Keith Collister

An Optimising Compiler from Haskell to Java Bytecode

Computer Science Tripos – Part II

Robinson College

Tuesday 14th May, 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.

I, Keith Collister of Robinson College, am content for my dissertation to be made available to the students and staff of the University.

Signed:

Date: Tuesday 14th May, 2019

Proforma

Candidate Number:	**2392F**
Project Title:	**An Optimising Compiler from Haskell to Java Bytecode**
Examination:	Computer Science Tripos – Part II, July 2019
Word Count:	11963 [1]
Lines of Code:	8569 [2]
Project Originator:	The dissertation author
Project Supervisor:	Dr. Timothy Jones

Original Aims

The aim of the project was to build an optimising compiler from a subset of Haskell to Java Bytecode, to explore compiler and language implementation beyond what was covered in the Tripos.

Work Completed

A larger-than-expected subset of Haskell is supported, with non-strict evaluation and a small number of optimisations. A number of extensions were implemented, including typeclasses.

Special Difficulties

None.

---

[1] Calculated using `texcount -1 -sum "$file"`

[2] Calculated using `wc -l "$file"`

Contents

Chapter 1

Introduction

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. The motivation behind this project was to explore the implementation of compilers beyond their coverage in the Tripos.

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: my compiler (the Haskell to Java Compiler, HJC) 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.

1.1 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[4]. It is also seeing increasing industrial usage by companies such as Facebook[1], J.P. Morgan[5], and Galois[2].

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, reducing the potential for bugs and allowing for aggressive optimisations; laziness improves efficiency in compositional code such as head ∘ sort, which runs in $O(n)$ rather than $O(n \log(n))$ under non-strict evaluation, and allow for convenient definitions such as powersof2 = 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, removing the need to implement a form of memory management.

1.2 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 does not 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.

Chapter 2

Preparation

This chapter covers various aspects of the preparation for the project, including some of the important concepts I learned about while planning the implementation of various stages, a general overview of the compiler stages, and a description of the tools, tests, and general software development attitude used.

2.1 Requirements Analysis

The requirements of the project were relatively simple: the primary goal was to produce a working compiler for a small subset of Haskell, then add optimisations to meet the success criteria. There were no requirements for interacting with other systems such as a build system. It was desirable to have a comprehensive test suite to ensure correctness, and this was planned in advance.

One issue discovered during requirements analysis was that implementing even a basic subset of Haskell required support for some features which were originally listed as extensions: 'simple' operations such as addition require support for functions, but also for typeclasses and Algebraic Datatypes (ADTs). This caused a reorganisation of the planned schedule during Michaelmas term.

Specifics of the software development processes used are described in Section 2.6, and the following sections on testability and tooling.

2.2 Concepts

There are a number of key concepts that I needed to learn about in order to design and implement various stages in the compiler: these are detailed in the following subsections.

2.2.1 Kinds

Kinds are an essential concept for the type system F_Ω used by Haskell, as described by B. Pierce[8]. System F_Ω introduces type constructors, terms in the 'type language' which act as 'functions' between types. `Int` is a traditional type as seen in the polymorphic and simply-typed lambda calculus, but `Maybe Int` is less traditional: `Maybe` is parameterised by a type. Kinds provide a language with which to describe these type constructors, and more importantly check that they are well-formed. After all, `Int Bool` makes no sense as a type, just as the term `1 True` makes no sense as an term.

A Kind is often described as the 'type of a type': we can say that `True :: Bool`, but looking to a type system 'one level up' we can say `Bool :: *` where `*` is the 'type' of a parameterless 'type constructor'. The type constructor `Maybe` has kind `* -> *`, as it takes a single type parameter: applying it to a type of kind `*` yields a type of kind `*`, such as `Maybe Int`, while applying it to a type with a different kind such as `Maybe Maybe` produces an invalid type.

Data and class declarations define type constructors, and the kinds of any parameters can be inferred by their usage. Consider that in the declaration `data Maybe a = Left | Right a`, we can see that `Maybe :: ` α `-> *` as it has one parameter (`a`, with as-yet unknown kind α), and from the data constructor branch `Right a` we can infer that $\alpha = *$, resulting in `Maybe :: * -> *`. On the other hand, `data A f a = A (f a)` defines a type constructor `A :: (* -> *) -> *` and `data Bool = True | False` simply defines `Bool :: *`.

All typeable terms in Haskell have a type with kind `*`: no values exist for types of other kinds. For example, `(1 :: Int) :: *`, but there are no values of type `Maybe`, only values of `Maybe Int` or `Maybe Bool`, and so on.

The grammar and rules defining kinds sufficient for this compiler are as follows:

$$\text{Kinds } K ::= * \mid K \to K \qquad \frac{\vdash T_1 :: K_1 \to K_2 \quad \vdash T_2 :: K_1}{\vdash T_1\, T_2 :: K_2}$$

The rule is identical to the function application rule from the simply typed lambda calculus, but defined on types instead of terms: this is the type system 'one level up' (more advanced systems for kinds include more rules). Checks for type well-formedness include a check that any type applications result in types of appropriate kinds. In particular, any type inferred for a value must have kind `*`.

2.2.2 Weak Head Normal Form

In a deterministic call-by-value language, `(1+2+3, \x -> x && True)` would evaluate to `(6, \x -> x && True)`. Haskell instead requires that terms are evaluated up to weak head normal form (WHNF), in which terms are evaluated to their 'head', and subexpressions need not have been evaluated. Haskell specifically defines the 'head' to be a literal, a fully or partially applied data constructor, or a partially applied function. Arguments to data constructors/functions are subexpressions, so need not have been evaluated.

In WHNF		Not in WHNF
1	`1`	1	`1 + 2`
2	`(+) 1`	2	`(\x -> x) 1`
3	`Just True`	3	`(1 + 2) + 3`
4	`(+) (1 + 2)`		
5	`x = 1:x`		

Evaluation of an expression up to WHNF corresponds to a form of **non-strict evaluation**: partial applications of functions or any data constructor applications do not force their arguments to be evaluated, but when a function is applied to all its arguments, it reduces to the body without necessarily having evaluated its arguments. In particular, the evaluation of a Haskell program is equivalent to its reduction to WHNF.

```
1      (\x y -> (+) x) (1 + 2) 3
2   →  (+) (1 + 2)
3   ↛
```

Figure 2.1: An example of evaluation to WHNF, demonstrating laziness.

This is an important concept for the code generation stage, which must generate code that implements evaluation to WHNF.

2.2.3 Administrative Normal Form

Administrative Normal Form (ANF, presented by Flanagan et al. [6]) is a style of writing programs in which all arguments to functions are trivial (a variable, literal, or other irreducible 'value' like a lambda expression). ANF is an alternative to Continuation Passing Style (CPS) as a style of intermediate language that is often seen as being simpler to manipulate.

The expression `f x (1 + 2)` in ANF would be `let y = 1 + 2 in f x y`.

ANF is quite convenient for conceptualising the 'thunk' implementation of lazy evaluation, where expressions are represented as (possibly shared) units of suspended computation: any variable that binds a non-trivial expression acts as the name of the thunk representing that expression, and function arguments now pass around references to expressions.

As part of the lowering process, the intermediate languages are converted into ANF as it makes generating thunk-based code quite intuitive.

2.2.4 Typeclasses

Typeclasses are a language feature used to provide statically typed ad-hoc polymorphism (overloading). The general usage of typeclasses is encapsulated by the following example:

```
1   class Functor f where
2       fmap :: (a -> b) -> f a -> f b
3   instance Functor Maybe where
4       -- fmap :: (a -> b) -> Maybe a -> Maybe b
5       fmap _ Nothing = Nothing
6       fmap f (Just x) = Just (f x)
7   instance Functor [a] where
8       -- fmap :: (a -> b) -> [a] -> [b]
9       fmap _ [] = []
10      fmap f (x:xs) = f x:fmap f xs
```

A typeclass is similar to an 'interface' in object-oriented languages: it defines a set of functions that must be implemented by all instances of the class. In the example above, the `Functor` class defines the `fmap` function and specifies its type, parameterised by a type variable `f`. The instances of the class must provide implementations of this function where `f` is replaced by the type being made an instance.

Functions of a class can be used on any instance of that class, such as in the expression

`fmap (*2) [1,2,3]` which evaluates to `[2,4,6]`, or `fmap (fmap (*2)) [Just 1, Nothing]` which evaluates to `[Just 2, Nothing]`.

Typeclasses are one of the most core features of Haskell: comparing values can be done using the functions provided by the `Eq` and `Ord` typeclasses, printing and reading values makes use of the `Show` and `Read` typeclasses, and all numeric types are instances of the `Num` typeclass.

The most commonly used implementation of typeclasses is dictionary passing, which is described in the section on Deoverloading (3.3.3).

2.2.5   Summary

To briefly summarise the concepts listed above:

- A Kind is the type of a type, usually used to describe type constuctors: `Maybe :: * -> *`.

- Weak head normal form corresponds to non-strict evaluation, by only evaluating to the outermost term.

- Administrative normal form intuitively corresponds to thunks by only allowing trivial arguments to be passed to functions.

- Typeclasses provide statically typed overloading, implemented using dictionary passing.

2.3   Compiler Structure: Big Picture

The general structure of HJC is standard: the specific components within each stage are discussed within the implementation section, but a general overview is useful for context.

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[3]) 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`).

> Dependency analysis computes a directed graph representing syntactic dependencies between the source declarations so that the typechecker can process them in dependency 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 removes typeclasses from the type-level by implementing them as datatypes 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 into more efficient forms (with respect to runtime performance or generated code size) while preserving their semantics.

Code Generation

> ILB is transformed into Java Bytecode (JVB), and a modified version of an existing library[3] 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.

2.4   Testability

Given the number of stages and the scale of the project, tests are important to ensure that each component has the intended behaviour.

Haskell is a good language for writing testable code in: pure functions are usually easier to unit-test than impure functions as their behaviour is only affected by the parameters, independent of any global mutable state. The pipeline of HJC is entirely pure, with impure code only for reading the source file and writing the compiled files. This made testing each stage reliable and strictly independent of the adjacent pipeline sections.

Regression tests were implemented for all major bugs discovered, and ensure that the compiler stages do not reintroduce incorrect behaviour.

Finally, end-to-end tests ensure that HJC successfully processes a given Haskell source file and that the executable produced computes the correct result, treating the compiler as a black box. This extremely coarse testing method was very effective for discovering the existence of bugs, which could then be tracked down using standard debugging techniques and isolated using the finer-grained unit and regression tests.

2.5   Tools

I chose to write HJC in Haskell as it has a number of desirable features for large projects: purity ensures that components of the project cannot interact in unexpected ways, and the static type system guarantees that modifications are checked for a shallow (type-level) degree of correctness across the entire system.

The natural choice of compiler for Haskell is the industry-leading Glasgow Haskell Compiler (GHC), and the Stack build system is also relatively uncontested for Haskell build tooling, ensuring reproducible builds through a strict dependency versioning system.

Documentation has been written using Haddock, a tool that generates documentation from the code and comments: this documentation is rebuilt on every successful build and provides an easily-navigable description of commented modules and functions.

Git was used for version control, allowing me to develop features on distinct branches, use bisection to find the commits which introduced bugs, and keep a remote repository of code on Github as a backup.

Continuous integration was performed using Travis CI, which ensures that tests are run on every pushed commit and that builds are reproducible: the project can be built and run on different machines.

The benchmarking framework was written in Python 3, and plots generated using `matplotlib`. The Java Microbenchmark Harness (JMH) was used for gathering runtime statistics about the performance of output programs.

2.6 Software Development Model

I mainly used the waterfall development model, building each stage in the pipeline sequentially and testing it both in isolation and in sequence with the previous stages. The only stage which broke this model was type inference, which required multiple refining iterations to properly implement.

This approach was effective for most stages as there was a well-defined set of unchanging requirements. When the approach failed to work it was due to an incomplete set of requirements, so an iterative approach was more suitable to introduce support for the new requirements.

