0.1 Success Criteria

The success criteria laid out in the project proposal have all been satisfied:

- Translate simple Haskell programs into executable Java bytecode.
- Reject ill-formed programs due to syntactic or type errors.
- Perform simple optimisations during translation.
- Perform evaluation using non-strict semantics.

Beyond these base requirements, a number of the extensions implementing language features have been completed successfully.

0.2 Language Features

The main goal of the project was to support a reasonable subset of Haskell. The planned subset encompassed functions, arithmetic, booleans, lists, simple typeclasses, and laziness: the following program is compilable using HJC (it is even included as a test), and demonstrates various use-cases of all of these features:

```
-- foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ e [] = e
foldl f e (x:xs) = foldl f (f e x) xs

-- sum :: Num a => [a] -> a
sum = foldl (+) 0

-- take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = undefined
take n (x:xs) = x:take (n-1) xs

-- ones :: Num a => [a]
ones = 1:ones

-- valid :: Bool
valid = sum (take 10 ones :: [Int]) == 10
```

Extensions which were successfully implemented add support for user-defined datatypes, user-defined typeclasses and instances, monads, and some syntactic features like operator sections and support for point-free notation: these can be demonstrated by the following program, which exemplifies the monad definition and use-cases:

```
data Maybe a = Nothing | Just a
data [] a = [] | a:[a]

class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
```

```haskell
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just
instance Monad [] where
    xs >>= f = concat (map f xs)
    return x = [x]

-- The monad instance for maybe can be interpreted as function application
-- with support for chaining failure
divide x y = if y == 0 then Nothing else Just (x / y)
x = divide 4 0 >>= divide 20 -- Evaluates to Nothing

-- The monad instance for lists can be interpreted as performing
-- non-deterministic computation: each step can have multiple results
countdown 0 = []
countdown n = n:countdown (n - 1)
onlyEven x = if even x then [x] else []
y = [1,2,3] >>= countdown -- Evaluates to [1,2,1,3,2,1]
z = y >>= onlyEven -- Evaluates to [2,2]
```

Each simple feature name necessarily glosses over many smaller constituent features necessary for use. For example, the 'lists' feature allows for lists to be created using either the plain constructor syntax (`1:(2:(3:[]))`) or syntactic sugar for lists (`[1,2,3]`), and matched using patterns (eg. `[x,y] = [1,2]`). However, there is no support for list comprehensions (eg. `[f x | x <- [1,2,3], even x]`) as they were not a high priority feature.

0.2.1 Correctness

Correctness of the various compiler stages has been empirically tested using a large set of unit, integration, and regression tests: these include tests of complete programs, such as those used for benchmarking.

At the time of writing, there are 245 tests. These are run both on my development machine (described in Section 0.3.1), and on machines provided by Travis CI[1] whenever a commit is pushed to my development GitHub repository. This ensures HJC works in a clean, reproducible environment and not just on my development system.

Bugs found and fixed during development have at least one associated regression test to ensure that they cannot reappear.

As HJC is developed using Haskell, some forms of compiler bugs that could affect the correctness of translation have been mitigated: in particular, type errors and bugs due to mutable state cannot occur.

[1] `https://travis-ci.org/hnefatl/dissertation-project`

0.3 Performance

Although performance was not an important aspect of the success criteria, it is still interesting to evaluate the effectiveness of optimisations on the output program, and the performance of HJC and its output compared to other Haskell to JVB compilers.

The two compilers used for comparisons are Eta and Frege. Eta is a fork of GHC that replaces the backend with one targeting JVB: it can take advantage of the powerful optimisations already available in the front- and middle-end of GHC, which is the world-leading Haskell compiler, so I expected it to perform better than both HJC and Frege. Frege is a from-scratch compiler that compiles to Java instead of JVB, then uses a Java compiler to produce JVB. I expect Frege to perform better than HJC , given the maturity of the project (under development since at least 2011).

Both Eta and Frege enable optimisations by default, so all metrics given for them have optimisations enabled. Metrics given for HJC are labelled to indicate whether or not optimisations have been applied.

0.3.1 Test Environment

Benchmarks were measured as the only active process on my development machine: a ThinkPad 13 running Debian 9 with 8GB RAM and an Intel Core i5-7200U CPU (2.5GHz).

0.3.2 Execution Speed

Figure 1 demonstrates the runtime performance of the benchmark programs after compilation by the different compilers. It is evident that the performance of programs compiled by HJC is significantly lower than those from Frege or Eta.

