Keith Collister

An Optimising Compiler from Haskell to Java Bytecode

Computer Science Tripos – Part II

Robinson College

February 28, 2019

Declaration of Originality

I, Keith Collister of Robinson College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Proforma

Name:	**Keith Collister**
Candidate Number:	
College:	**Robinson College**
Project Title:	**An Optimising Compiler from Haskell to Java Bytecode**
Examination:	**Computer Science Tripos – Part II, July 2019**
Word Count:	**1263**
Project Originator:	Keith Collister
Project Supervisor:	Dr. Timothy Jones

Contents

1 Introduction **9**

2 Preparation **11**

3 Implementation **13**

3.1 Implementation Details 14

3.1.1 Frontend 14

3.1.2 Preprocessor 15

3.1.3 Kind/Class Analysis 16

3.1.4 Dependency Analysis 16

4 Evaluation **19**

5 Conclusion **21**

Bibliography **21**

A Project Proposal **25**

Chapter 1

Introduction

Chapter 2

Preparation

Chapter 3

Implementation

The compiler is comprised of a number of stages and substages, as shown in Figure 3.1:

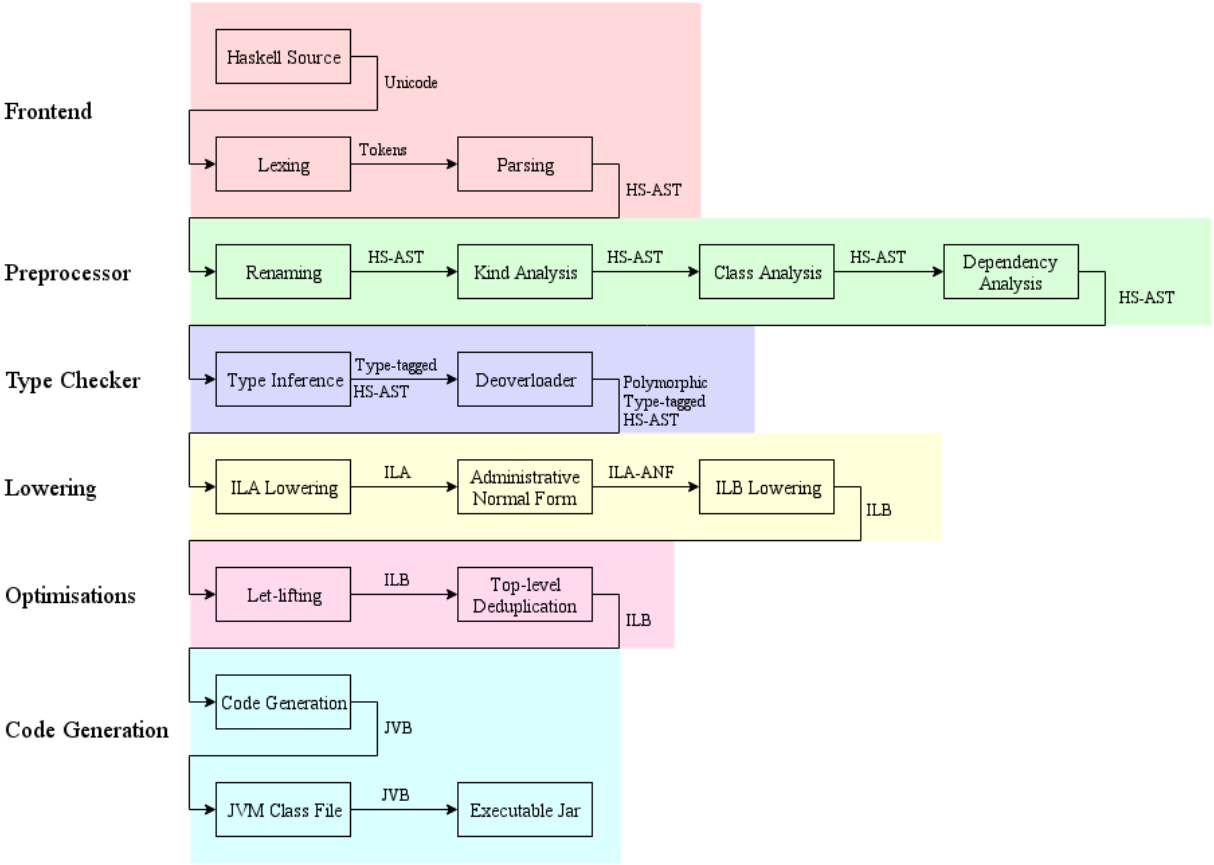

Figure 3.1

A brief overview of each stage is given here for a 'big picture' view of the compiler, followed by more detailed descriptions below.

Frontend

The frontend consists of standard lexing and parsing from Haskell source code into an Abstract Syntax Tree (AST). A modified version of an existing library (haskell-src[1]) is used.

Preprocessing

- The renamer renames each variable so that later stages can assume each variable name is unique: this reduces complexity by removing the possibility of variable shadowing (eg. `let x = 1 in let x = 2 in x`).

[1] https://github.com/hnefatl/haskell-src

- Kind and Class analysis both simply extract useful information about the declarations in the source so that stages of the type checker are simpler.
- Dependency analysis computes a partial order on the source declarations so that the typechecker can process them in a valid order.

Type Checker

- The type inference stage infers polymorphic overloaded types for each symbol, checks them against any user-provided type signatures, and alters the AST so that each expression is tagged with its type.
- Deoverloading converts polymorphic overloaded types to polymorphic types similar to those of System F, and alters the AST to implement typeclasses using dictionary-passing.

Lowering

The lowering stage transforms the Haskell source AST into Intermediate Language A (ILA), then rearranges that tree into Administrative Normal Form (ILA-ANF), before finally transforming it into Intermediate Language B (ILB).

