3 Benchmarks & methodology

In order to test our performance hypothesis I am going to use a series of programs and run them multiple times under different conditions in order to measure different aspects of performances. Typically, observing how memory usage and runtime varies depending on the optimisation we use.

Each benchmark will be compared to its control which will be the original compiler without any custom optimisation. The variable we are going to introduce is the new mutation instruction.

There are 2 types of benchmarks we want to write:

- synthetic benchmarks designed to show off our improvements.
- real-world benchmarks designed to test its influence on programs that were not specifically designed to show off our improvements.

The goal of the first category is to explore what is our "best case scenario" and then go from there. Indeed, if the best case scenario doesn't provide any results, then either there is something wrong with our implementation, or there is something wrong with our idea.

The second category aims to collect data about programs that are not built with a specific optimisation in mind such that we can observe how our changes manifest in everyday programs. This could give us insight into how to modify existing programs so that they take full advantage of linear types. Here is one possible course of action:

- Benchmark X doesn't show any improvement over the original compiler implementation.
- One function is changed from an unrestricted definition to a linear one.
- We find a performance improvement.

Synthetic benchmarks

Synthetic benchmarks are designed to show off a particular effect of a particular implementation. They are not representative of real world programs and are mostly there to establish a baseline so that individual variables can be tweaked with further testing. For this project I have designed 3 benchmarks which are all expected to highlight our optimisation in different ways.

Fibonnaci

One cannot have a benchmark suite without computing fibonacci, in this benchmark suite we are going to tweak the typical fibonacci implementation to insert an allocating function within our loop. Our mutation optimisation should get rid of this allocation and make use of mutation instead. Because of this we are going to look at three variants of the same program.

Traditional Fibonacci

This version is the one you would expect from a traditional implementation in a functional programming language

```
tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec 1 1 k
  where
    rec : Int -> Int -> Nat -> Int
    rec prev curr Z = prev + curr
    rec prev curr (S j) = rec curr (prev + curr) j
```

As you can see it does not perform any extraneous allocation since it only makes use of primitive values like `Int` which are not heap-allocated. If our optimisation works perfectly, we expect to reach the same performance signature as this implementation.

Allocating Fibonacci

This version of Fibonacci does allocate a new value for each call of the `update` function. We expect this version to perform worse than the previous one, both in memory and runtime, because those objects are allocated on the heap (unlike ints), and allocating and reclaiming storage takes more time than mutating values.

```
data FibState : Type where
  MkFibState : (prev, curr :  Int) -> FibState

next : FibState -> FibState
next (MkFibState prev curr) = MkFibState curr (prev + curr)

rec : FibState -> Nat -> Int
rec (MkFibState prev curr) Z = prev + curr
rec state (S j) = rec (next state) j

tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec (MkFibState 1 1) k
```

Mutating Fibonacci

This version is almost the same as the previous one except our `update` function should now avoid allocating any memory, while this adds a function call compared to the first version we do expect this version to have a similar performance profile as the first one

```
import Data.List
import Data.Nat

data FibState : Type where
  MkFibState : (prev, curr :  Int) -> FibState

%mutating
next : (1 _ : FibState) -> FibState
next (MkFibState prev curr) = MkFibState curr (prev + curr)


tailRecFib : Nat -> Int
tailRecFib Z = 1
tailRecFib (S Z) = 1
tailRecFib (S (S k)) = rec (MkFibState 1 1) k
  where
    rec : FibState -> Nat -> Int
    rec (MkFibState prev curr) Z = prev + curr
```

```
rec state (S j) = rec (next state) j
```

Real-world benchmarks

For our real world benchmarks we are going to use whatever is available to us. Since the ecosystem is still small we only have a handful of programs to pick from. For this purpose I've elected the following programs:

- The Idris2 compiler itself
- A Sat solver

The Idris2 compiler as benchmark

The Idris2 compiler itself has the benefit of being a large scale program with many parts that aren't immediately obvious if they would benefit from memory optimisation or not. Having our update statement be detected and replaced automatically will allow us to understand if our optimisation can be performed often enough, where and if it results in tangible performance improvements.

A simple SAT solver as benchmark

Sat solvers themselves aren't necessarily considered "real-world" programs in the same sense that compilers or servers are. However they have two benefits:

- You can make them arbitrarily slow to make the performance improvement very obvious by increasing the size of the problem to solve.
- They still represent a real-life case study where a program need to be fast and where traditional functional programming has fallen short compared to imperative programs, using memory unsafe operations. If we can implement a fast SAT solver in our functional programming language, then it is likely we can also implement fast versions of other programs that were traditionally reserved to imperative, memory unsafe programming languages.

Measurements

The benchmarks were run with a command link script written in idris itself which takes a source folder and recursively traverses it in order to find programs to execute and measure their runtime.

The analysis of the result was performed with another command line script which reads the output of out benchmark program and output data like minimum, maximum, mean and variance along with arrays for plotting our results on a graph.

