Homework Six CS 558

Cynthia Freeman

November 28, 2015

Introduction

As requested by the homework assignment, my code has been divided into the following modules: Abstract Syntax Module, Parser Module, Evaluation Module, Unification Module, and Constraint Typing Module. My code for main and also all my tests are at the end of this assignment.

Abstract Syntax Module

TypeVar is now included.

```
module AbstractSyntax where
  -- define the terms
  -- pulled from HW 5
data Term = Identifier {name :: String}  -- variable
  | Abstraction {variable :: Term, variableType :: Type, body :: Term}  -- abstraction
  | Application Term Term  -- application
  | Tru  -- constant true
  | Fls  -- constant false
  | If Term Term Term  -- conditional
  | Zero  -- constant zero
  | Succ Term  -- successor
  | Pred Term  -- predecessor
  | IsZero Term  -- zero test
  | Fix Term  -- fix
  deriving (Eq)
  -- show the terms defined above
  -- pulled from HW 5
instance Show Term where
```

```haskell
  show Tru = "True"
  show Fls = "False"
  show (If p c a) = "If " ++ show p ++ " then " ++
    show c ++ " else " ++ show a
  show Zero = "0"
  show (Succ t) | isNumericValue t = showAsNum t 1
     | otherwise = "(succ " ++ show t ++ ")"
    where showAsNum Zero num = show num
      showAsNum (Succ t) num =
        showAsNum t (num + 1)
  show (Pred t) = "(pred " ++ show t ++ ")"
  show (IsZero t) = "iszero " ++ show t
  show (Identifier n) = n
  show (Abstraction v vt b) = "abs (" ++ (show v) ++
    ":" ++ (show vt) ++ "." ++ (show b) ++ ")"
  show (Application t1 t2) = "app (" ++ (show t1) ++
    "," ++ (show t2) ++ ")"
  show (Fix t) = "fix(" ++ (show t) ++ ")"
  show (t) = "(" ++ (show t) ++ ")"

  -- define the types
data Type = TypePair {t1 :: Type, t2 :: Type}
  -- types of functions
   | TyBool
    -- types of booleans
   | TyNat
    -- types of natural numbers
   | TyVar TypeVar
    -- types of type variables
  deriving (Eq)

type TypeVar = String

  -- show the types defined above
instance Show Type where
  show (TypePair a b) = "arr (" ++ (show a) ++
    "," ++ (show b) ++ ")"
  show TyBool = "Bool"
  show TyNat = "Nat"
  show (TyVar varName) = varName

  -- define the typing context
data TypeContext = Empty   -- empty context
```

```
  | VariableBindings {bindings :: [Binding]}
    -- term variable binding
   -- this structure is used in TypeContext
   -- define the bindings of variables to their
   -- types in some sort of term
   -- eg : x: Bool has x as the identifierName and
   -- Bool as the identifierType
data Binding = VarBind {identifierName :: String,
  identifierType :: Type} deriving (Eq)
   -- input: a term
   -- output: whether or not the term is a value
   -- pulled from HW 5
isValue :: Term → Bool
isValue (Abstraction _ _ t) = True
isValue Tru = True
isValue Fls = True
isValue t = isNumericValue t
isValue _ = False
   -- input: a term
   -- output: whether or not the term is a
   -- numeric value
   -- pulled from HW 5
isNumericValue :: Term → Bool
isNumericValue Zero = True
isNumericValue (Succ t) = isNumericValue t
isNumericValue _ = False
```

Parser Module

Let is taken into account here. Let is a simple substitution which is done in parserLet. To deal with fresh variable generation in let, several new functions such as replaceVars is created. In addition, implicit abstraction can now be parsed.

```
module Parser where
import AbstractSyntax as S
import Evaluation
import ConstraintTyping
import Text.ParserCombinators.Parsec
```

```haskell
import Data.Either.Unwrap (fromRight)
  -- Note: a parser is a function
  -- that takes a string and returns
  -- a structure as output
  -- parser combinators are functions that
  -- accept several parsers and return a
  -- new parser as its output
  -- so we will make a lot of little parsers and
  -- combine them to make more complicated parsers
  -- using the Parsec library
  -- From page 5 of the HW 4 sheet:
parserArr = noSpace (string "arr")
parserLpar = noSpace (string "(")
parserRpar = noSpace (string ")")
  -- return what is in the parenthesis
parserParens pa =
  do
    parserLpar
    a <- pa
    parserRpar
    return a
parserComma = noSpace (string ",")
parserColon = noSpace (string ":")
parserFullstop = noSpace (string ".")   -- period
parserFi = noSpace (string "fi")
keywordSet = ["arr",",",":",".","fi","abs","app",
"Bool","Nat","true","false","if","then","else","0",
"succ","pred","iszero"]
parserIdentifier :: Parser Term
parserIdentifier = do
  x <- noSpace (many1 lower)
    -- if x has key words, error!
  return (Identifier x)
  -- note: many1 p applies the parser p one or more times
  -- (http://hackage.haskell.org/package/parsec-3.1.9/docs/
    -- Text-Parsec-Combinator.html)
  -- note: lower parses a lower case character
```

```
  -- (http://hackage.haskell.org/package/parsec-
    -- 3.1.9/docs/src/Text-Parsec-Char.htmlspaces)
parserExplicitAbs :: Parser Term
parserExplicitAbs =
  do
    noSpace (string "abs")
    parserLpar
    name ← parserIdentifier
    parserColon
    type1 ← parserType
    parserFullstop
    term1 ← parserTerm
    parserRpar
    return (Abstraction name type1 term1)
  -- find the used type variables in a term
  -- return these type variables in a list
  -- this function is necessary for generating
  -- fresh type variables because we can't use them again
  -- eg: usedTypeVariables (S.If S.Tru S.Fls S.Fls) = []
  -- eg: usedTypeVariables (S.Abstraction (S.Identifier "a")
    -- (S.TyVar "X") S.Tru) =["X"]
usedTypeVariables :: S.Term → [S.TypeVar]
usedTypeVariables (S.Abstraction t1 t2 t3) = (usedTypeVariablesHelper t2)
  ++ (usedTypeVariables t3)
    -- t1 is variable, t2 is variable type and t3 is body
  where
    usedTypeVariablesHelper :: S.Type → [S.TypeVar]
    usedTypeVariablesHelper (S.TyVar name) = [name]
    usedTypeVariablesHelper (S.TypePair t1 t2) =
      (usedTypeVariablesHelper t1) ++ (usedTypeVariablesHelper t2)
usedTypeVariables (S.Application t1 t2) = (usedTypeVariables t1)
++ (usedTypeVariables t2)
usedTypeVariables (S.If t1 t2 t3) = (usedTypeVariables t1) ++
(usedTypeVariables t2) ++ (usedTypeVariables t3)
usedTypeVariables (S.Succ t) = (usedTypeVariables t)
usedTypeVariables (S.Pred t) = (usedTypeVariables t)
usedTypeVariables (S.IsZero t) = (usedTypeVariables t)
usedTypeVariables (S.Fix t) = (usedTypeVariables t)
usedTypeVariables _ = []
```

```
moar_variables :: [S.TypeVar]
moar_variables = map (λx → [x]) ['a' .. 'z']
  -- generate fresh variables given a term
  -- fresh variables need to be in the set of variables
  -- defined above
  -- and cannot be in used variables of the given term
  -- eg: generateFreshVariable (S.Abstraction (S.Identifier "b")
    -- (S.TyVar "A") S.Tru) = B
  -- note: this is like saying (lambda b:X . True)
generateFreshVariable :: S.Term → S.Type
generateFreshVariable term = head
[S.TyVar x | x ← moar_variables, ¬ (elem x utvs)]
  where utvs = usedTypeVariables term

parserImplicitAbs :: Parser Term
parserImplicitAbs =
  do
    noSpace (string "abs")
    parserLpar
    name ← parserIdentifier
    parserFullstop
    term1 ← parserTerm
    parserRpar
    return (Abstraction name (generateFreshVariable term1)
      term1)

  -- looks thru the let term and updates the var names
replaceVars :: (Term, Int) → (Term, Int)
replaceVars ((Abstraction t1 t1Type body), idx) = case t1Type of
  (TyVar name) →
  (Abstraction t1 (TyVar (name ++ show (idx)))
    (fst (replaceVars (body, idx))), idx)
  otherwise → (Abstraction t1 t1Type
    (fst (replaceVars (body, idx))), idx)

replaceVars ((Application t0 t1), idx) =
  (Application (fst (replaceVars (t0, idx)))
    (fst (replaceVars (t1, idx))), idx)
replaceVars ((If t0 t1 t2), idx) = (If (fst (replaceVars (t0, idx)))
  (fst (replaceVars (t1, idx))) (fst (replaceVars (t2, idx))), idx)
replaceVars ((Pred t), idx) = (Pred (fst (replaceVars (t, idx))), idx)
replaceVars ((Succ t), idx) = (Succ (fst (replaceVars (t, idx))), idx)
```

