This project was successful: all of the success criteria were met and a number of extensions were completed, resulting in a sizeable subset of Haskell being supported. The optimisations implemented have a positive effect on the runtime behaviour of compiled programs, and while the programs are not as performant as might be desired, building a solid compiler for a significant subset of Haskell was the aim, rather than sheer performance.

I have learned a great deal about the design of compilers from this project, as well as the effects of language design choices and features on the different stages of a traditional compiler.

0.1 Hindsight

The only part of the project that I would redo, given the benefit of hindsight, is the typechecker.

The typechecker is the most complex part of the compiler: while some amount of complexity is unavoidable due to Haskell's relatively exotic type system, I feel that the 'complexity overhead' added by the implementation onto the complexity of the underlying concepts is higher here than in any other component.

If I could re-implement this component from scratch, I would focus less on type-checking performance (which was a design goal in the current implementation) and more on clarity and extensibility. This might involve implementing distinct passes for generating type/typeclass constraints and then solving them, rather than doing both in a single pass as in the current implementation. This would likely trade performance for clarity.

0.2 Further Work

The implementation of closures used by this compiler was an experiment: it differs from the approach taken by other Haskell to JVB compilers such as Frege and Eta. However, evaluation showed that this design is not very performant, so it would benefit the compiler to switch to the more efficient approach; although that is quite a large chunk of work.

Smaller, more achievable chunks of work include adding support for more language features. There are plenty of these in a large language like Haskell, ranging from syntactic sugar like list comprehensions to more complex features like typeclass/instance contexts.

Only a few optimisations were implemented in this project, mostly due to time constraints. Strictness analysis is an optimisation that I did not implement but expect would product sizeable performance improvements, as it reduces the overhead introduced by lazy evaluation in cases where the semantics of the program are unchanged. Inlining is another major optimisation, made easier to implement by Haskell's purity.

Another exciting extension would be replacing the typechecking component with the OutsideIn(X) framework described in Vytiniotis et al. [**?**], which would open up opportunities for supporting modern extensions to Haskell's type system such as GADTs and multi-parameter typeclasses.