CMPS 253, Final Project
Sara Sholes, ssholes
12 Oct, 2017

Introduction:

In this project, I implemented a type-checker an evaluator for a subset of Haskell. Since Haskell is a functional language and heavily based on lambda calculus, it provided an opportunity to implement several of the concepts discussed in class, such as lambda functions and application, let statements, and case statements. Additionally, since Haskell does not require explicit typing, it meant that I had a chance to also implement the Hindley-Milner type inference algorithm.

The typechecker and evaluator can process standalone Haskell expressions in an input file specified by the user using the `evalFile` function. The typechecker and evaluator can handle the Int, Bool, and String base types, lists, the binary operators $+$, $-$, $*$, $\&\&$, $||$, $++$, the comparison operators $<$, $>$, $==$, if statements, lambda functions, function application, let statements (both function let binding and regular variable let binding, and with some support for let polymorpism and resursive expressions), and case statements with patterns composed of lists/cons, literals, variables, and wildcards.

For each standalone statement, the original statement is printed, and then the type is evaluated and printed out, or an error is printed if the expression is not well-typed. If the statement is well-typed, then the value is also evaluated and printed out.

Overall Workflow & Notation

In order to avoid implementing lexing and parsing by hand, I used `parseFileContents` function from the module Language.Haskell.Exts.Simple to traslate Haskell code into a lexed and parsed AST. The program is written to process Exp type statements, which is an abstract syntax tree representing an expression. The syntax of these statements can is described by Bertram (2016). Throughout this paper, 'expression' and 'term' are used interchangably to indicate an `Exp` statement representing abstract syntax tree representing an expression.

After parsing via `parseFileContents`, an individual expression is preprocessed using the `desugar` function, which does three things. It converts let function statements to a let variable statement with a lambda function on the right hand side. It also replaces all multi-input lambda functions with a series of nested single-input, single-output lambda functions. Finally, for each let binding it substitutes the variable with the expression on the right hand side of the binding in the body of the let statement. In other types of expressions, the function is just recursively called on any subexpressions. The result of preprocessing will also be an expression of Exp type. Rules that describe procrocessing are prefixed with a 'P', and are expressed in terms of small step semantics (since that more closely describes the specific steps that occur).

The preprocessed expression, along with an empty context, is then passed to the `typeEval` function for type evaluation, and will return the type (which may be a type variable, a concrete type, or a compound

type expression) and a global context containing the type constraints required by the expression and it's subexpressions. Rules that describe type evaluation are prefixed with a 'T'.

The Hindley-Milner algorithm is then applied via the `unify` function to the type constraints to check whether the expression is well typed, and if so, return an assignment of type variables to types. Any type variables in the original return type of the expression are then replaced with their assigned type (or assigned a parametric type if there is no such assignment) via the `getReturnType` function, and then that type (or an error if the expression is not well-typed) is printed.

If the term is well-typed, then the preprocessed expression is passed to the `eval` function, which will evaluate the value of the expression, and the value of the expression will then be printed out. There are a few cases, such application of as recursive functions, which may type check but will fail during value evaluation, and the value evaluation error will then be printed out. Rules that describe value evaluation are prefixed with a 'E' and are expressed in terms of large step semantics.

Context Structure

The user defined datatypes `GlobalContext` and `OverallContext` represent the context used during type evaluation.

```
data GlobalContext =
    GlobalContext [AllTypes] [(AllTypes, AllTypes)]
    | ErrorContext String
     deriving (Show)

data OverallContext =
    OverallContext [(Name, AllTypes)] GlobalContext
     deriving (Show)
```

As the name suggests, the type `GlobalContext` stores the global context which persists for entire program. This includes the list $\mathcal{X}$ of type variables which have already been used (represented by the `[AllTypes]` field in the `GlobalContext` constructor), and the list $\mathcal{C}$ of type equivalence restraints (represented by the `[(AllTypes, AllTypes)]` field). Since this information is global and evaluation of any subexpression can add new used type variables or new constraints, it is passed as both an input and an output in the `getType` function which is used to type evaluation. This also means that when several sub-expressions are evaluated within an overall expression, the `GlobalContext` output by the type evaluation of the first sub-expression is passed as the input `GlobalContext` to the type-evaluation of the next sub-expression, and so on (this will not be explicitly stated when describing type evaluation of particular expression types).

Additionally, it means that the `GlobalContext` output by the type evaluation of an overall expression is the `GlobalContext` returned by it's subexpression (or the `GlobalContext` returned by it's last subexpression, if there are multiple), as well as any additional used type variables and constraints added by the expression itself. Simply put, new type variables and new can be added to by the type evaluation of any

expression and sub-expression, and these additions to the global context will persist through all remaining type evaluation. For all constraint typing rules, the set of used type variables introduced by any set of subexpressions are assumed to be distinct. Because of this, the store of used type variables $\mathcal{X}$ is omitted from typing constraint rules, apart from an specifying if a fresh type variable is used by an expression.

