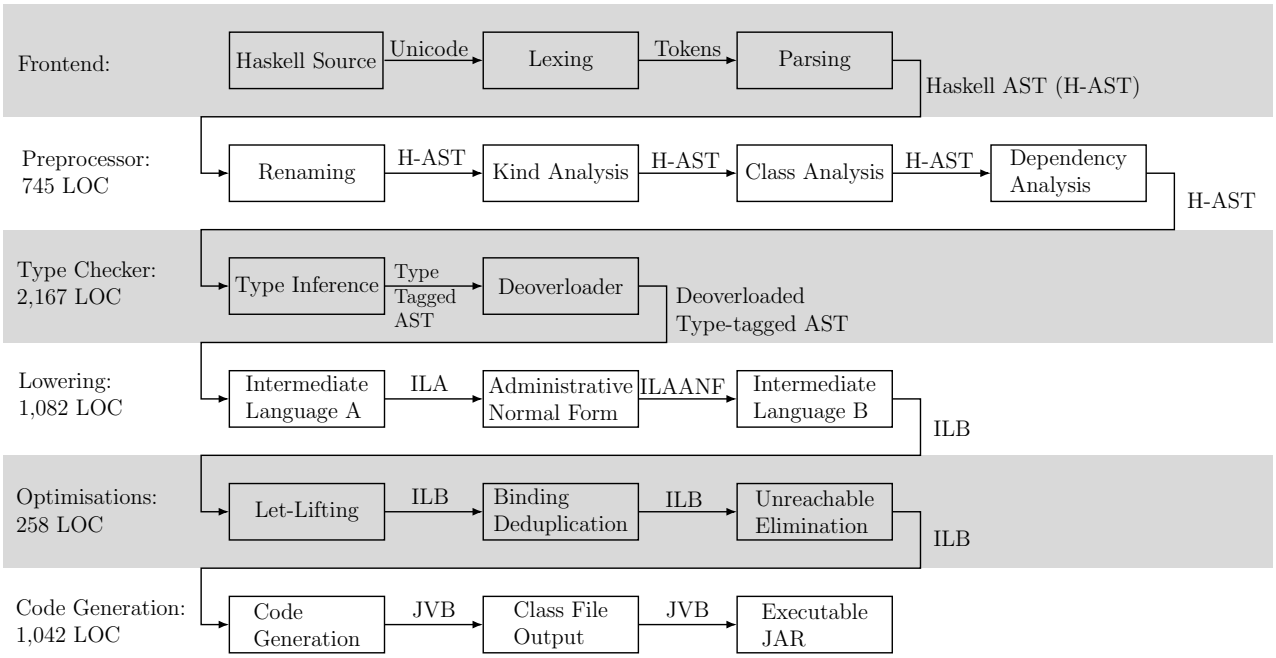

Figure 1

HJC consists of a number of stages and substages, as shown in Figure 1. A brief overview of each stage was given in the preparation chapter (Section **??**), but this chapter presents more in-depth descriptions. A summary is given at the end of the chapter, along with an overview of the code repository structure.

0.1 Frontend

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

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

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

Figure 2: An example of a multi-parameter typeclass

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

0.2 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.

