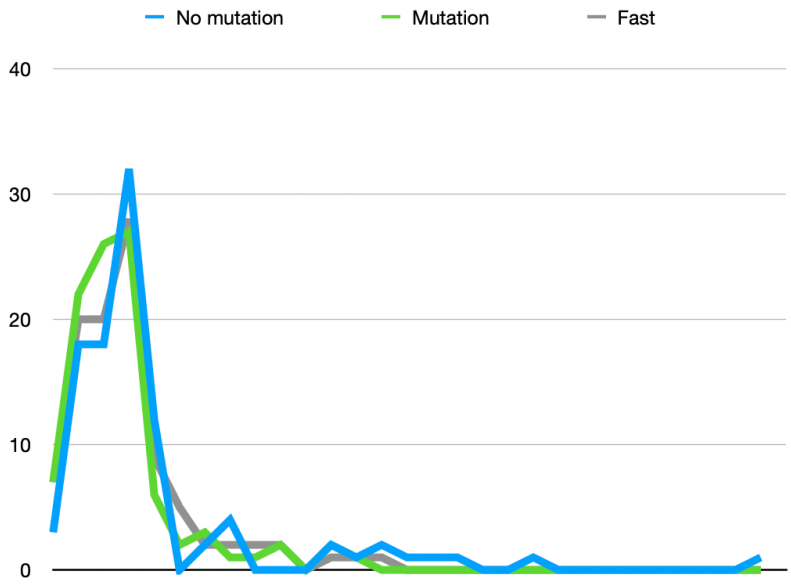
The hypothesis is as follows: Chez is a very smart and efficient runtime, and our example is small and simple. Because of this, we expect a small difference in runtime between those three versions. However, the memory pressure incurred in the second example will trigger the garbage collector to interfere with execution and introduce uncertainty in the runtime of the program. This should translate in our statistical model as a greater variance in the results rather than a strictly smaller mean.

The results

Here are there results of running our benchmarks 100 times in a row:

```
../Idris2_fib_benchmark/
    fibTestNoMutation.idr,5.76e-4,0.00119,6.733999999999996e-4,1.0818499999999999e-8
../Idris2_fib_benchmark/fibTest.idr,5.84e-4,8.41e-4,6.449e-4,2.3797899999999993e-9
../Idris2_fib_benchmark/
    fibTailRec.idr,5.88e-4,8.53e-4,6.499100000000001e-4,2.692301899999999e-9
```

```
[3, 18, 18, 32, 12, 0, 2, 4, 0, 0, 0, 2, 1, 2, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 1, 1]
[7, 22, 26, 27, 6, 2, 3, 1, 1, 2, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0]
[7, 20, 20, 28, 9, 5, 2, 2, 2, 2, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0]
```



This is the result of calling our data analysis program on the csv file generated by our benchmarking program. The benchmarking program was called with those options

```
build/exec/benchmarks -d ../idris2-fib-benchmarks -o results.csv -p $(which
    idris2dev)
```

And the statistical analysis program with no options except for the file:

```
build/exec/stats results_mutation_100_attempts_with_startup.csv
```

The results are in the following format:

```
name of benchmark, minimum, maximum, average, variance
```

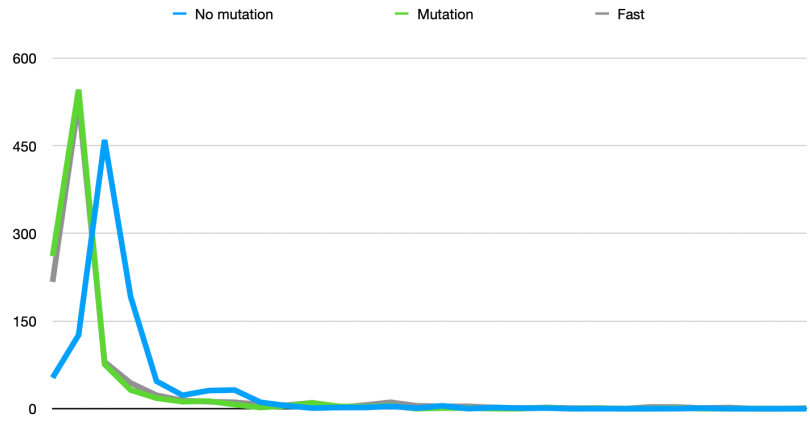
The three arrays at the end correspond to an aggergation of the data in “buckets”. Our statistical tool makes 30 “buckets” which represent the different time slots that each benchmark result falls into. The first bucket is the minimum time measured across all

benchmarks and the last bucket is the maximum time measured across all benchmarks. There are 28 other buckets in between those two extremities. The array represents the number of results that land for each bucket.