2.7 Starting Point

HJC uses a number of open-source packages from the de-facto Haskell standard library, such as `containers`, `text`, `mtl`, ... The full list is available in the `packages.yaml` configuration file in the root of the code repository.

The `haskell-src` lexing/parsing library for Haskell 98 source code was used, although forked and modified minorly (187++, 68--). The bytecode assembly library `hs-java` was forked and significantly modified and extended to meet the requirements of this project (1,772++, 1,431--).

No new languages had to be learnt for this project: I was already familiar with Haskell, Python 3, and Java. I had not worked with `haskell-src`, `hs-java`, `matplotlib`, or JMH before.

Chapter 3

Implementation
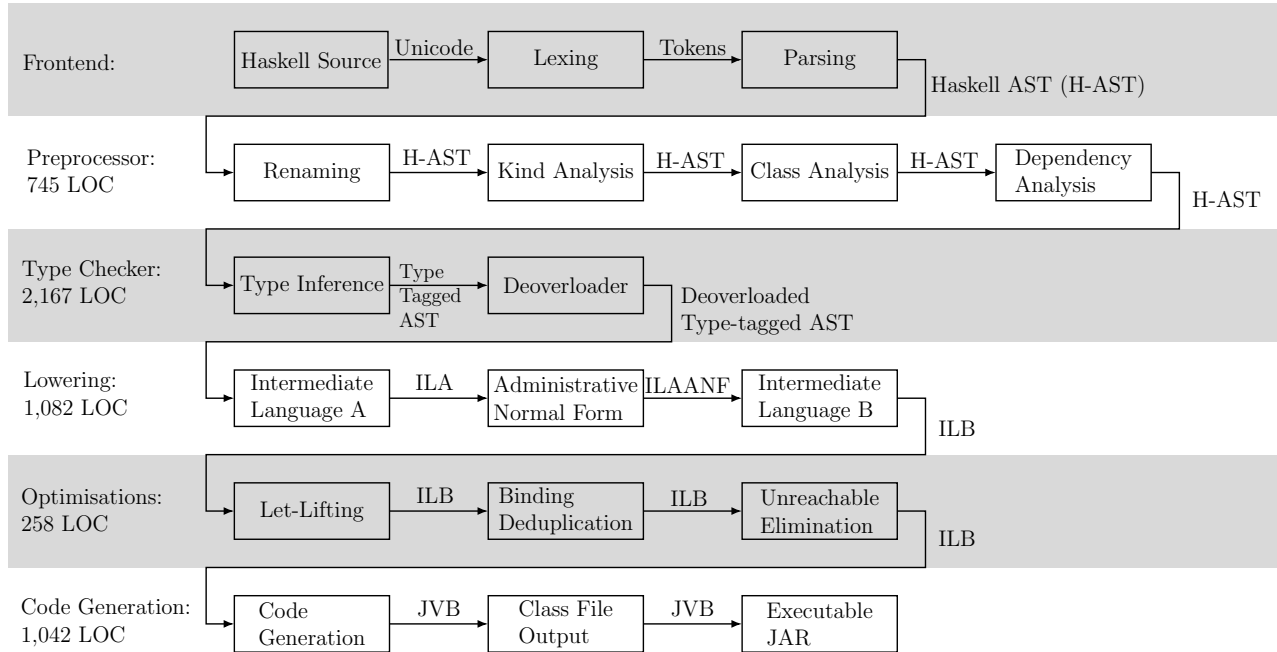
Figure 3.1

HJC consists of a number of stages and substages, as shown in Figure 3.1. A brief overview of each stage was given in the preparation chapter (Section 2.3), but this chapter presents more in-depth descriptions. A summary is given at the end of the chapter, along with an overview of the code repository structure.

3.1 Frontend

Lexing and parsing of Haskell source is performed using the `haskell-src`[1] library, which I modified to provide additional desirable features.

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

---

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

```haskell
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

3.2 Preprocessor

The preprocessing passes either make the Haskell source easier to deal with by later passes, or extract useful information to prevent subsequent passes from needing to.

3.2.1 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. 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:

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

Additionally, variables and type variables are in different namespaces: 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), despite the same name being used for a type variable and a variable:

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

To eliminate the potential for subtle bugs stemming from this feature, the renamer pass gives each distinct variable/type variable in the source a unique name (in the above example, the variable `x` might be renamed to `v0` and the type variable renamed to `tv0`, provided those names have not already been used).

> I removed the implementation explanation as it's pretty obvious and was an easy way to save words. Does it still read okay?

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, so renaming them would introduce unnecessary complexity.

3.2.2 Kind/Class Analysis

The typechecker and deoverloader require information about the kinds of any type constructors and the methods provided by different classes. This is tricky to compute during typechecking as those passes traverse the AST in dependency order. Instead, we just perform a traversal of the AST early in the pipeline to aggregate the required information.

3.2.3 Dependency Analysis

When typechecking, the order of processing declarations matters: we cannot infer the type of `foo` = `bar baz` until we have inferred the types of `1,431--` 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, with an edge from A to B if any bound variables/type variables/type constants at A are free in B. It is important to distinguish between variables/type variables and type constants, as otherwise name conflicts could occur (as type constants are not renamed). This separation is upheld in code by using different types for each, and is represented in the dependency graph in Figure 3.3 by colouring variables red and constants blue.

The strongly connected components (SCCs) 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. I use an existing Haskell library to compute the SCCs (`containers`[2]), which uses the algorithm presented by M. Sharir[9].

For example, from the dependency graph in Figure 3.3 we know that: we need to typecheck d_3, d_4, and d_5 together as they are contained within the same strongly-connected component so are mutually recursive; we have to typecheck d_2 last, after both other components. Typechecking declarations within the same component can proceed in an arbitrary order.

Special cases due to Typeclasses

This process given above 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, the multiple declarations of (`+`) in Figure 3.4a do not conflict, while `x` in Figure 3.4b does conflict.

---

[2]`https://hackage.haskell.org/package/containers`

```
data Bool = False | True          d₁
x = f True                        d₂
f y = g y                         d₃
g y = h y                         d₄
h y = f y                         d₅
```
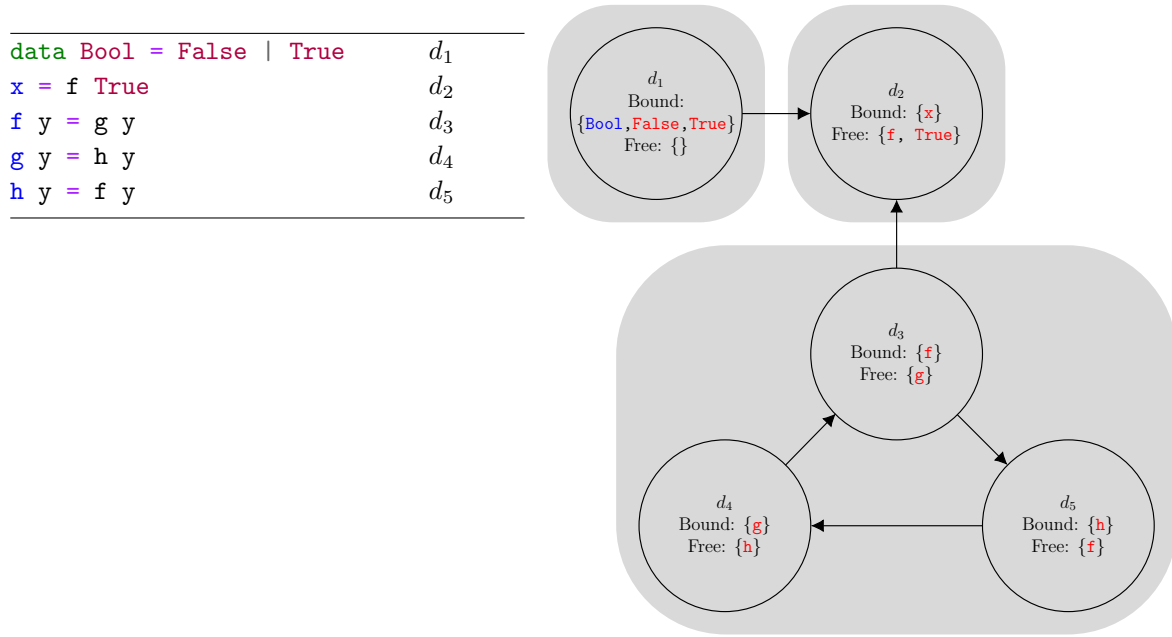

Figure 3.3: Code labelled with declaration numbers, and the corresponding dependency graph. Variables are in red text, type constants in blue. Strongly connected components are highlighted.
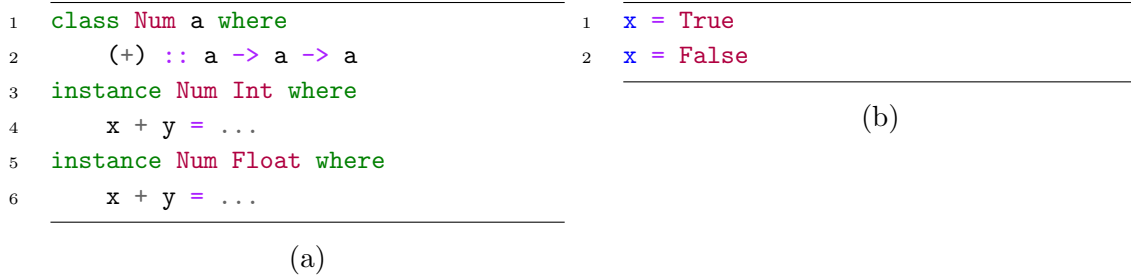
```
1   class Num a where              1   x = True
2       (+) :: a -> a -> a         2   x = False
3   instance Num Int where
4       x + y = ...                            (b)
5   instance Num Float where
6       x + y = ...
```

(a)

Figure 3.4

These declaration conflicts can be expressed as a binary symmetric predicate on declarations, as presented in Figure 3.5, where:

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

- ClassSym x c and ClassType x c represent Sym x and Type x inside the declaration for a class c, like class c where { x = True ; x :: Bool }.

- InstSym x c t represents a Sym x inside the declaration for a class instance c t, like instance c t where { x = True }.

Using this table we can see that the multiple declarations for (+) in Figure 3.4a are InstSym (+) Num Int and InstSym (+) Num Float so do not conflict, while the declarations for x in Figure 3.4b are both Sym x so do conflict.

- The dependencies generated by this technique are *syntactic*, not *semantic*: this is a subtle but important difference. The use of any ad-hoc overloaded variable generates dependencies on the class declaration that introduced the variable, but not the specific instance of the class that provides the definition of the variable used.

	Sym x_1	Type x_1	ClassSym x_1 c_1	ClassType x_1 c_1	InstSym x_1 c_1 t_1
Sym x_2	$x_1 = x_2$	False	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
Type x_2		$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
ClassSym x_2 c_2			$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
ClassType x_2 c_2				$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
InstSym x_2 c_2 t_2					$x_1 = x_2 \wedge (c_1 \neq c_2 \vee t_1 = t_2)$

Figure 3.5: The declaration conflict relation: entries below the diagonal are omitted as the predicate is symmetric.