0.2.1 Renaming

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

```
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 have not already been used).

Type constants such as `Bool` from `data Bool = False | True` and typeclass names like `Num` from `class Num a where ...` are not renamed: these names are already guaranteed to be unique by the syntax of Haskell, so renaming them would introduce unnecessary complexity.

0.2.2 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.

0.2.3 Dependency Analysis

When typechecking, the order of processing declarations matters: we cannot infer the type of `foo = bar baz` until we have inferred the types of `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, with an edge from A to B if any bound variables/type variables/type constants at A are free in B. It is important to distinguish between variables/type variables and type constants, as otherwise name conflicts could occur (as type constants are not renamed). This separation is upheld in code by using different types for each, and is represented in the dependency graph in Figure 3 by colouring variables red and constants blue.

The strongly connected components (SCCs) of the dependency graph correspond to sets of mutually recursive declarations, and the partial order between components gives us the order to typecheck each set. I use an existing Haskell library to compute the SCCs (`containers`[2]), which uses the algorithm presented by M. Sharir[**?**].

For example, from the dependency graph in Figure 3 we know that: we need to typecheck d_3, d_4, and d_5 together as they are contained within the same strongly-connected component so are mutually recursive; we have to typecheck d_2 last, after both other components. Typechecking declarations within the same component can proceed in an arbitrary order.

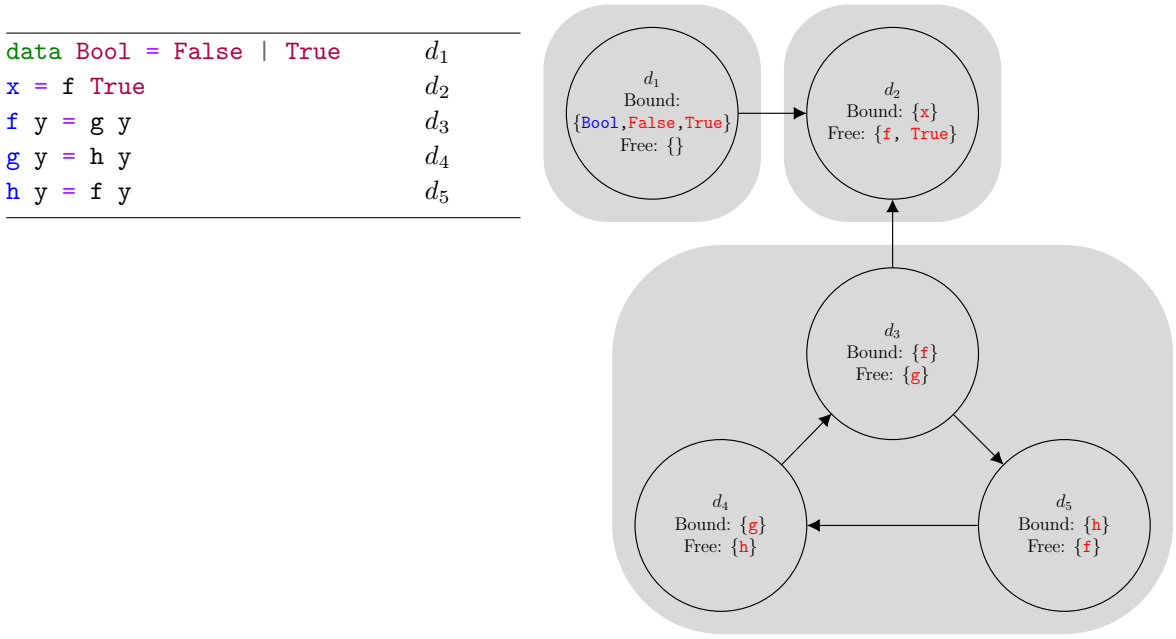

Figure 3: Code labelled with declaration numbers, and the corresponding dependency graph. Variables are in red text, type constants in blue. Strongly connected components are highlighted.

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

Special cases due to Typeclasses

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

- Typeclass member variables can be declared multiple times within the same scope. For example, the multiple declarations of `(+)` in Figure 4a do not conflict, while `x` in Figure 4b does conflict.

```haskell
class Num a where
    (+) :: a -> a -> a
instance Num Int where
    x + y = ...
instance Num Float where
    x + y = ...
```

(a)

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

(b)

Figure 4

  These declaration conflicts can be expressed as a binary symmetric predicate on declarations, as presented in Figure 5, where:
  - `Sym` x and `Type` x represent top-level declaration and type-signature declarations for a symbol x, like x `= True` and x `:: Bool`.
  - `ClassSym` x c and `ClassType` x c represent `Sym` x and `Type` x inside the declaration for a class c, like `class` c `where { x = True ; x :: Bool }`.
  - `InstSym` x c t represents a `Sym` x inside the declaration for a class instance c t, like `instance` c t `where { x = True }`.

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

  Using this table we can see that the multiple declarations for `(+)` in Figure 4a are `InstSym (+) Num Int` and `InstSym (+) Num Float` so do not conflict, while the declarations for `x` in Figure 4b are both `Sym x` so do conflict.

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

  Computing the true semantic dependency graph is too complicated to be done in this pass, so the problem is left and instead solved during the typechecking stage (Section 0.3.2).

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

Figure 6: The declaration of `foo` in `instance Foo [Bool]` semantically depends on the specific overload of `foo` defined in `instance Foo Bool`, and yet no dependency will be generated between the two instances as neither declaration binds `foo`.

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.

0.3 Type Checker

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

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

0.3.1 Definition of Types