Interestingly Frege produced more performant programs than Eta, which was not expected, given that Eta leverages the work done on the GHC compiler.

The inefficiency of programs output by HJC seems to stem primarily from my implementation of runtime closures (described in Section **??**). Figure 2 shows that almost 40% of a program's execution is spent in the functions associated with creating and calling closures: `enter`, `addArgument`, and `clone`. A more efficient implementation should have most of the runtime spent in one or more of the `Impl` functions (such as `_v19Impl`), which perform the actual logic of an expression of the Haskell program, rather than the bottleneck being in the runtime system.

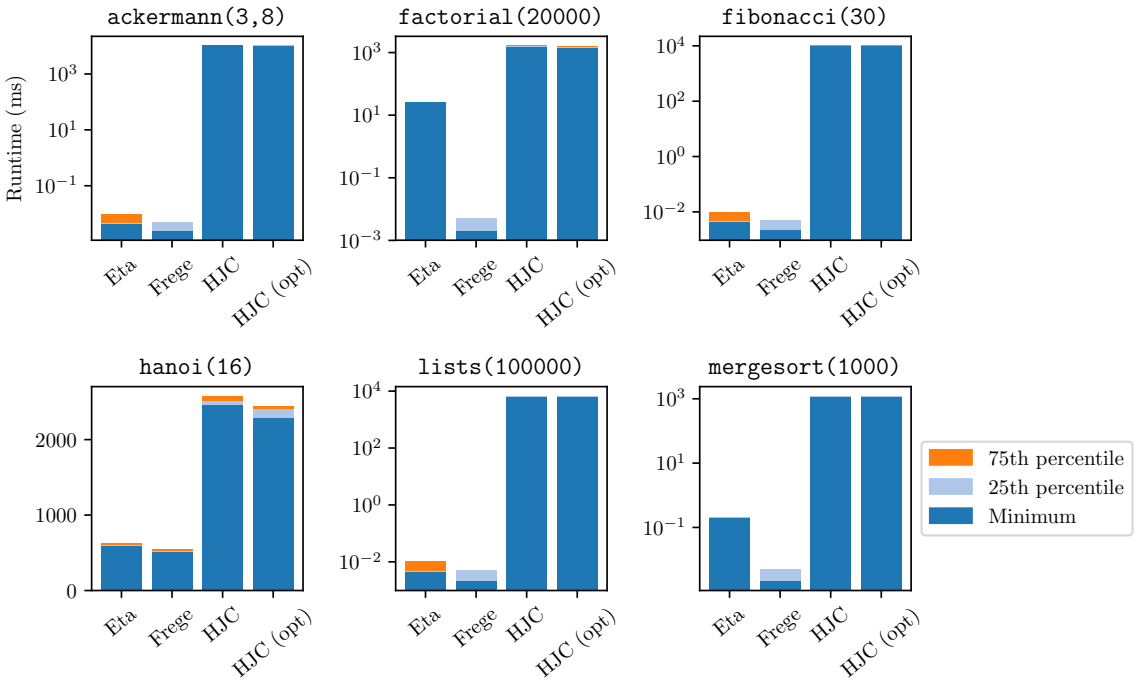

Figure 1: Minimum runtime of the benchmark programs. Minimum is plotted instead of median as it is more reflective of program speed without system load.