$replaceVars\ ((IsZero\ t), idx) = (IsZero$
$\quad (fst\ (replaceVars\ (t, idx))), idx)$
$replaceVars\ ((Fix\ t), idx) = (Fix\ (fst\ (replaceVars\ (t, idx))), idx)$
$replaceVars\ (a, idx) = (a, idx)$

-- looks for a var to replace with the let term
$subFV :: Term \rightarrow Term \rightarrow Term \rightarrow Int \rightarrow (Term, Int)$
$subFV\ (Abstraction\ t0\ aType\ t1)\ var\ val\ idx$
$\quad |\ var \equiv t0 = ((Abstraction\ t0\ aType\ t1), idx)$
$\quad |\ otherwise = ((Abstraction\ t0\ aType\ t1'), tidx)$
$\quad$ **where**
$\qquad (t1', tidx) = subFV\ t1\ var\ val\ idx$
$subFV\ (Application\ t0\ t1)\ var\ val\ idx =$
$\quad$ **let**
$\qquad (t0', tidx1) = subFV\ t0\ var\ val\ idx$
$\qquad (t1', tidx2) = subFV\ t1\ var\ val\ tidx1$
$\quad$ **in**
$\qquad ((Application\ t0'\ t1'), tidx2)$
$subFV\ (If\ t0\ t1\ t2)\ var\ val\ idx =$
$\quad$ **let**
$\qquad (t0', tidx1) = subFV\ t0\ var\ val\ idx$
$\qquad (t1', tidx2) = subFV\ t1\ var\ val\ tidx1$
$\qquad (t2', tidx3) = subFV\ t2\ var\ val\ tidx2$
$\quad$ **in**
$\qquad ((If\ t0'\ t1'\ t2'), tidx3)$
$subFV\ (Succ\ t)\ var\ val\ idx =$
$\quad$ **let**
$\qquad (t', tidx) = subFV\ t\ var\ val\ idx$
$\quad$ **in**
$\qquad ((Succ\ t'), tidx)$
$subFV\ (Pred\ t)\ var\ val\ idx =$
$\quad$ **let**
$\qquad (t', tidx) = subFV\ t\ var\ val\ idx$
$\quad$ **in**
$\qquad ((Pred\ t'), tidx)$
$subFV\ (IsZero\ t)\ var\ val\ idx =$
$\quad$ **let**
$\qquad (t', tidx) = subFV\ t\ var\ val\ idx$
$\quad$ **in**
$\qquad ((IsZero\ t'), tidx)$

```
subFV (Fix t) var val idx =
  let
    (t', tidx) = subFV t var val idx
  in
    ((Fix t'), tidx)
subFV term1 var term2 idx
   | var ≡ term1 = (fst (replaceVars (term2, idx)), idx + 1)
   | otherwise = (term1, idx)
parserLet :: Parser Term
parserLet =
  do
    noSpace (string "let")
    varName ← parserIdentifier
    noSpace (string "=")
    t1 ← parserTerm
    noSpace (string "in")
    t2 ← parserTerm
    return (fst (subFV t2 varName t1 0))
parserType :: Parser Type
parserType =
  do
    noSpace (string "Bool")
    return TyBool
   <|>
  do
    noSpace (string "Nat")
    return TyNat
   <|>
  do
    parserArr
    parserLpar
    t1 ← parserType
    parserComma
    t2 ← parserType
    parserRpar
    return (TypePair t1 t2)
parserTerm =
  do
    try $ parserLet
```

```
  <|>
  do
    try $ parserFix
  <|>
  do
    try $ parserIfthenelse
  <|>
  do
    try $ parserIszero
  <|>
  do
    try $ parserExplicitAbs
  <|>
  do
    try $ parserImplicitAbs
  <|>
  do
    try $ parserApp
  <|> try (parserBool) <|> try (parserNat) <|>
  try (parserParens parserTerm) <|>
  try (parserIdentifier)
    -- note: the try parser behaves like parse p
    -- but doesn't consume input when p fails
    -- (http://book.realworldhaskell.org/read/using-parsec.html)
parserApp =
  do
    noSpace (string "app")
    parserLpar
    t1 ← parserTerm
    parserComma
    t2 ← parserTerm
    parserRpar
    return (Application t1 t2)
parserTrue = noSpace (string "true")
parserFalse = noSpace (string "false")
parserIf = noSpace (string "if")
parserThen = noSpace (string "then")
parserElse = noSpace (string "else")
```

```
parserIfthenelse =
  do
    parserIf
    t1 ← parserTerm
    parserThen
    t2 ← parserTerm
    parserElse
    t3 ← parserTerm
    parserFi
    return (If t1 t2 t3)
parserZero = noSpace (string "0")
parserSucc = do
  noSpace (string "succ")
  t ← parserParens parserTerm
  return (Succ t)
parserPred = do
  noSpace (string "pred")
  t ← parserParens parserTerm
  return (Pred t)
parserIszero =
  do
    noSpace (string "iszero")
    t ← parserParens parserTerm
    return (IsZero t)
parserFix =
  do
    noSpace (string "fix")
    t ← parserParens parserTerm
    return (Fix t)
parserBool :: Parser Term
parserBool =
  (parserTrue ≫ return Tru) < | > (parserFalse ≫ return Fls)
    -- note: will only evaluate right if left fails
    -- (http://research.microsoft.com/pubs/65201/parsec-paper-letter.pdf)
    -- note: double arrow passes results of the first into the second
    -- (https://www.haskell.org/tutorial/monads.html)
parserNat :: Parser Term
parserNat = (parserZero ≫ return Zero) < | >
```

```
   parserSucc <|> parserPred
   -- a function for editing parsers to skip spaces
   -- input: a parser
   -- output: a parser that skips white spaces
noSpace :: Parser a → Parser a
noSpace parser = do
   spaces   -- spaces skips white space characters
   a ← parser
   spaces
   return a
```

