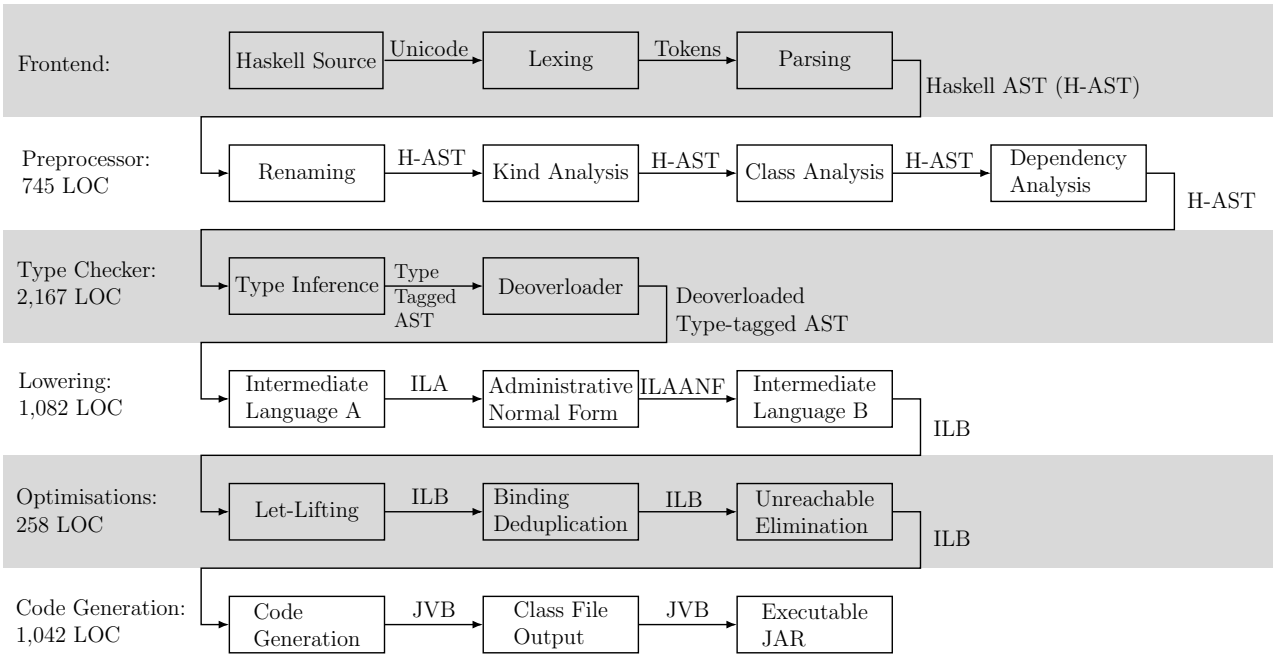


Figure 1

The compiler is comprised 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 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.

0.1 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.
- 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] 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

```haskell
class Convertable a b where
    convert :: a -> b
instance Convertable Bool Int where
    convert True = 1
    convert False = 0
```

Figure 2: An example of a multi-parameter typeclass

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 extract information while applying transformations.

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. 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:

```haskell
id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x
```

Additionally, variables and type variables are in different namespaces: the same token can refer to a variable and a type variable, even within the same scope. The following code is perfectly valid (but loops forever), despite the same name being used for a type variable and a variable:

```haskell
x :: x
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.

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 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[**?**].

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

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

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

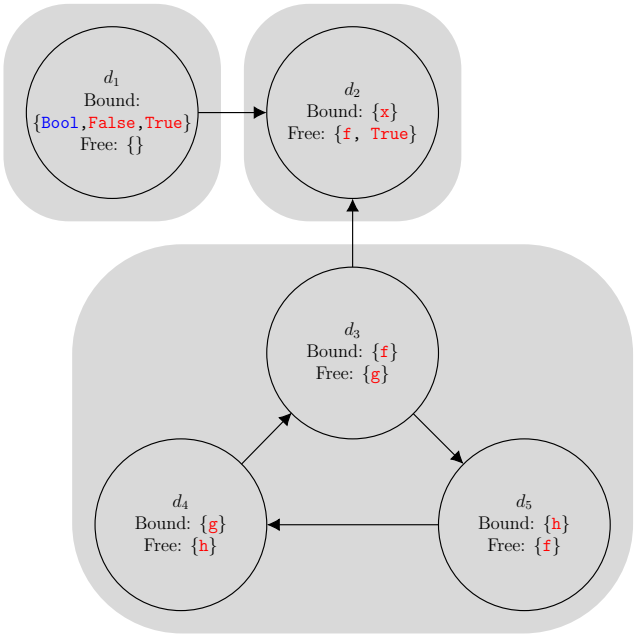


Figure 3: Code labelled with declaration numbers, and the corresponding dependency graph. Variables are in red text, type constants in blue. Strongly connected components are highlighted.

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 = ...
```

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

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

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

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

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.

