This project aimed to build an optimising compiler from Haskell to Java Bytecode, supporting a reasonable subset of Haskell and implementing at least one optimisation. Extensions included extending the supported subset of the language and implementing additional optimisations.

All of the major stages in a traditional optimising compiler pipeline were implemented, apart from lexing/parsing: verification (through type checking/inference), lowering into intermediate languages, optimisation transformations, and code generation. In order to implement these stages, I extended my existing knowledge of type systems, language design, and compiler design from the associated Part 1B and Part 2 courses.

The project was a success: the compiler can translate a reasonable subset of Haskell into executable JVB that evaluates lazily, with optimisations to improve both the size of the output program and the runtime performance.

Would be nice to include benefits of using my tool over an alternative, but realistically there aren't any: mine supports a smaller subset of Haskell, doesn't provide Java interop, and is slower than the others. The one plus is compilation speed, which isn't really enough to bring up.

0.1 Language Choice

Haskell is a mature purely functional programming language with lazy evaluation and static typing. It is popular in academia for its powerful type system, and has been used as inspiration for the dependently-typed language Idris[**?**]. It is also seeing increasing industrial usage by companies such as Facebook[**?**], J.P. Morgan[**?**], and Galois[**?**].

The semantics of Haskell are very different from other popular functional languages such as OCaml and F#: laziness and purity are unusual aspects of the language offering unique benefits: purity ensures that effects are restricted to explicitly marked portions of the code, which reduces the potential for bugs and allows for aggressive optimisations; laziness can improve efficiency in compositional code such as $\mathrm{head} \circ \mathrm{sort}$, which can run in $O(n)$ rather than $O(n\ \log(n))$ under non-strict evaluation, and allow for convenient definitions such as $\mathrm{powersof2} = \mathrm{map}\ (2\times)\ [0..]$. In addition, these semantics and their implementation are only lightly covered in the Tripos modules on compilers and language design, which made Haskell a very interesting language to implement.

Java Bytecode (JVB) is the strict, impure 'assembly language' for the Java Virtual Machine (JVM). It sits at a comfortable middle-ground between CISC and RISC instruction sets, with convenient utility instructions but without much bloat, making it a relatively enjoyable bytecode to work with. As it targets the JVM, JVB also benefits from automatic garbage collection, which made it a desirable target language for a project with a short time-frame such as this, as it removed the need to implement a form of memory management.

0.2 Existing Work

GHC is the industry-leading Haskell compiler, capable of generating high-performance code rivalling C. It takes advantage of purity to aggressively optimise code and can parallelise programs using only small hints from the programmer. GHC generates native code for a variety of architectures, and includes an LLVM backend: however, it doesn't target JVB.

There are two actively maintained compilers from dialects of Haskell to JVB that I am aware of: Eta and Frege. Both languages are dialects of Haskell with modifications to enable interoperation with Java with less effort. The Eta compiler is a fork of GHC, replacing the code generation stage with one targeting JVB. Frege also aims to provide high-quality Java interoperability, but targets Java rather than JVB directly. It was developed from scratch and is now written in the Frege language, as the compiler can bootstrap itself.