CS 558: Software Foundations - Homework 6

Lucas Nunno
lnunno@cs.unm.edu

December 13, 2013

Contents

1 Abstract Syntax

Datatype definition for Type and Term which will be the building blocks of our parser.

```
module AbstractSyntax where
data Term =
  Identifier {name :: String}
   | Abstraction {variable :: Term, variableType :: Type, body :: Term}
   | Application Term Term
   | Tru
   | Fls
   | If Term Term Term
   | Zero
   | Succ Term
   | Pred Term
   | IsZero Term
   | Fix Term
  deriving (Eq)
```

Most of this code is reused from the last assignment. Added Show definition for Fix term.

```
instance Show Term where
  show Tru                = "true"
  show Fls                = "false"
  show Zero               = "0"
  show (If a b c)         = "if " ++ (show a) ++ " then " ++ (show b) ++ " else " ++ (s
  show (Succ t)
     | isNumericValue t = showNumberIncrement t 1
     | otherwise          = "succ(" ++ (show t) ++ ")"
  show (Pred t)
     | otherwise          = "pred(" ++ (show t) ++ ")"
```

$$
\begin{array}{ll}
\quad show\ (IsZero\ t) & = \texttt{"iszero("} +\!\!+ (show\ t) +\!\!+ \texttt{")"} \\
\quad show\ (Identifier\ n) & = n \\
\quad show\ (Abstraction\ v\ vt\ b) & = \texttt{"abs("} +\!\!+ (show\ v) +\!\!+ \texttt{":"} +\!\!+ (show\ vt) +\!\!+ \texttt{"."} +\!\!+ (show\ b) +\!\!+ \texttt{"} \\
\quad show\ (Application\ t1\ t2) & = \texttt{"app("} +\!\!+ (show\ t1) +\!\!+ \texttt{","} +\!\!+ (show\ t2) +\!\!+ \texttt{")"} \\
\quad show\ (Fix\ t) & = \texttt{"fix("} +\!\!+ (show\ t) +\!\!+ \texttt{")"}
\end{array}
$$

$$
\begin{array}{l}
isValue :: Term \rightarrow Bool \\
isValue\ Tru \qquad\qquad\quad = True \\
isValue\ Fls \qquad\qquad\quad = True \\
isValue\ (Abstraction\ \_\ \_\ t) = True \\
isValue\ t \qquad\qquad\qquad = isNumericValue\ t
\end{array}
$$

$$
\begin{array}{l}
isNumericValue :: Term \rightarrow Bool \\
isNumericValue\ Zero = True \\
isNumericValue\ (Succ\ t) = isNumericValue\ t \\
isNumericValue\ \_ \qquad = False
\end{array}
$$

I re-added the show definition for succ, to make the number output more friendly when using factorial, times, and other functions that can have large numbers output.

$$
\begin{array}{ll}
showNumberIncrement\ Zero\ acc & = show\ acc \\
showNumberIncrement\ (Succ\ t)\ acc & = showNumberIncrement\ t\ (acc + 1)
\end{array}
$$

More data type definitions.

$$
\begin{array}{l}
\textbf{type}\ TypeVar = String \\
\textbf{data}\ Type = \\
\quad TypePair\ \{t1 :: Type, t2 :: Type\} \\
\quad \mid TBool \\
\quad \mid TNat \\
\quad \mid TVar\ TypeVar \\
\quad \textbf{deriving}\ (Eq)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Show\ Type\ \textbf{where} \\
\quad show\ (TypePair\ a\ b) = \texttt{"arr("} +\!\!+ (show\ a) +\!\!+ \texttt{","} +\!\!+ (show\ b) +\!\!+ \texttt{")"} \\
\quad show\ TBool = \texttt{"Bool"} \\
\quad show\ TNat\ \ = \texttt{"Nat"} \\
\quad show\ (TVar\ varName) = varName
\end{array}
$$

1.1 Type Contexts

Definition of our type context and the bindings of variable names to their types.

```
data Binding =
  VarBind {identifierName :: String, identifierType :: Type, identifierTerm :: Term}
  deriving (Eq)
data TypeContext =
  Empty
  | VariableBindings {bindings :: [Binding]}
```

2 Evaluation

Page 144 of TAPL was helpful when implementing fixed-point iteration in lambda calculus.

```
module Evaluation where
import Test.HUnit
import AbstractSyntax
import Data.List
import Data.Maybe
import Control.Monad (liftM)
eval1 :: TypeContext → Term → (Maybe Term, TypeContext)
eval1 context (Pred Zero)   = (Just Zero, context)
eval1 context (Pred (Succ t))
   | isNumericValue t        = (Just t, context)
eval1 context (Pred t)       = eval1Recursive Pred t context
eval1 context (Succ t)       = eval1Recursive Succ t context
eval1 context (IsZero Zero) = (Just Tru, context)
eval1 context (IsZero (Succ t))
   | isNumericValue t        = (Just Fls, context)
   | otherwise               = (Nothing, context)
eval1 context (IsZero t)     = eval1Recursive IsZero t context
eval1 context (If Tru t2 _) = (Just t2, context)
eval1 context (If Fls _ t3) = (Just t3, context)
eval1 context (If p t t1)    = eval1Recursive (λx → If x t t1) p context
eval1 context (Application t1 t2)
   | ¬ (isValue t1) = eval1Recursive (λx → Application x t2) t1 context   -- E-APP1
   | (isValue t1) ∧ (¬ (isValue t2)) = eval1Recursive (λx → Application t1 x) t2 context   -- E-A
   | otherwise = case t1 of    -- E-APPABS
     Abstraction var varType absBody → (Just (replace absBody var t2), context′)
       where
```

```
        context′              = addBinding context var varType t2
      otherwise → (Nothing, context)
eval1 context t@(Fix t1) =
  case t1 of
    Abstraction var varType absBody → (Just (replace absBody var t), context)
    otherwise → eval1Recursive Fix t1 context
eval1 context _               = (Nothing, context)

eval1Recursive f t context =
  let
    (term, context′)          = (eval1 context t)
  in
    ((liftM f) term, context′)

eval :: TypeContext → Term → Term
eval context t
  | isValue t = t
  | otherwise =
    case eval1 context t of
      (Nothing, _) → t
      (Just a, context′) → eval context′ a
```

Function to replace all instances of a nested term with a replacement term. Useful for variable binding.

```
replace :: Term → Term → Term → Term
replace (Application t0 t1) prevTerm replaceTerm = Application (replace t0 prevTerm replaceTe
replace a@(Abstraction t ttype body) prevTerm replaceTerm
  | t ≡ prevTerm = a    -- Don't replace a bound variable.
  | otherwise = Abstraction t ttype (replace body prevTerm replaceTerm)
replace (Succ t0) prevTerm replaceTerm = Succ (replace t0 prevTerm replaceTerm)
replace (Pred t0) prevTerm replaceTerm = Pred (replace t0 prevTerm replaceTerm)
replace (IsZero t0) prevTerm replaceTerm = IsZero (replace t0 prevTerm replaceTerm)
replace (If t0 t1 t2) prevTerm replaceTerm = If (replace t0 prevTerm replaceTerm) (replace t1
replace (Fix t) prevTerm replaceTerm = Fix (replace t prevTerm replaceTerm)
replace accTerm prevTerm replaceTerm
  | accTerm ≡ prevTerm = replaceTerm
  | otherwise          = accTerm
```

Function to find the given identifier in the list of bindings or Nothing if there's no binding found.

```
getTypeFromTypeContext :: TypeContext → Term → Maybe Type
getTypeFromTypeContext Empty (Identifier _) = Nothing
```

```haskell
getTypeFromTypeContext c (Identifier n) = bindingType
  where
    foundBinding = find (\b -> (identifierName b) == n) (bindings c)
    bindingType
      | foundBinding == Nothing = Nothing
      | otherwise                = Just (identifierType $ fromJust $ foundBinding)
getTypeFromTypeContext _ term           = error $ "Error: Can't look up a binding f
addBinding :: TypeContext -> Term -> Type -> Term -> TypeContext
addBinding Empty (Identifier a) t tterm = VariableBindings [VarBind a t tterm]
addBinding c (Identifier a) t tterm = VariableBindings ((VarBind a t tterm) : (bindings c))
addBinding _ t _ _ = error $ "Error: Cannot add a binding for non Identifier" +
```