Most type errors should only occur during the unification step, but in the case that the `getType` function is passed the abstract syntax tree of an expression that is not recognized, or the expression uses a variable which cannot be found in the variable store Γ, the `ErrorContext` constructor is used (the `String` field stores an appropriate error message) to create a global error context. If an error context is encountered at any stage of type evalution, that error is automatically returned as the result (meaning an expression will return an error is it any contains any sub-expressions which return an error).

The datatype `OverallContext` contains the `GlobalContext`, and in addition also includes the type variable store Γ (represented in the constructor by `[(Name, AllTypes)]`). Unlike the `GlobalContext`, the variable type store varies based on scope. The outer context which calls a sub-expression determines the global scope, but (with the exception of patterns discussed in section Case Statements and Appendix 1) type evaluation of the sub-expression does not affect the variable store of the enclosing expression. Thus, in order to ensure that the `getType` function has no unintended side effects on the variable store, `getType` takes `OverallContext` as an input (along with the expression being evaluated) and returns `GlobalContext` (along with the type of the evaluated expression).

Type Structure

The user-defined datatypes `AllTypes` and `BaseTypes` were created to respresent the different types that expressions can evaluate to, and `AllTypes` is the return type of the function `typeEval`.

```
data BaseTypes = IntType
    | BoolType
    | StringType
     deriving (Eq)

data AllTypes = Base BaseTypes
    | Error
    | TypeVar Int
    | ParamType Int
    | ArrowType AllTypes AllTypes
    | ListType AllTypes
      deriving (Eq)
```

For this project, three base types were defined to be Int, Bool, and String. To simiplify the code for unification, concrete types were grouped together via the `Base` constructor class, meaning that in order to check whether a type is a base type, you only need to check whether the `Base` constructor is used or not, rather that checking whether it is an Int, a Bool, or a String.

Since errors during type evaluation are indicated with the `ErrorContext` constructor, the AllType `Error` constructor is unneccesary, but is used to stand in for values whose type is unknown because of an error (such as the type of a variable expression for an unbound variable).

The `TypeVar` constructor, which represents variable types, takes an Int value in order to distinguish unique type variables. A list of previously used type variables is stored in the global context, and when a new type variable is generated, it is assigned the next unused integer value. Similarly, the `ParamType` constructor, which represents parametric types, also takes an Int value to distinguish unique parameter types. Parameter types are not used during type evaluation or type unification, they are just used in the final output result type, to distinguish from type variables that just haven't happened to be assigned yet.

The `ArrowType` constructor represents a single input, single output function type, where the first `AllType` specifies the type of the input parameter and the second `AllType` specifies the type of the output. The `ListType` constructor respresents a list type, and the `AllType` specifies the type of the elements of the list.

Value Structure

The user defined datatype `Value` is the type returned by the value evaluation performed by he `eval` function.

```
data Value = IntVal Integer
    | StringVal String
    | BoolVal Bool
    | ListVal [Value]
    | LambdaVal String Exp
    | ErrValue String
```

For the `IntVal`, `StringVal`, and `BoolVal` constructors, the second field stores the actual raw value of the result. The constructor `ListVal` is recursive; if it represents an empty List, the second field will be an empty list. Otherwise, it will be a list of two elements, where the first is a Value type representing the head value, and the second element is itself a `ListVal`-type Value. The first field of the `LambdaVal` constructor is the variable name of the input parameter, and the second is the function body expression.

Finally, the `ErrValue` constructor represents that an error occurred during value evaluation, and the second field contains the error message. For all expression types, if an error is encounted in any subexpression, the overall expression will also return an error with the same error message. Only cases where errors can originally occur during value evaluation will be mentioned.

Basic types

Literal values just evaluate to the their type (and if the type is a literal but is not Int, String, or Bool, and error is raised).

$$\text{CT-CONCRETE:} \quad \frac{(rawval, T) \in \{(intval, Int), (boolval, Bool), (stringval, String)\}}{\Gamma \vdash rawval : T \mid \emptyset}$$

while for variable references, the type of the variable is looked up in the store.

$$\text{CT-VAR:} \quad \frac{x : T \in \Gamma}{\Gamma \vdash t_1\ x : T \mid \emptyset}$$

In terms of value evaluation, literal values are assigned the appropriate value datatype with the given value. The rule for integers is given by

$$\text{E-LIT:} \quad \overline{intval \Downarrow Intval\ intval}$$

and the the value evaluation rules are equivalent for Bool and String literal values.

During value evaluation for all structures that can bind variable names (like lambda expressions and let expressions), the variables in the body of these structures are replaced by the terms on the right hand side of the binding before the body of the structure is evaluated to get a value (see the sections for these expressions for more details). Thus, if a variable is encountered during value evaluation, the variable must not be bound, and thus will return an error.

Lists

While a list may seem like a single simple expression, is actually behaves more like a compound expression of all it's elements, since the type of all elements in the list must match. The term $consOp$ is used instead of the actual Haskell syntax ":" to distinguish it from the syntax of the typing relation.

$$\text{CT-CONS:} \quad \frac{\Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2 \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_2 = Listtype\ T_1\}}{\Gamma \vdash t_1\ consOp\ t_2 : T_2 \mid \mathcal{C}'}$$

This rule specifies that the types of both sub-terms t_1 and t_2 are evaluated, and the resulting type is the type of the second term T_2. Additionally, the constraint $T_2 = Listtype\ T_1$ that the type T_2 of the second term must be a list type with element type of T_1, is added to the global context.