Evaluation and Typing Module

Little to no difference from previous homeworks. The detType function has been slightly edited.

```
module Evaluation where
import AbstractSyntax
import Data.Monoid
   -- the one step evaluation relation
   -- input: a term
   -- output: (Nothing,Context) or a
   -- (Maybe Term,Context) after 1 evaluation step
   -- similar to HW 4 except we take fix into account
eval1 :: TypeContext → Term → (Maybe Term, TypeContext)
eval1 context (Application t1 t2)
   | ¬ (isValue t1)                    =
   recursorCon (λt → Application t t2) t1 context
      -- E APP1
   | (isValue t1) ∧ (¬ (isValue t2)) =
   recursorCon (λt → Application t1 t) t2 context   -- E APP2
   | otherwise                         = case t1 of
     Abstraction var varType absBody
      → (Just (sub absBody var t2),
     (addBinding context var varType))
        -- E APP ABS
     otherwise
      → (Nothing, context)
```

```
eval1 context (If Tru t2 t3) = (Just t2, context)  -- E-IFTRUE rule
eval1 context (If Fls t2 t3) = (Just t3, context)  -- E-IFFALSE rule
eval1 context (If t1 t2 t3) =
recursorCon (λt → If t t2 t3) t1 context  -- E-IF rule
eval1 context (Succ t) = recursorCon Succ t context  -- E-SUCC rule
eval1 context (Pred Zero) = (Just Zero, context)  -- E-PREDZERO rule
eval1 context (Pred (Succ v))  -- E-PREDSUCC rule
  | isNumericValue v ≡ True = (Just v, context)
eval1 context (Pred t) = recursorCon Pred t context  -- E-PRED
eval1 context (IsZero Zero) = (Just Tru, context)  -- E-ISZEROZERO rule
eval1 context (IsZero (Succ v))  -- E-ISZEROSUCC rule
  | isNumericValue v ≡ True = (Just Fls, context)
  | otherwise = (Nothing, context)
eval1 context (IsZero t) = recursorCon IsZero t context  -- E-ISZERO rule
eval1 context (Fix t1) = case t1 of
(Abstraction var varType absBody) →
(Just (sub absBody var (Fix t1)), context)  -- E-FIX BETA rule
                          otherwise
                           → recursorCon Fix t1 context
                             -- E-FIX rule
eval1 context _ = (Nothing, context)
  -- helper function for recursorCon
  -- input: a function that normally cannot be applied to monads
  -- output: the function that can be applied to monads
  -- pulled from HW 5
liftM :: (Monad m) ⇒ (a → b) → m a → m b
liftM f m1 = do {x1 ← m1; return (f x1)}
  -- helper function for eval1
  -- input: a function and a term and a typing context
  -- output: evaluates the term by one step and applies
  -- the monadic function to it
  -- pulled from HW 5
recursorCon :: (Term → Term) → Term → TypeContext
  → (Maybe Term, TypeContext)
recursorCon f t context = ((liftM f) term, context2)
  where (term, context2) = (eval1 context t)
  -- helper function for eval1
  -- input: abstraction body term, variable term, term
  -- to replace variable with
```

```
-- output: the abstraction body where all free occurences
-- the first term are replaced with a second term
-- sub absBody var t2 will replace free occurences
-- of var with t2 in the absBody
-- pulled from HW 5
sub :: Term → Term → Term → Term
sub (Abstraction t0 t0type body) t absBody
  | t0 ≡ t      = Abstraction t0 t0type body   -- BOUND!
    -- Can only replace free variables!
  | otherwise = Abstraction t0 t0type (sub body t absBody)
sub (Application t0 t1) t absBody = Application
(sub t0 t absBody) (sub t1 t absBody)
sub (If t0 t1 t2) t absBody = If (sub t0 t absBody)
(sub t1 t absBody) (sub t2 t absBody)
sub (Succ t0) t absBody = Succ (sub t0 t absBody)
sub (Pred t0) t absBody = Pred (sub t0 t absBody)
sub (IsZero t0) t absBody = IsZero (sub t0 t absBody)
sub (Fix t0) t absBody = Fix (sub t0 t absBody)
sub aTerm t absBody
  | aTerm ≡ t = absBody
  | otherwise = aTerm
-- see Chapter 10 of TAPL's addBinding function
-- helper function for eval1
-- input: the current type context used, the term that
-- has the new binding, the new binding type for that term
-- output: the current type context along with the new term
-- and its binding
addBinding :: TypeContext → Term → Type → TypeContext
addBinding Empty (Identifier x) t = VariableBindings
  [VarBind x t]   -- a new context environment with only
    -- that new binding
addBinding notEmptyContext (Identifier x) t = VariableBindings
  ((VarBind x t) : (bindings notEmptyContext))
    -- simply cons the new binding to the existing
    -- context environment
addBinding someContext something t =
  error ("can't add binding to non identifier to ctx")
-- multistep evaluation
-- input: the typing context, the term to evaluate
```

```
  -- output: the term evaluated as much as possible using eval1
  -- pulled from HW 5
eval :: TypeContext → Term → Term
eval context t
   | isValue t = t   -- nothing more to do with a value
   | otherwise = case eval1 context t of (Nothing, someContext)
   → t
     (Just a, nextContext)
      → eval nextContext a
  -- see Chapter 10 TAPL's typeOf function
  -- input: the typing context, a term to find the type of
  -- output: the type of the term given using typing rules
detType :: TypeContext → Term → Type
detType someContext (Identifier x) = case
(getTypeFromContext someContext (Identifier x)) of
Just xtype → xtype   -- T-Var
Nothing → error ("could not find type for identifier")
detType someContext (Abstraction t1 typeT1 t2) =
  let
     nextContext = addBinding someContext t1 typeT1
     typeT2 = detType nextContext t2
  in
     TypePair typeT1 typeT2
detType someContext (Application t1 t2) = case (detType someContext t1) of
                        (TypePair typeT11 typeT12) → case typeT11 of
                           (TypePair a b) → case
                           (detType someContext t2) of
                              (TypePair typeT21 typeT22)
                               → if ((partialType typeT22) ≡ a)
                                 then (TypePair (partialType b)
                                                      (partialType typeT12)) else
                                                      error ("typefail both arrows")
                              _ → if
                                 ((detType someContext t2) ≡ a)
                                 then (TypePair (partialType b)
                                                      (partialType typeT12)) else
                                                      error ("typefail 1st arrow 2nd single")
                           _ → case (detType someContext t2) of
                              (TypePair typeT21 typeT22) → if
```

```
                              ((partialType typeT22) ≡ typeT11) then
                                       typeT12 else
                                         error ("typefail 1st single 2nd arrow")
                            _ →
                            if ((detType someContext t2) ≡ typeT11)
                              then typeT12 else error ("typefail both single")
                        _      → error ("type pair was not given")
detType anyContext Tru = TyBool   -- T-True
detType anyContext Fls = TyBool   -- T-False
detType someContext (If t1 t2 t3) = if (detType someContext t1) ≡
TyBool then (if (detType someContext t2) ≡ (detType someContext t3)
then (detType someContext t2) else
error ("can't have dif types for 2nd and 3rd cond"))
else error ("cannot have non boolean type for first conditional")  -- T-If
detType anyContext Zero = TyNat   -- T-Zero
detType someContext (Succ t) = if (detType someContext t) ≡ TyNat
then TyNat else error ("cannot have succ of non Nat type")  -- T-Succ
detType someContext (Pred t) = if (detType someContext t) ≡ TyNat
then TyNat else error ("cannot have pred of non Nat type")  -- T-Pred
detType someContext (IsZero t) = if (detType someContext t) ≡ TyNat
then TyBool else error ("cannot have iszero of non Nat type")  -- T-IsZero
detType someContext (Fix t1) = case typet1 of (TypePair t1 t2) → if
  (t1 ≡ t2) then t1 else
    error ("input neq output type in input func")
                          _ →
                      error ("fix was not given a generator function")
                        where typet1 = detType someContext t1

  -- helper function in the case of partial function application
  -- input: a type
  -- output: a type
partialType :: Type → Type
partialType b = case b of
  (TypePair c d) → partialType d
  _              → b

  -- helper function for getTypeFromContext
  -- input: a boolean function, a foldable
  -- structure such as a list
  -- output: the leftmost element of the foldable structure
  -- that satisfies the boolean function or Nothing if no element satisfies it
```

