Keith Collister

An Optimising Compiler from Haskell to Java Bytecode

Computer Science Tripos – Part II

Robinson College

May 11, 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: May 11, 2019

Proforma

Name:	**Keith Collister**
Candidate Number:	
College:	**Robinson College**
Project Title:	**An Optimising Compiler from Haskell to Java Bytecode**
Examination:	**Computer Science Tripos – Part II, July 2019**
Word Count:	**15302**
Lines of Code:	**8549**
Project Originator:	Keith Collister
Project Supervisor:	Dr. Timothy Jones

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.

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: the compiler 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.

Would be nice to include benefits of using my tool over an alternative, but realistically there aren't any: mine supports a smaller subset of Haskell, doesn't provide Java interop, and is slower than the others. The one plus is compilation speed, which isn't really enough to bring up.

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[3]. It is also seeing increasing industrial usage by companies such as Facebook[1], J.P. Morgan[4], 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, which reduces the potential for bugs and allows for aggressive optimisations; laziness can improve efficiency in compositional code such as head $\circ$ sort, which can run 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, as it removed 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 doesn't 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[7]. 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 System F and the 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're 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 'type constructor' that takes no parameters. 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 (of kind α), and from the data constructor branch `Right a` we can infer that α `= *`, 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 values 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, evaluation of terms is to normal form (NF): `(1+2+3, True && False)` is evaluated to `(6, False)`. An alternative form is weak head normal form (WHNF), in which terms are evaluated up 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

```
1  1
2  (+) 1
3  Just True
4  (+) (1 + 2)
5  x = 1:x
```

Not in WHNF

```
1  1 + 2
2  (\x -> x) 1
3  (1 + 2) + 3
```

Evaluation of an expression up to WHNF corresponds to a form of **non-strict evaluation**: partial applications of functions or any data constructor applications don't 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.

WHNF is vital for the code generation stage, as the evaluation method used is WHNF.

2.2.3 Administrative Normal Form

Administrative Normal Form (ANF, presented by Flanagan et al. [5]) 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 (but definitely different from) 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 the compiler 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[1]) is used.

Preprocessing

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

Dependency analysis computes a partial order on the source declarations so that the typechecker can process them in a valid order.

Type Checker

The type inference stage infers polymorphic overloaded types for each symbol, checks them against any user-provided type signatures, and alters the AST so that each expression is tagged with its type.

Deoverloading 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).

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

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 (hs-java[2]) is used to convert a logical representation of the bytecode into a set of class files, which are then packaged into an executable Jar file.

2.4 Testability

Given the number of stages in the compiler 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 the compiler 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 don't reintroduce incorrect behaviour.

Finally, end-to-end tests ensure that the compiler 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.

kc506: check this doesn't repeat stuff from the evaluation chapter

2.5 Tools

I chose to write this compiler 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.

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

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 of the compiler 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

The compiler 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

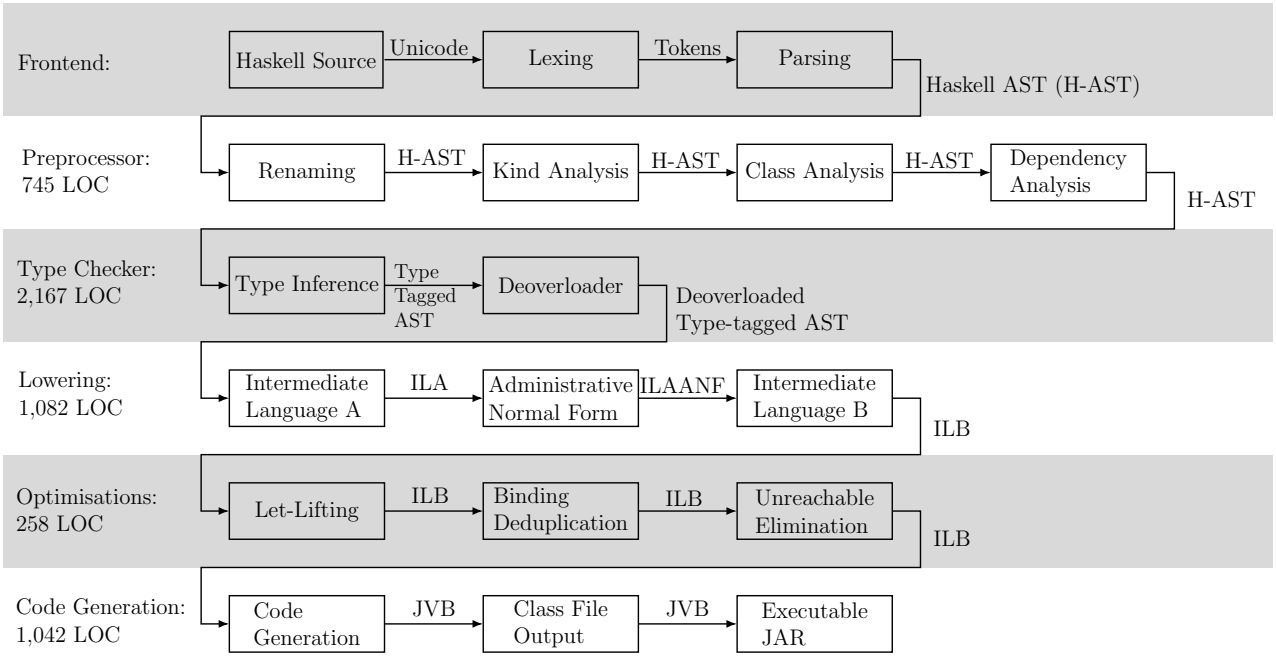


Figure 3.1

The compiler is comprised 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 will present 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 have modified to provide some additional desirable features:

- Lexing and parsing declarations for built-in constructors like list and tuple definitions (eg. `data [] a = [] | a:[a]`).
- Parsing data declarations without any constructors (eg. `data Int`)[2]. This allows for a convenient way of introducing built-in types.

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

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

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

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

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

Figure 3.2: An example of a multi-parameter typeclass

3.2 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 extract information while applying transformations.

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. For example, when typechecking the following code we might conflate the two uses of `x`, and erroneously infer that they have the same type. A similar problem arises with variable shadowing, when the scopes overlap. The problem also applies to any type variables present in the source – the type variable `a` is distinct between the two type signatures:

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

Additionally, variables and type variables 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:

```
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 haven't been already used).

Unique variable/type variable names are generated by prefixing the current value of an incrementing counter with either `v` for variable names or `tv` for type variable names. The renamer traverses the syntax tree maintaining a mapping from a syntactic variable/type variable name to an associated stack of unique semantic variable names (in Haskell, a `Map VariableName [UniqueVariableName]`):

- When processing the binding site of a new syntactic variable (eg. a let binding, a lambda argument, a pattern match...), a fresh semantic name is generated and pushed onto the stack associated with the syntactic variable.
- Whenever we leave the scope of a syntactic variable, we pop the top semantic name from the associated stack.
- When processing a use site of a syntactic variable, we replace it with the current top of the associated stack.

An analogously constructed mapping is maintained for type variables, but is kept separate from the variable mapping: otherwise the keys can conflict in code such as `x :: x`.

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

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

We compute the sets of free/bound variables/type variables/type constants for each declaration, then construct a dependency graph – each node is a declaration, and there's an edge from A to B if any of the bound variables/type variables/type constants at A are free in B. It is important to distinguish between variables/type variables and type constants, as otherwise name conflicts could occur (as we don't rename type constants). This separation is upheld in the compiler by using different types for each, and is represented in the dependency graph below by colouring 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. The compiler uses an existing Haskell library to compute the SCCs (`containers`[3]), which uses the algorithm presented by M. Sharir[8].

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're contained within the same strongly-connected component so are mutually recursive; we have to typecheck d_2 last, after both other components.

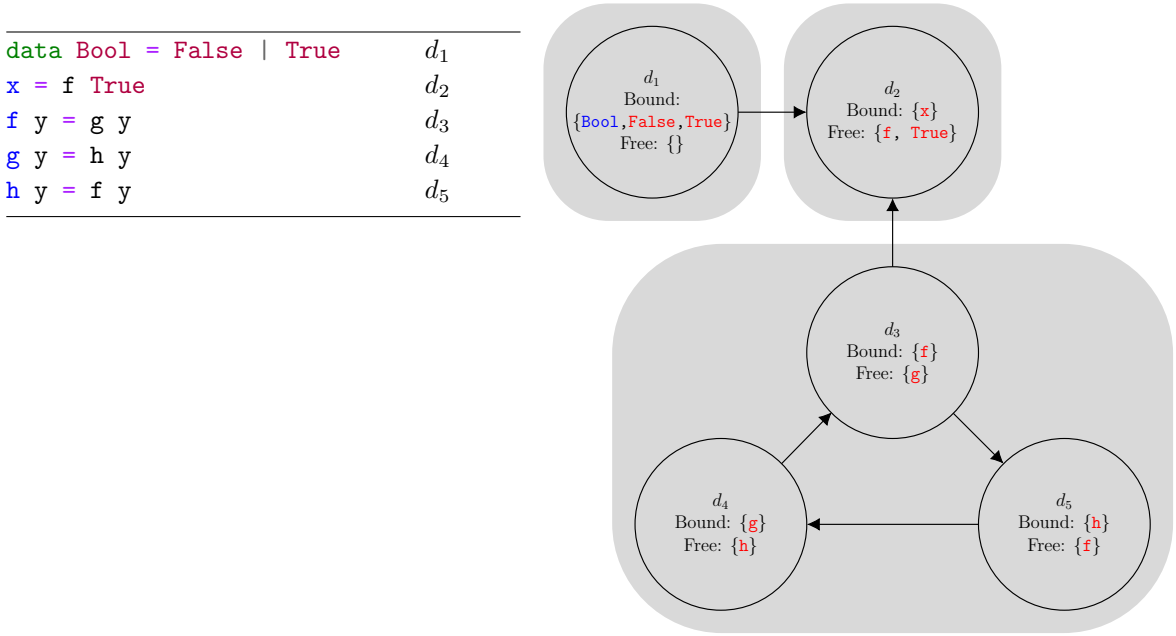


Figure 3.3: Code labelled with declaration numbers, and the corresponding dependency graph. Variables are in red text, type constants in blue. Strongly connected components are highlighted.

Typechecking declarations within the same component can proceed in an arbitrary order, we just need to ensure that the all of the type variables for the names bound by the declarations are available while processing each individual declaration.

Special cases due to Typeclasses

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

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

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

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

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

```
x = True
x = False
```

These declaration conflicts can be expressed as a binary symmetric predicate on declarations, as presented in Figure 3.4, 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 }`.

	`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 3.4: The conflict relation: the bottom triangle is omitted as the predicate is symmetric

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

- Another complication introduced by typeclasses is that variable declarations such as `id = \\x -> x` are usually treated as being the unique binding definition of `id`, and any other uses within the same level of scope must be free rather than binding (otherwise we have conflicting definitions).

  However, we treat binding declarations inside `instance` declarations as actually being free uses rather than binding uses, so that the instance declaration forms a dependence on the class declaration where the variables are bound, ensuring it is typechecked first.

- The dependencies generated by this technique are *syntactic*, not *semantic*: this is a subtle but very 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.

```
class Foo a where
    foo :: a -> Bool
instance Foo Bool where
    foo x = x
instance Foo [Bool] where
    foo xs = all foo xs
```

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` (`foo` is treated as being free within the declarations as described above): they will only generate dependencies to `class Foo a` (and to the declaration of `Bool` and `all`).

Computing the semantic dependencies is too complicated to be done in this pass, so the problem is left and instead solved during the typechecking stage. A full explanation is given later, but the approach used is to defer typechecking instance declarations until a different declaration requires the information, and then attempt to typecheck the instance declaration then, in a Just-In-Time manner.

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 alterations 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), as that compiler provides extensions such as GADTs and type families requiring a more complex type system.

The datatypes used to represent types are as follows:

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[4] to allow for type constraints such as `pure :: Functor f => α → fα`, in which `f` has kind $* \to *$.

Note that function types ($A \to B$) are represented as applications of the $\to$ type constructor. This simplifies logic in many places, as $\to$ can usually be treated in the same way as type constructors like `Either`.

A 'simple type' is then represented as any tree of applications between type variables and type constants: these are types such as `Int -> Maybe Bool`. Haskell has more complex types, however: overloaded and polymorphic types.

```
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 α => α → α → Bool` (the type constraint here being just `Eq α`). 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 α that are instance of the `Eq` typeclass.

A quantified/polymorphic type is an overloaded type with a set of type variables that are universally quantified over the type, meaning they must later be instantiated to a specific type/type variable (universally quantified variables are 'placeholder' variables). Haskell type signatures are implicitly quantified over all the contained type variables, but some extensions add explicit syntax: `id :: α → α`, `id :: forall α. α → α`, and `id :: ∀α. α → α` all mean the same.

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

Having covered the grammar used to described types within the typechecker, we now move on to the actual typechecking process. In the following sections, 'declaration' is assumed to refer to simple pattern binding declarations like `x = f y` unless otherwise mentioned: other declaration types are either easy to extend these approaches to, don't play much role in typechecking, or are sufficiently complicated that they are omitted for brevity.

Section giving overview of substitution and unification?

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

3.3.2 Type Inference

The implementation is inspired by the approach given by Mark P. Jones[6] 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 AST in dependency order as previously described in Section 3.2.3.

1. The first pass tags each subexpression with a type variable, then uses rules similar to the HM inference rules to infer the value of the type variable, usually using the type variables of subterms.

   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 ($\forall\alpha.\ \text{Num}\alpha \Rightarrow \alpha \to \alpha \to \alpha$ to $\text{Num}\beta \Rightarrow \beta \to \beta \to \beta$, where β is a fresh unique type variable), then moving the constraints from the type to the set of constraints built up while traversing this declaration to get just $\beta \to \beta \to \beta$. This is the ground type that's unified with the type variable used to tag the use of `(+)`, and the type constraints are stored for use after finishing traversing the declaration.

   After a pattern binding declaration has been fully traversed, types are generated for all the variable names bound by the patterns. 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 aren't 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 type variables used to tag each with the final expression type generated by the unification during the first pass.

   This can't be done efficiently during the first pass: consider the expression $(\texttt{(+)}^{t_1}\ \texttt{x}^{t_2})^{t_3}$ where the t_i are type variables tagging the expressions. Assume we've inferred that `x :: `α and `(+) :: `$\beta \to \beta \to \beta$ as described above, and that we've unified t_1 and t_2 with these types respectively. Inferring the type of the overall expression would proceed by unifying the type of the first formal argument of the function (β) with the first actual argument (α), and then using this substitution to type the return value of the function as $\alpha \to \alpha \to \alpha$, which we can now unify with t_3. Had we previously updated the subterm's tags to be their inferred concrete types we'd now have to update them again: β is no longer used as it's been unified with α, but our subterms may still contains uses of it.

3. The third pass checks that any user-provided type signatures (such as the user explicitly annotating `5 :: Float`) are valid compared to what was actually inferred: if the user-provided tag is more general than the inferred tag, we reject the program.

   This could be done during the second pass, but was kept as a distinct pass for clarity in the code.

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

3.3.2.1 Restricted Polymorphism

One departure from conventional polymorphic type systems is that Haskell's type system restricts polymorphism for some terms: let-bound variables are polymorphic over all their free type variables, while function parameters are never polymorphic. In practice, this means that in the code below, `f :: ` $\forall\alpha.\ \alpha \to a$ whereas `g :: ` $\alpha \to \alpha$. The difference in semantics ensures that type inference remains decidable.

```
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
```

3.3.2.2 Typeclass Order

A tricky part of the typechecking process is dealing with typeclasses, as dependency order isn't semantically accurate for typeclass instance declarations: the problem is detailed in Section 3.2.3.