Optimisations

Optimisations transform the intermediate languages while preserving their semantics.

At time of writing these are done on ILB, might change to ILAANF so should update this accordingly.

If any more optimisations are implemented, update the diagram and here.

Code Generation

ILB is transformed into Java Bytecode (JVB), and a modified version of an existing library (hs-java[2]) is used to convert a logical representation of the bytecode into a set of class files, which are then packaged into an executable Jar file.

3.1 Implementation Details

3.1.1 Frontend

Lexing and parsing of Haskell source is performed using the `haskell-src`[3] library, which I have modified to provide some additional desirable features:

- Lexing and parsing declarations for built-in constructors like list and tuple definitions (eg. `data [] a = [] | a:[a]`).
- Parsing data declarations without any constructors (eg. `data Int`)[4]. This is a valuable way of introducing built-in types.

[2] https://github.com/hnefatl/hs-java

[3] https://hackage.haskell.org/package/haskell-src

[4] Declarations of this form are invalid in the original Haskell 1998 syntax, but valid in Haskell 2010: see https://wiki.haskell.org/Empty_type

- Adding `Hashable` and `Ord` typeclass instances to the syntax AST, so that syntax trees can be stored in associative containers.

The syntax supported is a strict superset of Haskell 1998 and a strict subset of Haskell 2010, but my compiler does not have support for all of the features implied by the scope of the syntax. For example, multi-parameter typeclasses are parsed correctly as a feature of Haskell 2010 but get rejected by the deoverloading stage.

```
1 class Convertable a b where
2     convert :: a -> b
3 instance Convertable Bool Int where
4     convert True = 1
5     convert False = 0
```

Figure 3.2: An example of a multi-parameter typeclass

3.1.2 Preprocessor

Renaming

Haskell allows for multiple variables to share the same name within different scopes, which can increase the complexity of later stages in the pipeline. For example, when typechecking the following code we might conflate the two uses of `x`, and erroneously infer that they have the same type. A similar problem arises with variable shadowing, when the scopes overlap. The problem also applies to any type variables present in the source – the type variable `a` is distinct between the two type signatures:

```
1 id :: a -> a
2 id x = x
3
4 const :: a -> b -> a
5 const x _ = x
```