Lists themselves (of form $[t_1, t_2, t_3, ...t_n]$) are type checked in a similar manner.

$$\text{CT-EMPTY-LIST:} \quad \frac{X_n \text{ is fresh}}{\Gamma \vdash [\,] : Listtype\ X_n \mid \emptyset}$$

For empty lists [ ], a fresh type variable X_n is created, and the resulting type is a Listtype of this new type variable.

$$\text{CT-NONEMPTY-LIST:} \quad \frac{\Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash [t_2, ...t_n] : T_2 \mid \mathcal{C}_2 \quad \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_2 = Listtype\ T_1\}}{\Gamma \vdash [t_1, ...t_n] : T_2 \mid \mathcal{C}'}$$

For non-empty lists $[t_1, ...t_n]$, type evaluation is the same as a cons expression of the list's first element t_1 and the rest of the list $[t_2, ...t_n]$ (which will be the empty list if the original list was of length 1). In any following discussion of lists, only the cons rule will be stated, and the behavior of non-empty lists is equivalent to a cons between the head element and the list of the remaining elements.

In terms of value evaluation, an empty list simply produces an ListVal with an empty list as the value field

E-EMPTY-LIST: $\dfrac{}{[\,] \Downarrow ListVal\ [\,]}$

and for non-empty lists/cons operations, the head term t_1 is evaluated to get the resulting value v_1, the tail term t_2 is evaluated to get the resulting value v_2, and the result is a ListVal with a two element list as the value field, where v_1 is the first element and v_2 is the second element.

E-LIST-CONS: $\dfrac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2}{t_1 consOp t_2 \Downarrow ListVal\ (v_1\ consOp\ [v_2])}$

Note that the resulting ListVal produced is a recursive type.

Binary Operations

The type evaluation of other binary operations can be summarized as follows:

CT-ARITH-BIN-OPS: $\dfrac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = Int, T_2 = Int\} \quad op \in \{+, -, *\} \quad \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2}{\Gamma \vdash t_1\ op\ t_2 : Int \mid \mathcal{C}'}$

CT-BOOL-BIN-OPS: $\dfrac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = Bool, T_2 = Bool\} \quad op \in \{||, \&\&\} \quad \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2}{\Gamma \vdash t_1\ op\ t_2 : Bool \mid \mathcal{C}'}$

CT-STRING-BIN-OPS: $\dfrac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = String, T_2 = String\} \quad \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2}{\Gamma \vdash t_1\ +\!+\ t_2 : String \mid \mathcal{C}'}$

All three rules are implemented in terms of a single function `ctBoolArithExp` which takes the expected input/output type based on the operator type (Int, Bool, and String, respectively) as function inputs. The types of both of the input terms t_1 and t_2 are evaluated, and then constraints that the types T_1 and T_2 of both terms is the expected input/output type, are added to the global context. The resulting output type of the overall expression is the given input/output type.

Another similar rule is binary comparisons, which use the following typing constraint rule.

$$\text{CT-BIN-CMP:}\quad \frac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = T_2\} \quad op \in \{<, >, ==\} \quad \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2}{\Gamma \vdash t_1\ op\ t_2 : Bool \mid \mathcal{C}'}$$

In this case, the constraint $T_1 = T_2$ that the types of both sub-terms are equal, is added to the global context, and the resuting type is always a Bool. Otherwise, the mechanics are the same as for the previous three rules. Note that allowing T_1 and T_2 to be any types, so long as they are equal, is only valid because the operators $<$, $>$, and $==$ are defined for all three concrete types used in this project.

Value evaluation is similarly simple

$$\text{E-BIN-ARITH:}\quad \frac{op \in \{+, -, *\} \quad t_1 \Downarrow IntVal\ intval_1 \quad t_2 \Downarrow IntVal\ intval_2 \quad v_3 = IntVal\ (intval_1\ op\ intval_2)}{t_1\ op\ t_2 \Downarrow v_3}$$

$$\text{E-BIN-BOOL:}\quad \frac{op \in \{||, \&\&\} \quad t_1 \Downarrow BoolVal\ boolval_1 \quad t_2 \Downarrow BoolVal\ boolval_2 \quad v_3 = BoolVal\ (boolval_1\ op\ boolval_2)}{t_1\ op\ t_2 \Downarrow v_3}$$

$$\text{E-BIN-STRCAT:}\quad \frac{t_1 \Downarrow StringVal\ strval_1 \quad t_2 \Downarrow StringVal\ strval_2 \quad v_3 = StringVal\ (strval_1\ op\ strval_2)}{t_1\ +\!+\ t_2 \Downarrow v_3}$$

Both terms t_1 and t_2 are evaluated, the raw values are extracted from the value datatype constructor, the operation is performed on the two raw values. The resulting raw value is then wrapped with the constructor for the corresponding datatype. For comparison operations, the process is the same, except that the resulting raw value will be a boolean value, and will be wrapped in the BoolVal constructor.