To handle the potential 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.

3.3.2.3 The Ambiguity Check

There is a type-system correctness check called the 'ambiguity check', which ensures that for any variable with overloaded type, at its use-site the type variables are either resolved to a ground type such as `Int` or are exposed in the type of the expression they're being used in. This prevents programs which have ambiguous semantics (multiple possible runtime behaviours) from being reported as valid (this issue, along with an approach for fixing it called defaulting, is described in Section 11.5.1 of 'Typing Haskell in Haskell' [6]).

My compiler does not implement this check, which allows some programs to be accepted by the compiler which then crash at runtime (such cases can often be fixed by adding explicit type signatures). This feature was omitted in order to spend development time on the later stages of the compiler.

Check they actually crash at runtime: I feel like it should get caught in deoverloading...

3.3.2.4 Defaulting

As a defence against the issues detected by the ambiguity check, and to make Haskell code slightly cleaner, the Haskell Report defines a feature called 'defaulting'[5] which allows for ambiguous type variables appearing in `Num` constraints to be defaulted to a type specified on a per-module level.

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

This means that in the expression `const 1 2`, the type of `2` is defaulted to `Integer`, rather than raising an error.

This feature was not implemented, as the time taken to implement it was expected to be better spent elsewhere. This is a non-critical feature, as explicit type signatures can always be added (eg. `const 1 (2 :: Int)`).

3.3.2.5 Improvements

An improvement to the current approach would be to implement the OutsideIn(X) framework given by Vytiniotis et al. [9]. 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 and handling type classes more flexibly than the current implementation.

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 before the compiler begins lowering the program 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 essentially '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, except the v-table is being passed as an extrinsic argument, rather than being intrinsic to the object being operated on.

The approach used here is to perform a source-to-source transformation on the AST that replaces typeclass/instance declarations with datatype/value/function declarations. There are three parts to the transformation:

1. 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.

```
instance Eq Int where
    (==) = foo
    (/=) = bar
```

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

2. Any call-site of a function with an overloaded type needs to be passed the relevant dictionary: for example, `x = 1 + 2 :: Int` is translated into `x = (+) dNumInt 1 2 :: Int`.

   Similarly, any definition of a symbol which has an overloaded type needs to be extended with parameters to carry the implementation of all the typeclass constraints, in order to pass them down to the subexpressions within the function which require the implementations. Quite pleasingly, this replaces the `=>` symbol with `->`.

```
sum :: Num a => [a] -> a
sum xs = foldl (+) 0 xs
```

```
sum :: Num a -> [a] -> a
sum dNumA xs = foldl ((+) dNumA) 0 xs
```

   Note that this also affects declarations of symbols which aren't originally functions:

```
x :: Num a => a
x = sum [1,2,3,4]
```

```
x :: Num a -> a
x dNumA = sum dNumA [1,2,3,4]
```

These three steps can be performed in a single pass over the type-tagged Haskell AST, thanks to the information gathered in previous passes and the type annotations.

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 3.8.

My compiler 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.

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

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

Figure 3.8: 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 the compiler, 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 intermediate language, removing terms which are used for advanced language features like GADTs, as they are not supported by this compiler. Haskell

98 has hundreds of node types in its AST[6], whereas ILA has far fewer: this makes it far easier to transform.

3.4.1.1 Definition

```
data Expr = Var VariableName Type
          | Con VariableName Type
          | Lit Literal Type
          | App Expr Expr
          | Lam VariableName Type Expr
          | Let VariableName Type Expr Expr
          | Case Expr [VariableName] [Alt Expr]

data Literal = LiteralInt Integer
             | LiteralChar Char

data Alt a = Alt AltConstructor a

data AltConstructor = DataCon VariableName [VariableName]
                    | Default

data Binding a = NonRec VariableName a
               | Rec (Map VariableName a)
```

A Haskell program is lowered by this pass into a list of `Binding Expr`: a list of recursive or non-recursive bindings of expressions to variables.

One notable feature of ILA is that it carries type information: leaf nodes such as `Var` are tagged with a type. 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. The type annotations on ILA are not sufficient for such complete typechecking, but do allow for some sanity checks and are necessary for lower stages of the compiler such as code generation.

ILA is still quite high-level, so many of the language constructs have similar semantics to their Haskell counterparts. The main benefit in this lowering pass is to collapse redundant Haskell syntax (syntactic sugar) into a smaller grammar.

Most of these constructors have familiar usages, but some are more subtle:

- `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 vs 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, where the evaluated value of e is bound to each of the variables in vs. The additional binding variables can be useful when the scrutinee expression is reused within some number of the alts.

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

The alts in a `Case` expression, of the form `Alt` c b, match the value of evaluating the scrutinee against the data constructor c, then evaluates the b from whichever alt matched. `AltConstructor` represents the potential matches: either a data constructor with a number of variables to be bound, or a 'match-anything' default value.

3.4.1.2 Pattern Matching

Many syntax features in Haskell are just syntactic sugar, and are simple to desugar (list literals like `[1, 2]` are desugared to `1:(2:[])`). Others are slightly more verbose, such as converting `if x then y else z` expressions into `case x of { True -> y ; False -> z }` (`Bool`is just defined as an ADT in Haskell, there's no special language support for it).

Other language features are non-trivial to lower, such as the rich syntax Haskell uses for pattern matching. An example pattern match could be `Just x@(y:z) = Just [1, 2]`, binding `x = [1, 2]`, `y = 1`, and `z = [2]`. Multiple pattern matches can also be matched in parallel, as in function definitions:

```
1 f (x, Just y) = x + y
2 f (x, Nothing) = x
```

Additionally, pattern matches can occur in a number of places: pattern-binding declarations such as `let (x, y) = z in ...`, functions definitions like the example above, lambda expressions, and `case` expressions (`case Just 1 of { Nothing -> ... ; Just x -> ... }`). The heterogeneity of use sites demands a flexible approach to translating pattern matches that can be reused for each instance.

My initial implementation worked correctly for 'horizontal' patterns where there are a number of sequential patterns on independent variables, such as in `f (x, y) (Just z) = z`, but didn't support multiple disjoint patterns 'vertically', such as:

```
1 f Nothing = 0
2 f (Just x) = x
```

The implementation is somewhat fiddly, so a simplified declarative version 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:

$$
\begin{aligned}
\text{translate}(v : vs, v' : ps, b) &= \text{translate}(vs, ps, b)[v/v'] \\
\text{translate}(v : vs, (con\ ps') : ps, b) &= \texttt{case}\ v\ \texttt{of}\ \{\ con\ vs'\ \texttt{->}\ \text{translate}(vs' +\!\!+ vs, ps' +\!\!+ ps, b)\ \} \\
&\qquad \text{where } vs' \text{ are fresh variables} \\
\text{translate}(v : vs, (w\ \texttt{@}\ z) : ps, b) &= \text{translate}(v : vs, z : ps, b)[v/w] \\
\text{translate}([], [], b) &= b
\end{aligned}
$$

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:

translate([x], [y@(Just z)], (y, z))
→ translate([x], [Just z], (y, z))[x/y]
→ (case x of { Just u -> translate([u], [z], (y, z)) })[x/y]
→ (case x of { Just u -> translate([], [], (y, z))[u/z] })[x/y]
→ (case x of { Just u -> (y, z)[u/z] })[x/y]
→ 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' [10], 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 aren't 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`, there's no constructor corresponding to literals.

This is due to `case` expressions generally only making sense at a lower-level for data constructors, where there are a finite number of constructors which can be 'tagged' in some way in order to test a value for which constructor it uses. On the other hand, literals normally have a cumbersomely large (or infinite) number of 'constructors' (one can imagine the `Int` type, which is bounded, as being defined as `data Int = ... | -1 | 0 | 1 | ...`, but `Integer` cannot be defined in this way as it is unbounded).

As a result, pattern matches using literals are desugared into equality checks: the expression `case x of { 0 -> y ; 1 -> z ; _ -> w }` is essentially translated into the Haskell expression `if x == 0 then y else if x == 1 then z else w`, which is then lowered into ILA `case` expressions matching `True` and `False` as described above.