2.1 Determining Types

Functions for determining the types of terms. Chapter 10 in TAPL was referenced and drawn upon while writing this function.

```haskell
typeOf :: TypeContext -> Term -> Type
typeOf _ Tru  = TBool
typeOf _ Fls  = TBool
typeOf _ Zero = TNat
typeOf context t@(Succ t1) = if (typeOf context t1) == TNat
  then TNat
  else (error ("Error: Undefined type for " ++ (show t)))
typeOf context t@(Pred t1) = if (typeOf context t1) == TNat
  then TNat
  else (error ("Error: Undefined type for " ++ (show t)))
typeOf context t@(IsZero t1) = if (typeOf context t1) == TNat
  then TBool
  else (error ("Error: Undefined type for " ++ (show t)))
typeOf context (If t1 t2 t3) =
  if (typeOf context t1) == TBool then
    let
      tyT2 = typeOf context t2
      tyT3 = typeOf context t3
    in
      if tyT2 == tyT3 then
              tyT2
      else
```

```
        error $ "Error: Conditional results \"" ++ (show t2) ++ ": " ++ (show
  else
    error $ "Error: The initial test of a conditional must be a boolean. N
typeOf context t@(Identifier n) =
  case getTypeFromTypeContext context t of
    Just ttype → ttype
    Nothing → error $ "Error: No identifier " ++ n ++ " within this context."
typeOf context (Abstraction t1 typeT1 t2) =
  let
    context′ = addBinding context t1 typeT1 t2   -- Note that we're not actually using t2 within the
    typeT2 = typeOf context′ t2
  in
    TypePair typeT1 typeT2
typeOf context (Application t1 t2) =
  let
    typeT1 = typeOf context t1
    typeT2 = typeOf context t2
  in
    case typeT1 of
      TypePair typeT11 typeT12 → if typeT2 ≡ typeT11 then typeT12
                else
                  error $ "Error: Parameter type mismatch. Expecting " ++ (show ty
      _ → error $ "Error: Type pair expected not " ++ (show typeT1)
typeOf context (Fix t1) =
  let
    typeT1 = typeOf context t1
  in
    case typeT1 of
      TypePair tt1 tt2 → if tt1 ≡ tt2 then tt1
                else
                  error "Error: The function supplied to fix must have equival
      _ → error "Error: The argument to fix must be a function."
```

2.2 Unit Tests

```
typeNoTypeContext term = typeOf Empty term
basicTypeContext :: TypeContext
basicTypeContext = VariableBindings ([VarBind "a" TBool Tru, VarBind "b" TBool Fls, VarBind "c" TNat Zer
```

```
justTypeContext c t = fromJust $ getTypeFromTypeContext c t
eval1Result ctx term = fromJust $ fst $ eval1 ctx term
evaluationTests = TestList [
  "typeEvalA" ~: do
    typeNoTypeContext Zero ~?= (TNat)
  ,
  "typeEvalB" ~: do
    typeNoTypeContext Tru ~?= (TBool)
  ,
  "typeEval1" ~: do
    typeNoTypeContext (Succ (Succ Zero)) ~?= (TNat)
  ,
  "typeEval2" ~: do
    typeNoTypeContext (IsZero (Succ (Succ Zero))) ~?= (TBool)
  ,
  "typeEval3" ~: do
    typeNoTypeContext (If Tru Tru Fls) ~?= (TBool)
  ,
  "typeEval4" ~: do
    typeNoTypeContext (If Tru (Succ Zero) Zero) ~?= (TNat)
  ,
  "typeEval5" ~: do
    typeNoTypeContext (If Fls (Succ Zero) Zero) ~?= (TNat)
  ,
  "testTypeContext1" ~: do
    justTypeContext basicTypeContext (Identifier "a") ~?= (TBool)
  ,
  "testTypeContext2" ~: do
    justTypeContext basicTypeContext (Identifier "b") ~?= (TBool)
  ,
  "testTypeContext3" ~: do
    justTypeContext basicTypeContext (Identifier "c") ~?= (TNat)
  ,
  "testTypeContext3" ~: do
    getTypeFromTypeContext basicTypeContext (Identifier "ThisIsntInThere") ~?= (Nothing)
  ,
  "testTypeContext4" ~: do
    typeOf basicTypeContext (Identifier "a") ~?= (TBool)
  ,
```

```
"testTypeContext5" ~: do
   typeOf basicTypeContext (Identifier "c") ~?= (TNat)
,
"testTypeAbs1" ~: do
   typeOf basicTypeContext (Abstraction (Identifier "x") TBool Tru) ~?= (TypePair TBool TBool)
,
"testTypeAbs2" ~: do
   typeOf basicTypeContext (Abstraction (Identifier "x") TNat Tru) ~?= (TypePair TNat TBool)
,
"testTypeAbs2" ~: do
   typeOf basicTypeContext (Abstraction (Identifier "x") (TypePair TNat TBool) Tru) ~?= (TypePair (TypeP
,
"testTypeApp1" ~: do
   typeOf basicTypeContext (Application (Abstraction (Identifier "x") TBool Tru) Tru) ~?= (TBool)
,
"testTypeApp2" ~: do
   typeOf basicTypeContext (Application (Abstraction (Identifier "x") TBool (Identifier "x")) Tru) ~?= (TBoo
,
"testReplace1" ~: do
   replace (If (Identifier "x") Tru Fls) (Identifier "x") Tru ~?= (If Tru Tru Fls)
,
"testReplace2" ~: do
   replace (If (Identifier "x") (Identifier "x") (Identifier "y")) (Identifier "x") Tru ~?= (If Tru Tru (Identifie
,
"testReplace3" ~: do
   replace Tru (Identifier "x") Zero ~?= Tru
,
"testReplace4" ~: do
   replace (Abstraction (Identifier "x") TNat (Identifier "y")) (Identifier "y") (Abstraction (Identifier "y") T
,
"testReplace5" ~: do
   replace (Abstraction (Identifier "x") TNat (Identifier "y")) (Identifier "y") (Fix Tru) ~?= (Abstraction (Id
,
"testEval1" ~: do
   eval Empty (Pred (If Tru Zero (Succ (Succ Zero)))) ~?= Zero
,
"testEval2" ~: do
   eval Empty (Fix (Pred Zero)) ~?= Fix Zero
,
"testEval3" ~: do
```

```
    eval1Result Empty (Fix (Abstraction (Identifier "x") TNat (Identifier "x"))) ~?= (Fix (Abstraction (Identi
  ,
  "testEval4" ~: do
    eval1Result Empty (Fix (Abstraction (Identifier "x") TBool (IsZero (Identifier "x")))) ~?= (IsZero (Fix (A
  ]
```

