Homework 6

Posted Thursday March 21, Due Thursday April 4
(Adapted from the MIT Program Analysis)
50 points

1. Simple Type Inference Examples (10 points, 2 points each)

Give the simple type (i.e., no polymorphism) for the last named top-level function defined in each of the following snippets. Assume that all arithmetic operations take arguments of type `Int` and that all comparison operations return a result of type `Bool`, as in the typing rules discussed in class. If a type variable remains unconstrained, just leave it as is, e.g., t_1.

(1)
```
det a b c = (b*b) - 4*a*c
```

(2)
```
step (a,b) = (b,b+1)
```

(3)
```
sum f n =
  if (n<0) then 0 else sum f (n-1) + (f n)

sumSum f n = sum (sum f) n
```

(4)
```
loopy x = loopy x
```

(5)
```
isEven n =
   let
      even n = if (n==0) then True else odd (n-1)
      odd n = if (n==0) then False else even (n-1)
   in
      even n
```

2. Hindley Milner Type Inference Examples (10 points, 2 points each)

For each of the code snippets below state whether it can be typed in simple types (i.e., no polymorphism). If yes, give the type of the last named top-level function. If not, explain why it cannot be typed.

If a term cannot be typed in simple types, state whether it can be typed in Hindley Milner (i.e., with polymorphism). If yes, give the principal type of the last named top-level function. If not, explain why it cannot be typed.

(1)
```
f x = if x then x+3 else x*2
```

(2)
```
foo f =
   let func x = f x x
   in func
```

(3)
```
r g x y = if (g x) then g y
                   else 1+(g y)
```

(4)
```
s g =
   let h x = g (g x)
   in h (h True, h False)
```

(5)
```
id x = x

herbert y =
  if (id y) then (id 0)
            else (id 1)
```

3. Simple Type Inference in Haskell (30 points)

Now you will implement simple type inference (without polymorphism) in Haskell. Our small language is a standard extension of the pure λ-calculus with conditionals, arithmetic and general recursion (i.e., `let`). The language has the following expressions and statements:

```
data Exp =
    EVar Ident
  | EPrim Prim
  | ELambda Ident Exp
  | EApp Exp Exp
  | ECond Exp Exp Exp
  | EPlus Exp Exp
  | ELet Stmt Exp

data Stmt =
    SEmpy
  | SAssign Ident Exp
  | SSeq Stmt Stmt
```

(Note that we do not require you to handle the case of `ELet` with multiple bindings of the same variable. We consider such definitions ill-formed, but you to not need to detect this.)

The language has the following type expressions for simple types:

```
data BaseType = BTInt | BTBool
  deriving (Eq, Show)

data Type =
    TBase BaseType
  | TVar Ident
  | TArrow Type Type
  deriving (Eq, Show)
```

You will be writing a program that infers the type of a given lambda expression. You may choose to implement either Strategy 1 (constraint-based typing) or Strategy 2 (Algorithm W-like on-the-fly typing). We discuss the two strategies extensively in class.

Download starter code `hw6Code.zip` from the course web site: `http://www.cs.rpi.edu/~milanova/csci4450/hw6Code.zip`. Spend some time studying the code and the utility functions. The code uses Haskell features you may be unfamiliar with, such as records, exceptions, the symbol @ — syntactic sugar for "as", and others.

3.1. `unify.` In the code substitutions are represented as a mapping from type variables to types:

```
type Subst = [(TVar, Type)]
idSubst = []
```

Implement unification which produces a substitution that may bind additional variables:

```
unify :: Constraints -> Subst
```

Note that this function can either unify a pair of types, or it can unify a list of constraints, depending on how you choose to implement `inferTypes`.

Next, implement `inferTypes` which takes a constraint environment and an expression and returns a potentially modified constraint environment and a type:

```
inferTypes :: ConstraintEnv -> Exp -> (ConstraintEnv, Type)
```

Note that this signature of `inferTypes` fits with Strategy 1. If you choose to implement Strategy 2, you will need to change the signature to return a substitution map as well. As mentioned earlier, you can choose either Strategy 1 or Strategy 2.

3.2. `inferTypes cenv (EVar var)` **and** `inferTypes cenv (EPrim ...)`

3.3. `inferTypes cenv (ELambda ...), inferTypes cenv (EApp ...)`

3.4. `inferTypes cenv (ECond ...), inferTypes cenv (EPlus ...)`

3.5. `inferTypes cenv (ELet ...)` Note that ELet implements letrec, akin to Haskell itself (Haskell's let is letrec).

3.6. At the end, implement top-level `inferType` for testing on Submitty. It takes an expression and returns a type. It would simply wrap around your `inferTypes`.

When you are done, upload your files in Submitty. Upload a pdf file with solutions for problems 1 and 2. The file size limit is set to 5Mb. Upload `TypeInfer.hs`, `Data.hs` and `Utils.hs` for problem 3, or a zip file containing these three Haskell files.