Computer Science Tripos: Part II Project Proposal

An Optimising Compiler from Haskell to Java Bytecode

Keith Collister, kc506
Robinson College
date

Project Originator:	Keith Collister	
Project Supervisor:	Dr. Timothy Jones	**Signature:**
Director of Studies:	Prof. Alan Mycroft	**Signature:**
Overseers:	Dr. Andrew Rice	**Signature:**
	Prof. Simone Teufel	**Signature:**

Introduction and Description of the Work

The goal of this project is to implement an optimising compiler from a subset of the Haskell language to Java bytecode. A variety of optimisations will be implemented to explore their effect on compilation and execution time, as well as on the size of the output bytecode.

Haskell is a functional, pure, and non-strict language seeing increasing usage in industry and academia. Purity makes programs much simpler to reason about: a programmer can usually tell from the type of a function exactly what it can do, which makes it easier to avoid bugs.

Java Bytecode was chosen as the target language as it is portable and mature. While not as performant as native machine code, bytecode output by the compiler built during this project can be interpreted on almost any platform, rather than being restricted to e.g. only machines with an x86-64 processor. Other bytecodes like python bytecode aren't as well known, and lack existing Haskell libraries that provide an abstraction over them. Compiling to LLVM IR was considered, but would require implementing a garbage collector which is a significant piece of work that's not aligned with the aims of this project.

Starting Point

I intend to use Haskell to develop the compiler, and Python or Bash for quick utility scripts – I have experience with all of these languages.

I've preread the 2018 Optimising Compilers course[1] as preparation: my schedule involves writing optimisations before the module is lectured.

Resources Required

I will use my personal laptop to develop this project: a ThinkPad 13 running NixOS. I will use Git for version control, host the code on a public repository on GitHub, and use TravisCI for automated tests. I also intend to keep a backup repository on an MCS machine – my personal DS-Filestore allowance should be sufficient.

Should my laptop break or otherwise become unusable to complete the project, I have an older laptop running Debian 9 that I can use. It should only cost a few days to get it set up with a Haskell development environment.

I intend to use the GHC compiler[2] with the Stack toolchain[3] for development (both are available under BSD-style licenses).

Substance and Structure of the Project

The aim of the project is to develop an optimising compiler that can translate simple programs written in Haskell into Java bytecode that can be interpreted on platforms supporting the Java Runtime Environment.

As the focus of the project is on the implementation of various language features and optimisations operating on them, I intend to use an existing library for lexing and parsing (`haskell-src`[4]) which can produce an AST from Haskell 98. This will allow for more time to be devoted to those parts of the project which are more aligned with the aim.

Similarly, I intend to use the `hs-java` library[5] to handle the conversion from the logical representation of bytecode to the actual instruction sequences (the "assembly" part of the backend).

As Haskell is a lazy language, one of the major challenges will be to design a way to represent and perform lazy computation. This might be achieved using "thunks", in-memory representations of pending computations. GHC uses a directed graph to keep track of thunks.

Haskell is a very feature-rich language, and those features are often highly dependent on each other: simple things often touch many aspects of the language (for example, the simple numeric literal `5` which would have type `int` in C instead has type `Num t => t` in Haskell, involving type classes and

[1] https://www.cl.cam.ac.uk/teaching/1718/OptComp/
[2] https://www.haskell.org/ghc/
[3] https://github.com/commercialhaskell/stack
[4] https://hackage.haskell.org/package/haskell-src
[5] https://hackage.haskell.org/package/hs-java

type constraints. Simple operations like addition and subtraction on integers are provided by `Int`'s `Num` instance). In the schedule below I've detailed which areas I intend to focus on for language features – the subset should hopefully be small enough to be implementable in reasonable time, but large enough to allow for interesting programs to be written.

The project also aims to implement some optimisations, to improve the performance of the output of the compiler. These are also detailed in the schedule below, and include a mix of classical optimisations (e.g. peephole) and more exotic ones (strictness analysis).

Tests are a vital part of any engineering project. During the work on the project, I will write and maintain a test suite to ensure the various components of the compiler work as expected and guard against regressions. I intend to use the `tasty` framework[6] which provides a standard interface to `HUnit`[7] (for unit and regression tests) and `QuickCheck`[8] (for wonderful property-based tests) to implement this test suite.

Success Criteria

The primary goal of the project is to produce a compiler that can translate source code written in a small subset of Haskell into Java bytecode suitable for execution by the JVM, performing simple optimisations during the translation process. This big-picture goal can be subdivided into distinct aspects:

- To succeed as a Haskell compiler, the result of this project should be able to reject ill-formed programs for syntactic or type errors (within the scope of the subset of Haskell implemented), and convert well-formed programs into Java bytecode. The output programs should perform computation non-strictly.

  This can be evaluated by providing example valid/invalid programs and ensuring the compiler accepts/rejects them appropriately. Lazy evaluation can be tested similarly, using infinite lists and `undefined`.

- To succeed as an optimising compiler, it should produce bytecode which executes faster and/or in less memory than bytecode produced by a non-optimising compiler.

  This can be tested by enabling/disabling optimisations within this compiler and comparing the relative execution speeds/memory footprint, as well as the difference in size of the optimised bytecode compared to the non-optimised bytecode.

- To succeed as a stable and reliable piece of software, it should have a comprehensive test suite including unit tests and regression tests for bugs found during development.