As you can see the results are pretty consistent with our predictions but the values themselves aren't statistically significant. In order to get a better picture we are going to run the same benchmark 1000 times instead of 100.

```
../Idris2_fib_benchmark/
    fibTestNoMutation.idr,5.08e-4,0.001795,6.561829999999996e-4,1.4789385511000005e-
    8
../Idris2_fib_benchmark/
    fibTest.idr,5.08e-4,0.001753,5.882930000000001e-4,1.5392219150999998e-8
../Idris2_fib_benchmark/
    fibTailRec.idr,4.89e-4,0.001974,6.241300000000006e-4,3.8718697099999886e-8

[53, 126, 460, 192, 47, 23, 31, 32, 11, 5, 1, 2, 2, 4, 1, 5, 0, 2, 1, 1, 0, 0, 0, 0,
    0, 1, 0, 0, 0, 0]
[261, 546, 76, 32, 18, 12, 13, 7, 2, 5, 10, 4, 3, 5, 0, 1, 1, 0, 0, 2, 0, 1, 0, 0,
    1, 0, 0, 0, 0, 0]
[217, 527, 80, 44, 23, 14, 12, 11, 7, 6, 10, 2, 6, 11, 5, 4, 4, 2, 1, 2, 1, 1, 0, 3,
    3, 1, 2, 0, 0, 1]
```



The results are pretty similar which gives us a greater confidence in their accuracy.

There is however something we can do to improve our measurement and that is to subtract the startup time of the scheme runtime. Indeed every program is measured using the difference between the time it started and the time it ended. But this time also includes the time it takes to launch scheme and then execute a program on it. Indeed the following program:

```
main : IO ()
main = pure ()
```

Takes 0.13 seconds to run despite doing nothing. This time is the startup time and can go up to 0.3 seconds.

Results 2: Fibonacci without startup time

In order to remove the startup time we are going to change the emitted bytecode to wrap our main function inside a time-measuring function. Since the timer won't start until the program is ready to run the startup time will be eliminated. Running our empty program we get

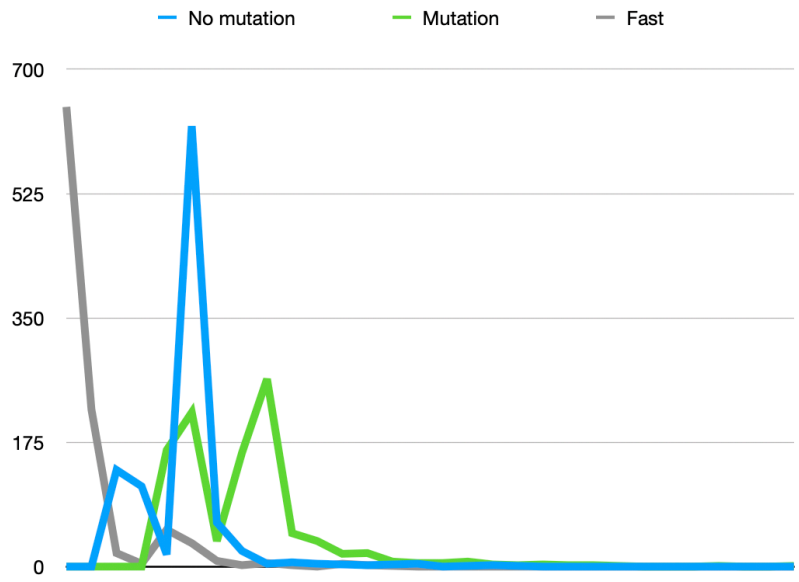
```
0.000000000s elapsed cpu time
```

Which is what we expect.

This time we will run our benchmarks 1000 times using the same command as before. We expect to see the same results but the difference should give us a greater interval of confidence. Running our statistical analysis gives us those results

```
../Idris2_fib_benchmark/
    fibTestNoMutation.idr,1.696760896,1.977075901,1.7447377120060026,9.1830322164425
    9e-4
../Idris2_fib_benchmark/
    fibTest.idr,1.734708117,2.152951106,1.786299514231,0.002247060292357963
../Idris2_fib_benchmark/
    fibTailRec.idr,1.65627213,1.881412963,1.6768551703740004,9.142437018832634e-4

[0, 0, 136, 113, 17, 620, 62, 22, 4, 6, 4, 3, 2, 3, 4, 0, 1, 2, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0]
[0, 0, 0, 0, 164, 217, 36, 160, 264, 47, 36, 18, 19, 7, 5, 5, 7, 3, 2, 3, 2, 2, 1,
    0, 0, 0, 1, 0, 0, 1]
[647, 221, 19, 4, 52, 33, 8, 2, 5, 2, 0, 4, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0]
```



As you can see the results aren't exactly as expected, both our *average* and our *variance* is higher than without any optimisation. A result that we definitely did not anticipate and goes against the belief that founded our hypothesis.