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
find p = getFirst . foldMap (\x -> First (if p x then Just x else Nothing))
```

```
  -- helper function for getTypeFromContext
  -- pulled from HW 5
  -- input: a Maybe value
  -- output: the value inside the Maybe...if there is no value, an error occurs!
fromJust          :: Maybe a -> a
fromJust Nothing = error ("Cannot grab value from Nothing!")
fromJust (Just x) = x
```

```
  -- see chapter 10 of TAPL for reference
  -- helper function for detType
  -- input: the typing context, the term to find the type of in the typing context
  -- output: the type of the term in the typing context or Nothing if identifier type
    -- is not found
getTypeFromContext :: TypeContext -> Term -> Maybe Type
getTypeFromContext Empty (Identifier x) = Nothing
getTypeFromContext someContext (Identifier x) = resultType
  where
    typeFound = find (\str -> (identifierName str) == x) (bindings someContext)
    resultType
      | typeFound == Nothing = Nothing
      | otherwise = Just (identifierType (fromJust typeFound))
getTypeContext _ term =
  error ("can't find a typing context")
```

Unification Module

Given an equation set, this module will unify the equations. The process either succeeds, halts with failure, halts with a cycle, or there is no match. It is assumed that a no match situation occurs when $f(t_1, ...t_n) = f(s_1, s_2, ...s_m)$ where $m \neq n$. This algorithm is based off of the canonical nondeterminininistic algorithm shown in Martelli and Montanari's paper *An Efficient Unification Algorithm*.

```
module Unification where
import Data.List (find)
```

```
  -- on paper, terms are
  -- constant symbols and variables
  -- if t1,...tn are terms and f is in An then f(t1,...tn) is a term
```

```
-- f(t1...tn) is represented by Fun f [t1 t2 ...tn]
-- variable v is represented by Var v
data Term v f = Fun f [Term v f] | Var v deriving (Show, Eq)

-- the equation is the lhs = rhs
type Equation v f = (Term v f, Term v f)

-- variable v is equal to term v f
-- essentially v will be replaced with Term v f
type Binding v f = (v, Term v f)

-- a substitution is a list of bindings
type Substitution v f = [Binding v f]

data EquationOutcome = HaltWithFailure | HaltWithCycle |
NoMatch | Success deriving (Show, Eq)

-- given a term (a variable or a function) and a substitution list (theta),
-- apply it to the term and output the resulting term
-- The find function takes a predicate and a structure and returns
-- the leftmost element of the structure matching the predicate,
-- or Nothing if there is no such element

-- in this situation, find is given a predicate that checks if the first
-- element in a given pair is equal to the variable x
-- it then checks every binding in the substitution list beta
-- and sees if it fulfills the predicate
-- corresponding 2nd element in the pair

-- given a function it applies the substitution for each of the args
-- in the function
applySubst :: (Eq v, Eq f) => Term v f -> Substitution v f -> Term v f
applySubst (Var x) theta =
  case Data.List.find (\(z, _) -> z == x) theta of
    Nothing -> Var x
    Just (_, t) -> t
applySubst (Fun f tlist) theta = Fun f (map (\t -> applySubst t theta) tlist)

-- given a list of equations, apply the given sub list to the equations and
-- output the resulting equations
applySubst' :: (Eq v, Eq f) => [Equation v f] -> Substitution v f -> [Equation v f]
applySubst' eql [] = eql
applySubst' [] theta = []
applySubst' eql theta = map (\(s, t) -> (applySubst s theta, applySubst t theta)) eql

-- given a list of equations and a list of subs,
```

-- apply all the subs to the equations from back to front
-- and return the resulting equation list

$applySubsts :: (Eq\ v, Eq\ f) \Rightarrow [Equation\ v\ f] \rightarrow [Substitution\ v\ f] \rightarrow [Equation\ v\ f]$
$applySubsts\ eql\ [\,] = eql$
$applySubsts\ eql\ ys = applySubsts\ (applySubst'\ eql\ x)\ xs$
  $\textbf{where}\ (x : xs) = reverse\ ys$

-- the unify that the professor really wants and to follow the cbt template...
-- given a list of equations, unifies them and returns
-- the outcome and the substitution list converted as an equation list

$unify :: (Eq\ v, Eq\ f) \Rightarrow [Equation\ v\ f] \rightarrow (EquationOutcome, [Equation\ v\ f])$
$unify\ eql =$
  $\textbf{let}$
    $(outcome, subs) = trueunify\ eql$
    $subeqns = convert\_subeqns\ subs$
  $\textbf{in}$
    $(outcome, subeqns)$

-- given a list of subs, converts them into equations
-- this is helper equation to conver a list of substitutions
-- into a list of equations which is necessary for
-- the professor's unify template

$convert\_subeqns :: (Eq\ v, Eq\ f) \Rightarrow [Substitution\ v\ f] \rightarrow [Equation\ v\ f]$
$convert\_subeqns\ [\,] = [\,]$
$convert\_subeqns\ (x : xs) =$
  $\textbf{let}$
    $a = fst\ (head\ x)$
    $b = snd\ (head\ x)$
  $\textbf{in}$
    $[(Var\ a, b)] + convert\_subeqns\ xs$

-- DIFFERS FROM TEACHER'S IMPLEMENTATION IN THAT
-- THE TYPE RETURNED IS A LIST OF SUBS
-- AND NOT A LIST OF EQNS
-- DEFINITIONALLY, THIS IS MORE CORRECT
-- given a list of equations, unifies them and returns the list of substitutions
-- and the equation outcome

$trueunify :: (Eq\ v, Eq\ f) \Rightarrow [Equation\ v\ f] \rightarrow (EquationOutcome, [Substitution\ v\ f])$
$trueunify\ eql = (outcome, subs)$
  $\textbf{where}\ (outcome, equationList, subs) = unifySub\ eql\ [\,]$

-- given a list of equations, unifies them and returns the
-- equation outcome, the resulting equation list (should be empty)

-- and the list of substitutions
$unifySub :: (Eq\ v, Eq\ f) \Rightarrow [Equation\ v\ f] \rightarrow [Substitution\ v\ f] \rightarrow (EquationOutcome,$
$[Equation\ v\ f], [Substitution\ v\ f])$
$unifySub\ (e : es)\ subs$
$\mid and\ [outcome \equiv Success, resultEquations \not\equiv [\,]] = unifySub$
$(applySubsts\ resultEquations\ resultSub)\ (resultSub +\!\!+\ subs)$
-- outcome is success,there are eqns -¿ apply subs and keep unifying
$\mid and\ [outcome \equiv Success, resultEquations \equiv [\,]] =$
$(outcome, applySubsts\ resultEquations\ resultSub, resultSub +\!\!+\ subs)$
-- outcome success, no eqns -¿ apply subs and done
$\mid outcome \not\equiv Success \qquad =$
$(outcome, resultEquations, resultSub +\!\!+\ subs)$
where $(outcome, resultEquations, resultSub) = unify1\ (e : es)$

-- given equation list, manip first element only
-- returns the outcome, the equation list, the sub list
$unify1 :: (Eq\ v, Eq\ f) \Rightarrow [Equation\ v\ f] \rightarrow (EquationOutcome,$
$[Equation\ v\ f], [Substitution\ v\ f])$

-- ALGORITHM 1 PART A (pg 261)
-- equation of form t = x
-- where t is not a variable and x
-- is a variable and rewrite it as x = t
$unify1\ ((Fun\ a\ b, Var\ c) : eqns) = (Success, ((Var\ c, Fun\ a\ b) : eqns), [\,])$

-- ALGORITHM 1 PART B
-- equation of form x = x
-- where x is a variable and erase it
-- otherwise make it a substitution
-- and continue unifying
$unify1\ ((Var\ a, Var\ b) : eqns)$
$\mid a \equiv b = (Success, eqns, [\,])$
$\mid otherwise = (Success, ((Var\ a, Var\ b) : eqns), [[(a, Var\ b)]])$

-- ALGORITHM 1 PART C
-- equation of the form
-- t' = t" where t' and t" are not variables
-- if the two root function symbols are different,
-- stop with failure; otherwise apply term reduction
-- ie:
-- term reduction:
-- f(t1,t2...tn) = f(s1,s2...sn) gives t1=s1, t2=s2,...tn=sn
-- fail:

```
-- f(t1,t2...tn) = g(s1,s2...sn) and f neq g is a failure
unify1 ((Fun a b, Fun c d) : eqns)
  | a ≢ c      = (HaltWithFailure, ((Fun a b, Fun c d) : eqns), [])
  | length (b) ≢ length (d) = (NoMatch, ((Fun a b, Fun c d) : eqns), [])
  | otherwise = (Success, (zip b d) ++ eqns, [])
-- ALGORITHM 1 PART D
-- 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, fail with cycle
-- otherwise apply variable elimination (apply the
-- substitution (t,x) to all other equations in
-- the set without deleting x =t equation)
unify1 ((Var a, Fun b c) : eqns)
  | isInside (Var a) (Fun b c) = (HaltWithCycle, ((Var a, Fun b c) : eqns), [])
  | otherwise = (Success, ((Var a, Fun b c) : eqns), [[(a, Fun b c)]])
-- no cases apply
unify1 [] = (Success, [], [])
-- checks if the 1st argument (a variable)
-- in inside the 2nd argument (a function)
-- True if yes, False if no
-- note that or someList returns True if some element
-- inside someList is True
isInside :: (Eq v, Eq f) ⇒ Term v f → Term v f → Bool
isInside (Var a) (Fun b c)
  | elem (Var a) c = True
  | otherwise      = or (map (isInside (Var a)) c)
isInside _ _ = False
```

Constraint Typing Module

The constraint typing rules follow from TAPL on page 322. In addition, I add the constraint rule for fix.

```
module ConstraintTyping where
import qualified AbstractSyntax as S
import qualified Unification as U
import Data.List
```

```
import Data.Maybe
  -- type equation
type TypeConstraint = (S.Type, S.Type)
  -- this is the constraint set which consists of
  -- a set of type equations
type TypeConstraintSet = [TypeConstraint]
  -- sub in the S.Type for S.TypeVar
type TypeSubstitution = [(S.TypeVar, S.Type)]
type TypeUnifVar = S.TypeVar
data TypeUnifFun = TypeUnifArrow | TypeUnifBool | TypeUnifNat deriving (Eq, Show)
  -- given a constraint set, encodes the type equations within
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.TyBool = U.Fun TypeUnifBool []
    enctype S.TyNat = U.Fun TypeUnifNat []
    enctype (S.TyVar xi) = U.Var xi
  -- given a substitution converted to equation form, decodes it into a type substitution
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.TyBool
    g (U.Fun TypeUnifNat []) = S.TyNat
    g (U.Var xi) = S.TyVar xi
  -- given a term, derives the term's type
reconstructType :: S.Term → Maybe S.Term
reconstructType t =
  let
    constraints = deriveTypeConstraints t   -- derive the constraint set
    unifencoding = encode constraints   -- encode the constraint set
    (unifoutcome, unifsolvedequations) = U.unify unifencoding
```

```
    -- unify the encoded constraints
  in
    case unifoutcome of
      U.Success →    -- if outcome of unification is successful
        let
          typesubst = decode unifsolvedequations   -- decode the substitution
          t′ = applyTSubToTerm typesubst t   -- apply the decoded sub to the term
        in
          Just t′
      U.HaltWithFailure → Nothing   -- if outcome of unification fails
      U.HaltWithCycle → Nothing
      U.NoMatch → Nothing
  -- CONSTRAINT RULES ON PAGE 322 IN TAPL
constraintRules :: S.Term → Id → IdBindingSet →
(NewId, TypeConstraintSet, IdBindingSet)
  -- CT VAR
constraintRules (S.Identifier x) ident bindingSet =
  let
    vName = removeId ident
    nextId = succ ident
  in
    case findBinding bindingSet (S.Identifier x) of
      Just name → (nextId, [(vName, name)], bindingSet)
      Nothing → (nextId, [ ], bindingSet)
  -- CT ABS
  -- if the body of abs is another abs
constraintRules (S.Abstraction variable variableType body) ident bindingSet =
  let
    vName = removeId ident
    vName′ = removeId (succ ident)
    vName″ = removeId (succ (succ ident))
    nextId = succ (succ (succ ident))
    bindingSet′ = (variable, variableType) : bindingSet
    (t_id, cset, bindingSet″) = constraintRules body nextId bindingSet′
    cset′ = changeConstraintset cset (removeId nextId) vName″
  in
    (t_id, (vName, S.TypePair vName′ vName″) :
      (vName′, variableType) : cset′, bindingSet″)
  -- CT APP – some sort of bug here
```