```haskell
data TypeVariable = TypeVariable TypeVariableName Kind
data TypeConstant = TypeConstant TypeVariableName Kind

data Type = TypeVar TypeVariable
          | TypeCon TypeConstant
          | TypeApp Type Type Kind

data Kind = KindStar
          | KindFun Kind Kind
```

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

```
data TypePredicate = IsInstance ClassName Type

data Qualified a = Qualified (Set TypePredicate) a
type QualifiedType = Qualified Type

data Quantified a = Quantified (Set TypeVariable) a
type QuantifiedType = Quantified QualifiedType
type QuantifiedSimpleType = Quantified Type
```

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

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

During type inference, types are almost always polymorphic and often overloaded (`(==) :: forall a. Eq a => a -> a -> Bool`, `head :: forall a. [a] -> a`, ...). After deoverloading (Section 0.3.3), types are never overloaded. This difference is enforced in code by using `QuantifiedType` and `QuantifiedSimpleType` respectively.

0.3.2 Type Inference

The implementation is inspired by the approach given by Mark P. Jones[**?**] and uses similar rules as the Hindley-Milner (HM) type inference algorithm. There are three passes over the source AST, each of which traverses the SCCs of the AST in dependency order as previously described in Section 0.2.3.

1. The first pass tags each subexpression of the declarations in the SCC with a type variable, then uses rules similar to the HM inference rules to infer the value of each type variable from subterm type variables.

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

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

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

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

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

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

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

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

Restricted Polymorphism

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

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

Typeclass Order

A tricky part of the typechecking process is dealing with typeclasses, as dependency order is not semantically accurate for typeclass instance declarations (mentioned in Section 0.2.3).

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

The Ambiguity Check

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

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

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

Summary

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

0.3.3 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.

0.3.3.1 Dictionary Passing

The expression `1 + 2` has type `Num a => a`, where the expression is 'overloaded' on the type of `a` (as `(+)` is overloaded). This overloading can be implemented by adding extra parameters ('dictionaries') to all overloaded functions, which provide the implementation of any overloaded operators such as `(+)`. This is similar to virtual method tables in object oriented languages, except the v-table is being passed as an extrinsic argument, rather than being intrinsic to the object being operated on.

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

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

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

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

```haskell
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

```haskell
data Eq a = Eq (a -> a -> Bool) (a -> a -> Bool)

(==), (/=) :: Eq a -> a -> a -> Bool
(==) (Eq eq _) = eq
(/=) (Eq _ neq) = neq
```

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

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

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

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

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

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

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

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

This approach can be easily extended to typeclasses with 'superclasses' (by storing superclass dictionaries inside subclass dictionaries) and instance declarations with contexts (by making functions that construct dictionaries from other dictionaries), such as those shown in Figure 10.

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

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

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

Figure 10: Typeclass contexts allow for expressing the semantics that 'in order for `m` to be a `Monad`, it must also be a `Functor`'. Instance contexts allow for the semantics 'for any `a` that is an instance of `Eq`, `Maybe a` is also an instance of `Eq`.'

0.4 Lowering and Intermediate Languages

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

0.4.1 Intermediate Language A

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

0.4.1.1 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 [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)
```

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

- `Con` represents a data constructor such as `True` or `Just`.
- `App` is application as expected, but covers both function applications and data constructor applications (eg. `App (Con "Just" (Int → Maybe Int)) (Var "x" Int)`).
- `Case` e as represents a multi-way switch on the value of an expression e (the 'head' or 'scrutinee'), matching against a number of possible matches ('alts') from the list as. The body b of the first alt `Alt` c b whose constructor c matches the value of the scrutinee is executed.

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

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

0.4.1.2 Pattern Matching

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

Other language features are non-trivial to lower, such as the rich syntax Haskell uses for pattern matching. Figure 11 demonstrates two uses of patterns.