```
1   class Foo a where
2       foo :: a -> Bool
3   instance Foo Bool where
4       foo x = x
5   instance Foo [Bool] where
6       foo xs = all foo xs
```

Figure 3.6: The declaration of `foo` in `instance Foo [Bool]` semantically depends on the specific overload of `foo` defined in `instance Foo Bool`, and yet no dependency will be generated between the two instances as neither declaration binds `foo`.

> Computing the true semantic dependency graph is too complicated to be done in this pass, so the problem is left and instead solved during the typechecking stage (Section 3.3.2).

To briefly summarise dependency analysis:

- Dependency analysis is required by typechecking in order to process declarations in the right order.

- The ordering between strongly connected components of the dependency graph corresponds to the order in which to process declarations, to ensure that all dependent declarations have been processed already.

  The components themselves correspond to mutually recursive functions that need to be typechecked 'together'.

- Typeclasses introduce some interesting special cases to the otherwise intuitive process.

3.3   Type Checker

Type inference and checking is the most complex part of the compiler pipeline. The type system implemented is approximately System F_ω (the polymorphically typed lambda calculus with type constructors) along with algebraic data types, and type classes to provide ad-hoc overloading. The approximation is due to a number of restrictions made by the Haskell Report to ensure that type inference is decidable.

This is a subset of the type system used by GHC (System F_C), which provides extensions such as GADTs and type families requiring a more complex type system.

3.3.1   Definition of Types

```
1  data TypeVariable = TypeVariable TypeVariableName Kind
2  data TypeConstant = TypeConstant TypeVariableName Kind
3
4  data Type = TypeVar TypeVariable
5            | TypeCon TypeConstant
6            | TypeApp Type Type Kind
7
8  data Kind = KindStar
9            | KindFun Kind Kind
```

Type variables have an associated kind[3] to allow for type constraints such as `pure :: Functor` $f \Rightarrow \alpha \to f\alpha$, in which `f` has kind $* \to *$. Function types $(A \to B)$ are represented as applications of the $\to$ type constructor, as it simplifies processing logic.

```
1  data TypePredicate = IsInstance ClassName Type
2
3  data Qualified a = Qualified (Set TypePredicate) a
4  type QualifiedType = Qualified Type
5
6  data Quantified a = Quantified (Set TypeVariable) a
7  type QuantifiedType = Quantified QualifiedType
8  type QuantifiedSimpleType = Quantified Type
```

A qualified/overloaded type is a simple type with type constraints/predicates attached, such as `Eq a => a -> a -> Bool` (the type constraints here being just `{Eq a}`). The type constraints act as restrictions on the valid types that can fulfil the type variable, or equivalently predicates which must hold on the variables: the type signature is only valid for `a` that are instances of the `Eq` typeclass.

A quantified/polymorphic type is a type with a set of quantifying type variables which must be instantiated before use: `forall a. Eq a => a -> a -> Bool`. Haskell type signatures are implicitly quantified over all the free type variables.

During type inference, types are almost always polymorphic and often overloaded (`(==) :: forall a. Eq a => a -> a -> Bool`, `head :: forall a. [a] -> a, ...`). After deoverloading (Section 3.3.3), types are never overloaded. This difference is enforced in code by using `QuantifiedType` and `QuantifiedSimpleType` respectively.

3.3.2   Type Inference

The implementation is inspired by the approach given by Mark P. Jones[7] and uses similar rules as the Hindley-Milner (HM) type inference algorithm. There are three passes over the source AST, each of which traverses the SCCs of the AST in dependency order as previously described in Section 3.2.3.

1. The first pass tags each subexpression of the declarations in the SCC with a type variable,

---

[3]Kinds were described in the Preparation chapter, Section 2.2.1.

then uses rules similar to the HM inference rules to infer the value of each type variable from subterm type variables.

Overloaded functions present a difference from the HM rules, as some expressions generate typeclass constraints on the type variables involved: using an overloaded function like (+) will first require instantiating its polymorphic type to an overloaded type (`forall a. Num a => a -> a -> a` to just `Num b => b -> b -> b`, where `b` is a fresh unique type variable), then moving the constraints from the type into a set of constraints built up while traversing this declaration to get just `b -> b -> b`. This is the ground type that is unified with the type variable used to tag the use of (+), and the type constraints are stored for use after finishing traversing the SCC.

After all declarations in an SCC have been fully traversed, types are generated for all the variable names bound. This involves adding explicit quantifiers and constraints to the simple type inferred for the top-level expression on the right-hand side of the binding. All type variables within the simple type that are not already in scope from eg. a class definition are added as universally quantified variables, and any constraints generated during processing the declaration involving type variables free in the simple type, are added as the qualifiers to the type.

2. The second pass simply traverses the AST again and updates the tag type variables with the final expression type generated by the unification during the first pass.

   This cannot be done efficiently during the first pass as the final type is only known after fully traversing each SCC.

3. The third pass checks that any user-provided type signatures are valid compared to what was actually inferred: if the user provided a more general tag than the inferred tag, we reject the program. This is distinct from the 3rd pass to maintain clarity.

An improvement to the current approach would be to implement the OutsideIn(X) framework given by Vytiniotis et al. [10]. This framework can work with Hindley-Milner type inference to handle more complex constraints than the current implementation, allowing support for GADTs and type families.

These three passes detail the inner workings, but there are still a number of tricky edge-cases to navigate:

Restricted Polymorphism

One departure from theoretical polymorphic type systems is that Haskell's type system restricts polymorphism for some terms to ensure that type inference remains decidable. Let-bound variables are polymorphic over all their free type variables, while function parameters are never polymorphic:

```
1  let f x = x in const (f True) (f 1) :: Bool  -- This is fine
2  (\g -> const (g True) (g 1)) (\x -> x)        -- This fails to typecheck
```

Typeclass Order

A tricky part of the typechecking process is dealing with typeclasses, as dependency order is not semantically accurate for typeclass instance declarations (mentioned in Section 3.2.3).

To handle potential dependency issues, such declarations are processed in a lazy manner. If a declaration requires an instance of a typeclass in order to typecheck then that typeclass is typechecked immediately, and all remaining instance declarations are processed after non-instance declarations have been processed.

The Ambiguity Check

There is a type-system correctness check called the 'ambiguity check', which ensures that at the use site of any variable with overloaded type has all type variables either resolved to a ground type or exposed in the type of the enclosing expression. This prevents programs which have ambiguous semantics (multiple possible runtime behaviours) from being reported as valid (this issue is described in Section 11.5.1 of 'Typing Haskell in Haskell' [7]).

HJC does not implement this check, which causes some programs to fail compilation during the code generation stage instead of during typechecking. This feature was omitted in order to spend development time on the later stages of the compiler.

Defaulting[4] is a user-friendly defence against the verbosity of giving unambiguous types. This feature was not implemented as the time required was expected to be better spent elsewhere.

Summary

Having completed the three passes of type inference and checking, and handling the edge cases associated with typeclasses, the output of the type inference component is a Haskell AST where expressions are tagged with their (possibly overloaded) types. This forms the input for the final component of the pipeline prior to lowering into intermediate languages.

3.3.3 Deoverloader

The deoverloading stage performs a translation which implements typeclasses using a technique named dictionary passing. This produces an AST tagged with types that no longer have type contexts.

3.3.3.1 Dictionary Passing

The expression `1 + 2` has type `Num a => a`, where the expression is 'overloaded' on the type of `a` (as `(+)` is overloaded). This overloading can be implemented by adding extra parameters ('dictionaries') to all overloaded functions, which provide the implementation of any overloaded operators such as `(+)`. This is similar to virtual method tables in object oriented languages,

---

[4]`https://www.haskell.org/onlinereport/decls.html#sect4.3.4`

except the v-table is being passed as an extrinsic argument, rather than being intrinsic to the object being operated on.

Dictionary passing is implemented using a single source-to-source transformation on the Haskell AST:

1. Each typeclass is replaced by a datatype which acts as the **type** of the v-tables for that class.

   Further, each function defined by a typeclass is made into a top-level function that extracts the implementation function from the passed dictionary (equivalent to extracting a function pointer entry from the v-table).

```
1  class Eq a where
2      (==) :: a -> a -> Bool
3      (/=) :: a -> a -> Bool
```

```
1  data Eq a = Eq (a -> a -> Bool) (a -> a -> Bool)
2
3  (==), (/=) :: Eq a -> a -> a -> Bool
4  (==) (Eq eq _) = eq
5  (/=) (Eq _ neq) = neq
```

2. Each instance declaration is replaced by a value of the corresponding class' datatype, providing an implementation of a v-table that can be passed around. Each of these values is a dictionary that can be passed to an overloaded function.

```
1  instance Eq Int where
2      (==) = foo
3      (/=) = bar
```

```
1  dEqInt :: Eq Int
2  dEqInt = Eq foo bar
```

3. Any use-site of a symbol with an overloaded type needs to be passed the relevant dictionary, and any definition of a symbol with an overloaded type needs to be extended with parameters for the dictionaries. Quite pleasingly, this replaces the => symbol with ->.

```
1  sum :: Num a => [a] -> a
2  sum xs = foldl (+) 0 xs
3
4  x :: Num a => a
5  x = sum [1,2,3,4]
```

```
1  sum :: Num a -> [a] -> a
2  sum dNumA xs = foldl ((+) dNumA) 0 xs
3
4  x :: Num a -> a
5  x dNumA = sum dNumA [1,2,3,4]
```

This approach can be easily extended to typeclasses with 'superclasses' (by storing superclass dictionaries inside subclass dictionaries) and instance declarations with contexts (by making functions that construct dictionaries from other dictionaries), such as those shown in Figure 3.10.

HJC does not support these uses as the development focus at this stage was on reaching a minimal working subset of Haskell. However, the existing implementation demonstrates that support is definitely achievable.

```
1   class Functor m => Monad m where
2          ...
```

```
1   instance Eq a => Eq (Maybe a) where
2        Just x == Just y = x == y
3        Nothing == Nothing = True
4        _ == _ = False
```

Figure 3.10: Typeclass contexts allow for expressing the semantics that 'in order for `m` to be a `Monad`, it must also be a `Functor`'. Instance contexts allow for the semantics 'for any `a` that is an instance of `Eq`, `Maybe a` is also an instance of `Eq`.'

3.4 Lowering and Intermediate Languages

There are two intermediate languages within HJC , imaginatively named Intermediate Languages A and B (ILA and ILB respectively). There is also a minor language named ILA-Administrative Normal Form (ILA-ANF), which is simply a subset of ILA that helps restrict the terms to those in Administrative Normal Form (ANF).

3.4.1 Intermediate Language A

ILA is a subset of GHC's Core IL, removing terms which are used for advanced language features like GADTs. This language is still high level: but while Haskell 98 has hundreds of node types in its AST[5], ILA has far fewer, making it far easier to transform.

3.4.1.1 Definition