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, which is somewhat unusual. Instead, lambda terms must immediately be bound to a variable: this restriction is enforced by the `AnfRhs` term in the grammar below. This design choice simplifies code generation.

```
data AnfTrivial = Var VariableName Type
                | Con VariableName Type
                | Lit Literal Type

data AnfApplication = App AnfApplication AnfTrivial
                    | TrivApp AnfTrivial

data AnfComplex = Let VariableName Type AnfRhs AnfComplex
                | Case AnfComplex Type [VariableName] [Alt AnfComplex]
                | CompApp AnfApplication
                | Trivial AnfTrivial

data AnfRhs = Lam VariableName Type AnfRhs
            | Complex AnfComplex
```

An ILA program is lowered from a list of `Binding Expr` to a list of `Binding AnfRhs` by this pass. The translation is quite simple compared to the other lowering passes – most of the terms are similar to those in ILA (including carrying type information), with notable exceptions being the introduction of `AnfApplication`, which restricts application arguments to purely trivial terms, and `AnfRhs`, to enforce that lambda terms can only be bound to variables.

3.4.3 Intermediate Language B

ILB is the final intermediate language of this compiler and is inspired by GHC's STG (Spineless Tagless G-Machine) IL. ILB maintains the ANF style from ILA-ANF. It has a number of extremely useful features for code generation: the only term that performs any evaluation of an expression is the `ExpCase` $e\ t\ vs\ as$ term (which evaluates e then branches to one of the as), and the only term which performs any memory allocation is the `ExpLit` $v\ r\ e$ term, which allocates memory on the heap to represent a datatype/literal/uneveluated expression then evaluates e.

Additionally, this language makes lazy evaluation 'explicit', in the sense that expressions to be evaluated are always encapsulated within an `RhsClosure` (thanks to ANF style which names each subexpression) that can be implemented as a not-yet-evaluated thunk.

```
data Arg = ArgLit Literal
         | ArgVar VariableName

data Exp = ExpLit Literal
         | ExpVar VariableName
         | ExpApp VariableName [Arg]
         | ExpConApp VariableName [Arg]
         | ExpCase Exp Type [VariableName] [Alt Exp]
         | ExpLet VariableName Rhs Exp

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.

  `ExpConApp` also requires all its arguments to be present: it cannot be a partial application. Haskell treats datatype constructors as functions, so the following is a valid program:

```
1 data Pair a b = Pair a b
2 x = Pair 1
3 y = x 2
```

  At the implementation level however, functions and data constructors are necessarily very different, so distinguishing them within this IL makes code generation easier.

- Right-hand side terms in ILA (`AnfRhs`) were either lambda expressions or a let-binding/case expression/...– in ILB, the only right-hand side term is a `RhsClosure`. A closure with no arguments is essentially a thunk, a term that exists purely to delay computation of an expression, while a closure with arguments is the familiar lambda term.

  ILB's `RhsClosure` takes a list of arguments, whereas ILA-ANF's lambda terms only take a single argument (multiple-argument functions are just nested single-argument lambdas). This is another translation aimed at making code generation easier. Single-argument lambdas allow for simpler logic when handling partial application in higher-level languages, but is inefficient in implementation. ILB is the ideal IL to perform this switch from the high-level convenient-to-modify grammar to a lower-level efficient representation.

- ILB only allows variables in many of the places where ILA-ANF allowed variables, literals, 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 aren't many terms to lower into bytecode. Implementing the semantics of these terms in Java Bytecode is complex, however.

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 the library by me to add support for Java 8 features required by the compiler, as well as a number of smaller improvements: 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`, as well as the base class for all ADTs definable within the language (`BoxedData`, described later). The compiler 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 later.

3.5.1 Heap Objects

Still figuring out where/how to move the big Java code dumps to increase clarity and keep things focused while not making the explanation confusing.

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.

Thunks are represented simply as closures without arguments: all of the closure logic described below is the same between thunks and functions.

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

```
1 public abstract class HeapObject implements Cloneable {
2     public abstract HeapObject enter();
3
4     @Override
5     public Object clone() throws CloneNotSupportedException {
6         return super.clone();
7     }
8 }
```

The abstract `enter` method evaluates the object to WHNF (described in Section 2.2.2) and returns a reference to the result, and 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 that can't be defined as an Haskell ADT, such as `Int`. Any such type is a subclass of the `Data` class, which is itself a subclass of the `HeapObject` class that a rather boring implementation of the abstract `enter` method. Any literal is already in WHNF, so evaluation to WHNF is trivial:

```
1 public abstract class Data extends HeapObject {
2     @Override
3     public HeapObject enter() {
4         return this;
5     }
6 }
```

Here is an example literal implementation for `Integer`, Haskell's arbitrary precision integral value type. It is implemented using Java's `BigInteger` class to perform all the computation. The copious uses of underscores is explained in Section 3.5.2.

```
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 `_make_Integer(String)` function is used by the compiler 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 can't be expressed in Haskell. Section 3.5.4 on Hooks covers this aspect of code generation in more detail.

3.5.1.2 Datatypes

An Haskell ADT can be represented simply by a class generated by the compiler 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 any arguments given to the constructor. An example generated class[7] for the datatype `data Maybe a = Nothing | Just a` might be:

```
1  public class _Maybe extends BoxedData {
```

[7]The compiler doesn't generate a class described in Java source as shown, it just generates the bytecode for the class directly.

```
    public _make_Nothing() {
        _Maybe x = new _Maybe();
        x.branch = 0;
        x.data = new HeapObject[] {};
        return x;
    }
    public _make_Just(HeapObject val) {
        _Maybe x = new _Maybe();
        x.branch = 1;
        x.data = new HeapObject[] { val };
        return x;
    }
}
```

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 a given arity, without any arguments having been applied yet but possibly including values of free variables in scope of the 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 A.

Now it's been moved to the appendices, is the following still clear enough?

A function f (either defined locally or at the top-level) in Haskell of arity n_a and using n_{fv} 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_{fv}), and returns a `HeapObject` representing the result of applying the function.
- `_make_`f, which takes n_{fv} 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.

`Function`'s `freeVariables` field has type `HeapObject[]` as we know at initialisation time exactly how many free variables the function has, and it doesn't change. The `arguments` field

is an `ArrayList<HeapObject>` so that we can handle partial applications and over-applications by only adding arguments when they're applied.

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.

   If we're using a local closure like `let add1 = (+) 1 in add1 2 * add1 3` then `add1` will be a local `Function` object with `inner` pointing to the implementation of `(+)` and one applied argument (a `Data` instance representing 1). Both `add1 2` and `add1 3` will mutate the object to add the argument being applied (see the next step for details), which leads to the `Function` object after `add1 3` having 3 stored arguments.

   Cloning the function essentially maintains the same references to arguments and free variables, but creates new (non-shared) containers to hold them, avoiding the above issue.

   This is a shallow clone – if we used a deep clone, recursively cloning the arguments and free variables, then we'd 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 `addArgument` on the cloned object for each argument in the application, storing them 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 hasn't 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've stored, then ensuring the result has been evaluated to WHNF by calling `enter`, then returning it.
   - The function is over-applied. Although this case looks complicated, it's 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.

In Haskell, the vast majority of expressions are function applications by the time the source has reached ILB. To provide lazy semantics, each expression has to be evaluatable without forcing other expressions, so each function implementation is quite small. This results in a lot of functions being generated. Using anonymous classes to implement Haskell functions would result in hundreds or thousands of small Java classes, whereas using Java functions results in far fewer classes and more functions inside a single class.

3.5.2 JVM Sanitisation

Haskell[8], Java[9], and JVB[10] all allow different sets of strings as valid identifiers: for example, in Java and JVB `Temp` is a valid variable name, but in Haskell it's not (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 `Data`) and constructor names in the Haskell source (eg. `data Function = Function`).

