CMPT 383 Comparative Programming Languages

Programming Assignment 4

This assignment is due by 11:59pm PT on Monday Apr 11, 2022. Please submit it to Canvas.

Requirements:

- This assignment must be your own work. No collaboration is permitted.
- You can learn the code on slides and start from it.
- You can only use library functions from the following modules: `Prelude`, `System.IO`, `System.Environment`, `Data.Map.Strict`, `Data.Set`, and `Control.Monad.State.Lazy`. Detailed information can be found on `https://hoogle.haskell.org`

Late policy:
You have two free late days for this assignment. Suppose you can get n (out of 10) points based on your code and report

- If you submit before the deadline, you can get all n points.
- If you submit between 11:59pm PT Apr 11 and 11:59pm PT Apr 13, you can still get n points.
- If you submit after 11:59pm PT Apr 13, you get 0 points.

(10 points) Consider the following fragment of the FUN language without type annotations

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

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

Here, e is the start symbol. c stands for an integer constant, b stands for a boolean constant, and x stands for an identifier (variable). The types that we consider for FUN programs are defined as follows

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

$$X \in \textbf{TVars}$$

Specifically, we consider primitive types (namely, Int and Bool), type variables, and function types of the form $T_1 \to T_2$ for FUN programs, where T_1 and T_2 can be primitive types, type variables, and function types. As is standard, the $\to$ operator is right-associative.

In this assignment, you need to write a Haskell program to perform type inference for expressions in the FUN language. Specifically, 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`.

Data Structures

This assignment requires using both `Map` and `Set` as data structures. Since many functions from these two modules have the same name, qualified imports are needed to resolve name ambiguity

```haskell
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set
```

Type System

The type system is defined by the following constraint-based typing rules.

$$\frac{\text{Int } c}{\Gamma \vdash c : \mathit{Int} \mid \{\}}\ \text{(CT-Int)} \qquad \frac{\text{Bool } b}{\Gamma \vdash b : \mathit{Bool} \mid \{\}}\ \text{(CT-Bool)} \qquad \frac{\begin{array}{c}\text{fresh } X \\ \Gamma[x \triangleleft X] \vdash e : T \mid C\end{array}}{\Gamma \vdash \texttt{lambda}\ x\,.\,e : X \to T \mid C}\ \text{(CT-Abs)}$$

$$\frac{\text{Ident } x \quad x \in \mathrm{dom}(\Gamma) \quad \Gamma(x) = T}{\Gamma \vdash x : T \mid \{\}}\ \text{(CT-Ident1)} \qquad \frac{\text{Ident } x \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash x : T_{err} \mid \{C_{err}\}}\ \text{(CT-Ident2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \\ \Gamma \vdash e_2 : T_2 \mid C_2 \\ C = C_1 \cup C_2 \cup \{T_1 = \mathit{Int},\ T_2 = \mathit{Int}\}\end{array}}{\Gamma \vdash e_1 + e_2 : \mathit{Int} \mid C}\ \text{(CT-Plus)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \\ \Gamma \vdash e_2 : T_2 \mid C_2 \\ C = C_1 \cup C_2 \cup \{T_1 = \mathit{Int},\ T_2 = \mathit{Int}\}\end{array}}{\Gamma \vdash e_1 - e_2 : \mathit{Int} \mid C}\ \text{(CT-Minus)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \\ \Gamma \vdash e_2 : T_2 \mid C_2 \\ C = C_1 \cup C_2 \cup \{T_1 = T_2\}\end{array}}{\Gamma \vdash e_1\ \texttt{==}\ e_2 : \mathit{Bool} \mid C}\ \text{(CT-Eq)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \\ \Gamma \vdash e_2 : T_2 \mid C_2 \quad \Gamma \vdash e_3 : T_3 \mid C_3 \\ C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \mathit{Bool},\ T_2 = T_3\}\end{array}}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : T_2 \mid C}\ \text{(CT-ITE)}$$

$$\frac{\begin{array}{c}\text{fresh } X_1, X_2 \\ \Gamma \vdash e_1 : T_1 \mid C_1 \\ \Gamma \vdash e_2 : T_2 \mid C_2 \\ C = C_1 \cup C_2 \cup \{T_1 = X_1 \to X_2,\ T_2 = X_1\}\end{array}}{\Gamma \vdash \texttt{app}\ e_1\ e_2 : X_2 \mid C}\ \text{(CT-App)} \qquad \frac{\begin{array}{c}\text{fresh } X \\ \Gamma[x \triangleleft X] \vdash e_1 : T_1 \mid C_1 \\ \Gamma[x \triangleleft X] \vdash e_2 : T_2 \mid C_2 \\ C = C_1 \cup C_2 \cup \{X = T_1\}\end{array}}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : T_2 \mid C}\ \text{(CT-Let)}$$