3 Evaluation Relation

3.1 Implementing the evaluation relation

```
module EvaluationRelation where
import Test.HUnit
import Data.Maybe
import AbstractSyntax
import Evaluation
data Context =
      CtxHole
      | CtxAppT   Context Term
      | CtxAppV   Term Context    -- where Term is a value
      | CtxIf     Context Term Term
      | CtxFix    Context
      | CtxSucc   Context
      | CtxPred   Context
      | CtxIsZero Context
      | CtxAbs    Term Type Context
      deriving (Show, Eq)
repeatChar :: Int -> Char -> String
repeatChar n c = take n $ repeat c
highlightTerm t = "|" ++ bracket ++ (show t) ++ bracket ++ "|"
  where
    bracket = repeatChar 2 '='
highlightRedex :: Context -> Term -> String
highlightRedex CtxHole t = highlightTerm t
highlightRedex (CtxAppT ctx t1) t = "app(" ++ (highlightRedex ctx t) ++ "," ++ (show t1) ++ "
highlightRedex (CtxAppV t1 ctx) t = "app(" ++ (highlightRedex ctx t) ++ "," ++ (show t1) ++ "
highlightRedex (CtxIf ctx t1 t2) t = "if " ++ (highlightRedex ctx t) ++ " then " ++ (show t
highlightRedex (CtxFix ctx) t = "fix(" ++ (highlightRedex ctx t) ++ ")"
```

```
highlightRedex (CtxSucc ctx) t = "succ(" ++ (highlightRedex ctx t) ++ ")"
highlightRedex (CtxPred ctx) t = "pred(" ++ (highlightRedex ctx t) ++ ")"
highlightRedex (CtxIsZero ctx) t = "iszero(" ++ (highlightRedex ctx t) ++ ")"
highlightRedex (CtxAbs v vt ctx) t = "abs(" ++ (show v) ++ ":" ++ (show vt) ++ "." ++ (highlig

fillWithTerm :: Context → Term → Term
fillWithTerm CtxHole t = t
fillWithTerm (CtxAppT ctx t1) t2 = Application t1 (fillWithTerm ctx t2)
fillWithTerm (CtxAppV t1 ctx) t2 = Application (fillWithTerm ctx t2) t1
fillWithTerm (CtxIf ctx t3 t4) t2 = If (fillWithTerm ctx t2) t3 t4
fillWithTerm (CtxSucc ctx) t = Succ $ fillWithTerm ctx t
fillWithTerm (CtxPred ctx) t = Pred $ fillWithTerm ctx t
fillWithTerm (CtxIsZero ctx) t = IsZero $ fillWithTerm ctx t
fillWithTerm (CtxFix ctx) t = Fix $ fillWithTerm ctx t

makeEvalContext :: Term → Maybe (Term, Context)
makeEvalContext (Application t1 t2)
  | isValue t2 =
    case t1 of
      a@(Abstraction _ _ _) → Just (Application a t2, CtxHole)
      otherwise → Just (t1, CtxAppV t2 CtxHole)
  | otherwise = Just (t2, CtxAppT CtxHole t1)
makeEvalContext (Abstraction var varType absBody) = Just (absBody, CtxAbs var varType Ctx
makeEvalContext f@(If t1 t2 t3)
  | isValue t1 = Just (f, CtxHole)
  | otherwise = Just (t1, CtxIf CtxHole t2 t3)
makeEvalContext s@(Succ t)
  | isValue s = Nothing
  | otherwise = Just (t, CtxSucc CtxHole)
makeEvalContext (Pred t)
  | isNumericValue t = Just (Pred t, CtxHole)
  | isValue t = Nothing
  | otherwise = Just (t, CtxPred CtxHole)
makeEvalContext (Fix t) = Just (t, CtxFix CtxHole)
makeEvalContext (IsZero t) = Just (t, CtxIsZero CtxHole)
makeEvalContext _ = Nothing

makeContractum :: Term → Term
makeContractum t = fromJust $ fst $ eval1 Empty t

machineStep :: Term → Maybe Term
machineStep t = do
  (t1, c) ← makeEvalContext t
```

```
    Just (fillWithTerm c (makeContractum t1))
machineEval :: Term → Term
machineEval t =
  case machineStep t of
    Just t' → machineEval t'
    Nothing → t
```

The tracing evaluation is defined as follows using the functions defined for machine evaluation. The tracing evaluator prints one line for each step of evaluation and brackets the redex being considered at that step.

```
traceEval :: Term → IO ()
traceEval t = do
  case makeEvalContext t of
    Just (t1, c) → do
      putStrLn $ "[CHOSEN REDEX]\n" ++ (highlightRedex c t1)
      case machineStep t of
        Just t' → do
          putStrLn $ "[RESULT]\n" ++ (show t')
          traceEval t'
        Nothing → putStrLn "Stuck!"
    Nothing → putStrLn "---FINISHED EVALUATING---"
```

3.2 Unit Tests

```
justMachineStep = fromJust ∘ machineStep
justmakeEvalContext = fromJust ∘ makeEvalContext
evalRelTests = TestList [
  "testEvalRelStep1" ~: do
    justMachineStep (Succ (If Tru Tru Fls)) ~? = (Succ Tru)
  ,
  "testEvalRelTot1" ~: do
    machineEval (Succ (If Tru (Succ Zero) Zero)) ~? = Succ (Succ (Zero))
  ,
  "testEvalRelTot2" ~: do
    machineEval (Pred Zero) ~? = Zero
  ,
  "testEvalRelTot3" ~: do
    machineEval (Pred (If Tru Zero (Succ (Succ Zero)))) ~? = Zero
```

```
  ,
  "testEvalRelTot4" ~: do
    machineEval (Application (Abstraction (Identifier "x") TNat (Succ (Succ (Identifier "x")))) (Succ Zero)) ~?
  ,
  "testMakeContext1" ~: do
    justmakeEvalContext (Succ (If Tru Tru Fls)) ~?= (If Tru Tru Fls, CtxSucc CtxHole)
  ,
  "testMakeContext2" ~: do
    justmakeEvalContext (IsZero (Pred (Succ Zero))) ~?= (Pred (Succ Zero), CtxIsZero CtxHole)
  ,
  "testMakeContext3" ~: do
    justmakeEvalContext (If (IsZero Zero) Tru Fls) ~?= (IsZero Zero, CtxIf CtxHole Tru Fls)
  ,
  "testMakeContext4" ~: do
    justmakeEvalContext (Application (IsZero Zero) (If Tru Tru Fls)) ~?= (If Tru Tru Fls, CtxAppT CtxHole (
  ,
  "testMakeContext5" ~: do
    justmakeEvalContext (Application (Abstraction (Identifier "x") TNat (Identifier "x")) (Succ (Succ Zero))) ~
  ,
  "testMakeContractum1" ~: do
    makeContractum (If Tru Tru Fls) ~?= Tru
  ,
  "testMakeContractum2" ~: do
    makeContractum (Pred (Succ Zero)) ~?= Zero
  ,
  "testMakeContractum3" ~: do
    makeContractum (IsZero Zero) ~?= Tru
  ,
  "testFillWithTerm1" ~: do
    fillWithTerm (CtxSucc CtxHole) Tru ~?= Succ Tru
  ,
  "testFillWithTerm2" ~: do
    fillWithTerm (CtxIsZero CtxHole) Zero ~?= IsZero Zero
  ,
  "testFillWithTerm3" ~: do
    fillWithTerm (CtxIf CtxHole Tru Fls) Tru ~?= If Tru Tru Fls
  ]
runEvalRelationTests = runTestTT evalRelTests
```

4 Unification

```
module Unification where
import Test.HUnit
import Data.List (find, nub)
import System.Environment (getArgs)
```

4.1 Datatype Definitions

```
data Term v f =
  Fun f [Term v f]
  | Var v
  deriving (Show, Eq)
type Equation v f     = (Term v f, Term v f)
type Binding v f      = (v, Term v f)
type Substitution v f = [Binding v f]
data EquationOutcome = HaltWithFailure | HaltWithCycle | NoMatch | Success deriving (Shou
```

4.2 Implementing the Unification Algorithm

The unification implementation below is the canonical nondeterministic algorithm as described in the Martelli and Montanari 1982 paper.

```
applySubst :: (Eq v, Eq f) ⇒ Term v f → Substitution v f → Term v f
applySubst (Var x) theta =
  case find (λ(z, _) → z ≡ x) theta of
    Nothing → Var x
    Just (_, t) → t
applySubst (Fun f tlist) theta = Fun f (map (λt → applySubst t theta) tlist)
applySubst′ :: (Eq v, Eq f) ⇒ [Equation v f] → Substitution v f → [Equation v f]
applySubst′ eql theta = map (λ(s, t) → (applySubst s theta, applySubst t theta)) eql
```

The occurs function returns true if the given variable occurs in the given term, false otherwise. Used to check for a cycle when defining variables.

```
occurs :: (Eq v, Eq f) ⇒ Term v f → Term v f → Bool
occurs v@(Var x) (Fun f tlist)
```

$$
\begin{aligned}
&\quad \mid \mathit{elem}\ v\ \mathit{tlist} = \mathit{True} \\
&\quad \mid \mathit{otherwise} = \mathit{or}\ \$\ \mathit{map}\ (\mathit{occurs}\ v)\ \mathit{tlist} \\
&\mathit{occurs}\ \_\ \_ = \mathit{False}
\end{aligned}
$$

$$\mathit{unify} :: (\mathit{Eq}\ v, \mathit{Eq}\ f) \Rightarrow [\mathit{Equation}\ v\ f] \rightarrow (\mathit{EquationOutcome}, [\mathit{Equation}\ v\ f])$$

