An Optimising Compiler from Haskell to Java Bytecode
Progress Report

Keith Collister, kc506
Robinson College
Thursday 31st January, 2019

Project Supervisor: Dr. Timothy Jones **Overseers:** Prof. Simone Teufel

Director of Studies: Prof. Alan Mycroft Dr. Andrew Rice

Success Criteria

The success criteria established in the project proposal were to support translating a basic subset of Haskell to lazily evaluating Java bytecode, performing simple optimisations.

My project currently does not quite meet those criteria: the compiler translates Java bytecode from a medium-sized subset of Haskell source code into executable bytecode, but it currently evaluates 'more strictly' than intended, and one core language feature (typeclass instances) currently has critical bugs when compiled to bytecode. No optimisations have been implemented so far.

The main cause of the delay was unexpected dependencies between different source language features. To define simple operations like addition, it is necessary to implement functions, datatypes, and typeclasses: this forced me to implement some of these features sooner than planned (during Michaelmas rather than at the start of Lent term). By implementing these to support simple operations, some planned features were delivered 'for free' earlier than planned (eg. lists, simply defined as `data [] a = [] | a:[a]` after implementing support for datatypes).

Another cause for delay was a period of unexpectedly high workload due to the Cloud Computing Unit of Assessment at the end of Michaelmas term, which halted development for two weeks.

Current Progress

The compiler contains 4 coarse stages:

1. Frontend: The Haskell source is parsed using an external library, which I have slightly modified to allow parsing built-in data constructors such as `(,)` and `[]`. Variables in the Haskell source are renamed to eliminate shadowing, reducing complexity in subsequent stages. Dependency analysis is performed on the source declarations: recursive relations are inferred and the order of compilation of declarations is decided.

2. Type Inference: Polymorphic overloaded type signatures (eg. $\forall\alpha.\ \text{Num}\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$) are inferred from the source declarations using the results of the dependency analysis. The Haskell AST is explicitly annotated with type information, and any user-provided type signatures are checked against the inferred signatures.

3. Middle-end: Overloaded functions (eg. `(+) :: Num a => a -> a -> a`) are deoverloaded into 'dictionary-passing' equivalents (`(+) :: Num a -> a -> a -> a`). Typeclasses are replaced with datatype and function declarations, and typeclass instances are replaced with variable declarations. These are all implemented as Haskell source-to-source transformations.

   The Haskell AST is lowered through two intermediate languages (ILs) and into Administrative Normal Form. The first IL provides a simple language that optimisation passes can easily

process, and the second IL makes allocation and (lazy) evaluation explicit for easier code generation.

4. Code Generation: The final IL is transformed into Java Bytecode. An external library is used to convert a logical description of the bytecode to actual binary data, and I have extended the library to support various JVM features required by the lowering (notably the `invokedynamic` and `lookupswitch` instructions).

New Schedule

Given that the project's schedule was effectively rearranged by implementing features sooner than expected, I've included a new workplan below:

- **4th Feb – 10th Feb**

  Debug currently failing bytecode for typeclass instances. Rewrite the translation of pattern matching to provide non-strict semantics.

  After this work package, the compiler will be able to transform a reasonably-sized subset of Haskell to executable Java Bytecode.

- **11th Feb – 3rd Mar**

  Implement optimisations: currently, I intend to implement a peephole pass, unreachable code/procedure elimination, dead store elimination, lambda+let lifting, and if there is sufficient time, strictness analysis.

After the end of these two work packages, I should be only a week behind the original proposed schedule, and intend to spend any remaining time fixing bugs and starting on my dissertation write-up.