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

- All names that have come from Haskell source are prefixed with an underscore, and any builtin classes are forbidden from starting with an underscore. This prevents name clashes.
- 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 is more restrictive than necessary, as JVB allows most unicode characters, but is a safe and simple defence against conflicts. Using $ symbols to mark the start and end of a sanitised character ensures that identifiers are uniquely decodable and prevents two distinct identifiers from clashing when sanitised (without delimiters, the valid Haskell identifiers π and `CF80` are sanitised into the same identifier: `_CF80`. With the delimiters, π is sanitised into `_$CF80$`).

3.5.3 Notable Instructions

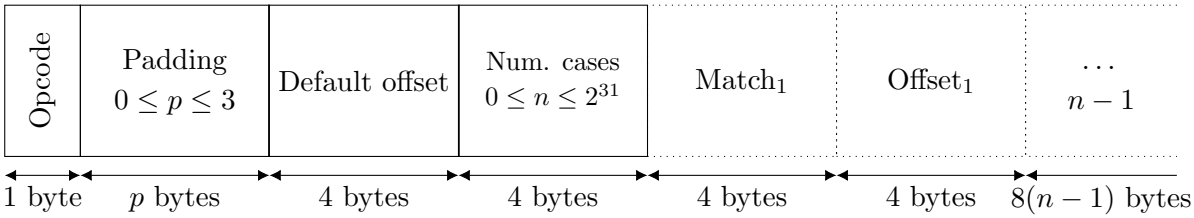
Candidate for complete deletion. Probably way too much detail, although it does demonstrate that codegen isn't trivial.

As mentioned earlier, the `hs-java` library is used to generate Java `.class` files from an in-memory representation of JVB, but support for a number of instructions were added to it: this section describes some of the more interesting ones that are heavily used by the compiler.

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

3.5.3.1 `lookupswitch`

The `lookupswitch` instruction is a low-level implementation of a `switch` statement in Java: it compares an `int` on the top of the stack ('scrutinee') with a set of values, jumping to an address paired with each value or to a default address if no values match. The interesting part of this instruction is that the length varies between usages:



- The first byte of the instruction is the opcode, `0xab`.
- Up to 3 bytes of padding follow, such that the next byte (the first byte of the 'Default offset' chunk) is an address that's a multiple of four bytes. JVM instruction addressing is local to a function, so the first instruction in a method has address 0.
- The subsequent 4 bytes constitute a signed `int` that gives the offset to jump to if the scrutinee doesn't match any of the values given in the instruction.

  For this description, 'offset' means 'signed relative address difference from the address of the opcode of the instruction to the address of the target': if a `lookupswitch` instruction is 30 bytes long, the offset used to jump to the immediately subsequent instruction would be 30, regardless of the position of the instruction in the file or the position of the offset within the instruction.
- The following 4 bytes form another signed `int` that's restricted to non-negative values, representing the number of value-offset pairs to match the scrutinee against, n.
- Next come n pairs of 4-byte values, each describing an `int` value to match the scrutinee against and an offset to jump to if the values match.

JVB uses variable-length instructions: the `nop` (no-op) instruction is just an opcode (`0x00`), a single byte, whereas the `goto` instruction is 3 bytes (the opcode, `0xa7`, followed by a two-byte operand forming the address to jump to).

The `lookupswitch` instruction is especially interesting because the length changes between uses of the same instruction: n and p effect the length of the instruction at runtime.

3.5.3.2 `invokedynamic`

Left because the code generation section looks like it's getting way too long: can write later if needed

- Basics of instruction, why needed (creation of `BiFunction` objects for `Function`).
- Boostrap methods, class attributes

3.5.4 Hooks

Many standard functions operating on primitive types like `Int` and `Char`, such as `(+)` and `(==)`, cannot be implemented in Haskell. These operations need to be implemented at the same level as `Int` is implemented, 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.

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

Figure 3.9: 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 can't provide an implementation for any of its member functions.

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

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

A hook is then added to the compiler 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.

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[11], 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 is instead given to higher-level optimisations.

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.

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

The example in Figure 3.10 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`.

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

```
f x = let
    g y = let
        z = 10
        x' = x + z
        y' = y + z
        in x' + y'
    in g
```

After

```
z = 10
f x = let
    x' = x + z
    g y = let y' = y + z in x' + y'
    in g
```

Figure 3.10

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.

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 doesn't need to initialise a new thunk containing that simple value each time it's used).

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 the compiler, 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 don't 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`. My compiler includes a standard library, which includes a subset of the definitions from the true Prelude.

With GHC, these are provided as importable modules from the `base` package, which is an implicit (opt-out) dependency of all packages. My compiler does not support modules or packages, instead using a single file per program, so the approach taken is to simply concatenate the file containing the standard library (`StdLib.hs`) with the input file: dependency analysis (described in Section 3.2.3) ensures that the definitions are compiled in a valid order, and unreachable code elimination (Section 3.6.3) removes uncessary 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.

  Typeclasses provide a way to implement strongly-typed ad-hoc overloading, allowing for the types of overloaded expressions to be known at compile time.
- After typechecking, types include typeclass 'contexts', eg. `Eq a => a -> a -> Bool`. These are implemented using dictionary passing, which introduces an extra parameter for each class constraint providing the 'implementation' of the typeclass for a type variable. This deoverloading process converts the context into formal parameters eg. `Eq a -> a -> a -> Bool`.
- 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 AST 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 represents evaluation of any kind is `Case`, and the only node representing memory allocation is `Let`.
- 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.

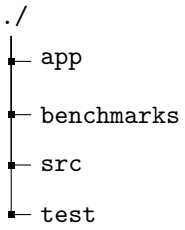
- 'Evaluating' a Haskell expression means reducing it to Weak Head Normal Form, where the expression is either a literal, fully/partially applied data constructor, or partially applied function. Arguments to functions or data constructors need not have been evaluated.
- Three optimisations have been 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 can't 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

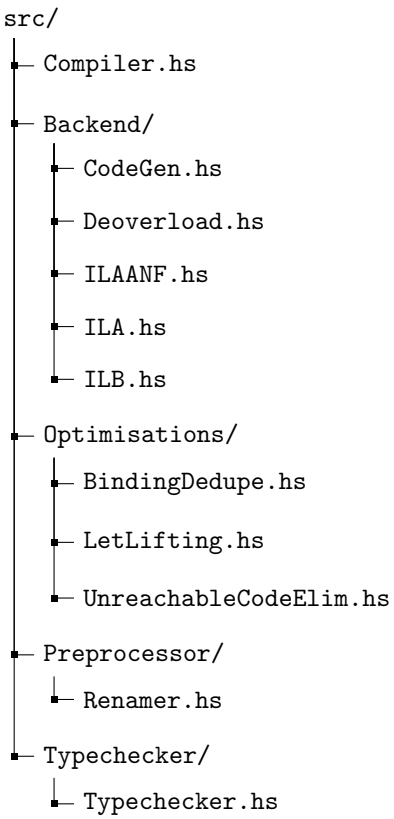
There's a loooot of whitespace here because of the tall directory layouts... I might have to just tweak spacing in the final draft

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. The important directories in the top-level of the repository are:



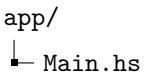
3.9.1 `compiler`

The vast majority of code written was in a library named `compiler`, which exposes an API for compiling Haskell programs – the code for this library is within the `src` directory. Files corresponding to major stages in the compiler pipeline are shown in the directory tree below. A number of source files have been omitted for clarity, as they contain utility functions or auxiliary code that is used by one of the major files. The `Compiler.hs` source file does not correspond to a single pipeline stage, but contains functions that string all of the other pipeline stages together.
This library contains 6344 lines of Haskell source code.