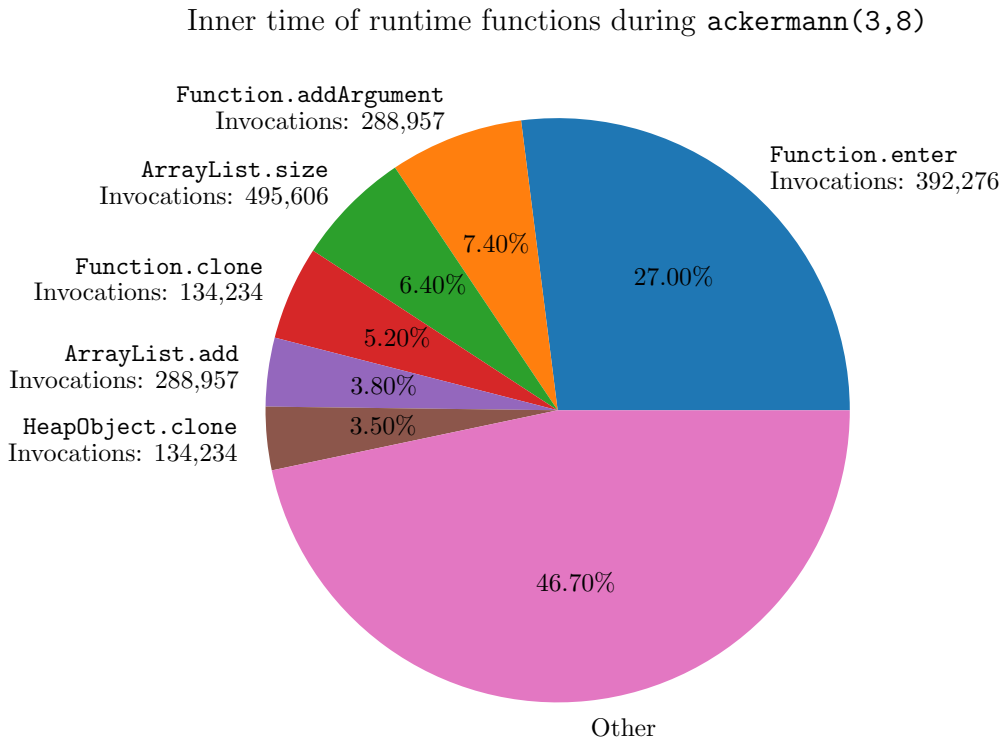

Figure 2: A breakdown of the percentage of execution time spent in each Java function in the generated code ('inner time') for the `ackermann` benchmark. Functions with inner time of less than 3% are grouped into a single wedge.

As described in the implementation chapter, HJC uses objects of a single class to represent closures, with objects of the class storing `Function` objects generated using the `invokedynamic` instruction and Java 8's support for anonymous functions. This is different from the approaches taken by both Eta and Frege, where closures are translated into anonymous classes that implement a Functional Interface[2] allowing objects of the classes to be invoked like anonymous methods.

This design choice was deliberate: before implementing HJC , it seemed like using a single class for all functions would be more efficient than using a new class for each function, as the equivalent Java code is significantly shorter. After performing the implementation and comparing the two different approaches, using functional interfaces with a new class for each function results in simpler code for the creation and evaluation of closures, while also eliminating significant amounts of overhead involved with creating the function objects used by HJC , which I expect plays a large role in the performance difference.

0.3.3 Compiler Performance

Figure 3 presents the minimum time taken to compile each benchmark program: HJC is significantly faster than both Eta and Frege to compile all benchmarks except for `mergesort`, which is significantly slower.

The poor performance appears to be due to the large number of bindings involved: processing the list of 1000 input numbers pushes the inefficient stages to their limits. The `hs-java` library used in code generation was known to perform poorly – efforts were made to improve the performance, in one case **halving** the time spent in code generation[3] for the `mergesort` benchmark, but as this was an unexpected work package I ran out of time before being able to improve the performance to a satisfactory level. This benchmark also revealed that the current let-lifting algorithm is possibly exponential-time. Neither component should take much work to improve up to acceptable scalability.

Figure 4 shows the same metrics for just the `ackermann` benchmark on HJC , so the times for each stage can be seen more clearly. It is clear that code generation is the most expensive stage in the main pipeline: profiling reveals that this is mostly due to the inefficient implementation of parts of the `hs-java` library.

It is interesting that HJC usually takes less time to process the input when performing optimisations than when not. From Figure 4 we can see this is due to a significant amount of the compilation time being spent in the code generation stage, and compressing and writing the compiled classes into a jar file: the unreachable code elimination optimisation described in Section **??** can significantly reduce the amount of code that reaches code generation, which also reduces the amount of bytecode that needs to be compressed and written to disk.

[2] `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`
[3] `https://github.com/hnefatl/hs-java/commit/5cbd5c3995161199442310ac30c101b0497a7f98`