Additionally, variables and type variables live in different worlds. The same token can refer to a variable and a type variable, even within the same scope – the following code is perfectly valid (but loops forever):

```
1 x :: x
2 x = x
```

To eliminate the potential for subtle bugs stemming from this feature, the renamer pass gives each semantically distinct variable and type variable in the source a unique name. It traverses the syntax tree maintaining a mapping from syntactic variable names to a stack of unique semantic variable names (`Map VariableName [UniqueVariableName]`):

- When a new syntactic variable comes into scope (eg. from a let binding, a lambda argument, a new type signature, a pattern match...), a fresh semantic variable name is generated and pushed onto the stack.
- Whenever we leave the scope of a syntactic variable, we pop a semantic name off its corresponding stack.
- When we see a use of a syntactic variable, we replace it with the current top of the corresponding stack.

A similar mapping is maintained for type variables, with the same rules for maintaining it.

Type constants such as `Bool` from `data Bool = False | True` and typeclass names like `Num` from `class Num a where ...` are not renamed: these names are already guaranteed to be unique by the syntax of Haskell, and renaming them means we need to carry more state through the compiler as to what they've been renamed to.

3.1.3 Kind/Class Analysis

The typechecker and deoverloader require information about the kinds of any type constructors (the 'type of the type', eg. `Int :: *` and `Maybe :: * -> *`), and the methods provided by different classes. This is tricky to compute during typechecking as that pass traverses the AST in dependency order. Instead, we just perform a traversal of the AST early in the pipeline to aggregate the required information.

3.1.4 Dependency Analysis

When typechecking, the order of processing declarations matters: we can't infer the type of `foo = bar baz` until we've inferred the types of `bar` and `baz`. The dependency analysis stage determines the order in which the typechecker should process declarations.

We compute the sets of free/bound variables/type variables/type constants for each declaration, then construct a dependency graph – each node is a declaration, and there's an edge from A to B if any of the bound variables/type variables/type constants at A are free in B. It is important to distinguish between the variables/type variables and type constants, as otherwise name conflicts could occur (we don't rename type constants). This separation is upheld in the compiler by using different types for each, and is represented in the dependency graph below by colouring symbol names blue or red.

The strongly connected components of the dependency graph correspond to sets of mutually recursive declarations, and the partial order between components gives us the order to typecheck each set. For example:

```
1 data Bool = False | True     -- d1
2 x = f True                   -- d2
3 f y = g y                    -- d3
4 g y = h y                    -- d4
5 h y = f y                    -- d5
```

Figure 3.3

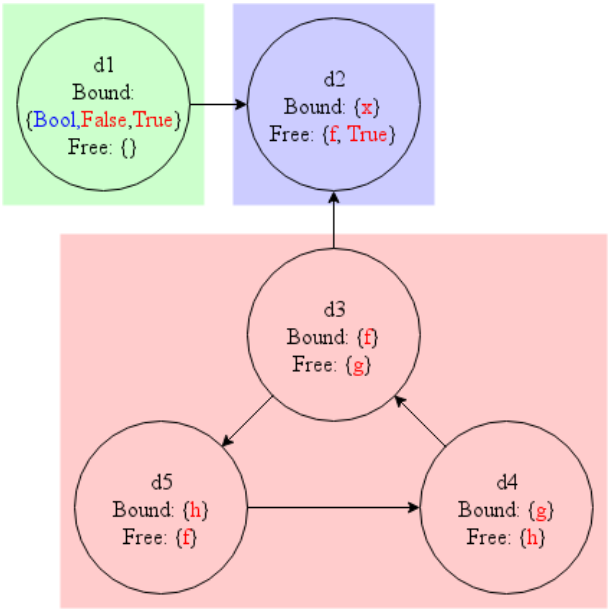

Figure 3.4: The dependency graph for Figure 3.3. Variables in red, type constants are in blue. Strongly connected components are highlighted.