```
constraintRules (S.Application t1 t2) ident bindingSet =
  let
    vName = removeId ident
    vName' = removeId (succ ident)
    vName'' = removeId (succ (succ ident))
    nextId = succ (succ (succ ident))
    (t1_id, cset, bindingSet1) = constraintRules t1 nextId bindingSet
    c1 = changeConstraintset cset (removeId nextId) vName'
    (t2_id, cset2, bindingSet2) = constraintRules t2 t1_id bindingSet1
    c2 = changeConstraintset cset2 (removeId t1_id) vName''
  in
    (t2_id, (vName, vName) : (vName', S.TypePair vName'' vName) :
      (c1 ++ c2), bindingSet2)
  -- CT ZERO
constraintRules (S.Zero) ident bindingSet =
  let
      -- generate needed variables
    vName = removeId ident
    nextId = succ ident
  in
    (nextId, [(vName, S.TyNat)], bindingSet)
  -- CT SUCC
constraintRules (S.Succ t) ident bindingSet =
  let
      -- generate needed variables
    vName = removeId ident
    vName' = removeId (succ ident)
    nextId = succ (succ ident)
    (t_id, cset, bindingSet') = constraintRules t nextId bindingSet
    cset' =
      case t of
        S.Identifier _ → changeConstraintset cset (removeId nextId) vName'
        S.Application _ _ → changeConstraintset cset (removeId nextId) vName'
        otherwise       → cset
  in
    (t_id, (vName, S.TyNat) : (vName', S.TyNat) : cset', bindingSet')
  -- CT PRED
constraintRules (S.Pred t) ident bindingSet =
  let
```

```
      -- generate needed variables
    vName = removeId ident
    vName' = removeId (succ ident)
    nextId = succ (succ ident)
    (t_id, cset, bindingSet') = constraintRules t nextId bindingSet
    cset' =
      case t of
        S.Identifier _ → changeConstraintset cset (removeId nextId) vName'
        S.Application _ _ → changeConstraintset cset (removeId nextId) vName'
        otherwise      → cset
  in
    (t_id, (vName, S.TyNat) : (vName', S.TyNat) : cset', bindingSet')
  -- CT IS ZERO
constraintRules (S.IsZero t) ident bindingSet =
  let
      -- generate needed variables
    vName = removeId ident
    vName' = removeId (succ ident)
    nextId = succ (succ ident)
    (t_id, cset, bindingSet') = constraintRules t nextId bindingSet
    cset' =
      case t of
        S.Identifier _ → changeConstraintset cset (removeId nextId) vName'
        S.Application _ _ → changeConstraintset cset (removeId nextId) vName'
        otherwise      → cset
    in
      (t_id, (vName, S.TyBool) : (vName', S.TyNat) : cset', bindingSet')
  -- CT TRUE
constraintRules (S.Tru) ident bindingSet =
  let
    vName = removeId ident
    nextId = succ ident
  in
    (nextId, [(vName, S.TyBool)], bindingSet)
  -- CT FALSE
constraintRules (S.Fls) ident bindingSet =
  let
      -- generate needed variables
    vName = removeId ident
```

```
    nextId = succ ident
  in
    (nextId, [(vName, S.TyBool)], bindingSet)
  -- CT IF
constraintRules (S.If t1 t2 t3) ident bindingSet =
  let
    vName0 = removeId ident   -- v1
    vName1 = removeId (succ ident)   -- v2
    vName2 = removeId (succ (succ ident))   -- v3
    vName3 = removeId (succ (succ (succ ident)))   -- v0
    nextId = succ (succ (succ (succ ident)))
    (t1_id, cset0, bindingSet1) = constraintRules t1 nextId bindingSet
    c1 = changeConstraintset cset0 (removeId nextId) vName0
    (t2_id, cset1, bindingSet2) = constraintRules t2 t1_id bindingSet1
    c2 = changeConstraintset cset1 (removeId t1_id) vName1
    (t3_id, cset2, bindingSet3) = constraintRules t3 t2_id bindingSet2
    c3 = changeConstraintset cset2 (removeId t2_id) vName3
  in
    (t3_id, (vName0, S.TyBool) : (vName1, vName2) : (vName3, vName1) :
      (c1 ++ c2 ++ c3), bindingSet3)
  -- CT FIX
constraintRules (S.Fix t) ident bindingSet =
  let
    vName0 = removeId ident
    vName1 = removeId (succ ident)
    vName2 = removeId (succ (succ ident))
    nextId = succ (succ (succ ident))
    (t_id, cset, bindingSet') = constraintRules t nextId bindingSet
  in
    (t_id, (vName0, vName1) : (vName0, vName2) : cset, bindingSet')
  -- catchall
constraintRules left ident bindingSet = (ident, [], bindingSet)
  -- now for deriving the type constraints given a term
  -- follows page 322 in TAPL
  -- given a term, get the contraint set
deriveTypeConstraints :: S.Term -> TypeConstraintSet
deriveTypeConstraints t = nub (constraintSet)   -- only unique solutions are allowed
  where
    (someId, constraintSet, someBindingSet) = constraintRules t (Id 'A') []
```

```
        -- 'A' is our first variable in our variable list
    -- binding variables types to their terms
type IdBinding = (S.Term, S.Type)
type IdBindingSet = [IdBinding]
    -- given a binding set and a term (specifically a variable term),
    -- will use the binding set to find
    -- the type variable that matches the term
findBinding :: IdBindingSet → S.Term → Maybe S.Type
findBinding bindingSet (S.Identifier name) =
    case find (λa → (fst a) ≡ (S.Identifier name)) bindingSet of
        Just binding → Just (snd binding)
        Nothing     → Nothing
    -- these are the variables we can work with
    -- ["A","B","C",...]
variables :: [S.TypeVar]
variables = map (λx → [x]) ['A' .. 'Z']
    -- for every type equation in the constraint set,
    -- if the left hand side of the equation equals to
    -- the first argument type then change the type equation in the constraint set
changeConstraintset :: TypeConstraintSet → S.Type → S.Type → TypeConstraintSet
changeConstraintset constraintSet type1 type2 =
    [if lhs ≡ type1 then (type2, rhs) else (lhs, rhs) | (lhs, rhs) ← constraintSet]
    -- necessary for generating the next variable name
newtype Id = Id Char deriving (Show, Eq)
type NewId = Id
    -- toVar is removeId
    -- eg: toVar (Id 'b') = b
removeId :: Id → S.Type
removeId (Id c) = S.TyVar [c]
    -- fromVar is getId
    -- eg: fromVar ("cccasdf") = Id 'c'
getId :: S.TypeVar → Id
getId var = Id (head var)
    -- toTypeVar is toTypeVar
    -- eg: toTypeVar (Id 'c') = "c"
toTypeVar :: Id → S.TypeVar
toTypeVar (Id c) = [c]
```

```
-- ways to:
-- get the next name in the list of potential list of names
-- given an id name
-- get the previous name in the list of potential list of names
-- given an id name
-- grab a name for an id from a list of potential names given index
-- return the index of an Id name from a list of potential names
-- Note:
-- the elemIndex function returns the index of the first element
-- in the given list which is equal to the query element,
-- or Nothing if there is no such element.
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' ..])