$$\text{E-CMP-OP:}\quad \frac{(Const, rawval) \in \{(IntVal, intval), (StringVal, stringint), (BoolVal, boolint)\} \quad op \in \{<, >, ==\} \quad t_1 \Downarrow Const\ rawval_1 \quad t_2 \Downarrow Const\ rawval_2 \quad v_3 = BoolVal\ (rawval_1\ op\ rawval_2)}{t_1\ op\ t_2 \Downarrow v_3}$$

If Statements

The rule for type evaluation of if statements is

$$\text{CT-IF:}\quad \frac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{T_1 = Bool, T_2 = T_3\} \quad \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2 \quad \Gamma \vdash t_3 : T_3 \mid \mathcal{C}_3}{\Gamma \vdash \text{ if } t_1 \text{ then } t_2 \text{ else } t_0 : Bool \mid \mathcal{C}'}$$

Type evaluation is relatively simple; the types of all three sub-terms are evaluated. Then the constraint $T_1 = Bool$ that the conditional sub-term t_1 is a Bool type, and the constraint $T_2 = T_2$ that the then and an else sub-terms evaluate to the same type, are added to the global context.

Value evaluation can be summarized by the rules

$$\text{E-IF-TRUE:} \quad \frac{t_1 \Downarrow (BoolVal\ True) \quad t_2 \Downarrow v_2}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow\ v_2}$$

and

$$\text{E-IF-FALSE:} \quad \frac{t_1 \Downarrow (BoolVal\ False) \quad t_2 \Downarrow v_3}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \Downarrow\ v_3}$$

Essentially, the control statement expression is evaluated, and if it is a BoolVal with value True, the result is the result of evaluation of the 'then' case expression. If it is a BoolVal with value False, the result is the result of evaluation of the 'else' case expression. Value evaluation is lazy; if the control statement is True, then the value of the 'else' case never gets evaluated, and vice versa.

Lambda Functions

There are multiple ways of defining a function in Haskell. The simplest is a anonymous lambda function, which takes the form:

```
\x y -> x + y
```

where the variables to the left of the arrow are the input parameters, and the function body is to the right of the arrow.

The other main way to define a function in Haskell is via a let statement. For example, the previous function could be re-written as a named function with the let statement:

```
let foo x y = x + y
```

To ensure that typechecking for functions was consistent, I treated the second case as syntactic sugar, where an AST representing a let defined function was translated to a regular let statement with the function name as the variable name in the let binding, and the equivalent anonymous lambda function as the right hand side of the binding.

The preprocessing rules to do this are (with $fArrow$ standing in the for Haskell syntax `->` to distinguish from the arrows in the evaluation rule syntax)

$$\text{P-LET-LAMBDA-1:} \quad \frac{t_1 \longrightarrow t_1'}{\text{let } x\ a_1\ a_2\ ..\ a_n\ =\ t_1 \text{ in } t_2 \longrightarrow \text{ let } x\ a_1\ a_2\ ..\ a_n\ =\ t_1' \text{ in } t_2}$$

$$\text{P-LET-LAMBDA-2:} \quad \frac{}{\text{let } x\ a_1\ a_2\ ..\ a_n\ =\ t_1' \text{ in } t_2 \longrightarrow \text{ let } x = \ \backslash a_1\ a_2\ ..\ a_n\ fArrow\ t_1' \text{ in } t_2}$$

Additionally, while the AST returned by Language.Haskell.Exts.Simple treated multiple input lambdas as a single lambda expression with a list of inputs, it was simpler to define type checking rules in terms of only single input, single output lambdas. Thus, during preprocessing, the `curryLambdas` function converts each multiple-input lambda function to a series of nested single input lambda functions. The preprocessing rule is shown below.

P-CURRY-LAMBDAS: $$\frac{}{\backslash a_1\ a_2\ ..\ a_n\ fArrow\ t_1' \longrightarrow\ \backslash a_1\ fArrow\ \backslash a_2\ fArrow\ ..\ \backslash a_n\ fArrow\ t_1'}$$

After proprocessing is complete, all functions are assumed to be single-input, single-output. The type evaluation performed by `getType` on a lambda expression can be described by the following rule.

CT-LAMBDA: $$\frac{X_n \text{ is fresh} \quad \Gamma, x : X_n \vdash t : T \mid \mathcal{C}}{\Gamma \vdash\ \backslash x\ funArrow\ t : X_n \to T \mid \mathcal{C}}$$

First, a new type variable X_n is generated and added to the global context's record of used type variables. Then, the assignment $x : X_n$ of the input variable to this new type variable is added to the variable store context to evaluate the type T of the lambda's body. The type of the overall lambda expression will be an arrow type $X_n \to T$ from the input variable's type variable to the type of the lambda expression's body, and is returned as the result.

Since a lambda function is considered a value, the value evaluation for lambdas just translates from an expression to an equivalent Value datatype.