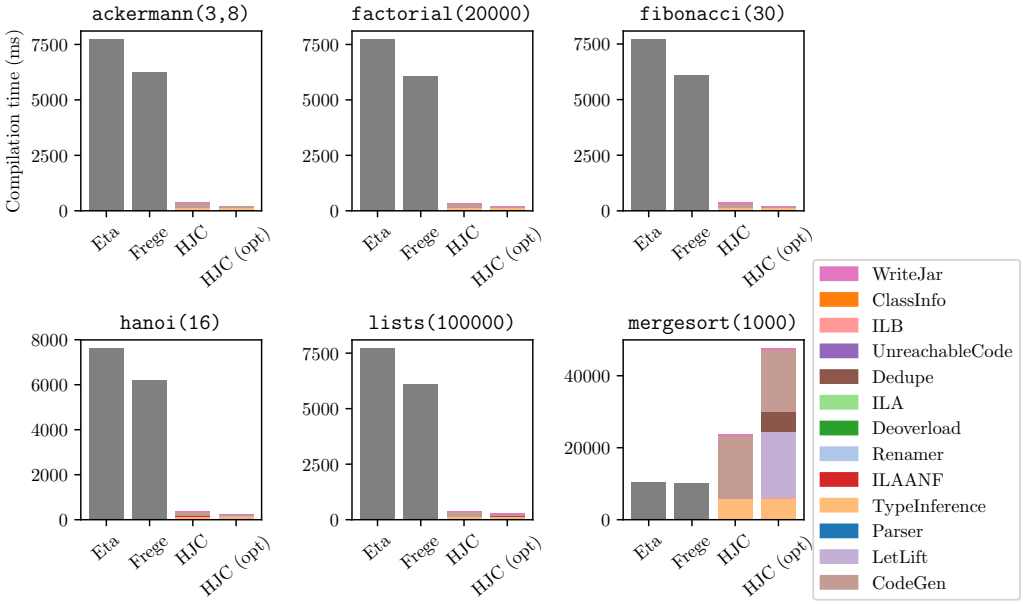

Figure 3: Minimum time taken to compile the benchmark program. The coloured sections indicate the amount of time spent in each stage.

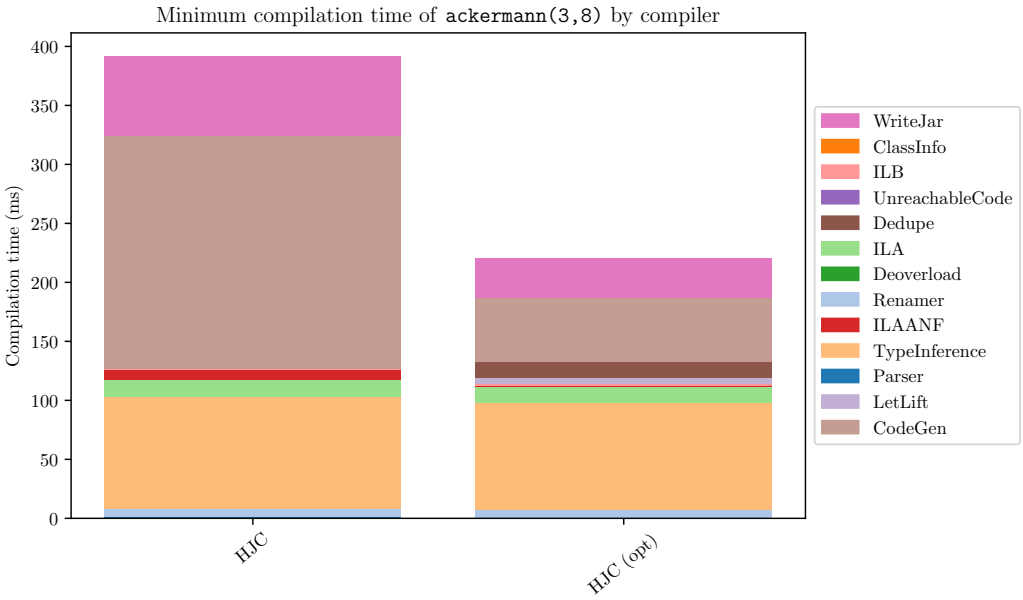

Figure 4: Runtime of each stage for the `ackermann` benchmark.

0.3.4 Executable Size

Figure 5 displays the compiled size of each benchmark program after compilation by the various compilers. All three compilers generate class files that implement the logic of the Haskell program, usually have a class file for each datatype defined in the Haskell program, and have class files providing runtime support (for example, each compiler has an equivalent to the

`Function` class described in Section **??**).

HJC produces the smallest executables for all benchmarks, with optimisations trimming off a reasonable amount of generated bytecode.