```
1   data Expr = Var VariableName Type
2           | Con VariableName Type
3           | Lit Literal Type
4           | App Expr Expr
5           | Lam VariableName Type Expr
6           | Let VariableName Type Expr Expr
7           | Case Expr [Alt Expr]
8
9   data Literal = LiteralInt Integer
10             | LiteralChar Char
11
12  data Alt a = Alt AltConstructor a
13
14  data AltConstructor = DataCon VariableName [VariableName]
15                    | Default
16
17  data Binding a = NonRec VariableName a
18               | Rec (Map VariableName a)
```

This pass lowers a Haskell program into a list of `Binding Expr`: recursive or non-recursive bindings of expressions to variables. Most of these constructors have familiar usages, but some are more subtle:

---

[5]https://hackage.haskell.org/package/haskell-src/docs/Language-Haskell-Syntax.html

- `Con` represents a data constructor such as `True` or `Just`.

- `App` is application as expected, but covers both function applications and data constructor applications (eg. `App (Con "Just" (Int → Maybe Int)) (Var "x" Int)`).

- `Case` *e as* represents a multi-way switch on the value of an expression *e* (the 'head' or 'scrutinee'), matching against a number of possible matches ('alts') from the list *as*. The body *b* of the first alt `Alt` *c b* whose constructor *c* matches the value of the scrutinee is executed.

One notable feature of ILA is that it carries type information, which is necessary for code generation but also allows for sanity checks: this is inspired by GHC's Core IL, which is fully explicitly typed under a variant of System F, allowing for 'core linting' passes in-between transformations to ensure they maintain type-correctness.

3.4.1.2 Pattern Matching

Many syntax features in Haskell are syntactic sugar, and are simple to desugar (eg. `[1, 2]` into `1:(2:[])`, or `if x then y else z` into `case x of { True -> y ; False -> z }`).

Other language features are non-trivial to lower, such as the rich syntax Haskell uses for pattern matching. Figure 3.11 demonstrates two uses of patterns.
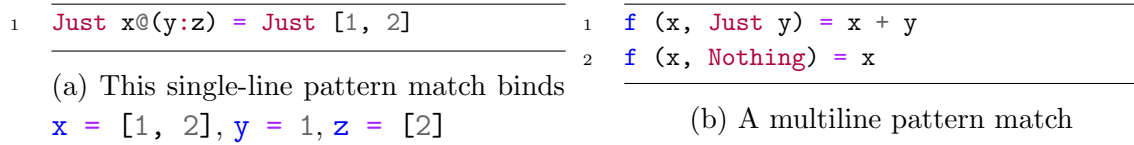
1 `Just x@(y:z) = Just [1, 2]`	1 `f (x, Just y) = x + y`
	2 `f (x, Nothing) = x`

(a) This single-line pattern match binds `x = [1, 2]`, `y = 1`, `z = [2]`

(b) A multiline pattern match

Figure 3.11

My initial implementation worked correctly for 'horizontal' patterns such as in Figure 3.11a, but did not support patterns stacked 'vertically' as in Figure 3.11b.

A simplified declarative version of the implementation is presented below as a function translating terms from the Haskell grammar (either patterns or variables) to terms of ILA. It demonstrates the translation of a few of the more interesting Haskell patterns. Syntactic terms of ILA are presented in monospaced font and meta-variables used to represent Haskell terms are given in script font:

$$\text{translate}(v : vs, v' : ps, b) = \text{translate}(vs, ps, b)[v/v']$$

$$\text{translate}(v : vs, (con\ ps') : ps, b) = \texttt{case}\ v\ \texttt{of}\ \{\ con\ vs'\ \texttt{->}\ \text{translate}(vs' +\!\!+ vs, ps' +\!\!+ ps, b)\ \}$$
$$\text{where } vs' \text{ are fresh variables}$$

$$\text{translate}(v : vs, (w\ @\ z) : ps, b) = \text{translate}(v : vs, z : ps, b)[v/w]$$

$$\text{translate}([], [], b) = b$$

In translate(vs, ps, b), vs is a stack of Haskell variables, ps is a stack of Haskell patterns to match the variables against, and b is a Haskell expression for the right-hand-side of the pattern match. The result is a recursively constructed ILA term that decomposes the variables in vs

using the patterns in *ps*. For example, the expression `case x of { y@(Just z) -> (y, z) }` would be translated as:

$$\text{translate}([\texttt{x}], [\texttt{y@(Just z)}], \texttt{(y, z)})$$
$$\rightarrow \text{translate}([\texttt{x}], [\texttt{Just z}], \texttt{(y, z)})[\texttt{x/y}]$$
$$\rightarrow (\texttt{case x of \{ Just u -> } \text{translate}([\texttt{u}], [\texttt{z}], \texttt{(y, z)}) \texttt{ \}})[\texttt{x/y}]$$
$$\rightarrow (\texttt{case x of \{ Just u -> } \text{translate}([], [], \texttt{(y, z)})[\texttt{u/z}] \texttt{ \}})[\texttt{x/y}]$$
$$\rightarrow (\texttt{case x of \{ Just u -> (y, z)}[\texttt{u/z}] \texttt{ \}})[\texttt{x/y}]$$
$$\rightarrow \texttt{case x of \{ Just u -> (x, u) \}}$$

However, as mentioned earlier, this implementation only worked for 'horizontal' patterns. The current implementation is now based off the approach given in Chapter 5 of 'The Implementation of Functional Programming Languages' [11], which is a more general version of my approach: the stack of patterns and the single body is replaced by a stack of 'rows', each of which is a stack of patterns along with a body representing the right-hand-side of the pattern match. The head patterns from each row are grouped into data constructors, variables or literals, and the recursion occurs on each group.

3.4.1.3 Literals are not patterns

In Haskell a pattern will eventually match against either a data constructor, a literal value, or 'anything' (with the wildcard pattern `_`). However, in the grammar for ILA's `AltConstructor`, we're missing a constructor for literals. This is due to restrictions during code generation: we can only perform `case` expressions on ADTs.

Pattern matches using literals are instead desugared into equality checks: the Haskell expression `case x of { 0 -> y ; 1 -> z ; _ -> w }` is effectively translated into `if x == 0 then y else if x == 1 then z else w`.

3.4.2 Intermediate Language A - Administrative Normal Form

ILA-ANF is a subset of ILA which uses a more restricted grammar to enforce more invariants on the language and guide the AST into Administrative Normal Form (as described in Section 2.2.3). The full definition of ILA-ANF is given below, and reuses the definitions of `Binding` and `Alt` from ILA.

In the case of ILA-ANF, 'trivial' terms are taken to be variables, data constructors, and literals. Note that this excludes lambda terms, as they are decidedly not trivial from the perspective of code generation. Instead, lambda terms must immediately be bound to a variable: this restriction is enforced by the `AnfRhs` term in the grammar below.

```
1  data AnfTrivial = Var VariableName Type
2                  | Con VariableName Type
3                  | Lit Literal Type
4
5  data AnfApplication = App AnfApplication AnfTrivial
6                      | TrivApp AnfTrivial
```

```
7
8   data AnfComplex = Let VariableName Type AnfRhs AnfComplex
9                  | Case AnfComplex Type [Alt AnfComplex]
10                 | CompApp AnfApplication
11                 | Trivial AnfTrivial
12
13  data AnfRhs = Lam VariableName Type AnfRhs
14             | Complex AnfComplex
```

3.4.3 Intermediate Language B

ILB is the final intermediate language of this compiler. It maintains a number of useful properties for code generation: the only term that evaluation an expression is the `ExpCase` e t as term, and the only term which performs any memory allocation is the `ExpLet` v r e term.

Additionally, this language makes lazy evaluation 'explicit', in the sense that expressions are always suspended within an `RhsClosure`.

```
1   data Arg = ArgLit Literal
2           | ArgVar VariableName
3
4   data Exp = ExpLit Literal
5           | ExpVar VariableName
6           | ExpApp VariableName [Arg]
7           | ExpConApp VariableName [Arg]
8           | ExpCase Exp Type [Alt Exp]
9           | ExpLet VariableName Rhs Exp
10
11  data Rhs = RhsClosure [VariableName] Exp
```

ILB is similar in grammar to ILA-ANF, and the translation pass is relatively simple. There are some key differences between the languages, that reflect the changes from a relatively high-level IL down to a lower-level one:

- There are now two terms for applications, one for functions (`ExpApp`) and one for data constructors (`ExpConApp`). The distinction is necessary for code generation, when a function application results in a jump to new executable code while a constructor application creates a new heap object.

- Right-hand side terms in ILA-ANF (`AnfRhs`) were either lambda expressions, or any other ILA-ANF expression – in ILB, the only right-hand side term is a `RhsClosure`. A closure with no arguments is a term that exists purely to delay computation of an expression.

- ILB only allows variables in many of the places where ILA-ANF allowed literals, variables, or 0-arity data constructors (like `True`). This is another step towards making laziness explicit, by keeping expressions simple so that only one step of the evaluation needs to happen at a time.

3.5 Code Generation

Code generation is, from the surface, quite a mechanically simple process. ILB is a small language, so there are few terms to lower into bytecode. However, implementing the semantics of these terms in Java Bytecode is complex.

The `hs-java` library was used to provide a Haskell representation of bytecode that could then be serialised to a Java `.class` file, but a number of modifications were made to add support for Java 8 features required by the project and improve performance: the forked project can be found at `https://github.com/hnefatl/hs-java`.

A number of Java classes have been written to provide the 'primitives' used by generated bytecode: including the implementation of Haskell's primitive datatypes like `Int` and `Char`. HJC is aware of these 'builtin' classes and uses a set of 'hooks' when generating code to provide Java implementations of Haskell functions. This is covered in more detail in Section 3.5.3.

3.5.1 Heap Objects

Literals, datatype values and closures are all represented at runtime by values on the heap, as they are all first-class values in Haskell, and will be referred to as 'objects': this intentionally overloads the terminology used by Java for an instance of a class, as the two concepts are essentially interchangeable here as Java objects are heap-allocated.

All objects on the heap inherit from a common abstract base class, `HeapObject`:

```java
public abstract class HeapObject implements Cloneable {
    public abstract HeapObject enter();

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

The abstract `enter` method evaluates the object to WHNF (described in Section 2.2.2) and returns a reference to the result. The `clone` method simply returns a shallow copy of the object: this method is critical for implementing function applications, described later.

3.5.1.1 Literals

Literals are builtin types such as `Int` that cannot be defined as an Haskell ADT. Any such type is implemented by a subclass of the `Data` class, which implements the trivial reduction to WHNF (all literals are already in WHNF).

```java
public abstract class Data extends HeapObject {
    @Override
    public HeapObject enter() {
        return this;
```

```
5        }
6    }
```

An example implementation of the `Integer` literal type using this class is given in Appendix A.

The classes implementing each literal provide their own type-specific initialisation functions, which are invoked at runtime depending on the type of the literal being constructed.

3.5.1.2 Datatypes

An Haskell ADT can be represented simply by a generated class which inherits from the `BoxedData` builtin abstract class:

```
1    public abstract class BoxedData extends Data {
2        public int branch;
3        public HeapObject[] data;
4    }
```

The `branch` field is used to identify which constructor of the type has been used, and the `data` field contains the arguments given to the data constructor. An example generated class[6] for the datatype `data Maybe a = Nothing | Just a` might be:

```
1    public class _Maybe extends BoxedData {
2        public _make_Nothing() {
3            _Maybe x = new _Maybe();
4            x.branch = 0;
5            x.data = new HeapObject[] {};
6            return x;
7        }
8        public _make_Just(HeapObject val) {
9            _Maybe x = new _Maybe();
10           x.branch = 1;
11           x.data = new HeapObject[] { val };
12           return x;
13       }
14   }
```

Note that as `BoxedData` inherits from `Data`, the `enter` method has the same simple implementation – as any data value is already in WHNF.

3.5.1.3 Closures

Closures are the most complicated objects stored on the heap. There are three main lifecycle stages of a closure:

- Creation: construction of a new closure representing a function of given arity, without any arguments applied yet but possibly including values of free variables in scope of the

---

[6]HJC does not generate a class described in Java source form, it just generates the bytecode for the class directly.

closure.

- Argument application: this may be a partial application or a total application, or even an over-application: consider `id` `(+1)` `5`, which evaluates to `6`. `id` has arity 1, but is applied to 2 arguments here.

- Evaluation: after a total application, reducing the function to its body (as specified by WHNF reduction).

These behaviours are provided by the `Function` builtin class, which is given in Appendix B.

A function f (either defined locally or at the top-level) in Haskell of arity n_a and using n_f free variables is translated into two Java functions:

- `_`f`Impl`, which takes two arrays of `HeapObject`s as arguments, one holding the arguments for the Haskell function (of length n_a) and one holding the free variables used by the Haskell function (of length n_f), returning a `HeapObject` representing the result of applying the function.

- `_make_`f, which takes n_f arguments representing the free variables of the Haskell function, and returns a Java `Function` object representing the closure, where the `inner` field points to the `_`f`Impl` function.

Haskell function applications are lowered into bytecode that:

1. Fetches the function, either by calling the appropriate `_make_` function with the free variables, or just loading a local variable if the function has already been partially applied and stored or passed as a function argument.

2. **Clones** the `Function` object. This step is subtle but vital, as each argument applied to the function mutates the `Function` object by storing additional arguments. Cloning the function maintains the same shared references to arguments and free variables, but creates new (non-shared) containers to hold them.

   This is a shallow clone – if we used a deep clone, recursively cloning the arguments and free variables, then we would lose the performance benefit of graph reduction where we can use an already computed value instead of recomputing it ourselves, and increase memory usage.

3. Invokes `True` on the cloned object for each argument in the application, storing them for later use.

4. Invokes `enter` on the function object. This will reduce the object to WHNF, which has three cases:

   - The function is partially applied, so has not yet received all of the necessary arguments to be evaluated. Such a function is already in WHNF, so we can just return it.

   - The function has exactly the right number of arguments, so WHNF demands we reduce it. This is implemented by calling the `inner` function that performs the actual implementation of the Haskell function with the free variables and arguments we have stored, then ensuring the result itself is in WHNF by calling `enter`, then returning it.