E-LAMBDA: $$\frac{}{\backslash x \to t \Downarrow\ LambdaVal\ x\ t}$$

Function Application

The type evaluation performed by `getType` on an application expression can be described by the following rule.

CT-APP: $$\frac{\begin{array}{c} X_n \text{ is fresh} \\ \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_1 = T_2 \to X_n\} \\ \Gamma \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2 : T_2 \mid \mathcal{C}_2 \end{array}}{\Gamma \vdash t_1\ t_2 : X_n \mid \mathcal{C}}$$

The type of the left and right sub-expressions t_1 and t_2 are evaluated. Then a fresh type variable X_n is generated to represent the overall type of the application expression, and is added to the global context. Finally, a new context rule $T_1 = T_2 \to X_n$ that the type of the left sub-expression must be an arrow type from the right sub-expression's type to the application expression's overall type, is added the global context.

The value of an application expression consists of evaluating the value of left term to get a `LambdaVal` type value. Then the right term t_2 is substituted for the input parameter x in the `LambdaVal` into the body t_3 of the `LambdaVal`. The new version of the lambda body sub-expression is then evaluated to get the value of the overall expression. The formal rule is shown below.

E-APP: $$\frac{t_1 \Downarrow v_1 \quad v_1 = LambdaVal\ x \to t_3 \quad [x \mapsto t_2]t_3 \Downarrow v_3}{t_1\ t_2 \Downarrow v_3}$$

Let Expression

Type and value evaluation of let expressions are implemented to support let polymorphism and type evaluation (but not value evaluation) of recursive let expressions.

In order to support let polymophism, the expression t_1 on the right hand side of a let binding is substituted for the variable x on the left hand side of the binding in all occurances in the body t_2 of the let expression. This allows each occurance of the right hand side to evaluate to different types if the expression itself is parametric, as discussed in Chapter 22 of Pierce.

All substitution of $x \mapsto t_1$ in all let expressions is performed with the `subExpr` method during preprocessing, which can be described by the preprocessing rules (where the t' indicates that a term has already been preprocessed)

$$\text{P-PATTLET-SUB1:} \quad \frac{t_1 \longrightarrow t_1' \quad t_2 \longrightarrow t_2'}{\text{let } x = t_1 \text{ in } t_2 \longrightarrow \text{ let } x = t_1' \text{ in } t_2'}$$

$$\text{P-PATTLET-SUB2:} \quad \frac{}{\text{let } x = t_1' \text{ in } t_2' \longrightarrow \text{ let } x = t_1' \text{ in } [x \mapsto t_1']t_2'}$$

The rule P-LET-SUB1 ensures that the body of the let expression gets preprocessed before the substitution occurs. This means that a variable which is bound in multiple nested expressions will be replaced by the expression it is bound to in the innermost revelant scope (ie, the expression `let x = 2 in let x = "a" x` will evaluate to `let x = 2 in let x = "a" "a"`, not `let x = 2 in let x = "a" 2`).

By the time type evaluation occurs and `getType` is called, all occurances of x in t_2 have been already replaced with t_1' to create the expression t_2'. Thus, type evaluation for the let expression follows the rule:

$$\text{CT-LET:} \quad \frac{\begin{array}{c} X_n \text{ is fresh} \\ \mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{X_n = T_1\} \\ \Gamma, x : X_n \vdash t_1 : T_1 \mid \mathcal{C}_1 \quad \Gamma \vdash t_2' : T_2 \mid \mathcal{C}_2 \end{array}}{\Gamma \vdash \text{ let } x = t_1 \text{ in } t_2' : T_2 \mid \mathcal{C}'}$$

Basically, when `typeEval` evaluates a let expression, the right hand side of the let binding is evaluated so that any new type variables it may generate and type constraints it may add are added to the global context, and then the body of the type of let expression is evaluated and returned as the result. This is to ensure correct typing even if the variable in the let binding is not used in the body of the let expression.

In addition, the fresh type variable X_n is generated to represent the type of the type of the right hand side, and the the assignment $x : X_n$ is added to the variable store context when evaluating the right hand side t_1 of the let binding. Additionally, the constraint $X_n = T_1$ that this new type variable must be the same type that as the type that the right hand side of the binding evaluates to is added to the global context. This allows for many recursive terms, like recursive function definitions, to type check correctly. However, in the case of polymorphic functions, the added constraint $X_n = T_1$ ensures that once x is bound to a specific type in the right hand side of the binding, x cannot evaluate to a different type in any other occuannces the right hand side term. This means that while the expressions

```
let len a = case a of
```

```
                [] -> 0
                h:t -> 1 + (len t)
            in len [1, 2, 4, 5]
```

and

```
let len a = case a of
                [] -> 0
                h:t -> 1 + (len t)
            in len [1, 2, 4, 5]
```

will correctly evaluate to type Int, the expression

```
let len a = case a of
                [] -> 0
                h:t -> 1 + (len t)
            in (len [1, 2, 3, 4]) + (len ["a", "b", "c"])
```

will encounter an error in unification, though in the actual Haskell language, this expression should should also type check to type Int.