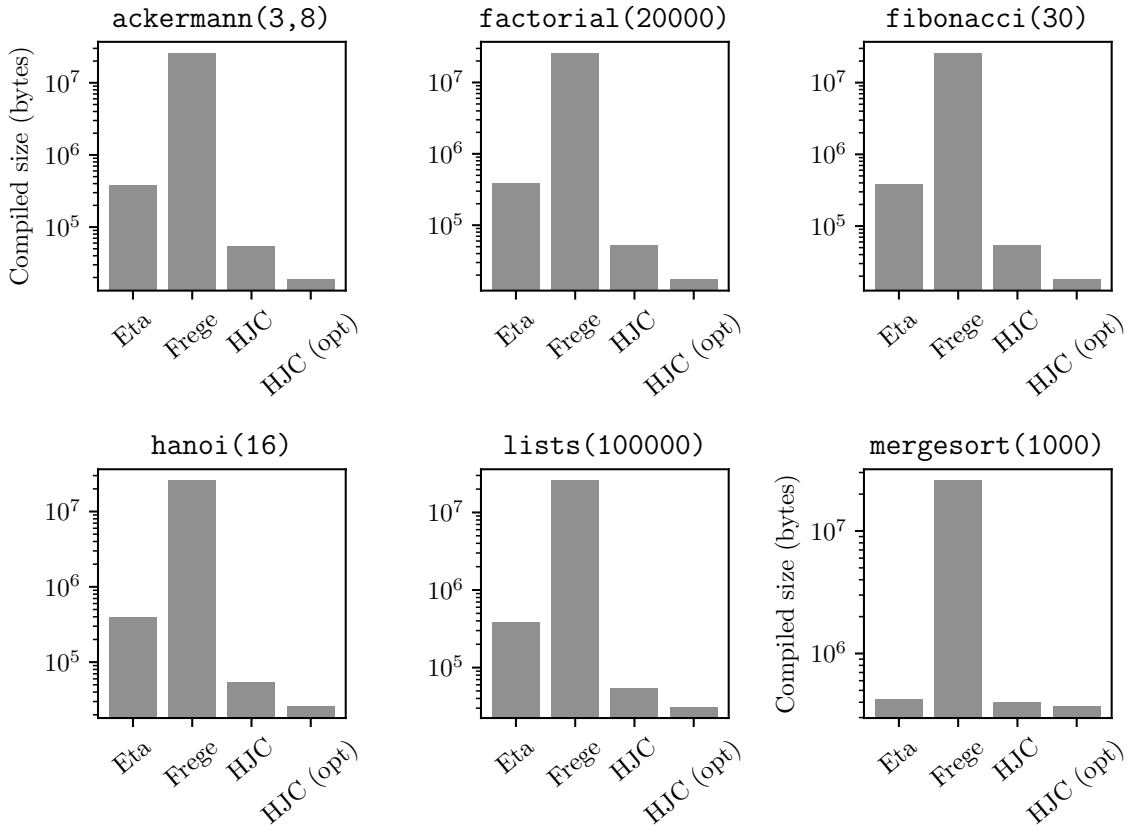

Figure 5: Compiled size of each benchmark program. This size includes the runtime system and the bytecode corresponding to the actual program.

0.3.5 Impact of Optimisations

Figure 6 shows that the implemented optimisations are successful in reducing the execution time of compiled programs, although none have a truly significant impact: speedups can be seen of up to 10%, but the performance remains poor compared to Frege and Eta.

Figure 7 demonstrates that the unreachable code elimination optimisation is effective at reducing the size of compiled programs, in some cases halving the size. The amount of removed code is relatively constant, as the majority of removable code in these benchmarks will be from the standard library, which produces a constant size overhead. For larger input programs, the space savings could be greater.