```
1  Just x@(y:z) = Just [1, 2]
```

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

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

(b) A multiline pattern match

Figure 11

My initial implementation worked correctly for 'horizontal' patterns such as in Figure 11a, but did not support patterns stacked 'vertically' as in Figure 11b.

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

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

In $\text{translate}(vs, ps, b)$, 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:

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

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' [**?**], 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.

0.4.1.3 Literals are not patterns

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

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

0.4.2 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 **??**). The full definition of ILA-ANF is given below, and reuses the definitions of `Binding` and `Alt` from ILA.

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

```
1  data AnfTrivial = Var VariableName Type
2                  | Con VariableName Type
3                  | Lit Literal Type
4
5  data AnfApplication = App AnfApplication AnfTrivial
6                      | TrivApp AnfTrivial
7
8  data AnfComplex = Let VariableName Type AnfRhs AnfComplex
9                  | Case AnfComplex Type [Alt AnfComplex]
10                 | CompApp AnfApplication
11                 | Trivial AnfTrivial
12
13 data AnfRhs = Lam VariableName Type AnfRhs
14             | Complex AnfComplex
```

0.4.3 Intermediate Language B

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

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

```
data Arg = ArgLit Literal
         | ArgVar VariableName

data Exp = ExpLit Literal
         | ExpVar VariableName
         | ExpApp VariableName [Arg]
         | ExpConApp VariableName [Arg]
         | ExpCase Exp Type [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.
- Right-hand side terms in ILA-ANF (`AnfRhs`) were either lambda expressions, or any other ILA-ANF expression – in ILB, the only right-hand side term is a `RhsClosure`. A closure with no arguments is a term that exists purely to delay computation of an expression.
- ILB only allows variables in many of the places where ILA-ANF allowed literals, variables, or 0-arity data constructors (like `True`). This is another step towards making laziness explicit, by keeping expressions simple so that only one step of the evaluation needs to happen at a time.

0.5 Code Generation

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

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

A number of Java classes have been written to provide the 'primitives' used by generated bytecode: including the implementation of Haskell's primitive datatypes like `Int` and `Char`. HJC is aware of these 'builtin' classes and uses a set of 'hooks' when generating code to provide Java implementations of Haskell functions. This is covered in more detail in Section 0.5.3.