- The function is over-applied. Although this case looks complicated, it is really only two simple steps: We pretend we have an application of exactly the right number of arguments as in the above case, then instead of returning the result we cast it to a `Function` object and perform a normal function application with all the leftover arguments.

All of the functions defined in a Haskell program are compiled into their pairs of Java functions within a single class, the 'main' class. Datatypes are compiled into their own classes which are then referenced by the main class. This approach to function compilation differs from the approaches taken by Scala and Kotlin (other languages targeting the JVM), which compile lambda expressions into anonymous classes.

3.5.2   JVM Sanitisation

JVM Sanitisation is a name conversion process used in the code generator to prevent conflicts and invalid variable names when everything has been lowered into JVB.

Haskell[7], Java[8], and JVB[9] all allow different sets of strings as valid identifiers: for example, `Temp` is a valid variable name in Java and JVB, but not in Haskell (identifiers with uppercase Unicode start characters are reserved for constructor names like `True`). `(+)` is a valid identifier in Haskell and JVB, but not in Java.

Additionally, name conflicts can occur between builtin classes used by the compiler (eg. `Function`) and constructor names in the Haskell source (eg. `data Function = Function`).

- All names that have come from Haskell source are prefixed with an underscore, and builtin classes are restricted from starting with an underscore.

- Any non-alphanumeric (Unicode) characters in a Haskell source identifier are replaced by their Unicode codepoint in hexadecimal, flanked on either side by $ symbols. This prevents variables from using characters disallowed by JVB while ensuring unique decodability.

3.5.3   Hooks

Many standard functions operating on primitive types like `Int`, such as `(+)` and `(==)`, cannot be implemented in Haskell. These operations need to be implemented at the same level as `Int`, in bytecode. However, *some* form of definition has to be given in the Haskell source, so that the typechecker can infer that eg. the instance exists.

Hooks solve this problem by allowing for methods implemented in bytecode to be injected during code generation, making them callable just like any function compiled from Haskell source. For example, integer addition is defined as:

---

[7]`https://www.haskell.org/onlinereport/lexemes.html`
[8]`https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.8`
[9]`https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.2.2`

```
1  instance Num Int where
2      (+) = ...
```

Figure 3.12: We want to be able to write an instance declaration in order to allow typechecking to see that `Int` is an instance of `Num`, but we cannot provide an implementation for any of its member functions.

```
1  instance Num Int where
2      (+) = primNumIntAdd
3  primNumIntAdd :: Int -> Int -> Int
```

A hook is then added to the code generation stage that generates functions named `_makeprimNumIntAdd` and `_primNumIntAddImpl`, as described in Section 3.5.1.3. The implementation of `_primNumIntAddImpl` is provided in the hook definition, and simply forwards its arguments to the `_Int::add` function shown in Section 3.5.1.1. The functions generated by the hook are, at the bytecode level, indistinguishable from functions generated by any other expression so can be called by any other function without issue.

Note that hooks are baked into the code generation stage, so this feature does not behave like a Foreign Function Interface – however, it would not be hard to extend this system to allow for programmer-defined hooks.

3.6 Optimisations

Many JVM implementations perform low-level optimisations on the bytecode while just-in-time compiling it to native machine code. The Jikes JVM, as of 2006[10], performs method inlining, common subexpression elimination, copy/constant propagation, peephole passes, instruction reordering and loop unrolling, as well as adaptive optimisations such as selective optimisation to improve startup time.

This somewhat reduces the need to perform low-level optimisations within the compiler, so focus was given instead to higher-level optimisations that targeted specific weaknesses in the implementation of HJC .

3.6.1 Let-lifting

Let lifting aims to move let-bound variables and functions 'up' the AST, to as close to the top-level as possible while preserving the semantics. This can reduce the number of heap allocations and eliminate the need for initialisation code, saving on program size and execution time.

The example in Figure 3.13 is somewhat contrived, but demonstrates the intent of the optimisation: `z` only has to be allocated and evaluated once each program, and `x'` only once per evaluation of `f` rather than each evaluation of `g`.

---

[10]http://www.complang.tuwien.ac.at/andi/ACACES06.pdf

Lifting bindings as high as possible can reduce the amount of duplicated work done by some subexpressions. Lifting bindings to the top level is particularly desirable as they can be implemented as a globally-accessible thunk, which can be referenced immediately by other code without needing initialisation.

Before	After

```
1  f x = let
2      g y = let
3          z = 10
4          x' = x + z
5          y' = y + z
6          in x' + y'
7      in g
```

```
1  z = 10
2  f x = let
3      x' = x + z
4      g y = let y' = y + z in x' + y'
5      in g
```

Figure 3.13

Let lifting is an ILB to ILB transformation, as ILB exposes the most uses of let-bindings: almost every subexpression in a computation is let-bound to a variable as ILB requires that all function applications only use trivial arguments. This provides a lot of freedom to move bindings around.

To implement let-lifting we traverse each top-level binding, building up and processing a dependency graph. Nodes in the graph represent let-bindings that we plan to rearrange, so are identified by a binding of a variable and a RHS expression. Node A depends on node B when the variable bound by B occurs free in the RHS of A. Any topological ordering of this graph gives us the reverse of a syntactically valid 'nesting' of let-bindings – the first binding in the reversed order is the one that depends on no other variables, so should go at the root of the rearranged right-leaning tree, with the second binding in the order being the 'body' of that first (`let x = 1 in (let y = 2 in ...`). This ensures that all bindings are in scope of their use-sites.

There are minor special cases arising for let-bindings which bind other let-bindings to a variable, and for lambda expressions (which cannot be relocated: ILB ensures that lambda expressions are at the top level of a binding). The most important special case is for bindings which are still available to be rearranged when we finish traversing the top-level binding AST. These bindings can be lifted entirely out of their original declaration and into new top-level definitions.

3.6.2 Binding Deduplication

The binding deduplication pass eliminates any bindings where the right-hand side of the binding is syntactically identical to that of another in-scope binding, and replaces references to the eliminated binding with references to the existing one. This is a rather unusual optimisation to perform, but it has significantly positive effects on program size, and can reduce computation in some cases (if the duplicated expression is a long-running computation, then it will only be run once).

As the intermediate languages are mostly in ANF, there are a great number of let-bindings. In particular, simple expressions like `True && True` in Haskell are converted into `let v1 = True in let v2 = True in (&&) v1 v2` in ILB. This is necessary in general in

order to provide lazy evaluation, but in common simple cases like this it introduces obvious redundancy.

Binding deduplication removes the redundancy by transforming the ILB version into `let v1 = True in (&&) v1 v1`. This reduces code size and removes a heap allocation (although more advanced compilers would likely pre-allocate such constants).

This optimisation is an ILB to ILB transformation like let-lifting, for similar reasons. In addition, this transformation is performed after let-lifting as it produces better results than before: after lifting bindings as high as possible there are more in-scope bindings at any program point, which increases the potential for bindings to be removed by this pass. In the `True && True` example given above, we would naturally converge to only having a single binding of `True` in the program, rather than multiple, which saves memory (as all values are allocated on the heap) and time (the program need not initialise a new thunk containing that simple value before each use).

The implementation simply traverses each binding (both top-level bindings and let-bindings) in the program, computing the hash of the bound expression's AST: when two hashes match, as long as the bindings are within scope of each other, one of the bindings is eliminated and all uses of its bound variable replaced with a use of the alternative binding's variable.

3.6.3 Unreachable Code Elimination

The unreachable code elimination pass removes all code that cannot be reached on some execution path from the entrypoint of the program. Reachable code is determined syntactically, as semantic reachability is intractable in general.

This optimisation pass is extremely effective as it can eliminate any unused functions from the 'standard library' included with HJC , described in Section 3.7, ensuring that only code necessary for the execution of the program is compiled.

This optimisation is performed on ILB after binding deduplication as this leads to more opportunities for removing unreachable code: eliminating bindings during the deduplication optimisation pass can only reduce the number of references to each variable in the program.

The implementation performs two passes over the AST: the first pass records all variables used along any execution path starting from `main`; the second pass removes any bindings which do not define those variables.

3.7 Prelude

The Haskell Standard Library is named the Prelude, and consists of a number of builtin modules that provide definitions of useful functions, datatypes, and typeclasses such as `map`, `Maybe`, and `Monad`. HJC includes a standard library, which includes a subset of the definitions from the true Prelude.

HJC does not support Haskell modules, instead using a single file per program, so the 'import' approach taken is to simply concatenate the standard library (`StdLib.hs`) with the input file.

Dependency analysis ensures that the definitions are compiled in a valid order, and unreachable code elimination (Section 3.6.3) removes unnecessary definitions to prevent program bloat.

3.8 Summary

To summarise the key ideas in the implementation:

- Lexing and Parsing is handled by a lightly modified third-party library.

- Variables and Type Variables in the source code are renamed to unique names to prevent naming conflicts.

- Dependency Analysis is necessary for determining the order in which to infer types for declarations, to ensure that the types of mutually recursive functions are correct.

- Type inference is implemented similarly to the Hindley-Milner algorithm, but typeclasses introduce a number of complications.

- After typechecking, types include typeclass 'contexts', eg. `Eq a => a -> a -> Bool`. These are implemented using dictionary passing, introducing an extra parameter for each class constraint providing the 'implementation' of the typeclass for a type variable.

- Intermediate Language A's (ILA) primary design goal is to collapse the hundreds of types of syntax node in the Haskell source AST into a significantly smaller core of less than 10 nodes. Type information is preserved for use by lower stages and for sanity checks.

- ILA-Adminstrative Normal Form (ILAANF) is a trivial subset of ILA to help guide ILA into ANF, to simplify later transformations.

- In Intermediate Language B (ILB), the only syntax node which causes evaluation of any kind is `Case`, and the only node causing memory allocation is `id`.

- All Haskell values (literals, data values, functions) are represented as Java objects on the heap.

- Lazy evaluation is implemented by translating each Haskell expression into a function, in order to delay evaluation until the function is called.

- Three optimisations have been implemented: let-lifting raises let-bound variables to as high as possible within their expression; binding deduplication removes duplicate bindings in the same scope, to 'clean up' the redundant bindings which can be introduced by let-lifting; unreachable code elimination removes all code that cannot be reached from the program entrypoint, to reduce program size.

  All optimisations operate on ILB, as this is where they can have the greatest impact.

3.9 Repository Structure

The main code repository for this project is available at `https://github.com/hnefatl/dissertation-project` – all of the code in this repository was written from scratch. Appendix C highlights the important files.

3.9.1 `compiler`

The vast majority of code written was in a library named `compiler`, within the `src` directory, which exposes an API for compiling Haskell programs. This library contains 6344 lines of Haskell source code.

3.9.2 `compiler-exe`

The compiler executable is imaginatively named `compiler-exe`, compiled from a single 97-line source file `Main.hs` in the `app` directory: it contains the application entry point and code for handling command-line arguments, before passing control over to the `compiler` library for the actual compilation process.

3.9.3 Tests

Tests are contained in the `test` directory. The directory layout intentionally mimics the layout of the `src` directory, so the tests for each major compiler stage are stored in a similarly named path.

There are 1418 lines of Haskell source code for the tests.

3.9.4 Benchmarks

The code for evaluating performance is in the `benchmarks` directory, and consists primarily of Python 3 scripts for representing, compiling, and executing a variety of benchmark programs using a number of compilers, along with a script for plotting the data.

There are 821 lines of Python source code for the benchmarking framework.

3.9.5 `hs-java`

The `hs-java` library[3] is a library developed by Ilya V. Portnov for generating Java Bytecode. During my project, I made a number of modifications to the library, resulting in quite significant changes from the original. My fork of the library is available at `https://github.com/hnefatl/hs-java`, and has Git additions/deletions of 1,772++, 1,431--: the modified version of the library contains 3569 lines of Haskell source.

Chapter 4

Evaluation

4.1 Success Criteria

The success criteria laid out in the project proposal have all been satisfied:

- Translate simple Haskell programs into executable Java bytecode.