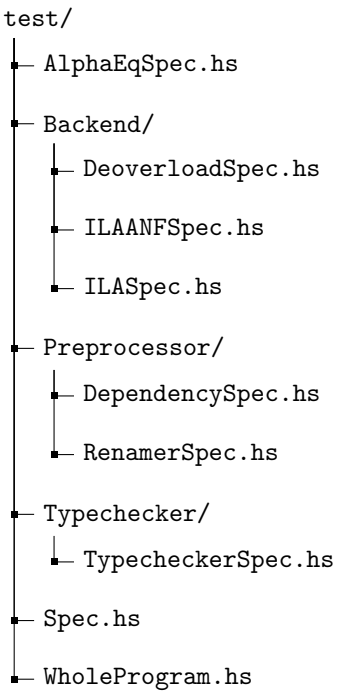
3.9.2 `compiler-exe`

The compiler executable is imaginatively named `compiler-exe`, and the single 97-line source file `Main.hs` resides 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.



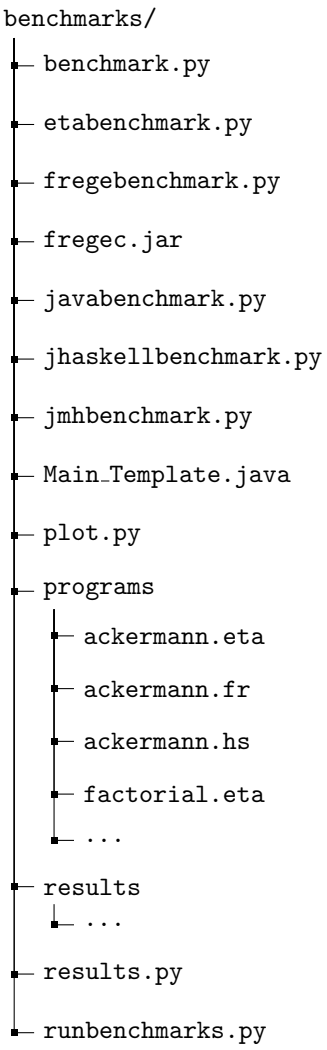
3.9.3 Tests

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 the performance of the compiler 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.
The `...benchmark.py` files define classes of benchmark and the `runbenchmarks.py` file creates objects of those classes to represent the various benchmarks and executes them.
The `programs` directory contains source files for the benchmarks, and the `results` directory contains the data obtained from the benchmarks.
There are 821 lines of Python source code for the benchmarking framework.

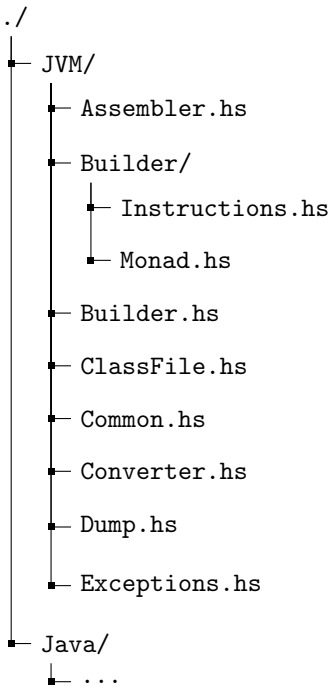


3.9.5 hs-java

The `hs-java` library[a] 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.

Find a place to reference the changes made, I feel like I've already duplicated this info once before.

[a] `https://hackage.haskell.org/package/hs-java`



`Assembler.hs` defines a representation of the JVB instruction set, `ClassFile.hs` defines a representation of Java Class files, and `Converter.hs` defines functions for converting the class file representation to and from a raw byte representation. The files within the `Builder` directory define a useful API for generating bytecode from Haskell using a monad named `GeneratorT`, and any other files simply contain utility functions.
The files under the `Java` directory define convenience variables for use when generating bytecode that refer to builtin Java classes such as `java.lang.Object`.

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.

Mention somewhere pros/cons of using JVB

4.2 Language Features

The planned subset of Haskell encompassed functions, arithmetic, booleans, lists, simple type-classes, and laziness. The following program is compilable using my compiler (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
```

Make a bigger deal out of the working bits? More demonstrations of laziness etc?

Successfully implemented extensions include 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:

Hard to give an example program using monads without ending up doing a monad tutorial!

```
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
    [] >>= f = []
    (x:xs) >>= f = (f x) ++ (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's no support for list comprehensions (eg. `[f x | x <- [1,2,3], even x]`) as they weren't a high priority feature.

4.2.0.1 Correctness

Correctness of the various stages of the compiler 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 the compiler works in a clean, reproducible environment and not just on my development system.

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

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

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

4.3 Performance

Although performance was not an important aspect of the success criteria, it's still interesting to evaluate the effectiveness of optimisations on the output program, and the performance of the compiler 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 expect it to perform better than both my compiler 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 my compiler, 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 my compiler are labelled to indicate whether or not optimisations have been applied.

4.3.1 Test Environment

Benchmarks were performed 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 Benchmark Approach

All of the compilers being compared output JVB, so a natural choice of benchmarking framework was the Java Microbenchmark Harness (JMH)[2]. This allows for accurate Java program benchmarking by handling JVM warmup, disabling garbage collection, etc. There are downsides to the framework though, notably that it doesn't record memory usage, and that it only appears to expose percentiles and histograms of the results through its Java API, not the raw data.

To measure compiler performance, a more naïve approach was taken: the execution times of 50 sequential compilations were recorded for each compiler.

For my compiler, the time taken to write the compiled class files to disk and compress them into a jar file was computed by measuring the difference between compilation runs which write

[2] `https://openjdk.java.net/projects/code-tools/jmh/`

to disk, and those which don't (using the `--no-write-jar` command-line-flag). This data is used in Figure 4.3.

4.3.3 Execution Speed

Might need to switch some of the plots to use a logarithmic y axis...

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 my compiler 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 wasn't expected.

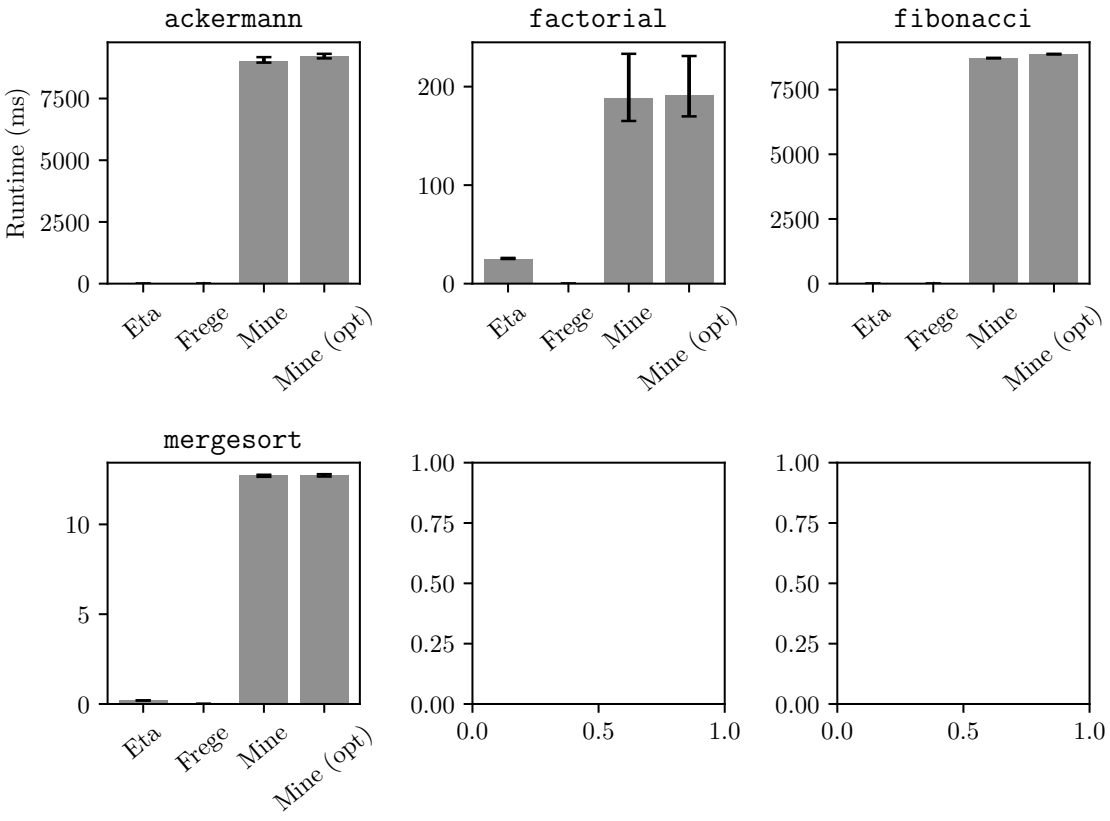


Figure 4.1: Median runtime in milliseconds of the benchmark programs. Error bars show 25th and 75th percentiles.