0.5.1 Heap Objects

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

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

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

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

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

0.5.1.1 Literals

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

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

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

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

0.5.1.2 Datatypes

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

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

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

```
public class _Maybe extends BoxedData {
    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.

0.5.1.3 Closures

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

- Creation: construction of a new closure representing a function of given arity, without any arguments applied yet but possibly including values of free variables in scope of the 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 function f (either defined locally or at the top-level) in Haskell of arity n_a and using n_f free variables is translated into two Java functions:

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

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

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

Haskell function applications are lowered into bytecode that:

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

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

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

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

4. Invokes `enter` on the function object. This will reduce the object to WHNF, which has three cases:
   - The function is partially applied, so has not yet received all of the necessary arguments to be evaluated. Such a function is already in WHNF, so we can just return it.
   - The function has exactly the right number of arguments, so WHNF demands we reduce it. This is implemented by calling the `inner` function that performs the actual implementation of the Haskell function with the free variables and arguments we have stored, then ensuring the result itself is in WHNF by calling `enter`, then returning it.
   - The function is over-applied. Although this case looks complicated, it is really only two simple steps: We pretend we have an application of exactly the right number of arguments as in the above case, then instead of returning the result we cast it to a `Function` object and perform a normal function application with all the leftover arguments.

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

0.5.2 JVM Sanitisation

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

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

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

- All names that have come from Haskell source are prefixed with an underscore, and builtin classes are restricted from starting with an underscore.
- Any non-alphanumeric (Unicode) characters in a Haskell source identifier are replaced by their Unicode codepoint in hexadecimal, flanked on either side by $ symbols. This prevents variables from using characters disallowed by JVB while ensuring unique decodability.

0.5.3 Hooks

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

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

Figure 12: We want to be able to write an instance declaration in order to allow typechecking to see that `Int` is an instance of `Num`, but we cannot provide an implementation for any of its member functions.

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 code generation stage that generates functions named `_makeprimNumIntAdd` and `_primNumIntAddImpl`, as described in Section 0.5.1.3. The implementation of `_primNumIntAddImpl` is provided in the hook definition, and simply forwards its

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

arguments to the `_Int::add` function shown in Section 0.5.1.1. The functions generated by the hook are, at the bytecode level, indistinguishable from functions generated by any other expression so can be called by any other function without issue.

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

0.6 Optimisations

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

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

0.6.1 Let-lifting

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

The example in Figure 13 is somewhat contrived, but demonstrates the intent of the optimisation: `z` only has to be allocated and evaluated once each program, and `x'` only once per evaluation of `f` rather than each evaluation of `g`.

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 13

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

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.

0.6.2 Binding Deduplication

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

As the intermediate languages are mostly in ANF, there are a great number of let-bindings. In particular, simple expressions like `True && True` in Haskell are converted into `let v1 = True in let v2 = True in (&&) v1 v2` in ILB. This is necessary in general in order to provide lazy evaluation, but in common simple cases like this it introduces obvious redundancy.

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

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

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

0.6.3 Unreachable Code Elimination

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

This optimisation pass is extremely effective as it can eliminate any unused functions from the 'standard library' included with HJC , described in Section 0.7, ensuring that only code necessary for the execution of the program is compiled.

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

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

0.7 Prelude

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

HJC does not support Haskell modules, instead using a single file per program, so the 'import' approach taken is to simply concatenate the standard library (`StdLib.hs`) with the input file. Dependency analysis ensures that the definitions are compiled in a valid order, and unreachable code elimination (Section 0.6.3) removes unnecessary definitions to prevent program bloat.

0.8 Summary

To summarise the key ideas in the implementation:

- Lexing and Parsing is handled by a lightly modified third-party library.
- Variables and Type Variables in the source code are renamed to unique names to prevent naming conflicts.
- Dependency Analysis is necessary for determining the order in which to infer types for declarations, to ensure that the types of mutually recursive functions are correct.

- Type inference is implemented similarly to the Hindley-Milner algorithm, but typeclasses introduce a number of complications.
- After typechecking, types include typeclass 'contexts', eg. `Eq a => a -> a -> Bool`. These are implemented using dictionary passing, introducing an extra parameter for each class constraint providing the 'implementation' of the typeclass for a type variable.
- Intermediate Language A's (ILA) primary design goal is to collapse the hundreds of types of syntax node in the Haskell source AST into a significantly smaller core of less than 10 nodes. Type information is preserved for use by lower stages and for sanity checks.
- ILA-Adminstrative Normal Form (ILAANF) is a trivial subset of ILA to help guide ILA into ANF, to simplify later transformations.
- In Intermediate Language B (ILB), the only syntax node which causes evaluation of any kind is `Case`, and the only node causing memory allocation is `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.
- Three optimisations have been implemented: let-lifting raises let-bound variables to as high as possible within their expression; binding deduplication removes duplicate bindings in the same scope, to 'clean up' the redundant bindings which can be introduced by let-lifting; unreachable code elimination removes all code that cannot be reached from the program entrypoint, to reduce program size.

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

0.9 Repository Structure

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

0.9.1 `compiler`

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

0.9.2 `compiler-exe`

The compiler executable is imaginatively named `compiler-exe`, compiled from a single 97-line source file `Main.hs` in the `app` directory: it contains the application entry point and code for

handling command-line arguments, before passing control over to the `compiler` library for the actual compilation process.

0.9.3 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.

0.9.4 Benchmarks

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

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

0.9.5 `hs-java`

The `hs-java` library[?] 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.