- Reject ill-formed programs due to syntactic or type errors.

- Perform simple optimisations during translation.

- Perform evaluation using non-strict semantics.

Beyond these base requirements, a number of the extensions implementing language features have been completed successfully.

4.2 Language Features

The main goal of the project was to support a reasonable subset of Haskell. The planned subset encompassed functions, arithmetic, booleans, lists, simple typeclasses, and laziness: the following program is compilable using HJC (it is even included as a test), and demonstrates various use-cases of all of these features:

```
1   -- foldl :: (b -> a -> b) -> b -> [a] -> b
2   foldl _ e [] = e
3   foldl f e (x:xs) = foldl f (f e x) xs
4
5   -- sum :: Num a => [a] -> a
6   sum = foldl (+) 0
7
8   -- take :: Int -> [a] -> [a]
9   take 0 _ = []
10  take _ [] = undefined
11  take n (x:xs) = x:take (n-1) xs
12
13  -- ones :: Num a => [a]
14  ones = 1:ones
15
16  -- valid :: Bool
17  valid = sum (take 10 ones :: [Int]) == 10
```

Extensions which were successfully implemented add support for user-defined datatypes, user-defined typeclasses and instances, monads, and some syntactic features like operator sections

and support for point-free notation: these can be demonstrated by the following program, which exemplifies the monad definition and use-cases:

```haskell
data Maybe a = Nothing | Just a
data [] a = [] | a:[a]

class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just
instance Monad [] where
    xs >>= f = concat (map f xs)
    return x = [x]

-- The monad instance for maybe can be interpreted as function application
-- with support for chaining failure
divide x y = if y == 0 then Nothing else Just (x / y)
x = divide 4 0 >>= divide 20 -- Evaluates to Nothing

-- The monad instance for lists can be interpreted as performing
-- non-deterministic computation: each step can have multiple results
countdown 0 = []
countdown n = n:countdown (n - 1)
onlyEven x = if even x then [x] else []
y = [1,2,3] >>= countdown -- Evaluates to [1,2,1,3,2,1]
z = y >>= onlyEven -- Evaluates to [2,2]
```

Each simple feature name necessarily glosses over many smaller constituent features necessary for use. For example, the 'lists' feature allows for lists to be created using either the plain constructor syntax (`1:(2:(3:[]))`) or syntactic sugar for lists (`[1,2,3]`), and matched using patterns (eg. `[x,y] = [1,2]`). However, there is no support for list comprehensions (eg. `[f x | x <- [1,2,3], even x]`) as they were not a high priority feature.

4.2.1 Correctness

Correctness of the various compiler stages has been empirically tested using a large set of unit, integration, and regression tests: these include tests of complete programs, such as those used for benchmarking.

At the time of writing, there are 245 tests. These are run both on my development machine (described in Section 4.3.1), and on machines provided by Travis CI[1] whenever a commit is pushed to my development GitHub repository. This ensures HJC works in a clean, reproducible environment and not just on my development system.

Bugs found and fixed during development have at least one associated regression test to ensure that they cannot reappear.

---

[1]`https://travis-ci.org/hnefatl/dissertation-project`

As HJC is developed using Haskell, some forms of compiler bugs that could affect the correctness of translation have been mitigated: in particular, type errors and bugs due to mutable state cannot occur.

4.3 Performance

Although performance was not an important aspect of the success criteria, it is still interesting to evaluate the effectiveness of optimisations on the output program, and the performance of HJC and its output compared to other Haskell to JVB compilers.

The two compilers used for comparisons are Eta and Frege. Eta is a fork of GHC that replaces the backend with one targeting JVB: it can take advantage of the powerful optimisations already available in the front- and middle-end of GHC, which is the world-leading Haskell compiler, so I expected it to perform better than both HJC and Frege. Frege is a from-scratch compiler that compiles to Java instead of JVB, then uses a Java compiler to produce JVB. I expect Frege to perform better than HJC , given the maturity of the project (under development since at least 2011).

Both Eta and Frege enable optimisations by default, so all metrics given for them have optimisations enabled. Metrics given for HJC are labelled to indicate whether or not optimisations have been applied.

4.3.1 Test Environment

Benchmarks were measured as the only active process on my development machine: a ThinkPad 13 running Debian 9 with 8GB RAM and an Intel Core i5-7200U CPU (2.5GHz).

4.3.2 Execution Speed

Figure 4.1 demonstrates the runtime performance of the benchmark programs after compilation by the different compilers. It is evident that the performance of programs compiled by HJC is significantly lower than those from Frege or Eta, but also that applying optimisations can produce a reasonable speedup.

Interestingly Frege produced more performant programs than Eta, which was not expected, given that Eta leverages the work done on the GHC compiler.

The inefficiency of programs output by HJC seems to stem primarily from my implementation of runtime closures (described in Section 3.5.1.3). Figure 4.2 shows that almost 40% of a program's execution is spent in the functions associated with creating and calling closures: `enter`, `addArgument`, and `clone`. A more efficient implementation should have most of the runtime spent in one or more of the `Impl` functions (such as `_v19Impl`), which perform the actual logic of an expression of the Haskell program, rather than the bottleneck being in the runtime system.
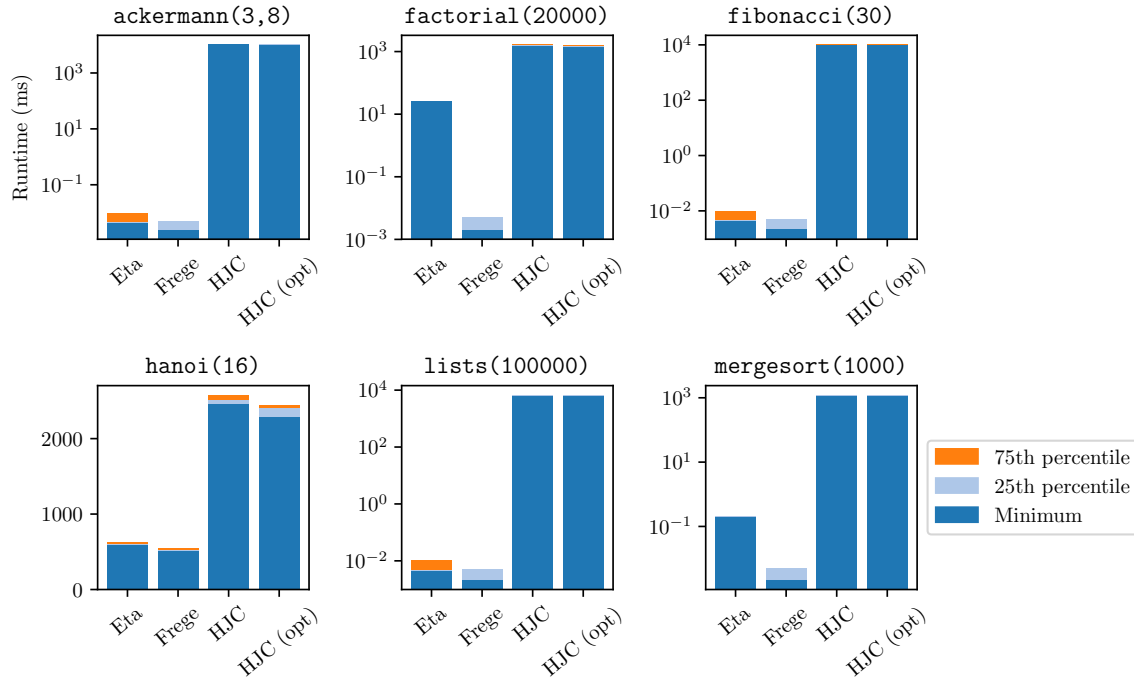
Figure 4.1: Minimum runtime of the benchmark programs. Minimum is plotted instead of median as it is more reflective of program speed without system load.

Inner time of runtime functions during `ackermann(3,8)`
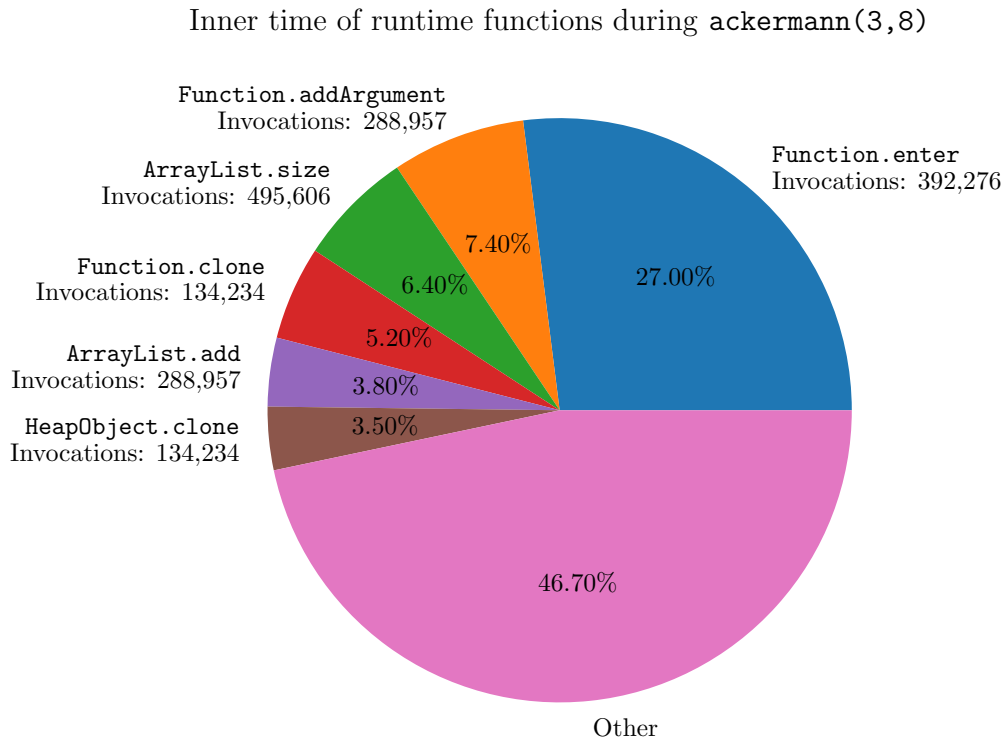

Figure 4.2: A breakdown of the percentage of execution time spent in each Java function in the generated code ('inner time') for the `ackermann` benchmark. Functions with inner time of less than 3% are grouped into a single wedge.

As described in the implementation chapter, HJC uses objects of a single class to represent closures, with objects of the class storing `Function` objects generated using the `invokedynamic` instruction and Java 8's support for anonymous functions. This is different from the approaches taken by both Eta and Frege, where closures are translated into anonymous classes that implement a Functional Interface[2] allowing objects of the classes to be invoked like anonymous methods.

This design choice was deliberate: before implementing HJC , it seemed like using a single class for all functions would be more efficient than using a new class for each function, as the equivalent Java code is significantly shorter. After performing the implementation and comparing the two different approaches, using functional interfaces with a new class for each function results in simpler code for the creation and evaluation of closures, while also eliminating significant amounts of overhead involved with creating the function objects used by HJC , which I expect plays a large role in the performance difference.

4.3.3   Compiler Performance

Figure 4.3 presents the minimum time taken to compile each benchmark program: HJC is significantly faster than both Eta and Frege to compile all benchmarks except for `mergesort`, which is significantly slower.

The poor performance appears to be due to the large number of bindings involved: processing the list of 1000 input numbers pushes the inefficient stages to their limits. The `hs-java` library used in code generation was known to perform poorly – efforts were made to improve the performance, in one case **halving** the time spent in code generation[3] for the `mergesort` benchmark, but as this was an unexpected work package I ran out of time before being able to improve the performance to a satisfactory level. This benchmark also revealed that the current let-lifting algorithm is possibly exponential-time. Neither component should take much work to improve up to acceptable scalability.