Idris-bench

Idris-bench is our benchmarking program and can be found at `https://github.com/andrevidela/idris-bench`.

Idris-bench takes the following arguments:

- `-p | --path IDRIS_PATH` the path to the idris2 compiler we want to use to compile our tests. This is used to test the different between different compiler versions. Typically running the benchmarks with our optimized compiler and running the benchmarks without our optimisation can be done by calling the program with two different version of the idris2 compiler.
- `-t | --testPath TEST_PATH` The path to the root folder containing the test files.
- `-o | --output FILE_OUTPUT` The location and name of the CSV file that will be written with our results.
    * Alternatively `--stdout` can be given in order to print out the results on the standard output.
- `-c count` The number of times each file has to be benchmarked. This is to get multiple results and avoid lucky/unluck variations.
- `--node` If the node backend should be used instead. If this flag is absent, the Chez backend will be used instead

Idris-stats

Once our results have been generated we are going to analyse them by computing the minimum time, maximum time, mean and variance of the collection of benchmark results. For this we are going to rely on another script: idris-stats. It take out CSV output and compute the data we need and output them as another CSV file. Again the code can be found here https://github.com/andrevidela/idris-bench/blob/master/script/idris-stats.idr.

The program takes the file to analyse as single argument. The file is expected to be a CSV file with the following format:

```
name of first benchmark, result1, result2, result3, etc
name of second benchmark, result1, result2, result3, etc
...
name of final benchmark, result1, result2, result3, etc
```

The first column is ignored for the purpose of data analysis.

For each row we compute the minimum value, the maximum value, the mean and the variance. As a reminder, the variance is computed as the sum of the square difference with the mean divided by the number of samples.

Expected results

Our expectation is that our programs will run fasters for 2 reasons:

- Allocation is slower than mutation
- Mutation avoids short lived variables that need to be garbage collected

Indeed allocation will always be slower than simply updating parts of the memory. Memory allocation requires finding a new memory spot that is big enough, writing to it, and then returning the pointer to that new memory address. Sometimes, allocation of big

buffers will trigger a reshuffling of the memory layout because the available memory is so fragmented that a single continuous buffer of memory of the right size isn't available.

Obviously all those behaviours are hidden from the programmer through *virtual memory* which allows to completely ignore the details of how memory is actually laid out and shared between processes. Operating systems do a great job a sandboxing memory space and avoid unsafe memory operations. Still, those operations happen, and make the performance of a program a lot less consistent than if we did not have to deal with it.

In addition, creating lots of short lived objects in memory will create memory pressure and trigger garbage collection during the runtime of our program. A consequence of automatic garbage collection is that memory management is now outside the control of the programmer and can trigger at times that are undesirable for the purpose of the program. Real-time applications in particular suffer from garbage collection because it makes the performance of the program hard to predict, an unacceptable trade-off when execution need to be guaranteed to run within a small time-frame.

Running the benchmarks

All the benchmarks were run on a laptop with the following specs:

- Intel core-i5 8257U (8th gen), 256KB L2 cache, 6MB L3 cache
- 16Gb of ram at 2133Mhz

While this computer has a base clock of 1.4Ghz, it features a boost clock of 3.9Ghz (a feature of modern CPUs called “turbo-boost”) which is particularly useful for single-core application like ours. However, turbo-boost might introduce an uncontrollable level of variance in the results since it triggers based on a number of parameters that aren't all under control (like ambient temperature, other programs running, etc). Because of this I've disabled turbo boost on this machine and run all benchmarks at a steady 1.4Ghz.

The performance hypothesis

Using mutation in an isolated case did not result in any performance improvement, even with pathological code that would typically showcase the Benedict of such benchmarks, no improvements were found.

My best hypothesis is that the Chez optimiser is smart enough to detect those cases and perform the optimisation were looking for before using having to do it.

While the chez backend is not intended to be the only backend for Idris 2 out goal isn't to make the runtime worse in order to show any improvement.

The next step is then to perform the optimisation automatically rather than manually on a large scale program such as the Idris 2 compiler itself.

Performing the optimization automatically

We want to optimise this case

```
let 1 v = a :: b in
    update v
```

The key observation is that a value `v` is constructed and bound linearly *and* its only use is performed in-scope.

Because we *know* it won't be used later, we can mutate this value instead of creating a new copy within the body of `update` and return the mutated reference instead of returning a newly allocated value.

However `update` might also be used in situations where it's argument isn't unique, for example update can be defined non-linearly

```
update : (1 _ : List Nat) -> List Nat
update [] = []
update (x :: xs) = S x :: xs
```

But by virtue of subtyping can still be called with non-linear arguments. Since a shared value can *also* be used exactly once, no invariant is broken. However, this breaks our performance fix since we cannot simply mutation whenever we deal with a linear function. We have to be sure that linearity implies uniqueness.

This is why we need our ad-how scoping rule. It ensures the variable isn't shared before calling a linear function.