Rewrite $term = variable$ as $variable = term$.

$$
\begin{aligned}
&\mathit{unify}\ ((f@(\mathit{Fun}\ \_\ \_), v@(\mathit{Var}\ \_)) : \mathit{eqns}) = \mathit{unify}\ \$\ (v, f) : \mathit{eqns} \\
&\mathit{unify}\ (f@(a@(\mathit{Var}\ \mathit{v1}), b@(\mathit{Var}\ \_)) : \mathit{eqns}) \\
&\quad \mid a \equiv b = \mathit{unify}\ \mathit{eqns} \quad \text{-- Erase redundant equations.} \\
&\quad \mid \mathit{otherwise} = (\mathit{result}, f : \mathit{rest}) \\
&\quad\ \ \mathbf{where} \\
&\quad\quad (\mathit{result}, \mathit{rest}) = \mathit{unify}\ \$\ \mathit{applySubst}'\ \mathit{eqns}\ [(\mathit{v1}, b)] \quad \text{-- Do the substitution and continue.}
\end{aligned}
$$

Equations of the form $t' = t''$. If the two root function symbols are different, stop with failure; otherwise, apply term reduction.

$$
\begin{aligned}
&\mathit{unify}\ e@(((\mathit{Fun}\ a\ b), (\mathit{Fun}\ c\ d)) : \mathit{eqns}) \\
&\quad \mid a \not\equiv c = (\mathit{HaltWithFailure}, e) \\
&\quad \mid \mathit{otherwise} = \mathit{unify}\ (\mathit{eqns} +\!\!+ (\mathit{zip}\ b\ d))
\end{aligned}
$$

Select any equation of the form

$$x = t$$

where x is a variable which occurs somewhere else in the set of equations and where $t \neq x$. If x occurs in t, then stop with failure; otherwise, apply variable elimination.

$$
\begin{aligned}
&\mathit{unify}\ e@(t@(v@(\mathit{Var}\ \mathit{varName}), f@(\mathit{Fun}\ \mathit{funcName}\ \mathit{tlist})) : \mathit{eqns}) \\
&\quad \mid \mathit{occurs}\ v\ f = (\mathit{HaltWithCycle}, e) \\
&\quad \mid \mathit{otherwise} = (\mathit{result}, t : \mathit{rest}) \\
&\quad\ \ \mathbf{where} \\
&\quad\quad (\mathit{result}, \mathit{rest}) = \mathit{unify}\ \$\ \mathit{applySubst}'\ \mathit{eqns}\ [(\mathit{varName}, f)]
\end{aligned}
$$

If no transformation applies, stop with success.

$$\mathit{unify}\ \mathit{eqns} = (\mathit{Success}, \mathit{eqns})$$

4.3 Testing

4.3.1 Example Unification Problems

$$
\begin{aligned}
&\mathit{s1} = \mathit{Fun}\ \texttt{"f"}\ [\mathit{Fun}\ \texttt{"g"}\ [\mathit{Fun}\ \texttt{"a"}\ [\,], \mathit{Var}\ \texttt{"X"}], \mathit{Fun}\ \texttt{"h"}\ [\mathit{Fun}\ \texttt{"f"}\ [\mathit{Var}\ \texttt{"Y"}, \mathit{Var}\ \texttt{"Z"}]]] \\
&\mathit{s2} = \mathit{Fun}\ \texttt{"g"}\ [\mathit{Var}\ \texttt{"Y"}, \mathit{Fun}\ \texttt{"h"}\ [\mathit{Fun}\ \texttt{"f"}\ [\mathit{Var}\ \texttt{"Z"}, \mathit{Var}\ \texttt{"U"}]]]
\end{aligned}
$$

```haskell
t1 = Fun "f" [Var "U", Fun "h" [Fun "f" [Var "X", Var "X"]]]
t2 = Fun "g" [Fun "f" [Fun "h" [Var "X"], Fun "a" []],
  Fun "h" [Fun "f" [Fun "a" [], Fun "b" []]]]
problem1 = [(s1, t1), (s2, t2)]
solution1 = unify problem1
a1 = Var "X"
p1 = [(a1, a1)] :: [Equation String String]
p2 = [(Fun "a" [], Var "X")] :: [Equation String String]
p3 = [(Fun "a" [], Fun "b" [])] :: [Equation String String]
p4 = [(Fun "a" [Var "X"], Fun "a" [Var "X"])] :: [Equation String String]
p5 = [(Var "X", Var "Y"), (Var "Y", Var "X")] :: [Equation String String]
p6 = [(Var "X", Var "Y"), (Fun "a" [], Var "X")] :: [Equation String String]
```

Example from the Martelli and Montanari paper.

```haskell
p7 = [
  (Fun "g" [Var "X2"],
    Var "X1"),
  (Fun "f" [Var "X1", Fun "h" [Var "X1"], Var "X2"],
    Fun "f" [Fun "g" [Var "X3"], Var "X4", Var "X3"])
  ] :: [Equation String String]
p7' = [
  (Fun "f" [Var "X1", Fun "h" [Var "X1"], Var "X2"],
    Fun "f" [Fun "g" [Var "X3"], Var "X4", Var "X3"])
  ] :: [Equation String String]
p9 = [
  (Fun "f" [Var "X", Var "Y"],
    Fun "f" [Fun "a" [], Fun "b" []])
  ] :: [Equation String String]
p10 = [
  (Fun "f" [Var "X", Var "Y"],
    Fun "f" [Fun "a" [], Fun "b" []]),
  (Fun "g" [Var "X", Var "Y"],
    Fun "g" [Fun "c" [], Fun "d" []])
  ] :: [Equation String String]
p11 = [
  (Fun "f" [Var "X", Var "Y"],
    Fun "f" [Fun "a" [], Fun "b" []]),
  (Fun "g" [Var "S", Var "T"],
    Fun "g" [Fun "c" [Var "X"], Var "Y"])
```

```
  ] :: [Equation String String]
p12 = [
  (Fun "f" [Var "X", Var "Y"],
    Fun "f" [Fun "a" [], Fun "b" [Var "Z", Var "Q", Var "T"]]),
  (Fun "g" [Var "S", Var "T"],
    Fun "g" [Fun "c" [Var "X"], Var "Y"])
  ] :: [Equation String String]
p13 = [
  (Fun "f" [Var "X", Var "Y"],
    Fun "f" [Fun "a" [], Fun "b" [Var "Z", Var "Q", Var "R"]]),
  (Fun "g" [Var "S", Var "T"],
    Fun "g" [Fun "c" [Var "X"], Var "Y"]),
  (Fun "qconst" [], Var "Q"),
  (Fun "zconst" [], Var "Z"),
  (Var "R", Var "X")
  ] :: [Equation String String]
```