Figure 4.4 shows the same metrics for just the `ackermann` benchmark on HJC , so the times for each stage can be seen more clearly. It is clear that code generation is the most expensive stage in the main pipeline: profiling reveals that this is mostly due to the inefficient implementation of parts of the `hs-java` library.

It is interesting that HJC usually takes less time to process the input when performing optimisations than when not. From Figure 4.4 we can see this is due to a significant amount of the compilation time being spent in the code generation stage, and compressing and writing the compiled classes into a jar file: the unreachable code elimination optimisation described in Section 3.6.3 can significantly reduce the amount of code that reaches code generation, which also reduces the amount of bytecode that needs to be compressed and written to disk.

---

[2]`https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`
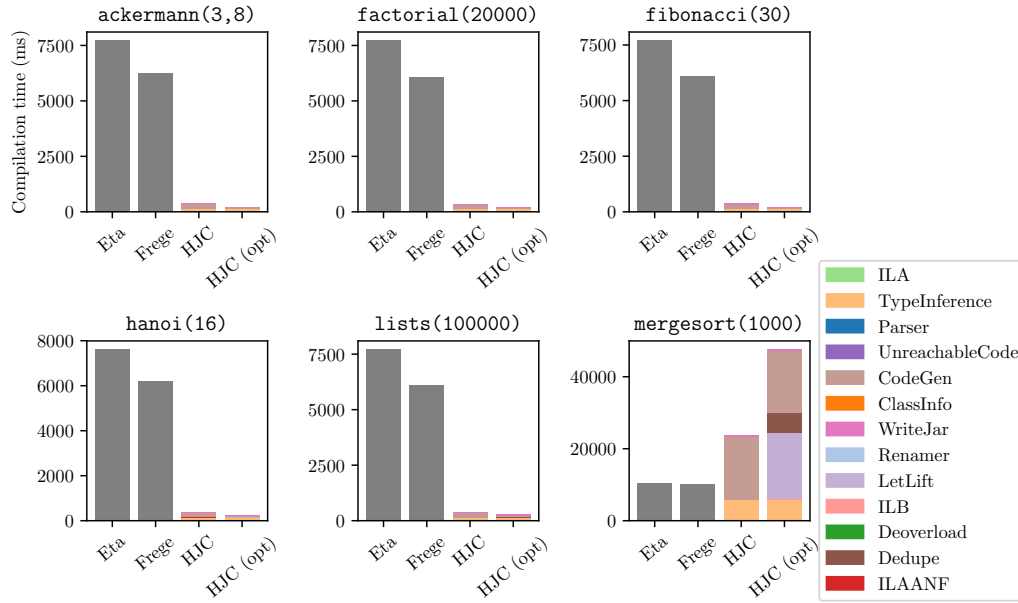[3]`https://github.com/hnefatl/hs-java/commit/5cbd5c3995161199442310ac30c101b0497a7f98`

Figure 4.3: Minimum time taken to compile the benchmark program. The coloured sections indicate the amount of time spent in each stage.
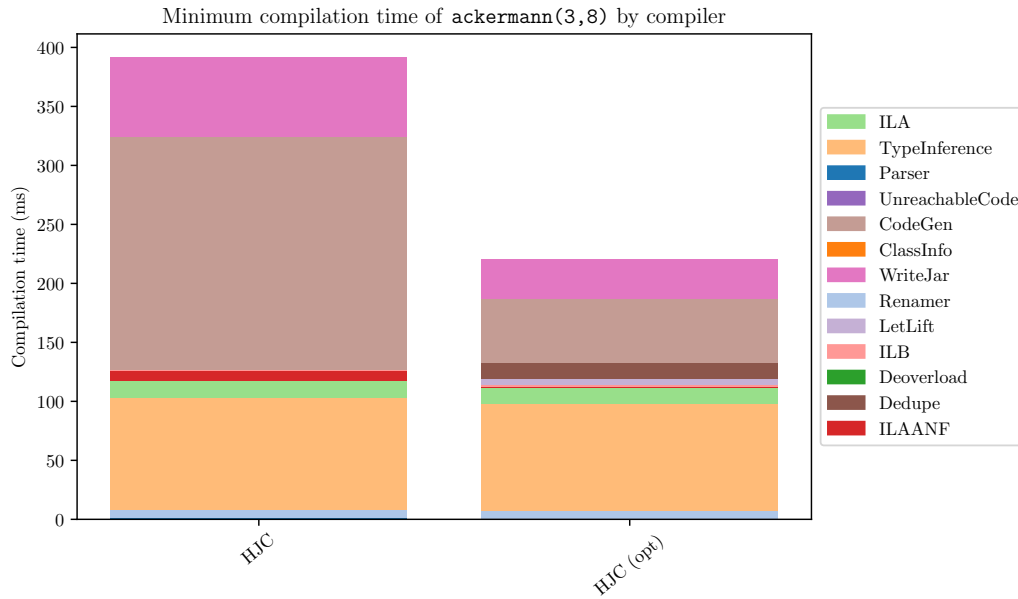

Figure 4.4: Runtime of each stage for the `ackermann` benchmark.

4.3.4 Executable Size

Figure 4.5 displays the compiled size of each benchmark program after compilation by the various compilers. All three compilers generate class files that implement the logic of the Haskell program, usually have a class file for each datatype defined in the Haskell program, and have class files providing runtime support (for example, each compiler has an equivalent to the

`Function` class described in Section 3.5.1).

HJC produces the smallest executables for all benchmarks, with optimisations trimming off a reasonable amount of generated bytecode.
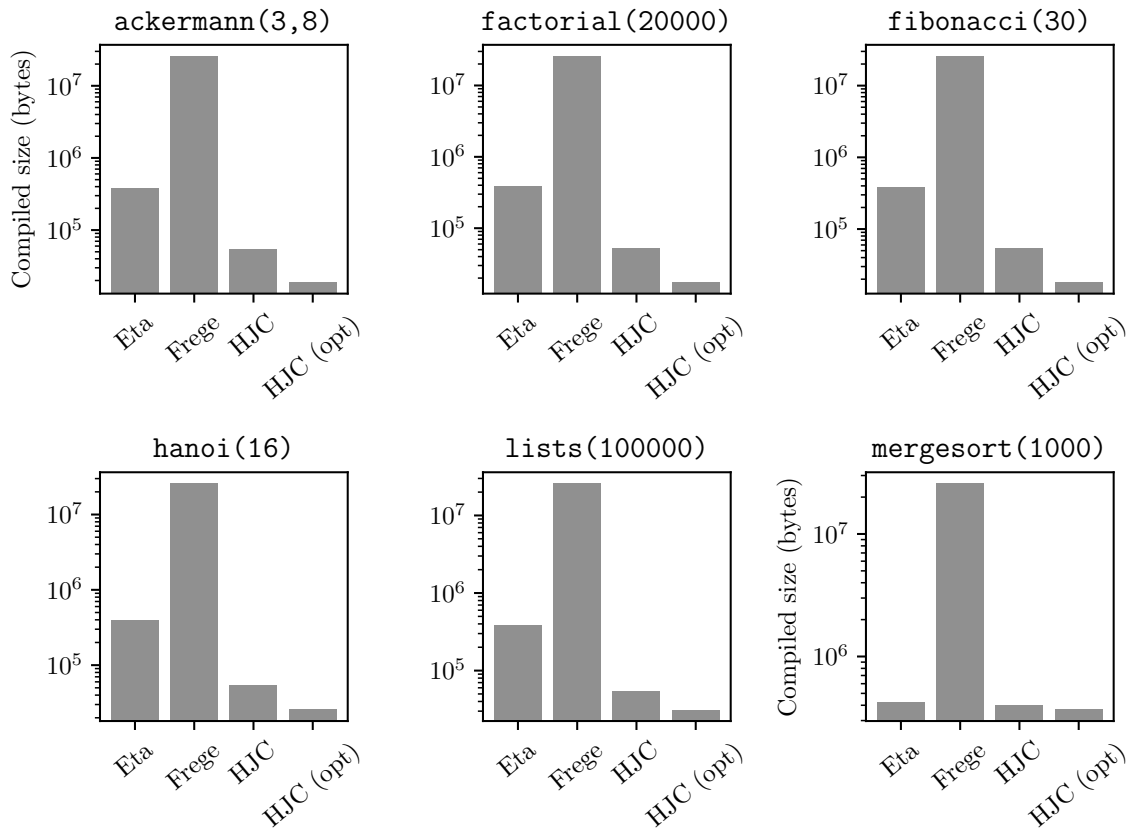

Figure 4.5: Compiled size of each benchmark program. This size includes the runtime system and the bytecode corresponding to the actual program.

4.3.5   Impact of Optimisations

Figure 4.6 shows that the implemented optimisations are successful in reducing the execution time of compiled programs, although none have a truly significant impact: speedups can be seen of up to 10%, but the performance remains poor compared to Frege and Eta.

Figure 4.7 demonstrates that the unreachable code elimination optimisation is effective at reducing the size of compiled programs, in some cases halving the size. The amount of removed code is relatively constant, as the majority of removable code in these benchmarks will be from the standard library, which produces a constant size overhead. For larger input programs, the space savings could be greater.
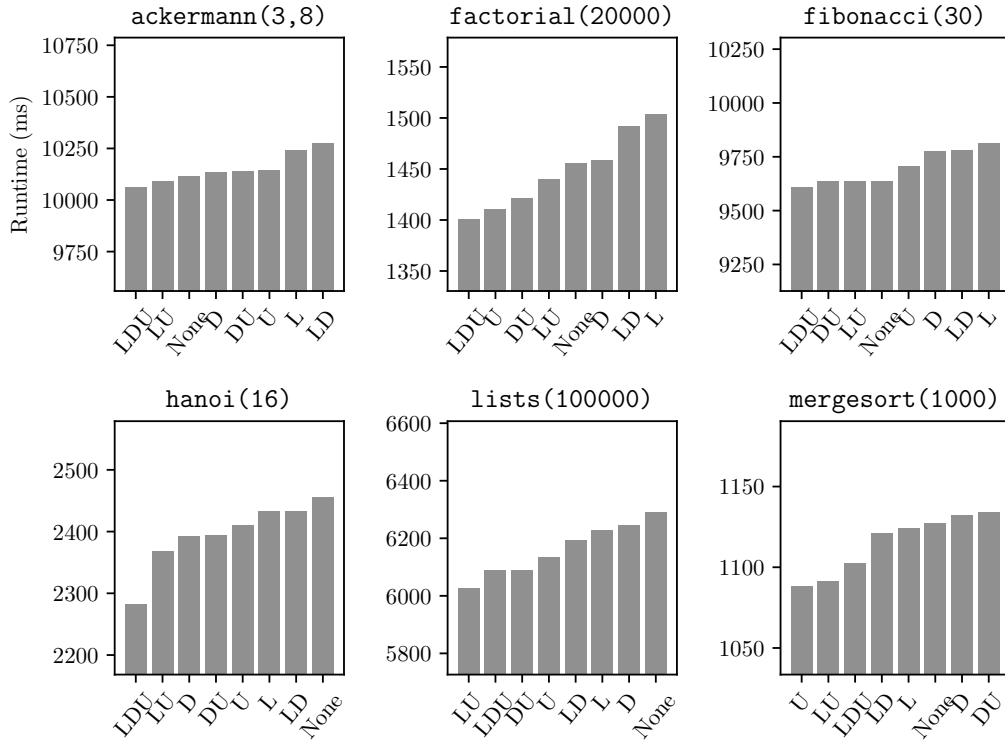
Figure 4.6: Minimum runtime of the benchmarks after compilation with various optimisations: L stands for let-lifting, D for binding deduplication, and U for unreachable code elimination.
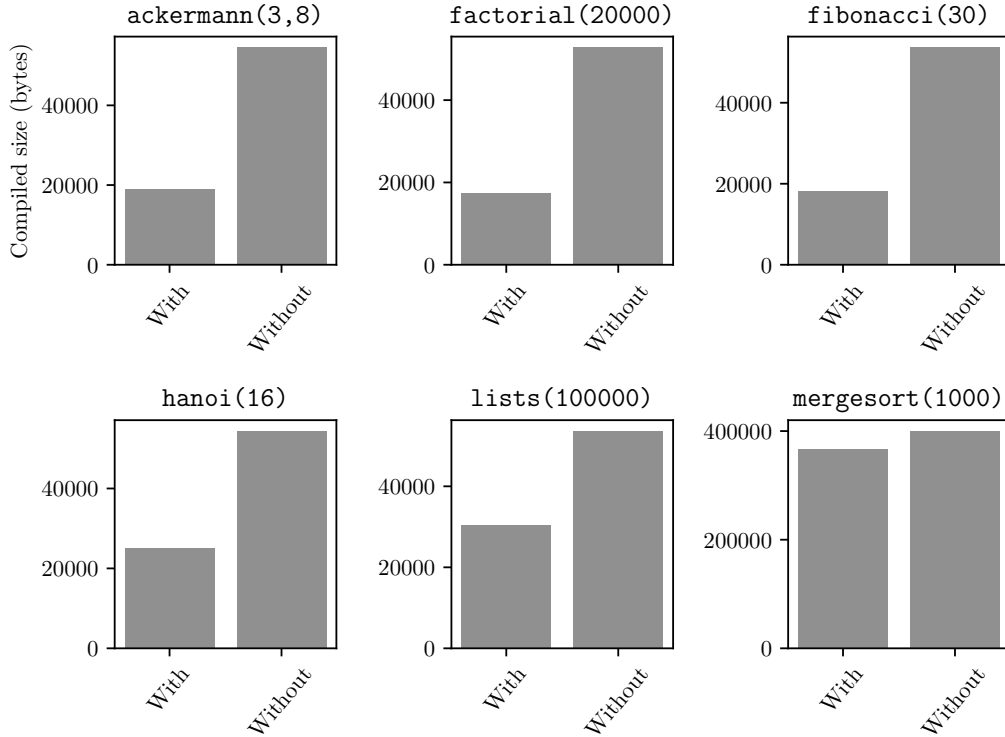