Since this issue only arises if there is an occurance of the variable x in the right hand side of the binding, this issue does not affect non-recursive polymorphic terms.

In terms of value evaluation, since the expression has already been preprocessed by the time it is passed to the `eval` function for value evaluation, all let bindings have already been substituted into their corresponding body expressions. Thus, the result will simply be the result of evaluating the let body.

E-LET: $$\frac{t_2' \Downarrow v_2}{\text{let } x = t_1 \text{ in } t_2' \Downarrow\ v_2}$$

Though type checking does work for recursive expressions (with the exception of re-used polymorpic expressions previously discussed), in many cases value evaluation will not work for recursive expressions, since substituting the entire body of a recursive let expression would result in an infinite loop of recursive calls in the `subExpr` function. Standalone recursively defined functions will evaluate to their expected value, but recursively defined functions within a application expression will throw an error during value evaluation, since the body of the function is only evaluated once the input parameter is bound. T

There is possibly a work around for functions that converge, that involves substituting in only the relevent parts of the expressions body until a base case is reached and discussed in Chapter 22 of Pierce, but I did not get a chance to implement this.

Case Statements

For this project, case statements with patterns composed of lists, variables, and literal values can be type-checked and evaluated. Evaluation of case statements is described by the statement below ($caseArrow$ is

used to to stand in for the Haskell syntax -> used in to seperate the pattern and body of case statement alternatives)

$$\text{CT-CASE:}\quad \frac{\begin{array}{c}\mathcal{C}' = \mathcal{C}_0 \cup (\bigcup_{i\in 1..n}(\mathcal{C}_{ib} \cup \mathcal{C}_{ia} \cup \{T_0 = T_{ip}\} \cup \{T_i = T_{i+1} \text{ if } i < n\})) \\ \Gamma \vdash t_0 : T_0 \mid \mathcal{C}_0 \\ \text{for each } i,\ \Gamma \vdash p_i : T_{ip}, \Gamma'_i \mid \mathcal{C}_{ia} \quad \Gamma'_i \vdash t_i : T_i \mid \mathcal{C}_{ib}\end{array}}{\Gamma \vdash \text{case } t_0 \text{ of } p_i\ caseArrow\ t_i^{i\in 1..n} : T_n \mid \mathcal{C}'}$$

First, the type of the control expression is evaluated. Then, for each case alternative, the type of the pattern is evaluated. Pattern type evaluation rules are similar to regular typing rules (see Appendix 1), except that when a variable is encountered in a pattern, instead of looking up variables in the variable store, a fresh type variable is generated, and will be added to the variable store context. The other difference is that, because these bindings need to be included in the variable store context for evaluating the body of the case alternative, pattern type evaluation returns the variable store context in addition to the type of the pattern (denoted by the format $\Gamma \vdash p : T, \Gamma'...$). Pattern binding was implemented this way to allow for recursive pattern matching evaluation, which allows the any of the infinite possible list matching patterns (like `a:b:c:tail`) to be matched.

After the pattern has been type evaluated, the constraint $T_{ip} = T_0$ that the case alternative pattern's type T_{ip} is equivalent to control expression type T_0 is added to the global context.

Next, given the new variable store context generated by the case alternative's pattern type evaluation, the body of the case alternative is evaluated. Finally, add constraint $T_i = T_{i+1}$ if $i < n$ that the type of each case alternative's body (except for the last case) is equivalent to that of the next case alternative.

Case evaluation can be described by the rule

$$\text{E-CASELIST:}\quad \frac{pAlt(p_j, t_0, t_j) = \text{ Just } t_{jnew} \quad \text{for all } k \in 1..(j-1),\ pAlt(p_k, t_0, t_k) = \text{ Nothing} \quad t_{jnew} \Downarrow v_j}{\text{case } t_0 \text{ of } (p_i\ caseArrow\ t_i)^{i\in 1..n} \Downarrow v_j}$$

For each ith case alternative (proceeding from the first case alternative in order), the control expression t_0, the case alternative pattern p_0, and the case alternative body t_i are all passed to the the pAlt rule (which corresponds to the `ePat` method in the code). Depending on the structure of the case alternative pattern p_i, the structure of t_0 is checked to see if it matches (see Appendix 2 for formal pattern matching rules).

For a variable pattern x, t_0 will always match and the variable x in the case alternative's body will be replaced with the term t_0. A wildcard _ pattern always matches the given expression t_0. If the pattern is a literal value, then the input term t_0 will be evaluated and it matches if it evaluates to a literal with the same value as the pattern literal.
For lists, p_j and t_0 match if both are empty lists. If they are both cons operations (or non-empty lists or some combination of the two), then `ePat` method is recursively called to match the heads and tails of the pattern and control expression, and binding any variables found in the pattern to the corresponding sub-expression in the control expression.

The 'Maybe' monad is used to indicate whether the a case alternative sucessfully matched the control statement or not. If the pattern of a case alternative and the control expression match, then the 'Just' constructor and the body expression t_{jnew} of the case alternative (with substitution of any variables bound by the pattern) is returned. If they do not match, then 'Nothing' is returned.