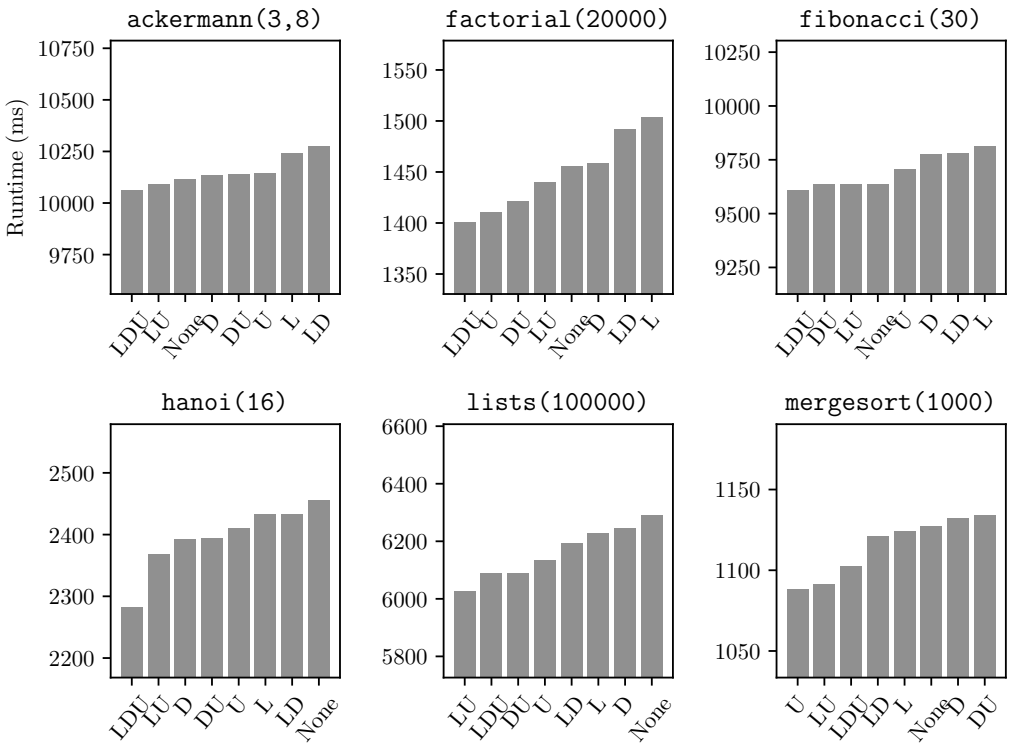

Figure 6: Minimum runtime of the benchmarks after compilation with various optimisations: L stands for let-lifting, D for binding deduplication, and U for unreachable code elimination.

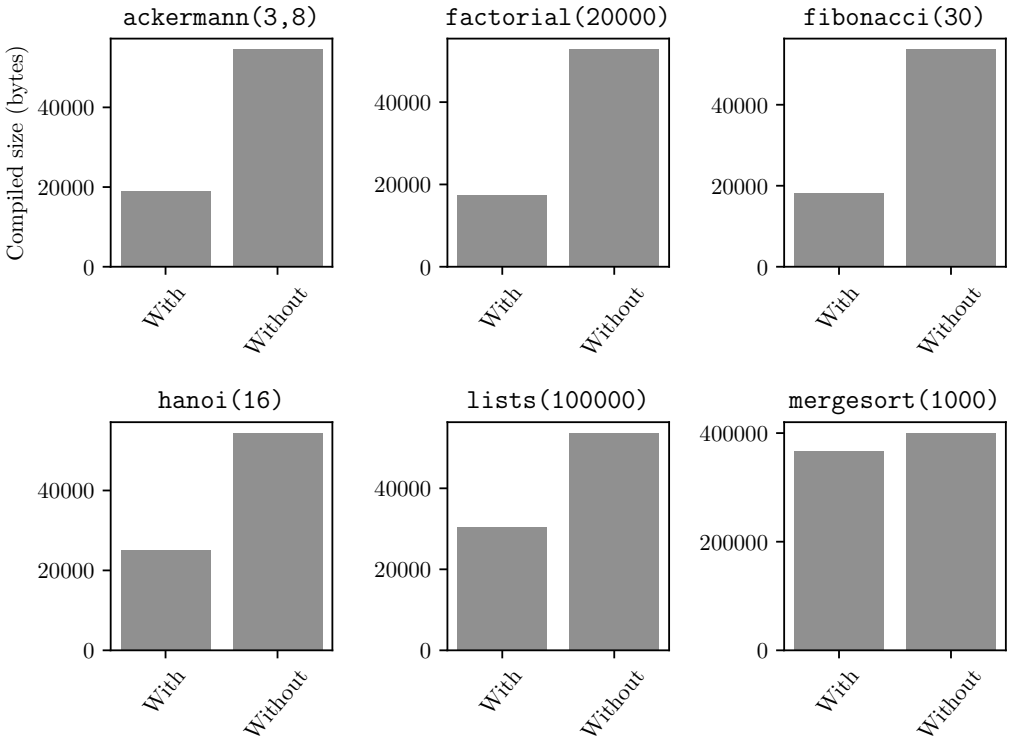

Figure 7: Executable size from HJC with and without the unreachable-code-elimination optimisation.