Given an expression e, if $\Gamma \vdash e : T \mid C$ where Γ is an empty map, and if constraints C are satisfied with a most general unifier σ, then the type of expression e is $T\sigma$.

Representation of Expressions

Variable IDs are assumed to be strings:

```
type VarId = String
```

To avoid the complication of parsing, an expression of the FUN language is represented as a value of the following `Expr` type

```
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)
```

As indicated by the names of data constructors, `CInt` denotes an integer constant, `CBool` denotes a boolean constant, `Var` denotes a variable (identifier), `Plus` denotes the `+` operator, `Minus` denotes the `-` operator, `Equal` denotes the `==` operator, `ITE` denotes the if-then-else expression, `Abs` denotes the function abstraction (lambda), `App` denotes the function application, and `LetIn` denotes the let-in expression. There is no type annotation for `Abs` and `LetIn`. Also note that there is no data constructor for parenthesized expressions. For example,

- `x1` is represented as `Var "x1"`
- `1 + 2` is represented as `Plus (CInt 1) (CInt 2)`
- `if True then 1 else 2` is represented as `ITE (CBool True) (CInt 1) (CInt 2)`
- `lambda x.x` is represented as `Abs "x" (Var "x")`
- `app (lambda x.x) 1` is represented as `App (Abs "x" (Var "x")) (CInt 1)`
- `let x = 1 in x` is represented as `LetIn "x" (CInt 1) (Var "x")`

Representation of Types

A FUN type is represented as a value of the following `Type` data type in Haskell

```
data Type = TInt
          | TBool
          | TError
          | TVar Int
          | TArr Type Type
          deriving (Eq, Ord, Read, Show)
```

where `TInt` represents the *Int* type, `TBool` represents the *Bool* type, and `TArr` represents the function type. For example, type $Int \to Bool \to Int$ in FUN is represented as `TArr TInt (TArr TBool TInt)`. `TError` represents a special error type, corresponding to the T_{err} symbol in the typing rules. It should never occur in well-typed expressions. `TVar` represents a type variable. For example, the type variable X_1 can be represented as `TVar 1`.

Representation of Constraints

A constraint is represented as a value of the `Constraint` data type.

```
data Constraint = CEq Type Type
                | CError
                deriving (Eq, Ord, Read, Show)
```

Here, `CEq` represents an equality constraint. For instance, $Int = Int$ should be represented as `CEq TInt TInt`. `CError` represents a special constraint that is never satisfied, corresponding to the C_{err} symbol in the typing rules.

The following aliases are also defined for clarity:

```
type ConstraintSet = Set.Set Constraint
type ConstraintList = [Constraint]
```

Inclusion of Definitions

Note that you need to use exactly the same definition of `VarId`, `Expr`, `Type`, `Constraint`, `ConstraintSet`, `ConstraintList`, and their deriving clauses as written in this document. Otherwise, you will lose points because potential grading scripts may not work as expected.

Detailed Steps

For function and type names mentioned in the following steps, please use **exactly the same names** as provided in this document.

1. Define the `Env` type for the typing environment

   ```
   type Env = Map.Map VarId Type
   ```

   Also use `State` from the `Control.Monad.State.Lazy` module to define the `InferState` type constructor (exact name)

```
type InferState a = State Int a
```

Observe that `InferState` is a monad.

2. Write a function `getFreshTVar :: InferState Type` returning a monadic value that yields a different type variable every time it is performed.

3. Write a function `infer :: Env -> Expr -> InferState (Type, ConstraintSet)` returning a monadic value that conducts constraint-based typing when it is performed. To start with, you might use the following (incomplete) code snippet

```
infer :: Env -> Expr -> InferState (Type, ConstraintSet)
infer g (CInt _)  = return (TInt, Set.empty)
infer g (Abs x e) = do y <- getFreshTVar
                       (t, c) <- infer (Map.insert x y g) e
                       return (TArr y t, c)
```

   You can also modify the code snippet if you want.