4.4 Unit Tests

The unit tests provided are intended to cover many of the unification use cases and corner cases such that the implementation under test can be determined as correct according to the spec.

```
unificationTests = TestList [
  "occurstest1" ~: do
    occurs (Var "X") (Fun "a" [Var "X"]) ~?= True

  ,
  "occurstest2" ~: do
    occurs (Var "X") (Fun "a" [Var "Y"]) ~?= False

  ,
  "occurstest3" ~: do
    occurs (Var "X") (Fun "a" [Fun "b" [Var "X"]]) ~?= True

  ,
  "occurstest4" ~: do
    occurs (Var "X") (Fun "a" [Fun "haha"
      [Fun "foo" [Var "X"]]]) ~?= True

  ,
  "occurstest5" ~: do
    occurs (Var "X") (Fun "a" [Fun "haha"
      [Fun "foo" [Var "Y"]]]) ~?= False
```

```
,
"occurstest6" ~: do
  occurs (Var "X") (Fun "a" [Fun "haha"
    [Fun "foo" [Var "Z", Var "Y", Var "X"]]]) ~?= True
,
"occurstest7" ~: do
  occurs (Var "X") (Fun "a"
    [Fun "haha" [Fun "foo" [Var "Z", Var "Y", Var "HAX"]]]) ~?= False
,
"test1" ~: do
  unify p1 ~?= (Success, [])
,
"test2" ~: do
  unify p2 ~?= (Success, [(Var "X", Fun "a" [])])
,
"test3" ~: do
  unify p3 ~?= (HaltWithFailure, p3)
,
"test4" ~: do
  unify p4 ~?= (Success, [])
,
"test5" ~: do
  unify p5 ~?= (Success, [(Var "X", Var "Y")])
,
"test6" ~: do
  unify p6 ~?= (Success, [(Var "X", Var "Y"), (Var "Y", Fun "a" [])])
,
"test7" ~: do
  unify p7 ~?= (Success, [(Var "X1", Fun "g" [Var "X2"]),
    (Var "X4", Fun "h" [Fun "g" [Var "X2"]]), (Var "X2", Var "X3")])
,
"test8" ~: do
  fst solution1 ~?= HaltWithCycle
,
"test9" ~: do
  unify p9 ~?= (Success, [(Var "X", Fun "a" []), (Var "Y", Fun "b" [])])
,
"test10" ~: do
  fst (unify p10) ~?= HaltWithFailure
,
```

```
"test11" ~: do
  unify p11 ~?= (Success,
    [(Var "X", Fun "a" []),
    (Var "Y", Fun "b" []),
    (Var "S", Fun "c" [Fun "a" []]),
    (Var "T", Fun "b" [])])
,
"test12" ~: do
  fst (unify p12) ~?= HaltWithCycle
,
"test13" ~: do
  unify p13 ~?= (Success, [(Var "Q", Fun "qconst" []),
    (Var "Z", Fun "zconst" []),
    (Var "R", Var "X"),
    (Var "X", Fun "a" []),
    (Var "Y", Fun "b" [Fun "zconst" [], Fun "qconst" [], Fun "a" []]), (Var "S", Fun "c" [Fun "a" []]),
    (Var "T", Fun "b" [Fun "zconst" [], Fun "qconst" [], Fun "a" []])])
]
```

5 Parser Combinator Definitions

Definitions of our parser combinators.

```
module Parser where
import Test.HUnit
import Data.Either.Unwrap (fromRight)
import Text.ParserCombinators.Parsec
import AbstractSyntax
import Evaluation
import ConstraintTyping
true = ignoreWhitespace $ string "true"
false = ignoreWhitespace $ string "false"
bool :: Parser Term
bool =
  (true >> return Tru)
  <|> (false >> return Fls)
zero  = ignoreWhitespace $ string "0"
tSucc =
```

```
  do
    ignoreWhitespace $ string "succ"
    t ← parens term
    return (Succ t)
tPred =
  do
    ignoreWhitespace $ string "pred"
    t ← parens term
    return (Pred t)
nat :: Parser Term
nat =
  (zero ≫ return Zero)
  <|>
  tSucc
  <|>
  tPred
```

Utility function that takes a parser and ignores the whitespace around it. Useful since our language does not depend on whitespace for syntactic meaning.

```
ignoreWhitespace :: Parser a → Parser a
ignoreWhitespace parser = do
  spaces
  a ← parser
  spaces
  return a
```

Note that we're going to define an identifier as a string of one or more lower case letters. The specification didn't formally define what an identifier can contain, so we're going to be consistent with the book and class examples and assume that it is a string of lower case letters.

```
identifier :: Parser Term
identifier = do
  l ← ignoreWhitespace (many1 lower)
  return $ Identifier l
arr    = ignoreWhitespace $ string "arr"
pAbs   = ignoreWhitespace $ string "abs"
app    = ignoreWhitespace $ string "app"
lpar   = ignoreWhitespace $ string "("
```

```
rpar     = ignoreWhitespace $ string ")"
parens p =
  do
      lpar
      a ← p
      rpar
      return a
comma    = ignoreWhitespace $ string ","
colon    = ignoreWhitespace $ string ":"
fullstop = ignoreWhitespace $ string "."
iszero   = ignoreWhitespace $ string "iszero"
fix      = ignoreWhitespace $ string "fix"

pIf :: Parser Term
pIf =
  do
      ignoreWhitespace $ string "if"
      t1 ← term
      ignoreWhitespace $ string "then"
      t2 ← term
      ignoreWhitespace $ string "else"
      t3 ← term
      ignoreWhitespace $ string "fi"
      return (If t1 t2 t3)

pLet :: Parser Term
pLet =
  do
      ignoreWhitespace $ string "let"
      name ← identifier
      ignoreWhitespace $ string "="
      t1 ← term
      ignoreWhitespace $ string "in"
      t2 ← term
      return (replace t2 name t1)

pType :: Parser Type
pType =
  do
      ignoreWhitespace $ string "Bool"
      return TBool
   <|>
```

```
  do
      ignoreWhitespace $ string "Nat"
      return TNat
  <|>
  do
      arr
      lpar
      t1 ← pType
      comma
      t2 ← pType
      rpar
      return (TypePair t1 t2)
explicitAbs :: Parser Term
explicitAbs =
  do
      pAbs
      lpar
      name ← identifier
      colon
      type1 ← pType
      fullstop
      term1 ← term
      rpar
      return (Abstraction name type1 term1)
implicitAbs :: Parser Term
implicitAbs =
  do
      pAbs
      lpar
      name ← identifier
      fullstop
      term1 ← term
      rpar
      return (Abstraction name (generateFreshVariable term1) term1)
term =
  do
      try $ pLet
  <|>
  do
```

```
        try $ pIf
  <|>
  do
        try $ fix
        t <- parens term
        return (Fix t)
  <|>
  do
        try $ iszero
        t <- parens term
        return (IsZero t)
  <|>
  do
        try $ explicitAbs
  <|>
  do
        try $ implicitAbs
  <|>
  do
        try $ app
        lpar
        t1 <- term
        comma
        t2 <- term
        rpar
        return (Application t1 t2)
  <|> try (bool) <|> try (nat) <|> try (parens term) <|> try (identifier)
```

5.1 Unit Tests

Some helper functions and constants for running our unit tests.

```
parseEither p input = parse p "" input
parseRight p input = fromRight $ parseEither p input
```

```
parserTests = TestList [
  "variableTest1" ~: do
    parseRight term "a" ~?= (Identifier "a")
```

```
,
"variableTest2" ~: do
  parseRight term "(a)" ~?= (Identifier "a")
,
"IfTest1" ~: do
  parseRight term "if true then true else false fi" ~?= (If Tru Tru Fls)
,
"typeTest1" ~: do
  parseRight pType "Bool" ~?= (TBool)
,
"typeTest2" ~: do
  parseRight pType "Nat" ~?= (TNat)
,
"typeTest3" ~: do
  parseRight pType "arr(Bool,Bool)" ~?= (TypePair TBool TBool)
,
"typeTest4" ~: do
  parseRight pType "arr(Nat,arr(Nat,Bool))" ~?= (TypePair TNat (TypePair TNat TBool))
,
"letTest1" ~: do
  parseRight pLet "let a = true in a" ~?= Tru
,
"letTest2" ~: do
  parseRight pLet "let a = 0 in (iszero(a))" ~?= IsZero Zero
,
"letTest3" ~: do
  parseRight pLet "let a = true in let b = 0 in a" ~?= Tru
,
"letTest4" ~: do
  parseRight term "let a = true in let b = 0 in if a then succ(b) else b fi" ~?= If Tru (S
,
"implicitTest1" ~: do
  parseRight term "abs(x.x)" ~?= Abstraction (Identifier "x") (TVar "A") (Identifier "x")
,
"implicitTest2" ~: do
  parseRight term "abs(x.abs(y.y))" ~?= Abstraction (Identifier "x") (TVar "B") (Abstraction (Identifie
]
```

6 Constraint-Based Typing

```haskell
module ConstraintTyping where

import Test.HUnit
import Data.Maybe
import Data.List
import qualified AbstractSyntax as S
import qualified Unification as U
import Evaluation (replace)

type TypeConstraint = (S.Type, S.Type)
type TypeConstraintSet = [TypeConstraint]
type TypeSubstitution = [(S.TypeVar, S.Type)]
```

6.1 Implemented Functions

The following are the definitions required to complete the constraint based typing implementation. The functions deriveTypeConstraints and applyTypeSubstitutionToTerm were required to be completed before a working constraint based typing system could be formed.