If the case alternative match succeeds and a 'Just' t_{jnew} expression is returned, then the expression t_{jnew} is evaluated and it's resulting value is returned as the overall result value. If the case alternative match fails and 'Nothing' is returned, then the next case alterative is evaluated. If the last case alternative is reached and none of the cases match, then an error is returned.

The reason why value evaluation of both the case alternative's body and (other than for case alternatives with literal value pattern expressions) the control expression is delayed is to be able to use the `subExpr` function which takes expressions as inputs for binding in any recursive calls to the `ePat` method for subpattern matching.

Type Inference

Since Haskell is not explictly typed, type inference must be performed in order to know the types of all terms, and ensure that a term can type check. After type evaluation performed by the `getType` function, in addition to the overall type (possibly a type variable on compound type containing a type variable) of the term, the global context which contains the set of type constraint added during type evaluation is also returned.

If an error was not thrown during type evaluation, the basic Hindley-Milner unification algorithm, described by Milner (1978), is applied using the `unify` function. The `unify` function takes as input the set of constraints from type evaluation, and will return an assignment of type variables to other types if the expression is well-typed, and will return a unification error otherwise. To encompass these two possiblilties, the following datatype `UnifyResult` was created.

```
data UnifyResult =
    TypeVarAssignment [(AllTypes, AllTypes)]
    | UnificationErr String
     deriving (Show)
```

Given an input list of constraints, the algorithm removes a constraint from the constraint list. If both types in this constraint are identical, then the constraint is removed and the algorithm is recursively called on the rest of the constraints. If one type in the given constraint is a type variable, then replace all occurrences of that specific type variable in the rest of the constraints with the other type in the given constraint. Then recursively call the algorithm, and add the assignment of the type variable to the other type to the resulting set of assignments of type variables. To handle unification errors, the method `conTVAssign` is used to add a new type variable assignment to the set of type variable assignments returned by a recursive call. If the recursive call returned a unification error, then the overall result will be a unification error, and otherwise the overall set of all type assignments in returned. Thus, type unification errors in any recursive calls will be

mean that the overall result will also be a unification error.

If both types in the given constraint are compound types, then the constraint is replaced with a constraint equating the corresponding subtypes (ie. the constraint $T_0 \rightarrow T_1 = S_0 \rightarrow S_1$ would be replaced with the the two new constraints $T_0 = S_0$ and $T_1 = S_1$ in the recursive call, or likewise $Listtype\ T_0 = Listtype\ S_0$ would be replaced by $T_0 = S_0$), and the algorithm is recursively called on this new set of constraints. If the constraints does not satisfy any of the previous conditions, then a unification error is returned.

This process continues until all constraints have been removed or a unification error has been returned.

After unification is complete, any type variables that may be left in the return type of the overall expression can be substituted for concrete types (here, concrete types refers to the base types Int, Bool, String, and any compound list or arrow types made up of these basy types). The function `getReturnType` takes the resulting assignment and the overall expression's return type, and substitutes in other types for the type variables in the return type.

When a concrete base type (Int, String, or Bool) is encountered, the `getReturnType` function will return the type as is. For any compound types (a Listtype or Arrow type), `getReturnType` will recursively be called on all of the component types, and the types resulting from the resursive calls are then re-assembled back into the original compound type. When a type variable is ecnountered, `getReturnType` will look up the type variable in the assignment list, and then recursively calls `getReturnType` on it's assigned type, and then return the result of this recursive call. When a type variable is encountered that is not assigned to any other type, then that means the type variable is not constrained to a particular concrete type. Thus, the variable type is a parametric type. Since there can be multiple distinct parametric types in a given expression, the function `getReturnType` will assign to a parametric type with a number to identify it. The identifying number for a parametric type is assigned to be the identifying number of the variable type it was made from. Note that in this function, it is possible for type variables to be assigned to other type variables, which is acceptable, since the function is resursively called on the other variable type (it just means that both will be assigned the same overall type).

Conclusion

Overall, the program can effectively type check and evaluate many standalone expressions of Haskell code in an input file. The program can be run by compiling main.hs, and then running:

```
evalFile relative_path/testfile.hs
```

A set of test files testing different types of expressions are inculded in the 'testcases' folder.

The most challenging aspects of the project was implementing type inference and value evaluation of case statements. The overall program structure, such as the user-defined datatypes representing types, context, and values, required several stages of redesign in order to work for all expressions and to most cleanly follow the logical type and evaluation rules. Future work could include extending pattern matching in case statements to more types of patterns, and also to make the implementation of pattern handling in type evaluation and value evaluation more consistent. The main improvement which could be made would enable

value evaluation for recursive types.

Bibliography:

[1] Benjamin Pierce. "Types and Programming Languages", MIT Press, 2002.

[2] Felgenhauer, Bertram. "haskell-src-exts-simple-1.19.0.0: A simplified view on the haskell-src-exts AST." (2016). https://hackage.haskell.org/package/haskell-src-exts-simple-1.19.0.0/docs/Language-Haskell-Exts-Simple-Syntax.html.