From the dependency graph, we know we have to typecheck $d2$ last, after both the trio of functions and after the datatype declaration. We also know that we need to typecheck the trio of functions together, as they're mutually recursive.

This process works for languages without ad-hoc overloading, like SML. However, in Haskell there are some complications introduced by typeclasses:

- Typeclass member variables can be declared multiple times within the same scope. For example:

```
1 class Num a where
2     (+) :: a -> a -> a
3 instance Num Int where
4     x + y = ...
5 instance Num Float where
6     x + y = ...
```

  Here the multiple declarations of + don't conflict: this is a valid program. However, the following program does have conflicting variables, as `x` is not a typeclass member and is not declared inside an `instance` declaration:

```
1 x = True
2 x = False
```

  These declaration conflicts can be expressed in a grid, where:

  - `SymDef` x and `SymType` x represent top-level declaration and type-signature declarations for a symbol x, like x `= True` and x `:: Bool`.

- `ClassSymDef` x c and `ClassSymType` x c represent `SymDef` x and `SymType` x inside the declaration for a class c, like `class` c `where { x = True ; x :: Bool }`.
- `InstSymDef` x c t represents a `SymType` x inside the declaration for a class instance $c\ t$, like `instance` c t `where { x = True }`.

	`SymDef` x_1	`SymType` x_1	`ClassSymDef` x_1 c_1	`ClassTypeDef` x_1 c_1	`InstSymDef` x_1 c_1 t_1
`SymDef` x_2	$x_1 = x_2$	`False`	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
`SymType` x_2		$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
`ClassSymDef` x_2 c_2			$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
`ClassTypeDef` x_2 c_2				$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
`InstSymDef` x_2 c_2 t_2					$x_1 = x_2 \wedge (c_1 \neq c_2 \vee t_1 = t_2)$

- The variables declared within a class declaration are treated as being the binding instance of those variables, whereas the declarations of those variables within the instance declarations are treated as being free uses of them. This is
- Instance usage is actually syntactic not semantic.

Chapter 4

Evaluation

Chapter 5

Conclusion

Bibliography

Appendix A

Project Proposal

Computer Science Tripos: Part II Project Proposal

An Optimising Compiler from Haskell to Java Bytecode

Keith Collister, kc506
Robinson College
Tuesday 16th October, 2018

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 produced 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 produced 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 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.

[1] https://github.com/Frege/frege
[2] https://eta-lang.org/
[3] https://eta-lang.org/docs/user-guides/eta-user-guide/introduction/what-is-eta#motivation

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 licences).

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.

The project also aims to implement some optimisations to improve the performance of the 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 keeps track of thunks at runtime using a directed graph.

[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
[8] https://www.cl.cam.ac.uk/~am21/papers/sofsem92b.ps.gz

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 resulting bytecode should perform computation non-strictly.

I also hope to identify the cases in which the optimisations produce code that uses fewer resources than when non-optimised (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.

Additional 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 bytecode, and the size of output bytecode (number of instructions or raw byte size). These should allow for critical evaluation on a number of axes, such as "speed-up of optimised bytecode 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 bytecode running with and without the optimisation enabled.

[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

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 emitted bytecode, 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 also 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.

- 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.

- Using the JVM will impose a performance impact compared to compiling to native machine code – this overhead is hard to measure and reduces the informativeness of comparisons to other compilers like GHC. It would be beneficial to find a way of calculating this overhead, to improve the quality of data obtained during evaluation.

[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

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 type checking 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 produced 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, etc.

- **14th Jan – 3rd Feb**

  Progress report and presentation.

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

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]`).

- **4th Feb – 24th Feb**

  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.

- **25th Feb – 24th Mar**

  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 section.

- **25th Mar – 14th Apr**

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

  Near the start of this work chunk, I intend to submit a full draft to my DoS and Supervisor.

- **15th Apr – 28th Apr**

  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 and concentrate fully on revision and the final term.