Page 322 of TAPL was helpful when writing the deriveTypeConstraints function. An effort was made to transcribe figure 22-1 in Haskell as close as possible to the original descriptions.

```haskell
newtype Id = Id Char deriving (Show, Eq)
instance Enum Id where
  succ (Id name)     = (Id (succ name))
  pred (Id name)     = (Id (pred name))
  toEnum int         = Id $ ['A' ..] !! int
  fromEnum (Id name) = fromJust $ elemIndex name ['A' ..]

toVar :: Id → S.Type
toVar (Id c) = S.TVar [c]

fromVar :: S.TypeVar → Id
fromVar var = Id (head var)

toTypeVar (Id c) = [c]

type NextId = Id

  -- We have to keep track of our Identifiers and the type variables they're associated with.
type IdBinding = (S.Term, S.TypeVar)
```

```haskell
type IdBindingSet = [IdBinding]
variables = map (λx → [x]) ['A' ..] :: [S.TypeVar]
generateFreshVariable :: S.Term → S.Type
generateFreshVariable term = head [S.TVar x | x ← variables, ¬ (x ∈ fvs)]
  where
    fvs = freeVariables term
freeVariables :: S.Term → [S.TypeVar]
freeVariables a@(S.Abstraction t ttype body) = (fvs ttype) ++ (freeVariables body)
  where
    fvs :: S.Type → [S.TypeVar]
    fvs f@(S.TVar name) = [name]
    fvs f@(S.TypePair t1 t2) = (fvs t1) ++ (fvs t2)
freeVariables (S.Application t1 t2) = (freeVariables t1) ++ (freeVariables t2)
freeVariables (S.If t1 t2 t3) = (freeVariables t1) ++ (freeVariables t2) ++ (freeVariables t3)
freeVariables (S.Succ t1) = (freeVariables t1)
freeVariables (S.Pred t1) = (freeVariables t1)
freeVariables (S.IsZero t1) = (freeVariables t1)
freeVariables (S.Fix t1) = (freeVariables t1)
freeVariables t = []
getBinding :: IdBindingSet → S.Term → Maybe S.TypeVar
getBinding idbs t@(S.Identifier name) =
  case find (λa → (fst a) ≡ t) idbs of
    Just u → Just (snd u)
    Nothing → Nothing
getBinding _ _ = error "Can't get binding for non-identifier."
replaceConstraint :: TypeConstraintSet → S.Type → S.Type → TypeConstraintSet
replaceConstraint cstSet t t' = [if tvar ≡ t then (t', ttype) else u | u@(tvar, ttype) ← cstSet]
deriveTypeConstraints :: S.Term → TypeConstraintSet
deriveTypeConstraints t = nub $ solution   -- Remove all redundant constraints.
  where
    (_, solution, _) = constraintHelper t (Id 'A') []   -- Delegate to our helper function.
    constraintHelper :: S.Term → Id → IdBindingSet → (NextId, TypeConstraintSet, IdBindingS
    constraintHelper t@(S.Pred t1) i idbs =
      let
        v0 = toVar i
        v1 = toVar $ succ i
        nextid = succ $ succ i
        (t1id, c1, idbs1) = constraintHelper t1 nextid idbs
```

```
    c1'    =
      case t1 of
        S.Identifier _ → replaceConstraint c1 (toVar nextid) v1
        S.Application _ _ → replaceConstraint c1 (toVar nextid) v1
        otherwise → c1
  in
    (t1id, (v0, S.TNat) : (v1, S.TNat) : c1', idbs1)
constraintHelper t@(S.Succ t1) i idbs =
  let
    v0 = toVar i
    v1 = toVar $ succ i
    nextid = succ $ succ i
    (t1id, c1, idbs1) = constraintHelper t1 nextid idbs
    c1'    =
      case t1 of
        S.Identifier _ → replaceConstraint c1 (toVar nextid) v1
        S.Application _ _ → replaceConstraint c1 (toVar nextid) v1
        otherwise → c1
  in
    (t1id, (v0, S.TNat) : (v1, S.TNat) : c1', idbs1)
constraintHelper t@(S.IsZero t1) i idbs =
  let
    v0 = toVar i
    v1 = toVar $ succ i
    nextid = succ $ succ i
    (t1id, c1, idbs1) = constraintHelper t1 nextid idbs
    c1'    =
      case t1 of
        S.Identifier _ → replaceConstraint c1 (toVar nextid) v1
        S.Application _ _ → replaceConstraint c1 (toVar nextid) v1
        otherwise → c1
  in
    (t1id, (v0, S.TBool) : (v1, S.TNat) : c1', idbs1)
constraintHelper t@(S.If t1 t2 t3) i idbs =
  let
    v1 = toVar i
    v2 = toVar $ succ i
    v3 = toVar $ succ $ succ i
    v0 = toVar $ succ $ succ $ succ i
    nextid = succ $ succ $ succ $ succ i
```

```
    (t1id, c1, idbs1) = constraintHelper t1 nextid idbs
    c1'     = replaceConstraint c1 (toVar nextid) v1
    (t2id, c2, idbs2) = constraintHelper t2 t1id idbs1
    c2'     = replaceConstraint c2 (toVar t1id) v2
    (t3id, c3, idbs3) = constraintHelper t3 t2id idbs2
    c3'     = replaceConstraint c3 (toVar t2id) v3
  in
    (t3id, (v1, S.TBool) : (v2, v3) : (v0, v2) : (c1' ++ c2' ++ c3'), idbs3)
constraintHelper t@(S.Application t1 t2) i idbs =
  let
    v0      = toVar i
    v1      = toVar $ succ i
    v2      = toVar $ succ $ succ i
    nextid = succ $ succ $ succ i
    (t1id, c1, idbs1) = constraintHelper t1 nextid idbs
    c1'     = replaceConstraint c1 (toVar nextid) v1
    (t2id, c2, idbs2) = constraintHelper t2 t1id idbs1
    c2'     = replaceConstraint c2 (toVar t1id) v2
  in
    (nextid, (v0, v0) : (v1, S.TypePair v2 v0) : (c1' ++ c2'), idbs2)
constraintHelper t@(S.Abstraction tvar ttype tbody) i idbs =
  let
    v0      = toVar i
    v1      = toVar $ succ i
    v2      = toVar $ succ $ succ i
    nextid = succ $ succ $ succ i
    idbs'   =
      case ttype of
        S.TVar name -> (tvar, name) : idbs
        otherwise -> idbs
    (t1id, c2, idbs1) = constraintHelper tbody nextid idbs'
    c2' = replaceConstraint c2 (toVar nextid) v2
  in
    (t1id, (v0, S.TypePair v1 v2) : (v1, ttype) : c2', idbs1)
constraintHelper t@(S.Tru) i idbs =
  let
    v0 = toVar i
    nextid = succ i
  in
    (nextid, [(v0, S.TBool)], idbs)
```

```
constraintHelper t@(S.Fls) i idbs =
  let
    v0 = toVar i
    nextid = succ i
  in
    (nextid, [(v0, S.TBool)], idbs)
constraintHelper t@(S.Zero) i idbs =
  let
    v0 = toVar i
    nextid = succ i
  in
    (nextid, [(v0, S.TNat)], idbs)
constraintHelper t@(S.Identifier _) i idbs =
  let
    v0 = toVar i
    nextid = succ i
  in
    case getBinding idbs t of
      Just tvar → (nextid, [(v0, S.TVar tvar)], idbs)
      Nothing → (nextid, [ ], idbs)
constraintHelper _ i idbs = (i, [ ], idbs)
```