Update to use min with error bars showing 25th+50th quartile or similar

The inefficiency of programs output by my compiler 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.

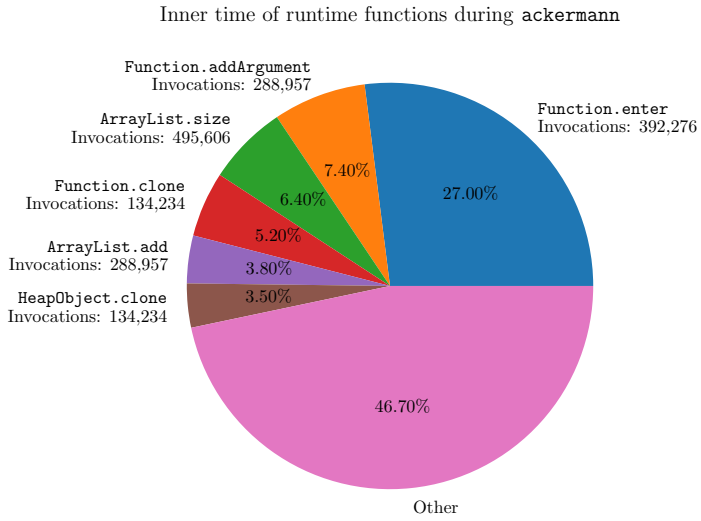


Figure 4.2: A breakdown of the percentage of execution time spent in each Java function in the generated code ('inner time') for the `ackermann` benchmark. Functions with inner time of less than 3% are grouped into a single wedge.

As described in the implementation chapter, my compiler 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[3] allowing objects of the classes to be invoked like anonymous methods.

This design choice was deliberate: before implementing my compiler, 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 implementation and having compared 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, which I expect plays a large role in the performance difference.

4.3.4 Compiler Performance

Figure 4.3 presents the minimum time taken to compile each benchmark program: my compiler is faster than both Eta and Frege to compile all benchmarks.

It's interesting that the compiler takes less time to process the input when performing optimisations than when not. This appears to be due to a significant amount of the compilation time being spent in compressing and writing the compiled classes into a jar file: the unreachable code elimination optimisation described in Section 3.6.3 can massively reduce the amount of code that reaches code generation, which in turn reduces the amount of bytecode that needs to be compressed and written to disk.

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`

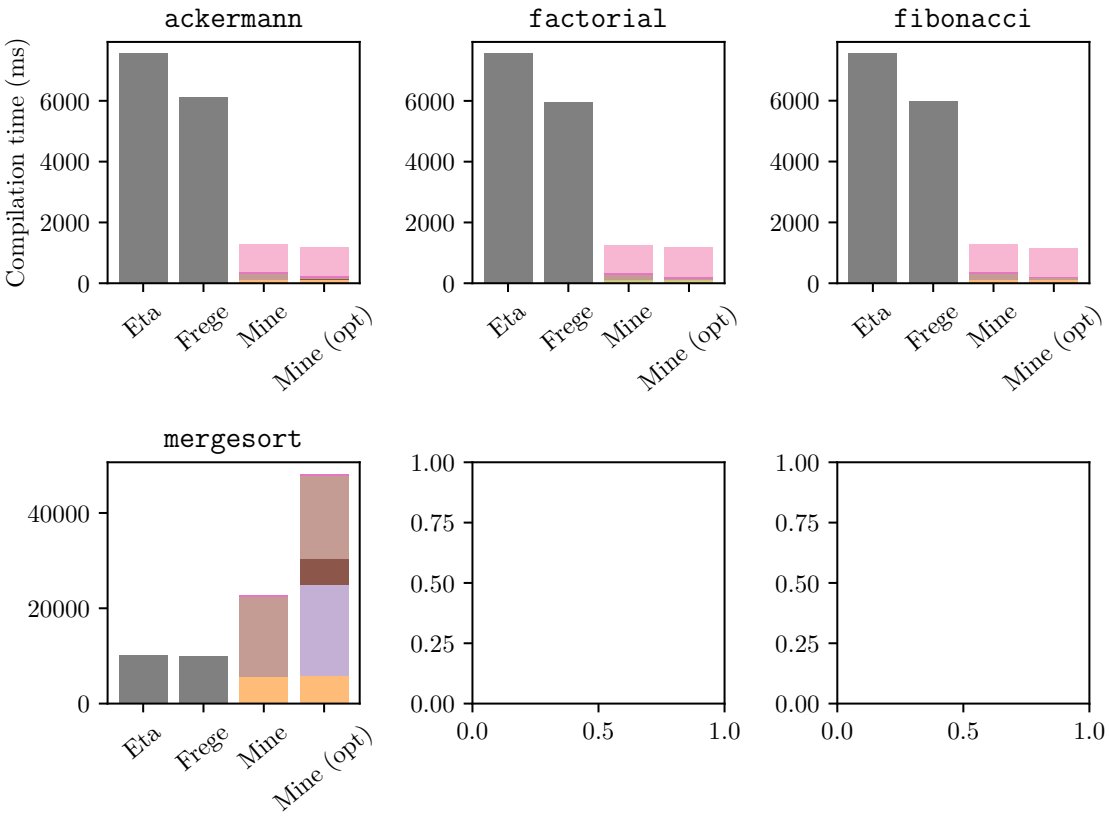


Figure 4.3: Minimum time taken to compile the benchmark program. The dark grey section for the times from my compiler is the time taken writing the compiled classes to a compressed jar file.

4.3.5 Executable Size

Data for Eta here is outdated for all but `factorial`, it's quite a bit smaller but still larger than Frege and mine.

Figure 4.4 displays the compiled size of each benchmark program after compilation by the various compilers. All three compilers generate a fixed-size set of class files that implement the logic of the Haskell program, have class files providing runtime support (for example, each compiler has an equivalent to the `Function` class described in Section 3.5.1), and usually have a class file for each datatype defined in the Haskell program. This metric includes specifically the class files implementing the logic from the Haskell code and the runtime files, but does not count the size of the implementation of datatypes or other files. This combination was chosen because all the compilers generate class files for all datatypes regardless of whether they're used by the program, and Frege and Eta implement many more datatypes than my compiler's standard library provides: this causes the size of their executables to be primarily due to datatype implementation, obscuring the size due to the program logic itself.

Also of note is that Eta and my compiler perform the Java equivalent of static linking, where the executable jar contains both the program logic and all the runtime files required, so that it can execute portably on any machine supporting Java. Frege performs the equivalent of dynamic linking, requiring the compiler's jar to be on the Java classpath when running its

output executables as the runtime files are all stored inside. This made accurately measuring executable size slightly trickier.

