```
infer :: Env -> Expr -> InferState (Type, ConstraintSet)
infer g (CInt _) = return (TInt, Set.empty)
infer g (CBool _) = return (TBool, Set.empty)
infer g (Var x) = case Map.lookup x g of
  Just t -> return (t, Set.empty)
  Nothing -> return (TError, Set.empty)
infer g (Plus e1 e2) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer g e2
  let c = Set.unions [c1, c2, Set.fromList [CEq t1 TInt, CEq t2 TInt]]
  return (TInt, c)
infer g (Minus e1 e2) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer g e2
  let c = Set.unions [c1, c2, Set.fromList [CEq t1 TInt, CEq t2 TInt]]
  return (TInt, c)
infer g (Equal e1 e2) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer g e2
  let c = Set.unions [c1, c2, Set.fromList [CEq t1 t2]]
  return (TBool, c)
infer g (ITE e1 e2 e3) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer g e2
  (t3, c3) <- infer g e3
  let c = Set.unions [c1, c2, c3, Set.fromList [CEq t1 TBool, CEq t2 t3]]
  return (t2, c)
infer g (Abs x e) = do
  tVar <- getFreshTVar
  (t, c) <- infer (Map.insert x tVar g) e
  return (TArr tVar t, c)
infer g (App e1 e2) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer g e2
  tVar <- getFreshTVar
  let c = Set.unions [c1, c2, Set.fromList [CEq t1 (TArr t2 tVar)]]
  return (tVar, c)
infer g (LetIn x e1 e2) = do
  (t1, c1) <- infer g e1
  (t2, c2) <- infer (Map.insert x t1 g) e2
  return (t2, Set.union c1 c2)
```

for me the infer function took me the longest to get it down. I couldn't understand a way to do the unions but as I looked more into the attributes of set/map I found the union functions. But even with doing this I was unable to get the proper results that I was aiming for. Mainly due to the function being long and hard to read. I could not figure out a more readable way of doing this. Overall this assignment took me the longest to do.

```
unify :: ConstraintList -> Maybe Substitution
unify [] = Just Map.empty
unify (CEq t1 t2 : cs) =
  case (t1, t2) of
    (TInt, TInt) -> unify cs
    (TBool, TBool) -> unify cs
    (TVar x, _) | t2 `elem` tvars (TVar x) -> Nothing
                | otherwise -> fmap (\s -> composeSub s (Map.singleton (TVar x)
t2)) (unify (applySubToCstrList (Map.singleton (TVar x) t2) cs))
    (_, TVar x) | t1 `elem` tvars (TVar x) -> Nothing
                | otherwise -> fmap (\s -> composeSub s (Map.singleton (TVar x)
t1)) (unify (applySubToCstrList (Map.singleton (TVar x) t1) cs))
    (TArr a b, TArr c d) -> unify (CEq a c : CEq b d : cs)
    _ -> Nothing
```

Initially I struggles with this function. Because I knew it was something along the lines of t2 elem tvars but for the second part I wasn't sure how I was suppose to do the fmap. Took me a while to figure out I had to use (\s -> composeSub s (Map.singleton..... also I got lucky by finding the singleton value for map. I wasn't looking for it but while glancing over the documentations I saw it. Which made it easy to get the values

These are some test cases that were provided over discord that I ended up using:

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")

Abs "x" (App (Abs "y" (Var "y")) (Var "x"))

Var "x1"

Var "x1"

```
Var "x1"

LetIn "x" (CBool True) (Var "x")

Minus (CInt 5) (CInt 3)

Equal (CInt 2) (CInt 2)

Equal (CBool True) (CBool False)

App (Abs "x" (Plus (Var "x") (CInt 1))) (CInt 3)

LetIn "x" (Plus (CInt 2) (CInt 3)) (Equal (Var "x") (CInt 5))

LetIn "x" (CInt 5) (LetIn "y" (CInt 3) (Plus (Var "x") (Var "y")))

App (Abs "x" (Abs "y" (Plus (Var "x") (Var "y")))) (CInt 2)

Abs "x" (Plus (Var "x") (Var "x"))

LetIn "x" (CInt 2) (App (Abs "y" (Plus (Var "x") (Var "y"))) (CInt 3))

LetIn "x" (CInt 2) (LetIn "y" (CInt 3) (Equal (Var "x") (Var "y")))

Abs "x" (Abs "y" (ITE (Equal (Var "x") (CInt 42)) (CBool False) (Var "y")))

App (CBool True) (CInt 2)

Minus (CBool True) (CBool False)

LetIn "x" (CBool True) (Plus (Var "x") (CBool False))
```