Note that we only care about substituting the types that are annotated in abstractions, since this is the only point where these type variables can occur in our language.

```
replaceTVar :: TypeSubstitution → S.Type → S.Type
replaceTVar typesubst tv@(S.TVar tvarName) =
  let
    substs = [tt | (tvar, tt) ← typesubst, tvar ≡ tvarName]
  in
    if (length substs) > 0
      then
        replaceTVar typesubst (head substs)
      else
        tv
replaceTVar typesubst tp@(S.TypePair t1 t2) = S.TypePair (replaceTVar typesubst t1) (replace
replaceTVar typesubst t = t

applyTypeSubstitutionToTerm :: TypeSubstitution → S.Term → S.Term
applyTypeSubstitutionToTerm typesubst a@(S.Abstraction t ttype body) = S.Abstraction t (repla
```

```
applyTypeSubstitutionToTerm typesubst (S.Application t1 t2) =
  S.Application (applyTypeSubstitutionToTerm typesubst t1) (applyTypeSubstitutionToTerm typ
applyTypeSubstitutionToTerm typesubst (S.If t1 t2 t3) =
  S.If (applyTypeSubstitutionToTerm typesubst t1) (applyTypeSubstitutionToTerm typesubst t2)
applyTypeSubstitutionToTerm typesubst (S.Succ t1) = S.Succ (applyTypeSubstitutionToTerm typ
applyTypeSubstitutionToTerm typesubst (S.Pred t1) = S.Pred (applyTypeSubstitutionToTerm ty
applyTypeSubstitutionToTerm typesubst (S.IsZero t1) = S.IsZero (applyTypeSubstitutionToTerm
applyTypeSubstitutionToTerm typesubst (S.Fix t1) = S.Fix (applyTypeSubstitutionToTerm types
applyTypeSubstitutionToTerm _ t = t

unifyTerm t =
  let
    constraints = deriveTypeConstraints t
    unifencoding = encode constraints
    u@(unifoutcome, unifsolvedequations) = U.unify unifencoding
  in
    u
```

6.2 Provided Code

The following code was provided as a template in order to implement constraint based typing. The template was updated to work with the AbstractSyntax module that I have defined previously.

```
reconstructType :: S.Term → Maybe S.Term
reconstructType t =
  let
    constraints = deriveTypeConstraints t
    unifencoding = encode constraints
    (unifoutcome, unifsolvedequations) = U.unify unifencoding
  in case unifoutcome of
    U.Success →
      let
        typesubst = decode unifsolvedequations
        t′ = applyTypeSubstitutionToTerm typesubst t
      in
        Just t′
    U.HaltWithFailure → Nothing
    U.HaltWithCycle → Nothing
type TypeUnifVar = S.TypeVar
```

```haskell
data TypeUnifFun = TypeUnifArrow | TypeUnifBool | TypeUnifNat
  deriving (Eq, Show)

encode :: TypeConstraintSet → [U.Equation TypeUnifVar TypeUnifFun]
encode = map (λ(tau1, tau2) → (enctype tau1, enctype tau2))
  where
    enctype :: S.Type → U.Term TypeUnifVar TypeUnifFun
    enctype (S.TypePair tau1 tau2) = U.Fun TypeUnifArrow [enctype tau1, enctype tau2]
    enctype S.TBool                = U.Fun TypeUnifBool []
    enctype S.TNat                 = U.Fun TypeUnifNat []
    enctype (S.TVar xi)            = U.Var xi

decode :: [U.Equation TypeUnifVar TypeUnifFun] → TypeSubstitution
decode = map f
  where
    f :: (U.Term TypeUnifVar TypeUnifFun, U.Term TypeUnifVar TypeUnifFun)
      → (S.TypeVar, S.Type)
    f (U.Var xi, t) = (xi, g t)
    g :: U.Term TypeUnifVar TypeUnifFun → S.Type
    g (U.Fun TypeUnifArrow [t1, t2]) = S.TypePair (g t1) (g t2)
    g (U.Fun TypeUnifBool []) = S.TBool
    g (U.Fun TypeUnifNat [])  = S.TNat
    g (U.Var xi)              = S.TVar xi
```

6.3 Unit Tests

```haskell
justreconstructType = fromJust ∘ reconstructType

constraintBasedTypingTests = TestList [
  "substTest1" ~: do
    applyTypeSubstitutionToTerm [("X", S.TBool)] (S.Abstraction (S.Identifier "a") (S.TVar "X") S.Tru)
    ~? = (S.Abstraction (S.Identifier "a") S.TBool S.Tru)
  ,
  "substTest2" ~: do
    applyTypeSubstitutionToTerm [("X", S.TBool), ("Y", S.TNat)] (S.Abstraction (S.Identifier "a") (S.TVar "
    ~? = (S.Abstraction (S.Identifier "a") S.TBool S.Tru)
  ,
  "substTest3" ~: do
    applyTypeSubstitutionToTerm [("X", S.TBool), ("Y", S.TNat)] (S.Abstraction (S.Identifier "a") (S.TVar "
    ~? = (S.Abstraction (S.Identifier "a") S.TBool (S.Abstraction (S.Identifier "b") S.TNat S.Tru))
  ,
```

```
"substTest4" ~: do
  applyTypeSubstitutionToTerm [("X", S.TBool), ("Y", S.TNat)] (S.Application (S.Abstraction (S.Identifier "
  ~? = (S.Application (S.Abstraction (S.Identifier "a") S.TBool S.Tru) (S.Abstraction (S.Identifier "b") S.T
,
"substTest5" ~: do
  applyTypeSubstitutionToTerm [("X", S.TBool), ("Y", S.TNat)] (S.Abstraction (S.Identifier "a") (S.TypePai
  ~? = (S.Abstraction (S.Identifier "a") (S.TypePair S.TBool S.TNat) S.Tru)
,
"constrTest1" ~: do
  deriveTypeConstraints (S.Pred (S.Identifier "x"))
  ~? = [(S.TVar "A", S.TNat), (S.TVar "B", S.TNat)]
,
"constrTest2" ~: do
  deriveTypeConstraints (S.Succ (S.Identifier "x"))
  ~? = [(S.TVar "A", S.TNat), (S.TVar "B", S.TNat)]
,
"constrTest3" ~: do
  deriveTypeConstraints (S.IsZero (S.Identifier "x"))
  ~? = [(S.TVar "A", S.TBool), (S.TVar "B", S.TNat)]
,
"constrTest4" ~: do
  deriveTypeConstraints (S.If (S.Identifier "x") S.Tru S.Fls)
  ~? = [(S.TVar "A", S.TBool), (S.TVar "B", S.TVar "C"),
  (S.TVar "D", S.TVar "B"),
  (S.TVar "B", S.TBool),
  (S.TVar "C", S.TBool)]
,
  -- "constrTest6" : do
  -- deriveTypeConstraints (S.Application (S.Identifier "x") S.Tru)
  -- ?= [(S.TVar "B",S.TypePair (S.TVar "C") (S.TVar "A")),(S.TVar "C",S.TBool)]
  -- ,
"constrTest7" ~: do
  deriveTypeConstraints (S.Abstraction (S.Identifier "x") S.TBool S.Tru)
  ~? = [(S.TVar "A", S.TypePair (S.TVar "B") (S.TVar "C")), (S.TVar "B", S.TBool), (S.TVar "C", S.TBo
,
"constrTest9" ~: do
  deriveTypeConstraints (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Identifier "x"))
  ~? = [(S.TVar "A", S.TypePair (S.TVar "B") (S.TVar "C")), (S.TVar "B", S.TVar "X"), (S.TVar "C", S.
,
```