One possible explanation is that scheme performs JIT compilation and correctly identifies the hot-loop in our unoptimized example but is unable to perform such optimization with our mix of pure and mutating code.

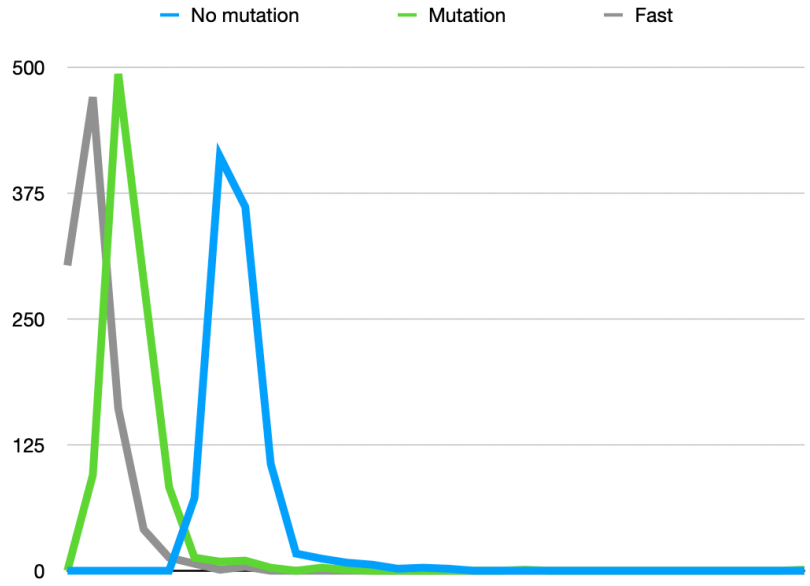
Results 3: Fibonacci without startup time, small loop

In order to test the JIT hypothesis we are going to run the same test, *without* startup time but with a much smaller loop so that the results are measured in milliseconds rather than seconds. This should be enough to prevent the runtime from identifying the loop and performing its optimisation.

In order to reduce the running time from seconds to milliseconds we simply change the loop count from `8*10^6` to `8*10^4` reducing it by two orders of magnitude reduces the running time accordingly.

```
../Idris2_fib_benchmark/
    fibTestNoMutation.idr,0.007216185,0.008861124,0.007520532116999987,3.58278518563
    47296e-8
../Idris2_fib_benchmark/
    fibTest.idr,0.006543267,0.010942671,0.006867369243000004,5.3106313711037986e-8
../Idris2_fib_benchmark/
    fibTailRec.idr,0.006385357,0.007625528,0.006624731209000001,2.4002041892751334e-
    8

[0, 0, 0, 0, 0, 72, 411, 361, 106, 17, 12, 8, 6, 2, 3, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0]
[0, 95, 493, 288, 83, 13, 9, 10, 3, 0, 3, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 1]
[303, 470, 161, 41, 13, 7, 1, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0]
```



And indeed this data does not disprove our JIT hypothesis (but it does not confirm it either). However, since this thesis is not about the intricacies of the scheme runtime we are going to let the issue rest for now.

Those results showcase two things: That our optimisation works, and that it is not significant enough to be strictly superior to other forms of optimisations. Ideally the best way to test our optimisation would be to write our own runtime which runs on *bare metal* or some approximation of it (WASM/LLVM) which would (probably) be even faster than scheme, and give us more control over which optimisations play nicely together and which ones are redundant or even harmful to performance.

Idris2 has an alternative javascript backend, however, the javascript code generation is unable to translate our programs into plain loops free of recursion. Because of this, our

benchmark exceeds the maximum stack size and the program aborts. When the stack size is increased the program segfaults.

Safe inlining

As we've seen in the context review, one of the use-cases for linear types is to detect where control flow allows for safe inlining of functions. In the following snippet, `y` cannot be inlined without duplicating computation.

```
let x = 1 + 2
    y = x + 3 in
    y + y
```

Indeed inlining it would result in

```
let x = 1 + 2
    x + 3 + x + 3
```

where the `+ 3` operation is performed twice. `x` however can be inlined safely:

```
let y = 1 + 2 + 3 in
    y + y
```

the `1 + 2` operation is performed only once after inlining.

One area where linearity and inlining comes into play is when defining effects. Indeed, a linear state monad could have the following `bind` signature:

```
(>>=) : (1 _ : LState s a) -> (1 f : a -> LState s b) -> LState s b
(>>=) (LState s) c = LState (\x => …)
```

Which indicates that the state is inspected linearity and the function is applied exactly once. Since programs using `bind` compose with themselves we often see the following sequence of operations:

```
inital >>= f >>= g
```

which once inlined results in

```
LState (\x => let v = initial >>= f)
                  g v)
```

```
LState (\x => let v = x (LState (\y => let w = y initial
                                           f w))
                  g v)
```