Extensions

There are plenty of interesting extensions to the proposed work:

[6] https://hackage.haskell.org/package/tasty
[7] https://hackage.haskell.org/package/HUnit
[8] https://hackage.haskell.org/package/QuickCheck

- There are many more features to Haskell than those mentioned in this proposal, ranging from syntactic sugar to features in their own right: infix operators, operator sections, point-free notation, user-defined datatypes, type instances, monads, do-notation for monads, do-notation for applicative functors, GADTs, user input/output, etc.

  It would be fun to increase the size of the implemented subset of Haskell, both in order to write more interesting programs and also to explore the effectiveness of existing optimisations on the new changes.

- There exist many more optimisations that could be investigated: there are over 60 "big picture" optimisations listed on the GHC's "using optimisations" page[9].

- One of the greatest attractions of pure languages is the relative ease with which they can be parallelised: any subexpressions can be evaluated at any time without effecting the result of the computation. GHC provides a concurrency extension including a number of simple language constructs and runtime environment options that allow for parallel computations to be expressed in a clean way. It would be extremely interesting to implement something similar (but sadly also very time consuming and far beyond the scope of this project).

- The Haskell Prelude[10] is the "standard library" of Haskell: as it is written in Haskell, it might be possible to compile parts of it using the compiler developed during this project, allowing it to be used in programs. However, this would require quite a significant level of support for the language in the compiler.

- A very cool demonstration for the project would be to compile the project using the compiler developed during the project (bootstrapping). This would require extensive language support though (at the very least, support for monads), which is likely infeasible to be completed.

- One potential advantage of using the JVM as a target is that it may be possible to provide a foreign function interface between Java code and Haskell code.

Schedule

I intend to treat tests as part of a feature: when the schedule lists a certain feature as being deliverable in a slot, that implicitly includes suitable tests for it.

- **15th Oct – 21st Oct**

  General project setup: creating a version-controlled repository of code with continuous integration to run tests.

  Create a simple frontend for converting a given file into an AST using the `haskell-src` package. Focusing on single files for the moment, as without proper support for modules, multiple source files don't make sense.

- **22nd Oct – 11th Nov**

[9] https://downloads.haskell.org/ ghc/master/users-guide/using-optimisation.html
[10] https://www.haskell.org/onlinereport/standard-prelude.html

Implement a typechecking pass over the AST, *including support for typeclasses*. This is one of the most uncertain duration parts of the project, because while the Hindley-Milner type system is well understood and frequently implemented, the extension of type classes seems less comprehensively covered, although there are still some strong leads[11].

After this work, the frontend should be functional and the compiler should be able to reject ill-formed source code either due to syntactic or type errors.

- **12th Nov – 2nd Dec**

  Create a simple (non-optimising) backend for experimenting with lazy evaluation. This should just perform a minimally-featured translation from the frontend's AST to executable bytecode (supporting e.g. basic arithmetic and conditional expressions), but performing evaluation *lazily*.

- **3rd Dec – 16th Dec**

  The goal of this week is to implement a peephole pass to collapse sequences of instructions into more efficient versions. The sequences to be collapsed will need to be decided at the time, based on inspection of the bytecode output by the compiler, and more peephole rules can be added as other transformations are implemented.

  After this work is complete, the absolutely minimal success criteria should have been met, taking pressure off the rest of the planned work.

- **17th Dec – 23rd Dec**

  This week is a slack week, to catch up on anything that fell behind, or to spend time cleaning up any parts of the existing implementation that are messy/fragile/poorly designed.

- **24th Dec – 13th Jan**

  Implement user-defined functions, supporting currying, partial application, recursion, and laziness. Depending on how long this takes, this may be a convenient time to implement a number of smaller related features, such as:

  - Pattern matching.
  - `case` expressions.
  - `let ... in ...` expressions.
  - `... where ...` expressions.

- **14th Jan – 3rd Feb**

  Progress report and presentation, along with another feature:

  Introduce lists: these are one of the most frequently used data structures in Haskell, and form the basis for many algorithms. They also give many opportunities to demonstrate that the implementation of lazy evaluation works correctly (e.g. by careful analysis of expressions like `let l = 1:l in take 5 l`, which should give `[1,1,1,1,1]`).

[11] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.53.3952&rep=rep1&type=pdf

- **4th Feb – 17th Feb**

  Implement an optimisation pass performing common subexpression elimination. In a lazy language, where we can expect to see significant buildup of unevaluated computations in memory, ensuring that we don't have two large blobs of memory devoted to computing the same thing is likely worthwhile.

- **18th Feb – 10th Mar**

  Implement strictness analysis, and accompanying optimisations. The optimisation opportunities revealed by strictness analysis should reduce compiled code size and memory usage, by eagerly evaluating expressions that are guaranteed to require evaluation during program execution.

- **11th Mar – 7th Apr**

  During these weeks, I intend to focus on writing the dissertation.

  Implement some micro-benchmarks to demonstrate the effectiveness of the optimisations, for use in the evaluation.

- **8th Apr – 21st Apr**

  In these weeks I hope to balance work on the dissertation with revision.

- **22nd Apr – 5th May**

  I now expect to switch fully to revision, making only critical changes to the dissertation.

  At the end of these weeks I hope to submit the dissertation.