Keeping track of the order which the variables are generated made testing this difficult, the more illustrative tests are the reconstructive tests below.

```
"reconstructTypeTest1" ~: do
   justreconstructType (S.Pred S.Zero)
    ~? = (S.Pred S.Zero)
,
"reconstructTypeTest2" ~: do
   justreconstructType (S.Succ S.Zero)
    ~? = (S.Succ S.Zero)
,
"reconstructTypeTest3" ~: do
   justreconstructType (S.IsZero S.Zero)
    ~? = (S.IsZero S.Zero)
,
"reconstructTypeTest4" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.IsZero (S.Identifier "x")))
    ~? = (S.Abstraction (S.Identifier "x") S.TNat (S.IsZero (S.Identifier "x")))
,
"reconstructTypeTest5" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Pred (S.Identifier "x")))
    ~? = (S.Abstraction (S.Identifier "x") S.TNat (S.Pred (S.Identifier "x")))
,
"reconstructTypeTest6" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Succ (S.Identifier "x")))
    ~? = (S.Abstraction (S.Identifier "x") S.TNat (S.Succ (S.Identifier "x")))
,
"reconstructTypeTest7" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.If (S.Identifier "x") S.Tru S.Fls))
    ~? = (S.Abstraction (S.Identifier "x") S.TBool (S.If (S.Identifier "x") S.Tru S.Fls))
,
"reconstructTypeTest8" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.If (S.Identifier "x") (S.Identifier "x")
    ~? = (S.Abstraction (S.Identifier "x") S.TBool (S.If (S.Identifier "x") (S.Identifier "x") (S.Identifier "x")))
,
"reconstructTypeTest9" ~: do
   justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Pred (S.If (S.Identifier "x") S.Zero S.
    ~? = (S.Abstraction (S.Identifier "x") S.TBool (S.Pred (S.If (S.Identifier "x") S.Zero S.Zero)))
,
"reconstructTypeTest10" ~: do
```

```
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Succ (S.If (S.Identifier "x") S.Zero S.
     ~?= (S.Abstraction (S.Identifier "x") S.TBool (S.Succ (S.If (S.Identifier "x") S.Zero S.Zero)))
,
"reconstructTypeTest11" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.IsZero (S.If (S.Identifier "x") S.Zero S
     ~?= (S.Abstraction (S.Identifier "x") S.TBool (S.IsZero (S.If (S.Identifier "x") S.Zero S.Zero)))
,
    -- Tests with 2 variables.
"reconstructTypeTest12" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
     ~?= (S.Abstraction (S.Identifier "x") S.TNat (S.Abstraction (S.Identifier "y") S.TBool (S.If (S.Identifier "
,
"reconstructTypeTest13" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
     ~?= (S.Abstraction (S.Identifier "x") S.TNat (S.Abstraction (S.Identifier "y") S.TBool (S.If (S.Identifier "
,
"reconstructTypeTest14" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
     ~?= (S.Abstraction (S.Identifier "x") S.TNat (S.Abstraction (S.Identifier "y") S.TBool (S.If (S.Identifier "
,
    -- "reconstructTypeTest15"  : do
    -- justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.If (S.Application (S.Identifier "
    --  ?= (S.Abstraction (S.Identifier "x") S.TNat (S.Abstraction (S.Identifier "y") S.TBool (S.If (S.Identif
    -- ,
    -- Tests with 3 variables.
"reconstructTypeTest16" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
     ~?= (S.Abstraction (S.Identifier "x") S.TBool (S.Abstraction (S.Identifier "y") S.TNat (S.Abstraction (S.Id
,
"reconstructTypeApp1" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.IsZero (S.Application (S.Identifier "x"
     ~?= (S.Abstraction (S.Identifier "x") (S.TypePair S.TBool S.TNat) (S.IsZero (S.Application (S.Identifier "x
,
"reconstructTypeApp2" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.If (S.Application (S.Identifier "x") S.T
     ~?= (S.Abstraction (S.Identifier "x") (S.TypePair S.TBool S.TBool) (S.If (S.Application (S.Identifier "x")
,
"reconstructTypeApp3" ~: do
    justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.If (S.Application (S.Identifier "x") S.Z
     ~?= (S.Abstraction (S.Identifier "x") (S.TypePair S.TNat S.TBool) (S.If (S.Application (S.Identifier "x") S
```

```
,
"reconstructTypeApp4" ~: do
  justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Succ $ S.Pred (S.Application (S.Identifi
   ~? = (S.Abstraction (S.Identifier "x") (S.TypePair S.TNat S.TNat) (S.Succ $ S.Pred (S.Application (S.Identi
,
"reconstructTypeApp5" ~: do
  justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
   ~? = (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TypePair S.TBool S
,
"reconstructTypeApp6" ~: do
  justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
   ~? = (S.Abstraction (S.Identifier "x") (S.TypePair S.TBool S.TNat) (S.Abstraction (S.Identifier "y") (S.TV
,
"reconstructTypeApp7" ~: do
  justreconstructType (S.Abstraction (S.Identifier "x") (S.TVar "X") (S.Abstraction (S.Identifier "y") (S.TVar
   ~? = (S.Abstraction (S.Identifier "x") (S.TypePair S.TBool S.TNat) (S.Abstraction (S.Identifier "y") (S.Typ
]
```

6.4 Examples for Let

1. `let x = 0 in succ(x)`

```
---Term:---
1
---Type:---
Nat
---Normal form:---
1
```

2. `let x = abs(a.succ(a)) in app(x,0)`

```
---Term:---
app(abs(a:Nat.succ(a)),0)
---Type:---
Nat
---Normal form:---
1
```

3.
```
let x = 0 in
    let y = true in
        if y then succ(x) else x fi
```

```
---Term:---
if true then 1 else 0 fi
---Type:---
Nat
---Normal form:---
1
```

4. ```
   let id = abs(a.a) in
       app(id,0)
   ```

   ```
   ---Term:---
   app(abs(a:Nat.a),0)
   ---Type:---
   Nat
   ---Normal form:---
   0
   ```

5. ```
   let id = abs(a.a) in
       let zero = app(id,0) in
           let tru = app(id,true) in
               zero
   ```

   ```
   ---Term:---
   app(abs(a:Nat.a),0)
   ---Type:---
   Nat
   ---Normal form:---
   0
   ```

6. ```
   let succtwo = abs(a.succ(succ(a))) in
       let succfour = abs(b.app(succtwo,app(succtwo,b))) in
           app(succfour,0)
   ```

   ```
   ---Term:---
   app(abs(b:Nat.app(abs(a:Nat.succ(succ(a))),app(abs(a:Nat.succ(succ(a))),b))),0)
   ---Type:---
   Nat
   ---Normal form:---
   4
   ```

7. ```
   let not = abs(b. if b then false else true fi) in
           app(not,true)
   ```

```
---Term:---
app(abs(b:Bool.if b then false else true fi),true)
---Type:---
Bool
---Normal form:---
false
```

7 Driver

7.1 How to compile and run the program

The following cabal packages are required to run the program: either-unwrap, HUnit, and parsec. To install these packages, run the command below.

```
$ cabal install either-unwrap
$ cabal install HUnit
$ cabal install parsec
```

You should then be able to compile the program with.

```
$ ghc PCF.lhs
```

The program is then run as follows.

```
$ ./PCF tests
```

To run the unit tests and

```
$ ./PCF [/path/to/my/file.pcf]
```

To run the program on a given pcf file. Note that it also works on tlbn files.

7.2 Imports

List of imports for our parser.

import *System.Environment* (*getArgs*)
import *System.IO* (*openFile*, *IOMode* (*ReadMode*), *hGetContents*)
import *Text.ParserCombinators.Parsec*
import *Test.HUnit*
import *Data.Either.Unwrap* (*fromRight*)
import *Data.Maybe*
import *AbstractSyntax*
import *Evaluation*

```
import EvaluationRelation
import Parser
import Unification
import ConstraintTyping
```

7.3 Main Execution

Main method that takes the first argument as a file path to a TLBN file and parses and evaluates it as described in the spec. Special case, if the first argument is the string "tests" then the TLBN will run and output the results of the unit test suite.

```
parseTLBNFile fileName = do
  file ← openFile fileName ReadMode
  text ← hGetContents file
  return $ parse term fileName text
main = do
  args ← getArgs
  let arg0 = head args
  if (length args) < 1
    then error "Error: The first argument should be a path to a PCF file c
    else
    if arg0 ≡ "tests" then (runTests >>= print) else do
      parseResult ← parseTLBNFile arg0
      putStrLn "---Term:---"
      let parseTerm = fromJust $ reconstructType $ fromRight parseResult
      print parseTerm
      putStrLn "---Type:---"
      let termTypeResult = (typeOf Empty parseTerm)
      print termTypeResult
      putStrLn "---Normal form:---"
      let evalResult = eval Empty parseTerm
      print evalResult
```

7.4 Unit Tests

Runs the tests in the other modules.

```
concatTestList (TestList ls1) (TestList ls2) = TestList (ls1 ++ ls2)
concatTestList′ testLs = foldr concatTestList (head testLs) (tail testLs)
```

$$
\begin{aligned}
&\mathit{runTests} = \mathbf{do} \\
&\quad \mathit{runTestTT}\ \$\ \mathit{concatTestList}'\ [\mathit{parserTests}, \mathit{evaluationTests}, \mathit{evalRelTests}, \mathit{unificationTests}, \mathit{constraintBasedTyping}
\end{aligned}
$$

7.5 Test Outputs

```
Counts {cases = 109, tried = 109, errors = 0, failures = 0}
```