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 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:

0.3.1 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 0.3.3), 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 ??.

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 AST in dependency order as previously described in Section 0.2.3.

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 $\texttt{((+)}^{t_1}\ \texttt{x}^{t_2}\texttt{)}^{t_3}$ 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:

0.3.2.1 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
```

0.3.2.2 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 0.2.3.

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.

0.3.2.3 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' [**?**]).

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...

0.3.2.4 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)`).

0.3.2.5 Improvements

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 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.

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 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 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 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`.'

0.4 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).

0.4.1 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.

0.4.1.1 Definition

```haskell
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

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

variables can be useful when the scrutinee expression is reused within some number of the alts.

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.

0.4.1.2 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' [**?**], 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 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.

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, 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.

0.4.3 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
```

```
10
11 data Rhs = RhsClosure [VariableName] Exp
```

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

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

  `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.

0.5 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.

0.5.1 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 **??**) 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.

0.5.1.1 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() {
```

```
        return this;
    }
}
```

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 0.5.2.

```
import java.math.BigInteger;

public class _Integer extends Data {
    public BigInteger value;
    public static _Integer _make_Integer(BigInteger x) {
        _Integer i = new _Integer();
        i.value = x;
        return i;
    }
    public static _Integer _make_Integer(String x) {
        return _make_Integer(new BigInteger(x));
    }

    public static _Integer add(_Integer x, _Integer y) {
        return _make_Integer(x.value.add(y.value));
    }
    ... // Analogous functions for subtraction and multiplication

    public static boolean eq(_Integer x, _Integer y) { ... }

    public static String show(_Integer x) { ... }
}
```

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:

```
ldc          210       // String 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 0.5.4 on Hooks covers this aspect of code generation in more detail.

0.5.1.2 Datatypes

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

```
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 any arguments given to the constructor. An example generated class[7] 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 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 **??**.

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

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.

0.5.2 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

[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`

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 `_$CF80$`).

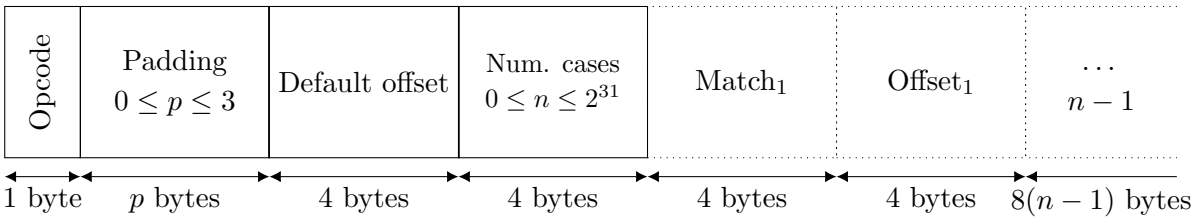
0.5.3 Notable Instructions

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.

0.5.3.1 `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.

0.5.3.2 `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

0.5.4 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 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 0.5.1.3. The implementation of `_primNumIntAddImpl` is provided in the hook definition, and simply forwards its arguments to the `_Int::add` function shown in Section 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.

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[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.

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 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 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

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

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.

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.

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 the compiler, 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 don't 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`. 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 0.2.3) ensures that the definitions are compiled in a valid order, and unreachable code elimination (Section 0.6.3) removes uncessary 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.

  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`.