Figure 4.7: Executable size from HJC with and without the unreachable-code-elimination optimisation.

Chapter 5

Conclusions

This project was successful: all of the success criteria were met and a number of extensions were completed, resulting in a sizeable subset of Haskell being supported. The optimisations implemented have a positive effect on the runtime behaviour of compiled programs, and while the programs are not as performant as might be desired, building a solid compiler for a significant subset of Haskell was the aim, rather than sheer performance.

I have learned a great deal about the design of compilers from this project, as well as the effects of language design choices and features on the different stages of a traditional compiler.

5.1   Hindsight

The only part of the project that I would redo, given the benefit of hindsight, is the typechecker.

The typechecker is the most complex stage: while some amount of complexity is unavoidable due to Haskell's relatively exotic type system, I feel that the 'complexity overhead' added by the implementation onto the complexity of the underlying concepts is higher here than in any other component.

If I could re-implement this component from scratch, I would focus less on type-checking performance (which was a design goal in the current implementation) and more on clarity and extensibility. This might involve implementing distinct passes for generating type/typeclass constraints and then solving them, rather than doing both in a single pass as in the current implementation. This would likely trade performance for clarity.

5.2   Further Work

The implementation of closures was an experiment: it differs from the approach taken by other Haskell to JVB compilers such as Frege and Eta. However, evaluation showed that this design is not very performant, so it would benefit HJC to switch to the more efficient approach; although that is quite a large chunk of work.

Smaller, more achievable chunks of work include adding support for more language features. There are plenty of these in a large language like Haskell, ranging from syntactic sugar like list comprehensions to more complex features like typeclass/instance contexts.

Only a few optimisations were implemented in this project, mostly due to time constraints. Strictness analysis is an optimisation that I did not implement but expect would product sizeable performance improvements, as it reduces the overhead introduced by lazy evaluation in cases where the semantics of the program are unchanged. Inlining is another major optimisa-

tion, made easier to implement by Haskell's purity.

Another exciting extension would be replacing the typechecking component with the OutsideIn(X) framework described in Vytiniotis et al. [10], which would open up opportunities for supporting modern extensions to Haskell's type system such as GADTs and multi-parameter typeclasses.

Finally, it would not be too hard to extend the existing 'Hooks' implementation to provide the programmer with a foreign function interface (mentioned in Section 3.5.3), which would add significant expressive power.

Bibliography

[1] Fighting spam with Haskell. `https://code.fb.com/security/fighting-spam-with-haskell`. Accessed: 09/05/2019.

[2] Haskell at Galois. `https://galois.com/about/haskell/`. Accessed: 09/05/2019.

[3] `hs-java` forked package page. `https://github.com/hnefatl/haskell-src`. Accessed: 14/05/2019.

[4] Idris Language Homepage. `https://www.idris-lang.org/`. Accessed: 09/05/2019.

[5] J.P. Morgan GitHub repository list. `https://github.com/jpmorganchase?language=haskell`. Accessed: 09/05/2019.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[7] Mark P. Jones. Typing Haskell in Haskell. `https://web.cecs.pdx.edu/~mpj/thih/thih.pdf`, 2000.

[8] Benjamin C. Pierce. Types and programming languages, 2002.

[9] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7, 1980.

[10] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x): Modular type inference with local assumptions. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/jfp-outsidein.pdf`, 2011.

[11] Philip Wadler and Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

Appendix A

Implementation of `Integer`

Here is an example implementation of `Integer`, Haskell's arbitrary precision integral value type: it is essentially a wrapper around Java's `BigInteger` class. The copious uses of underscores is explained in Section 3.5.2.

```java
1  import java.math.BigInteger;
2
3  public class _Integer extends Data {
4      public BigInteger value;
5      public static _Integer _make_Integer(BigInteger x) {
6          _Integer i = new _Integer();
7          i.value = x;
8          return i;
9      }
10     public static _Integer _make_Integer(String x) {
11         return _make_Integer(new BigInteger(x));
12     }
13
14     public static _Integer add(_Integer x, _Integer y) {
15         return _make_Integer(x.value.add(y.value));
16     }
17     ... // Analogous functions for subtraction and multiplication
18
19     public static boolean eq(_Integer x, _Integer y) { ... }
20
21     public static String show(_Integer x) { ... }
22  }
```

The `x = [1, 2]` function is used to construct `Integer` literals: it allows a Java `_Integer` object to be constructed from a Java string representation. For example, the bytecode that creates the Haskell literal 2 would load the string `"2"` from the constant pool then invoke the creation method:

```
1  ldc           210        // String 2
2  invokestatic 16          // Method
↪    tmp/_Integer._make_Integer:(Ljava/lang/String;)Ltmp/_Integer;
```

The `add`, `eq`, etc. methods are Java implementations of the functions required by Haskell's `Num`, `Eq` and `Show` typeclass instances for `Integer`. For 'builtin' types, the implementation of these typeclass functions need to be given in Java, as they cannot be expressed in Haskell. Section 3.5.3 on Hooks covers this aspect of code generation in more detail.

Appendix B

The `Function` class

```java
import java.util.ArrayList;
import java.util.function.BiFunction;

public class Function extends HeapObject {
    private BiFunction<HeapObject[], HeapObject[], HeapObject> inner;
    private HeapObject[] freeVariables;
    private ArrayList<HeapObject> arguments;
    private int arity = 0;
    private HeapObject result = null;

    public Function(BiFunction<HeapObject[], HeapObject[], HeapObject> inner, int arity,
    ↪  HeapObject[] freeVariables) {
        this.inner = inner;
        this.arity = arity;
        this.freeVariables = freeVariables;
        arguments = new ArrayList<>();
    }

    @Override
    public HeapObject enter() {
        // Check if we've got a cached value
        if (result != null) {
            return result;
        }

        if (arguments.size() < arity) {
            return this;
        }
        else if (arguments.size() > arity) {
            try {
                Function fun = (Function)inner
                    .apply(arguments.subList(0, arity).toArray(new HeapObject[0]),
                    ↪  freeVariables)
                    .enter()
                    .clone();
                for (HeapObject arg : arguments.subList(arity, arguments.size()))
                    fun.addArgument(arg);
                result = fun.enter();
                return result;
            }
            catch (CloneNotSupportedException e) {
                throw new RuntimeException(e);
            }
        }
        else {
```

```java
44          result = inner.apply(arguments.toArray(new HeapObject[0]),
   ↪     freeVariables).enter();
45          return result;
46      }
47  }

49  public void addArgument(HeapObject arg) {
50      arguments.add(arg);
51  }

53  @Override
54  public Object clone() throws CloneNotSupportedException {
55      Function f = (Function)super.clone();
56      f.inner = inner;
57      f.arity = arity;
58      f.freeVariables = freeVariables.clone();
59      f.arguments = new ArrayList<>(arguments);
60      return f;
61  }
62 }
```

Appendix C

Repository Structure Diagrams

The top-level repository structure is within 4 directories:

```
./
├── app
├── benchmarks
├── src
└── test
```

C.1   compiler

```
src/
├── Compiler.hs
├── Backend/
│   ├── CodeGen.hs
│   ├── Deoverload.hs
│   ├── ILAANF.hs
│   ├── ILA.hs
│   └── ILB.hs
├── Optimisations/
│   ├── BindingDedupe.hs
│   ├── LetLifting.hs
│   └── UnreachableCodeElim.hs
├── Preprocessor/
│   └── Renamer.hs
└── Typechecker/
    └── Typechecker.hs
```

C.2   compiler-exe
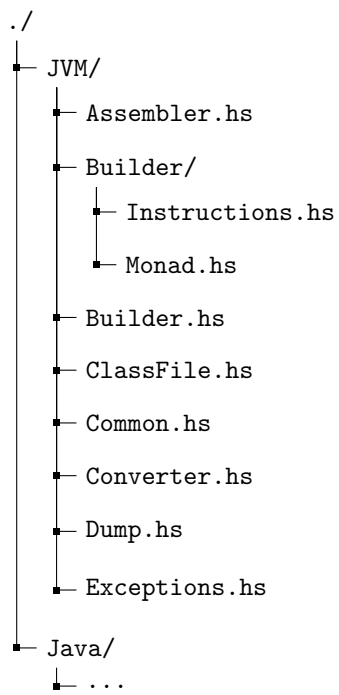
```
app/
└── Main.hs
```

C.3   Tests

```
test/
├── AlphaEqSpec.hs
├── Backend/
│   ├── DeoverloadSpec.hs
│   ├── ILAANFSpec.hs
│   ├── ILASpec.hs
├── Preprocessor/
│   ├── DependencySpec.hs
│   ├── RenamerSpec.hs
├── Typechecker/
│   ├── TypecheckerSpec.hs
├── Spec.hs
├── WholeProgram.hs
```

C.4 Benchmarks

```
benchmarks/
├── benchmark.py
├── etabenchmark.py
├── fregebenchmark.py
├── fregec.jar
├── javabenchmark.py
├── jhaskellbenchmark.py
├── jmhbenchmark.py
├── Main_Template.java
├── plot.py
├── programs
│   ├── ackermann.eta
│   ├── ackermann.fr
│   ├── ackermann.hs
│   ├── factorial.eta
│   └── ...
├── results
│   └── ...
├── results.py
└── runbenchmarks.py
```

C.5 `hs-java`

```
./
├── JVM/
│   ├── Assembler.hs
│   ├── Builder/
│   │   ├── Instructions.hs
│   │   └── Monad.hs
│   ├── Builder.hs
│   ├── ClassFile.hs
│   ├── Common.hs
│   ├── Converter.hs
│   ├── Dump.hs
│   └── Exceptions.hs
└── Java/
    └── ...
```

Chapter 6

Project Proposal

An Optimising Compiler from Haskell to Java Bytecode

Candidate 2392F

Robinson College

Tuesday 16th October, 2018

Project Originator:	Candidate 2392F	
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

---

[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 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

---

[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

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.

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).

---

[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

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.

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].

---

[14]https://github.com/ghc/nofib
[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

- 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.

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.

---

[18] https://www.haskell.org/onlinereport/standard-prelude.html

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

- **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.

  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.