301398647 **CMPT 383: Programming Assignment 4 Report** **Evan Mangat**

Step 1:

No work done here, all work is provided in the P4.pdf, I merely added both the given statements to the code.

Step 2:

Uses the State monad to keep track of a counter n that is incremented every time getFreshTVar is called. It then returns a new type variable TVar n using the current value of n.

Step 3:

By far the longest function in this assignment, though oddly not the most complex. This implementation uses the State monad to generate fresh type variables as needed. The infer function is defined recursively for each Expr case, using the do notation to sequence the sub-inferences and combine their results with sets of constraints using Set.union and Set.singleton. The helper function inferBinaryOp is for Plus and Minus. The provided code snippet was very helpful and gave me a place to start from for the design.

Step 4:

This function first creates an empty environment Map.empty, and then calls infer with this environment and the input expression e. The resulting monadic value is then evaluated with an initial state of 0 using evalState, which returns the result of the computation as a tuple of a Type and a ConstraintSet.

Step 5:

Simple function. It takes a ConstraintSet as input and returns a ConstraintList by converting the set to a list using the toList function from the Data.Set module.

Step 6:

The design for this was inspired by similar functions we've created for past assignments, such as eval in P1 due to similarity in design nature (base cases with recursion for higher levels) . It recursively applies the substitution sigma to each sub-type of Type, leaving the base types TInt, TBool, and TError unchanged. For each type variable TVar x, it looks up its substitution in the map sigma and replaces it with the corresponding type. Finally, for function types TArr t1 t2, it applies sigma to both t1 and t2.

Step 7:

This function recursively applies the substitution sigma to each constraint in the input ConstraintList. The applySubToCstr helper function is used to apply the substitution to a single

constraint. If the constraint is CError, the result is also CError. Otherwise, the substitution is applied to each of the constraint's types using the applySub function defined earlier.

Step 8:

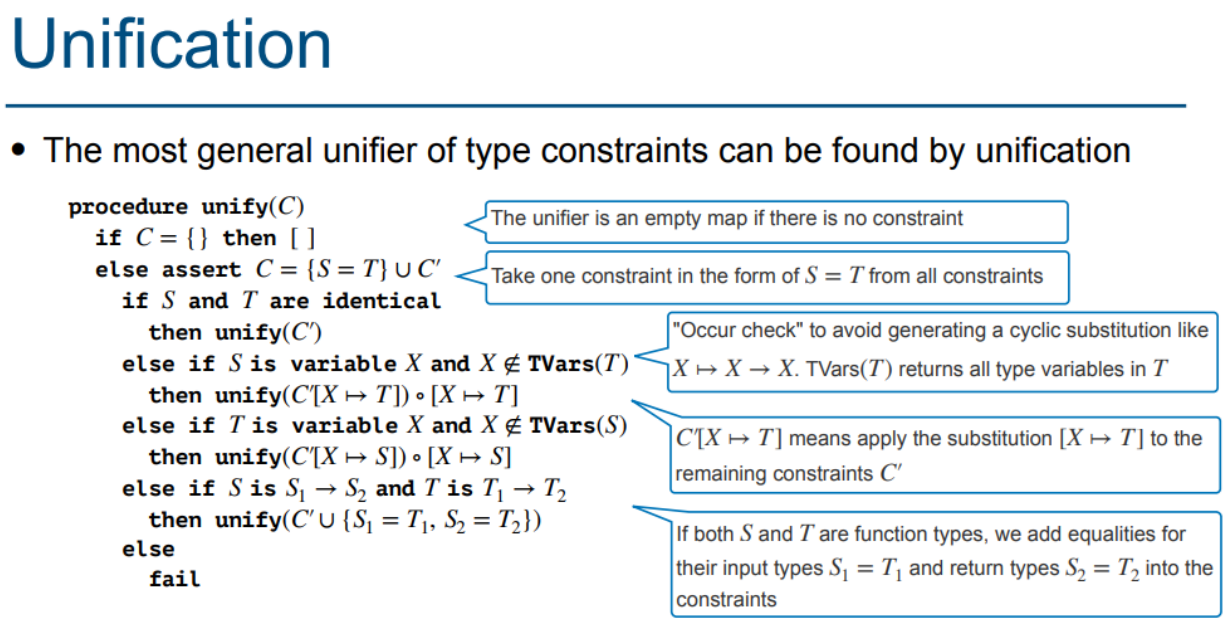
Very simple function. The algorithm/design was lifted from our lecture notes (Slide 25, Lecture 20). The composeSub function applies s1 to all types in s2 using applySub from Step 6 and then merges the resulting map with s1.

Step 9:

Like in Step 6, the design for this was inspired by similar functions we've created for past assignments, such as eval in P1. This function recursively traverses the input Type and accumulates all the type variables encountered in a Set. The TInt, TBool, and TError cases are trivial, as these types do not contain any type variables. The TVar case adds the type variable to the result set. The TArr case combines the results of recursively calling tvars on the two component types using the Set.union function.

Step 10:

The hardest function to implement in my opinion. It took quite a bit of thinking, and I scrapped the original design all together due to a bug where it output TVar 1 instead of Tint when appropriate. The biggest saving grace was the pseudocode for Unify given on Slide 28 of Lecture 20. It helped immensely. Additionally utilizing a helper function made the design simpler and easier to think through. In terms of what it does it just performs unification on a list of constraints to find a most general unifier, returning nothing if it can't be satisfied in accordance with this:



Step 11:

In and of itself, this function is relatively simple. However, it essentially ties together all the previous steps done. It utilizes inferExpr and unify to achieve a result. By using these two it is by extension using all the previous functions because those are used by inferExpr and unify. The algorithm is simply run the given Expr through inferExpr to yield (Type, ConstraintSet) which is then given to Unify to output either Just Type or nothing.

Step 12:

I based this function on typeCheck from P3. I decided to do this upon reading the function description and its typing being nearly identical. The small tweak that was necessary however was the addition of checking for the case of TError being included in v which leads to an overall output of Type Error rather than TError and other outputs such as TError (TInt) for example.

Main:

Finally, the main is identical to P3 as our final function is the same type Expr -> String. However, I needed to add one more helper function in readExpr. This was not in the previous steps like in P3 and we had no other function that was type String -> Expr so I added it to read the input. The result is a main identical to P3 with the addition of the readExpr helper function before it.

Sample run of the program. exprs.txt contains the following test case from P4.pdf:

```
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")
```

```
PS C:\Users\evanm> ghc ./P4_Evan_Mangat.hs
[1 of 1] Compiling Main             ( P4_Evan_Mangat.hs, P4_Evan_Mangat.o )
Linking P4_Evan_Mangat.exe ...
PS C:\Users\evanm> ./P4_Evan_Mangat exprs.txt
Type Error
TInt
TInt
TArr (TVar 1) (TVar 1)
TInt
TInt
```