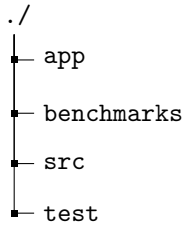
- Intermediate Language A's (ILA) primary design goal is to collapse the hundreds of types of syntax node in the Haskell source AST into a significantly smaller core of less than 10 AST nodes. Type information is preserved for use by lower stages and for sanity checks.
- ILA-Adminstrative Normal Form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

0.9 Repository Structure

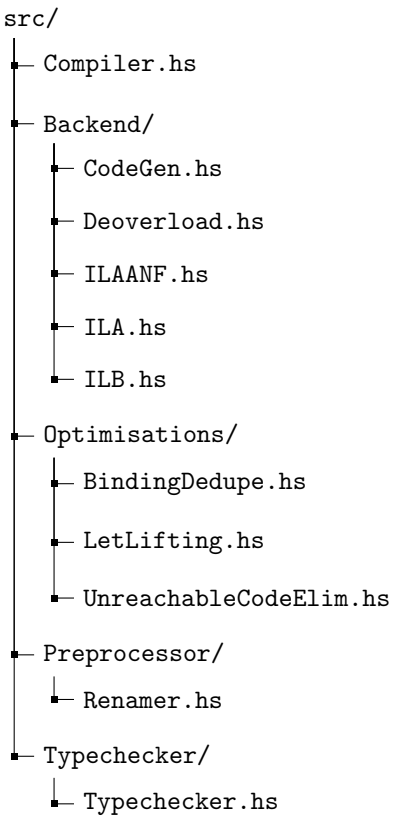
There's a loooot of whitespace here because of the tall directory layouts... I might have to just tweak spacing in the final draft

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:



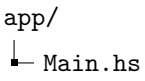
0.9.1 `compiler`

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.



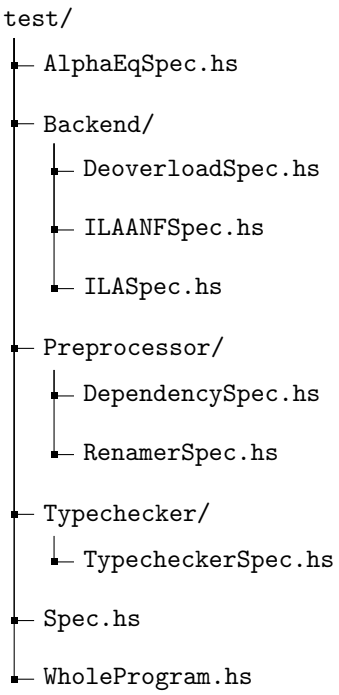
0.9.2 `compiler-exe`

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.



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.



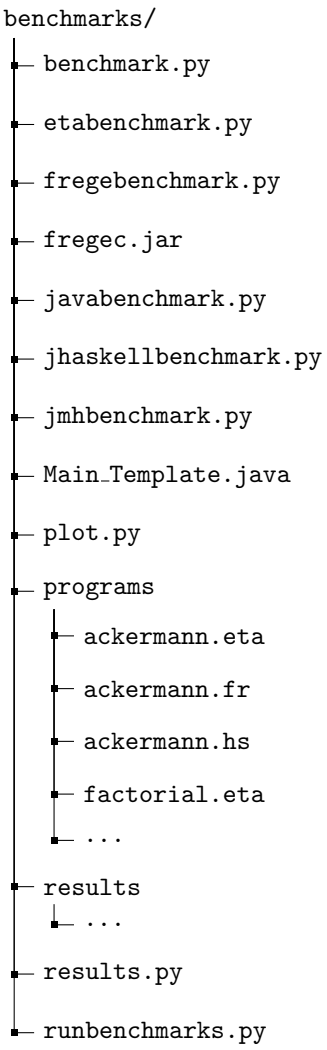
0.9.4 Benchmarks

The code for evaluating 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.

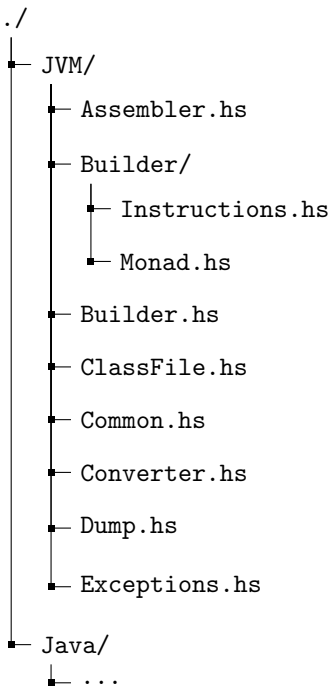


0.9.5 `hs-java`

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 (described ????????), 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`.