[3] Diehl, Stephen. "Hindley-Milner Inference" (n.d.). http://dev.stephendiehl.com/fun/006_hindley_milner.html.

[4] Milner, Robin. "A Theory of Type Polymorphism in Programming." (1978). Journal of Computer and System Sciences, 17. p348-375.

Appendix 1: Type Evaluation Pattern Rules

$$\text{CT-PATT-VAR:}\quad \frac{X_n \text{ is fresh} \quad \Gamma' = \Gamma, x : X_n}{\Gamma \vdash x \ : \ X_n, \Gamma' \ \mid \ \mathcal{C}}$$

$$\text{CT-PATT-LIT:}\quad \frac{(T, rawval) \in \{(Int, intval), (String, stringval), (Bool, boolval)\}}{\Gamma \vdash rawval \ : T, \Gamma \ \mid \ \mathcal{C}}$$

$$\text{CT-PATT-WILDCARD:}\quad \frac{X_n \text{ is fresh}}{\Gamma \vdash \_ \ : X_n, \Gamma \ \mid \ \mathcal{C}}$$

$$\text{CT-PATT-EMPTY-LIST:}\quad \frac{X_n \text{ is fresh}}{\Gamma \vdash [\,] \ : Listtype\ X_n, \Gamma \ \mid \ \mathcal{C}}$$

$$\text{CT-PATT-NONEMPTY-LIST:}\quad \frac{\mathcal{C}' = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{T_2 = Listtype\ T_1\} \quad \Gamma \vdash p_1 : T_1, \Gamma_1 \ \mid \ \mathcal{C}_1 \quad \Gamma_1 \vdash p_2 : T_2, \Gamma_2 \ \mid \ \mathcal{C}_2}{\Gamma \vdash p_1\ consOp\ p_2 \ : Listtype\ T_1, \Gamma_2 \ \mid \ \mathcal{C}'}$$

Appendix 2: Value Evaluation Pattern Rules

$$\text{E-ALT-VAR:}\quad \frac{t_{jnew} = [x \mapsto t_0]t_j}{pAlt(x,\ t_0,\ t_j)\ =\ \text{Just}\ t_{jnew}}$$

$$\text{E-ALT-LIT-1:}\quad \frac{(rawval, Const) \in \{(intval, IntVal), (boolval, BoolVal), (stringval, StringVal)\} \quad t_0 \Downarrow Const\ rawval_j \quad rawval_j = v_p}{pAlt(v_p,\ t_0,\ t_j)\ =\ \text{Just}\ t_j}$$

$$\text{E-ALT-LIT-2:}\quad \frac{(rawval, Const) \in \{(intval, IntVal), (boolval, BoolVal), (stringval, StringVal)\} \quad t_0 \Downarrow Const\ rawval_j \quad rawval_j \neq v_p}{pAlt(v_1,\ t_0,\ t_j)\ =\ \text{Nothing}}$$

$$\text{E-ALT-WILDCARD:}\quad \frac{}{pAlt(\_,\ t_0,\ t_j)\ =\ \text{Just}\ t_j}$$

$$\text{E-ALT-EMPTY-LIST-1:}\quad \frac{t_0\ =\ [\,]}{pAlt([\,],\ t_0,\ t_j)\ =\ \text{Just}\ t_j}$$

$$\text{E-ALT-EMPTY-LIST-2:}\quad \frac{t_0 \neq [\,]}{pAlt([\,],\ t_0,\ t_j)\ =\ \text{Nothing}}$$

$$\text{E-ALT-NONEMPTY-LIST-1:}\quad \frac{pAlt(p_t,\ t_{0t},\ t_{jtemp})\ =\ \text{Just}\ t_{jnew} \quad pAlt(p_h,\ t_{0h},\ t_j)\ =\ \text{Just}\ t_{jtemp} \quad t_0\ =\ t_{0h}\ consOp\ t_{0t}}{pAlt(p_h\ consOp\ p_t,\ t_0,\ t_j) =\ \text{Just}\ t_{jnew}}$$

$$\text{E-ALT-NONEMPTY-LIST-2:}\quad \frac{pAlt(p_h,\ t_{0h},\ t_j)\ =\ \text{Nothing} \quad t_0\ =\ t_{0h}\ consOp\ t_{0t}}{pAlt(p_h\ consOp\ p_t,\ t_0,\ t_j)\ =\ \text{Nothing}}$$

$$\text{E-ALT-NONEMPTY-LIST-3:}\quad \frac{pAlt(p_t,\ t_{0t},\ t_{jtemp})\ =\ \text{Nothing} \quad pAlt(p_h,\ t_{0h},\ t_j)\ =\ \text{Just}\ t_{jtemp} \quad t_0 = t_{0h}\ consOp\ t_{0t}}{pAlt(p_h\ consOp\ p_t, t_0, t_j) =\ \text{Nothing}}$$

$$\text{E-ALT-NONEMPTY-LIST-4:}\quad \frac{t_0\ \neq\ t_{0h}\ consOp\ t_{0t}}{pAlt(p_h\ consOp\ p_t,\ t_0,\ t_j)\ =\ \text{Nothing}}$$