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 are less 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 is not aligned with the aims of this project.

Similar projects exist, like Frege[1] and Eta[2], that both aim to provide a fully-featured Haskell compiler for programs running on the JVM, with the ability to interoperate with Java. The Eta project aims to accelerate the uptake of Haskell in industry by interfacing with a widely used

[1]https://github.com/Frege/frege
[2]https://eta-lang.org/

imperative language[3]. The motivation behind this project is instead simply individual learning – Haskell has a number of aspects which are not covered in undergraduate courses, such as type-classes and lazy evaluation, which I am very interested in learning how to implement.

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 have preread the 2018 Optimising Compilers course[4] 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[5] with the Stack toolchain[6] 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.

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 type constraints). I intend to implement typeclasses[7], aspects of functions (currying, partial application, recursion), arithmetic, boolean operations, lists (and functions for manipulating them such as `map` and `foldl`). The implementation of many of these features will be different from in conventional languages due to the impact of typeclasses and laziness. These features should cover most of the novel aspects of Haskell that are feasible to be implemented, so should be the most educational to implement.

[3] https://eta-lang.org/docs/user-guides/eta-user-guide/introduction/what-is-eta#motivation
[4] https://www.cl.cam.ac.uk/teaching/1718/OptComp/
[5] https://www.haskell.org/ghc/
[6] https://github.com/commercialhaskell/stack
[7] http://homepages.inf.ed.ac.uk/wadler/papers/classhask/classhask.ps

The project also aims to implement some optimisations to improve the performance of the output of the compiler. These include classical optimisations like peephole analysis, but also strictness analysis[8], which is exclusively useful for lazy languages, so again offers good educational value. I intend to research and implement existing impactful techniques, rather than try to invent new optimisation techniques.

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.

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`[9]) which can produce an AST from Haskell 98 – similarly, the actual assembly of the textual bytecode will be handled by the `hs-java` library[10]. This will allow for more time to be devoted to those parts of the project which are more aligned with the aim.

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[11] which provides a standard interface to `HUnit`[12] (for unit and regression tests) and `QuickCheck`[13] (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, attempting simple optimisations during the translation process.

The compiler 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.

I also hope to identify the cases in which the optimisations produce code that uses less resources than the non-optimised output (either CPU time, memory, or disk space).

Evaluation

To evaluate the success criteria, I plan to use a suite of test programs designed to probe various areas of the compiler, based off GHC's `nofib`[14] repository. Some tests from that suite will likely use features that my compiler does not support, and I intend to modify or discard them depending on how close they are to being supported.

[8] https://www.cl.cam.ac.uk/~am21/papers/sofsem92b.ps.gz
[9] https://hackage.haskell.org/package/haskell-src
[10] https://hackage.haskell.org/package/hs-java
[11] https://hackage.haskell.org/package/tasty
[12] https://hackage.haskell.org/package/HUnit
[13] https://hackage.haskell.org/package/QuickCheck
[14] https://github.com/ghc/nofib

Extra test programs will also be written, to specifically demonstrate features of the compiler: for example a simple program like `let l = 1:l in take 5 l` (with result `[1,1,1,1,1]`) is a good demonstration of lists and laziness. These might be carefully crafted, e.g. to demonstrate the effect of the peephole pass on non-optimised versus optimised code. Combined, these sets of programs should form a broad range of inputs to ensure that the compiler behaves as expected.

Specific metrics that I aim to capture data about are the time taken to compile a program with and without optimisations enabled, the execution time and memory footprint of non-optimised and optimised output programs, and the size of output programs (number of instructions or raw byte size). These should allow for critical evaluation on a number of axes, such as "speed-up of optimised output over non-optimised" against "extra time taken during compilation" or "change in output size". The effects of strictness analysis should also be visible by comparing the memory usage of programs running with and without the optimisation enabled.

To gather data about the performance of the compiler, I intend to use the rich profiling options built in to GHC, together with the `criterion`[15] and `weigh`[16] packages for reproducible benchmarks.

To gather data about the performance of the test programs, I intend to leverage mature JVM tooling by using an existing JVM profiler such as JProfiler or Java VisualVM.

Extensions

There are many interesting extensions to the proposed work:

- 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, GADTs, user input/output, etc.

  Increasing the size of the implemented subset of Haskell would allow for writing more interesting programs, and also exploring 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[17].

- One of the greatest attractions of pure languages is the relative ease with which they can be parallelised: any sub-expressions can be evaluated at any time without effecting the result of the computation. GHC provides a concurrency extension to make such parallel programming easy – it would be interesting to implement such a feature but likely far beyond the scope of this project.

- The Haskell Prelude[18] 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.

[15] https://hackage.haskell.org/package/criterion
[16] https://hackage.haskell.org/package/weigh
[17] https://downloads.haskell.org/~ghc/master/users-guide/using-optimisation.html
[18] https://www.haskell.org/onlinereport/standard-prelude.html

- 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.

- **22nd Oct – 11th Nov**

  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[19].

  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.

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

- **24$^{\text{th}}$ Dec – 13$^{\text{th}}$ 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, etc.

- **14$^{\text{th}}$ Jan – 3$^{\text{rd}}$ 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]`).

- **4$^{\text{th}}$ Feb – 17$^{\text{th}}$ 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 never have two large allocations of memory representing a partial computation of the same thing is likely worthwhile.

- **18$^{\text{th}}$ Feb – 10$^{\text{th}}$ 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.

- **11$^{\text{th}}$ Mar – 7$^{\text{th}}$ 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.

- **8$^{\text{th}}$ Apr – 21$^{\text{st}}$ Apr**

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

- **22$^{\text{nd}}$ Apr – 5$^{\text{th}}$ 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.