Rauf Shimarov
301397321
CMPT 383, Spring 2022

Programming Assignment 4

We will write a Haskell program to perform type inference for expressions in the FUN language.

$$
\begin{array}{rcl}
e & ::= & c \mid b \mid x \mid \texttt{'('}\ e\ \texttt{')'} \\
& \mid & e\ \texttt{'+'}\ e \mid e\ \texttt{'-'}\ e \mid e\ \texttt{'=='}\ e \\
& \mid & \texttt{'if'}\ e\ \texttt{'then'}\ e\ \texttt{'else'}\ e \\
& \mid & \texttt{'lambda'}\ x\ \texttt{'.'}\ e \\
& \mid & \texttt{'app'}\ e\ e \\
& \mid & \texttt{'let'}\ x\ \texttt{'='}\ e\ \texttt{'in'}\ e
\end{array}
$$

$$c \in \mathbf{Int} \quad b \in \mathbf{Bool} \quad x \in \mathbf{Ident}$$

Given an expression e, the program computes the type of e if it is well-typed. If e is ill-typed, the program should output Type Error. The types we consider are the following:

$$t \ ::= \ \texttt{'}Int\texttt{'} \mid \texttt{'}Bool\texttt{'} \mid X \mid \texttt{'('}\ t\ \texttt{')'} \mid t\ \texttt{'}\rightarrow\texttt{'}\ t$$

$$X \in \mathbf{TVars}$$

The program will follow all the rules defined in the Type System (page 2, P4.pdf)

The following are the data types that are defined in the program:

```
type VarId = String
data Expr = CInt Int
          | CBool Bool
          | Var VarId
          | Plus Expr Expr
          | Minus Expr Expr
          | Equal Expr Expr
          | ITE Expr Expr Expr
          | Abs VarId Expr
          | App Expr Expr
          | LetIn VarId Expr Expr
          deriving (Eq, Ord, Read, Show)
```

```
data Type = TInt
          | TBool
          | TError
          | TVar Int
          | TArr Type Type
          deriving (Eq, Ord, Read, Show)
data Constraint = CEq Type Type
                | CError
                deriving (Eq, Ord, Read, Show)
type ConstraintSet = Set.Set Constraint
type ConstraintList = [Constraint]
type Env = Map.Map VarId Type
type InferState a = State Int a
type Substitution = Map.Map Type Type
```

1. *Type VarID* (Variable IDs) are assumed to be strings
2. *data Expr* represents FUN grammar
3. *data Type* represents a FUN type. The type is either Bool, Int, Variable, function type, or a type error
4. *data Constraint* represents a constraint, where CEq represents an equality constraint (e.g., Int = Int) and CError represents a special constraint that is never satisfied
5. *type ConstraintSet* is a set of constraints
6. *type ConstraintList* is a list of constraints
7. *type Env* represents the typing environment, which stored the mapping from the variables to their corresponding types
8. *type InferState a* is a monad that will help us to implement the fresh function
9. *type* Substitution is a map from type variables to types

The algorithm of the program closely follows the algorithm discussed in the lectures. Given an expression *expr*, we first "infer" its type and set of constraints using the rules that are defined in the Type System. We then apply the unification algorithm to the resulted list of constraints and

get the most general unifier. Finally, we apply this most general unifier to the resulted type, that possibly include type variables, and get the type of *expr*.

I will go over each function of the program:

```
1.  getFreshInt :: InferState Int
2.  getFreshTVar :: InferState Type
3.  infer :: Env -> Expr -> InferState (Type, ConstraintSet)
4.  inferExpr :: Expr -> (Type, ConstraintSet)
5.  toCstrList :: ConstraintSet -> ConstraintList
6.  applySub :: Substitution -> Type -> Type
7.  applySubToCstrList :: Substitution -> ConstraintList -> ConstraintList
8.  composeSub :: Substitution -> Substitution -> Substitution
9.  tvars :: Type -> Set.Set Type
10. unify :: ConstraintList -> Maybe Substitution
11. isFuncType :: Type -> Bool
12. typing :: Expr -> Maybe Type
13. typeInfer :: Expr -> String
14. readExpr :: String -> Expr
15. main :: IO ()
```

1. ***getFreshInt:*** yields a different integer every time it is called using a state monad *InferState.*
2. ***getFreshVar:*** generates and yields a fresh variable. The function makes sure that a new type variable does not occur anywhere else in the derivation by using *getFreshInt*.
3. ***infer:*** implements every rule defined in the Type System. Given an expression, it pattern matches a certain rule and recursively conducts constraint-based typing. The function returns a type and set of constraints of the expression as a monadic value.
4. ***inferExpr:*** uses *infer* function to perform a constraint-based typing, "evals" the monadic value returned by *infer* function and returns it as a result. It calls evalState on the monadic value with 1 as a second argument. The value of the second argument doesn't really matter since it will be handled by the *relabel* function anyways.
5. ***toCstrList:*** calls *Set.toList* to convert a constraint set to a constraint list.
6. ***applySub:*** given substitution σ and type T, it applies σ to T and finds σT. The function follows the recursive logic defined lecture 21, slide 24. If the type is a concrete type, it returns the concrete type. If the type is a type variable, then it looks up the type of this type variable in the substitution and returns this type. If the type is a function type T1 -> T2, it reclusively applies the substation to both T1 and T2.

7. ***applySubToCstrList:*** given a substitution σ and a constraint list, it gets each constraint from the list of the form T1 = T2 and applies σ to both T1 and T2 using applySub function.
8. ***composeSub:*** takes two substitutions σ1, σ2 as input and produces σ1 ◦ σ2 as output. The function follows the logic defined in lecture 21, slide 25. In short, σ1 ◦ σ2 is union of (X ↦ Tσ1 for each (X ↦ T) ∈ σ2) and (X ↦ T for each (X ↦ T) ∈ σ with X ∉ dom(σ2)).
9. ***tvars:*** returns all type variables in a FUN type. The function recursively traverses a type and inserts type variables into a set, if any.
10. ***unify:*** performs unification on a list of constraints to find a most general unifier. The function closely follows the pseudocode shown in the lecture 21, slide 28. In short, it takes out one constraint of the form S = T from the input list of constraints at a time. If the constraint S and T are syntactically equivalent, then we continue to the rest of constraints. If S is a type variable, say X, and X is not in the set of all type variables in T, then we apply substitution [X |-> T] to the remaining constraints, unify the resulted substitution, and compose with [X |-> T]. We follow similar logic for the case when T is a type variable. If both S and T are function types, we add equalities for their input types S1 = S2 and return types T1 = T2 into the constraints. Else, return *Nothing.*
11. ***typing:*** performs constraint-based typing by calling *infer* function, unifies the resulted list of constraints by calling *unify* function, applies substitution to the resulted type from the infer function by calling *applySub*, and returns the resulted type of the expression. If expression is ill-typed, return *Nothing.*
12. ***typeInfer:*** it gets the type of the expression by calling *typing* function and returns it using the provided relabel function. If typing function returns *Nothing*, then return "Type Error".
13. ***readExpr:*** takes a *String* an turns it to *Expr.*
14. ***main:*** handles input/output. It reads the expressions from a text file and prints their types.