4. Write a function `inferExpr :: Expr -> (Type, ConstraintSet)` that takes a FUN expression as input and produces as output the result of constraint-based typing. Hint: you need to "eval" the monadic value defined by `infer`.

5. Find a function in the `Set` module to implement `toCstrList :: ConstraintSet -> ConstraintList`. It can convert a constraint set to a constraint list.

6. Recall that a substitution is a map from type variables to types. Define the type for substitutions

```
type Substitution = Map.Map Type Type
```

   Write a function `applySub :: Substitution -> Type -> Type` that takes a substitution σ and a type T as input and produces $T\sigma$ as output, i.e., applying σ to T.

7. Write a function `applySubToCstrList :: Substitution -> ConstraintList -> ConstraintList` that applies a substitution to a constraint list.

8. Write a function `composeSub :: Substitution -> Substitution -> Substitution` that takes two substitutions σ_1, σ_2 as input and produces $\sigma_1 \circ \sigma_2$ as output.

9. Write a function `tvars :: Type -> Set.Set Type` that returns all type variables in a FUN type.

10. Write a function `unify :: ConstraintList -> Maybe Substitution` that performs unification on a list of constraints to find a most general unifier. If the constraints cannot be satisfied, it returns `Nothing`.

11. Write a function `typing :: Expr -> Maybe Type` that puts the constraint-based typing and unification together. In particular, `typing` takes a FUN expression e as input and generates e's type as output if e is well-typed. If e is ill-typed, `typing` returns `Nothing`.

12. Write a simple function `typeInfer :: Expr -> String` (exact name) that takes a FUN expression as input and produces a string as output representing the type inference result. Specifically,

    - If the expression is well-typed and a value `v` of type `Type` is obtained, generate the output using `show (relabel v)`. **NOTE** that you need to use the `relabel` function as defined in the appendix. Otherwise, you will lose points because potential grading scripts may not work as expected.
    - If the expression is ill-typed, output string "`Type Error`".

13. Write a `main` to handle IO and put everything together.

The program must be in a form that GHC can compile. It needs to take one command-line argument denoting the path to the expression file. Each line of the file contains a string representing a FUN expression, and the program needs to print the result of `typeInfer` on each expression to the console.

Sample Input and Output

Suppose we have a file called `exprs.txt` that contains the following six lines:

```
Var "x1"
Plus (CInt 1) (CInt 2)
ITE (CBool True) (CInt 1) (CInt 2)
Abs "x" (Var "x")
App (Abs "x" (Var "x")) (CInt 1)
LetIn "x" (CInt 1) (Var "x")
```

After compiling, we can run the executable and get

```
$ ./p4_firstname_lastname exprs.txt
Type Error
TInt
TInt
TArr (TVar 1) (TVar 1)
TInt
TInt
```

Deliverable

A zip file called `p4_firstname_lastname.zip` that contains at least the followings:

- A file called `p4_firstname_lastname.hs` that contains the source code of your Haskell program. You can have multiple source files if you want, but you need to make sure `ghc p4_firstname_lastname.hs` can compile.
- A report called `p4_firstname_lastname.pdf` that explains the design choices, features, issues (if any), and anything else that you want to explain about your program.

Appendix

Here is a function `relabel :: Type -> Type` that can relabel type variables using `TVar 1`, `TVar 2`, ... It can also ensure identical old type variables are relabeled to the same new type variables.

```haskell
type RelabelState a = State (Map.Map Int Int) a

relabel :: Type -> Type
relabel t = evalState (go t) Map.empty
  where
    go :: Type -> RelabelState Type
    go TInt         = return TInt
    go TBool        = return TBool
    go TError       = return TError
    go (TVar x)     = do m <- get
                         case Map.lookup x m of
                           Just v  -> return (TVar v)
                           Nothing -> do let n = 1 + Map.size m
                                         put (Map.insert x n m)
                                         return (TVar n)
    go (TArr t1 t2) = do t1' <- go t1
                         t2' <- go t2
                         return (TArr t1' t2')
```