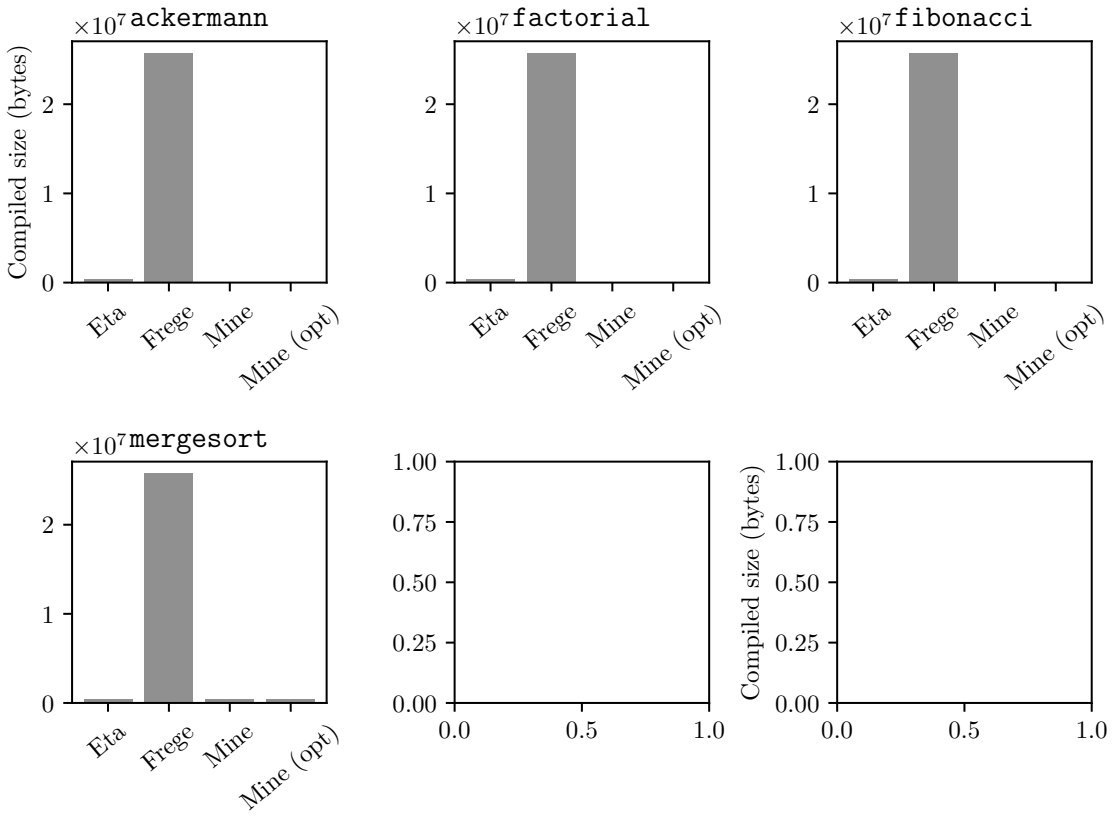


Figure 4.4: Compiled size of each benchmark program. This size includes the runtime system and the bytecode corresponding to the actual program.

4.3.6 Impact of Optimisations

Yeah this doesn't look good: check this is correct + work out why?

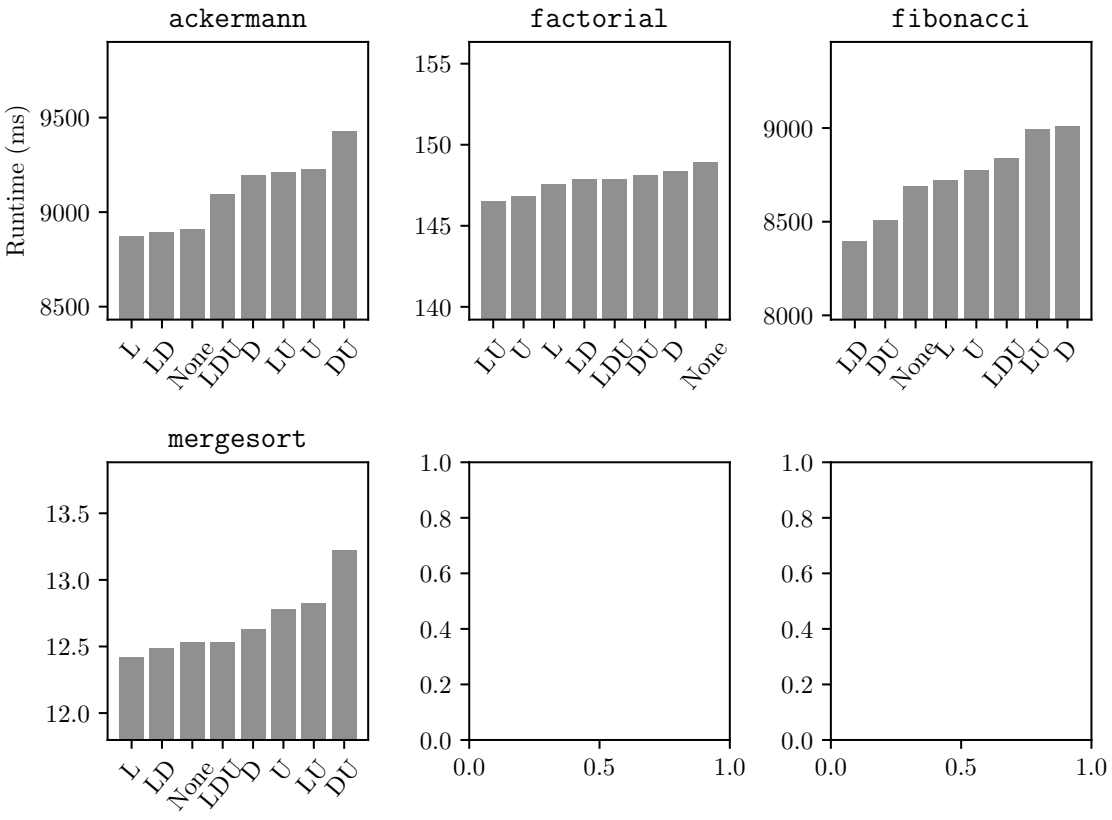


Figure 4.5: Runtime of the benchmarks after compilation with various optimisations: L stands for let-lifting, D for binding de-duplication, and U for unreachable code elimination.

4.4 Schedule

This is a WIP rehash of the stuff from the progress report explaining why the schedule got messed up, I feel it'll help explain why optimisations didn't get much splotlight.

The core features of Haskell are tightly coupled: simple features such as arithmetic operators require a significant level of support for other language features. For example, the `(+)` function relies on:

Typeclasses: `(+)` is defined by the `Num` typeclass in order to allow ad-hoc overloading.

Typeclass instances: The types that can be used as arguments to the overloaded functions, and the implementation of the overloads, are defined by typeclass instances.

Datatypes: The most common implementation of typeclasses involves translating classes into datatypes and instances into values of the datatype.

It would be possible to implement a function like `(+)` which only worked for integers and avoid all of the dependencies on other language features, but then the language simply wouldn't be Haskell: it would resemble a lazy variant of ML's semantics.

All of these language features are very expensive to implement, as they span multiple layers of the compiler: the type checker needs to be able to infer and check types based on the usage of these features, they need to be translatable into intermediate languages, and the code generator needs to be able to produce bytecode reflecting the semantics.

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 marked positive effect on the runtime behaviour of compiled programs, and while the programs are not as performant as might be desired, performance was never a critical goal for the compiler.

check!

I've 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 part of the compiler: 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 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 used by this compiler was an experiment: it differs from the approach taken by other Haskell to JVB compilers such as Frege and Eta. Having learned from the evaluation that this design is not very performant, it would benefit the compiler 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 them, 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 was unable to 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

optimisation, 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. [9], which would open up opportunities for supporting modern extensions to Haskell's type system such as GADTs and multi-parameter typeclasses.

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

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

[5] 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.

[6] Mark P. Jones. Typing haskell in haskell, 2000.

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

[8] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis, 1980.

[9] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein(x): Modular type inference with local assumptions, 2011.

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

Appendix A

The `Function` class

Line-break appropriately

```
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);
            }
```

```
42          }
43          else {
44              result = inner.apply(arguments.toArray(new HeapObject[0]),
                ↪  freeVariables).enter();
45              return result;
46          }
47      }
48
49      public void addArgument(HeapObject arg) {
50          arguments.add(arg);
51      }
52
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  }
```

Chapter 6

Project Proposal

Computer Science Tripos: Part II Project Proposal

An Optimising Compiler from Haskell to Java Bytecode

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

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

Introduction and Description of the Work

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

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

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

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

[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.