  -- for applying the given subsitution to a term
applyTSubToTerm :: TypeSubstitution → S.Term → S.Term
applyTSubToTerm tSub (S.Abstraction var vtype body) =
  S.Abstraction var (subTypeVariable tSub vtype) (applyTSubToTerm tSub body)
applyTSubToTerm tSub (S.Application t1 t2) =
  S.Application (applyTSubToTerm tSub t1) (applyTSubToTerm tSub t2)
applyTSubToTerm tSub (S.Succ t) = S.Succ (applyTSubToTerm tSub t)
applyTSubToTerm tSub (S.Pred t) = S.Pred (applyTSubToTerm tSub t)
applyTSubToTerm tSub (S.IsZero t) = S.IsZero (applyTSubToTerm tSub t)
applyTSubToTerm tSub (S.If t1 t2 t3) = S.If (applyTSubToTerm tSub t1)
(applyTSubToTerm tSub t2) (applyTSubToTerm tSub t3)
applyTSubToTerm tSub (S.Fix t) = S.Fix (applyTSubToTerm tSub t)
applyTSubToTerm tSub left = left

unifyTerm :: S.Term → (U.EquationOutcome, [U.Equation TypeUnifVar TypeUnifFun])
unifyTerm t =
  let
    constraintSet = deriveTypeConstraints t
    unifencoding = encode constraintSet
    (outcome, equations) = U.unify unifencoding
  in
    (outcome, equations)

  -- for applying the given substituion to a type
subTypeVariable :: TypeSubstitution → S.Type → S.Type
```

```
subTypeVariable tsub (S.TyVar name) =
  let
    allsubs = [asub | (var, asub) ← tsub, var ≡ name]
      -- find all subs that would apply to the given type
  in
    if (length allsubs) > 0 then
      (subTypeVariable tsub (head allsubs)) else (S.TyVar name)
subTypeVariable tsub (S.TypePair t1 t2) = S.TypePair
(subTypeVariable tsub t1) (subTypeVariable tsub t2)
subTypeVariable tsub t = t
```

Main

To run the following code, $Data.Either.Unwrap$ needs to be installed. Similar to my $fromMaybe$ in Homework 3, $Data.Either.Unwrap$ consists of $fromJust$ which is a useful tool for pulling values out of monads. In addition, the Parsec libraries are needed for parsing. To evaluate the expression in the file $test.TLBN$ simply perform:

$ ghc -o main main.hs
$ ./main test.TLBN

Now for the main code which imports all the modules above:

```
import System.IO (openFile, IOMode (ReadMode), hGetContents)
import System.Environment (getArgs)
import Text.ParserCombinators.Parsec
import Data.Monoid
import Data.Either.Unwrap (fromRight)
import AbstractSyntax
import Parser
import Evaluation
import Unification
import ConstraintTyping
parseFile filePath = do
  file ← openFile filePath ReadMode
  text ← hGetContents file
  return (fromRight $ parse parserTerm filePath text)
main = do
```

```
args ← getArgs
let firstarg = head args
parseResult ← parseFile firstarg
putStrLn "---Term:---"
let parseTerm = fromJust $ reconstructType parseResult
  -- fromRight is used to grab the value out of a
  -- Right Value (recall fromMaybe)
print parseTerm
putStrLn "--Type:--"
let termType = detType Empty parseTerm
print termType
putStrLn "--Normal Form:--"
let evalResult = eval Empty parseTerm
print evalResult
```

Test Logs

Tests for **Unification** follow. We will test the following sets of equations:

```
  -- problem in hw sheet
s1 = Fun "f" [Fun "g" [Fun "a" [], Var "X"], Fun "h" [Fun "f" [Var "Y", Var "Z"]]]
s2 = Fun "g" [Var "Y", Fun "h" [Fun "f" [Var "Z", Var "U"]]]
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" []]]]
problem = [(s1, t1), (s2, t2)]
  -- martelli and montanari paper
a1 = Fun "g" [Var "X2"]
a2 = Fun "f" [Var "X1", Fun "h" [Var "X1"], Var "X2"]
b1 = Var "X1"
b2 = Fun "f" [Fun "g" [Var "X3"], Var "X4", Var "X3"]
problem1 = [(a1, b1), (a2, b2)]
m1 = Fun "g" [Var "X2"]
n1 = Fun "g" [Var "X3", Var "X1"]
problem2 = [(m1, n1)]
q1 = Fun "g" [Var "x1"]
l1 = Fun "f" [Var "x1"]
```

$problem3 = [(q1, l1)]$
$z1 = Fun$ "P" $[Var$ "a", Var "y"$]$
$z2 = Fun$ "P" $[Var$ "x", Fun "f" $[Var$ "b"$]]$
$problem4 = [(z1, z2)]$
$z11 = Fun$ "P" $[Var$ "a", Var "x", Fun "f" $[Fun$ "g" $[Var$ "y"$]]]$
$z12 = Fun$ "P" $[Var$ "z", Fun "f" $[Var$ "z"$]$, Fun "f" $[Var$ "u"$]]$
$problem5 = [(z11, z12)]$

Here are the test logs:

$*$ *Unification* $>$ *unify problem*
(*HaltWithCycle*, [(*Var* "Z", *Var* "X"), (*Var* "Y", *Var* "X"),
(*Var* "U", *Fun* "g" [*Fun* "a" [], *Var* "X"])])
$*$ *Unification* $>$ *unify problem1*
(*Success*, [(*Var* "X4", *Fun* "h" [*Fun* "g" [*Var* "X3"]]),
(*Var* "X2", *Var* "X3"), (*Var* "X1", *Fun* "g" [*Var* "X2"])])
$*$ *Unification* $>$ *unify problem2*
(*NoMatch*, [])
$*$ *Unification* $>$ *unify problem3*
(*HaltWithFailure*, [])
$*$ *Unification* $>$ *unify problem4*
(*Success*, [(*Var* "y", *Fun* "f" [*Var* "b"]), (*Var* "a", *Var* "x")])
$*$ *Unification* $>$ *unify problem5*
(*Success*, [(*Var* "u", *Fun* "g" [*Var* "y"]), (*Var* "x", *Fun* "f" [*Var* "z"]),
(*Var* "a", *Var* "z")])

Tests for **constraint based type checking** follow. We will derive type constraints on letters a to m. Types will be reconstructed for n and o. Type reconstruction will be tested more thoroughly when we go over implicit abstraction examples.

-- TEST FOR VAR
$a = S.Identifier$ "x"
-- TEST FOR ABS
$b = S.Abstraction$ ($S.Identifier$ "x") ($S.TyVar$ "y") ($S.Identifier$ "z")
-- TEST FOR APP
$c = S.Application$ ($S.Identifier$ "x") ($S.Identifier$ "y")
-- TEST FOR ZERO
$d = S.Zero$
-- TEST FOR SUCC

$e = S.Succ\ (S.Identifier\ \texttt{"x"})$

-- TEST FOR PRED

$f = S.Pred\ (S.Identifier\ \texttt{"x"})$

-- TEST FOR ISZERO

$g = S.IsZero\ (S.Identifier\ \texttt{"x"})$

-- TEST FOR TRUE

$h = S.Tru$

-- TEST FOR FALSE

$i = S.Fls$

-- TEST FOR IF

$j = S.If\ (S.Identifier\ \texttt{"x"})\ (S.Identifier\ \texttt{"y"})\ (S.Identifier\ \texttt{"z"})$

-- MORE TESTS

$k = S.If\ (S.Tru)\ (S.Tru)\ (S.Tru)$

$l = S.Abstraction\ (S.Identifier\ \texttt{"x"})\ (S.TyVar\ \texttt{"y"})\ (S.Succ\ (S.Identifier\ \texttt{"x"}))$

$m = S.Application\ (S.Abstraction\ (S.Identifier\ \texttt{"x"})$
$(S.TyVar\ \texttt{"y"})\ (S.Identifier\ \texttt{"z"}))\ (S.Tru)$

$n = S.Abstraction\ (S.Identifier\ \texttt{"x"})\ (S.TyVar\ \texttt{"y"})$
$(S.Succ\ (S.Identifier\ \texttt{"x"}))$

$o = S.Abstraction\ (S.Identifier\ \texttt{"this"})\ (S.TyVar\ \texttt{"y"})$
$(S.If\ (S.Identifier\ \texttt{"this"})\ (S.Zero)\ (S.Zero))$

$p = S.Abstraction\ (S.Identifier\ \texttt{"this"})\ (S.TyVar\ \texttt{"A"})$
$(S.If\ (S.Identifier\ \texttt{"this"})\ (S.Zero)\ (S.Zero))$

Here are the test logs:

$*\ ConstraintTyping > deriveTypeConstraints\ a$
$[\,]$

$*\ ConstraintTyping > deriveTypeConstraints\ b$
$[(A, arr\ (B, C)), (B, y)]$

$*\ ConstraintTyping > deriveTypeConstraints\ c$
$[(A, A), (B, arr\ (C, A))]$

$*\ ConstraintTyping > deriveTypeConstraints\ d$
$[(A, Nat)]$

$*\ ConstraintTyping > deriveTypeConstraints\ e$
$[(A, Nat), (B, Nat)]$

$*\ ConstraintTyping > deriveTypeConstraints\ f$
$[(A, Nat), (B, Nat)]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ g$
$[(A, \mathit{Bool}), (B, \mathit{Nat})]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ h$
$[(A, \mathit{Bool})]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ i$
$[(A, \mathit{Bool})]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ j$
$[(A, \mathit{Bool}), (B, C), (D, B)]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ k$
$[(A, \mathit{Bool}), (B, C), (D, B), (B, \mathit{Bool}), (D, \mathit{Bool})]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ l$
$[(A, \mathit{arr}\ (B, C)), (B, y), (C, \mathit{Nat}), (E, \mathit{Nat}), (E, y)]$

$* \mathit{ConstraintTyping} > \mathit{deriveTypeConstraints}\ m$
$[(A, A), (B, \mathit{arr}\ (C, A)), (B, \mathit{arr}\ (E, F)), (E, y), (C, \mathit{Bool})]$

$* \mathit{ConstraintTyping} > \mathit{reconstructType}\ n$
$\mathit{Just}\ \mathit{abs}\ (x : \mathit{Nat} \circ (\mathit{succ}\ x))$

$* \mathit{ConstraintTyping} > \mathit{reconstructType}\ o$
$\mathit{Just}\ \mathit{abs}\ (\mathit{this} : \mathit{Bool}.\mathit{If}\ \ \mathit{this}\ \mathbf{then}\ 0\ \mathbf{else}\ 0)$

Here are those examples again along with explanations of results that were listed above for letters a to m. We begin with what the program considers each type variable representation. CS stands for resulting constraint set.

-- TEST FOR VAR
-- CS:
-- empty

$a = S.\mathit{Identifier}\ \texttt{"x"}$

-- TEST FOR ABS
-- x has type variable B
-- whole expression has type variable A
-- body z has type variable C
-- CS:
-- A is B to C
-- B is y

$b = S.\mathit{Abstraction}\ (S.\mathit{Identifier}\ \texttt{"x"})\ (S.\mathit{TyVar}\ \texttt{"y"})\ (S.\mathit{Identifier}\ \texttt{"z"})$

-- TEST FOR APP
-- whole expression has type variable A
-- x has type variable B

-- y has type variable C
-- CS:
-- B is C to A
$c = S.Application\ (S.Identifier\ \texttt{"x"})\ (S.Identifier\ \texttt{"y"})$
-- TEST FOR ZERO
-- whole expression has type variable A
-- CS:
-- A is Nat
$d = S.Zero$
-- TEST FOR SUCC
-- x has type variable A
-- whole expression has type variable B
-- CS:
-- A is Nat
-- B is Nat
$e = S.Succ\ (S.Identifier\ \texttt{"x"})$
-- TEST FOR PRED
-- x has type variable A
-- whole expression has type variable B
-- CS:
-- A is Nat
-- B is Nat
$f = S.Pred\ (S.Identifier\ \texttt{"x"})$
-- TEST FOR ISZERO
-- x has type variable B
-- whole expression has type variable A
-- CS:
-- A is Bool
-- B is Nat
$g = S.IsZero\ (S.Identifier\ \texttt{"x"})$
-- TEST FOR TRUE
-- whole expression has type variable A
-- CS:
-- A is Bool
$h = S.Tru$
-- TEST FOR FALSE
-- whole expression has type variable A
-- CS:

-- A is Bool

$i = S.Fls$

-- TEST FOR IF
-- x has type variable A,
-- y has type variable B,
-- z has type variable C
-- whole expression has type variable D
-- CS:
-- A is Bool,
-- B is C
-- B is D

$j = S.If\ (S.Identifier\ \texttt{"x"})\ (S.Identifier\ \texttt{"y"})\ (S.Identifier\ \texttt{"z"})$

-- MORE TESTS

-- Assuming we have form If t1 t2 t3
-- t1 has type variable A
-- t2 has type variable B
-- t3 has type variable C
-- whole expression has type variable D
-- CS:
-- A is Bool
-- B is C
-- B is D
-- B is Bool (we explicitly listed it as Tru)
-- D is Bool (we explicitly listed it as Tru)

$k = S.If\ (S.Tru)\ (S.Tru)\ (S.Tru)$

-- x has type variable B
-- whole expression has type variable A
-- body has type variable C
-- term that Succ is working on has type variable E
-- CS:
-- A is B to C
-- B is y
-- C is Nat
-- E is y
-- E is Nat

$l = S.Abstraction\ (S.Identifier\ \texttt{"x"})\ (S.TyVar\ \texttt{"y"})\ (S.Succ\ (S.Identifier\ \texttt{"x"}))$

-- assuming app firstTerm secondTerm
-- whole expression has type variable A

-- first term has type variable B
-- second term has type variable C
-- x has type variable E
-- body z has type variable F
-- CS:
-- B is C to A
-- B is E to F
-- E is y
-- C is Bool (second term is explicitly Tru)

$m = S.Application\ (S.Abstraction\ (S.Identifier\ \texttt{"x"})\ (S.TyVar\ \texttt{"y"})\ (S.Identifier\ \texttt{"z"}))\ (S.Tru)$

Tests for **implicitly typed lambda binding** follow. Some types are not given but these types are reconstructed which you can see below. We will test the following examples using test.TLBN:

app(abs(f.abs(y.app(f,app(f,y)))),abs(x:Nat.x))

app(abs(f.f),abs(x:Nat.x))

app (abs(x.x),0)

abs (this.if this then 0 else 0 fi))

app(fix(abs(f:arr(Nat,arr(Nat,Nat)).abs(x:Nat.abs(y:Nat.if iszero(x)then y else app(app(f,pred(x)),succ(y))fi)))), succ(succ(0))),succ(succ(succ(0))))app(fix(abs (f:arr(Nat,arr(Nat,Nat)).abs(x:Nat.abs(y.if iszero(x) then y else app(app(f,pred(x)), succ(y)) fi)))), succ(succ(0))),succ(succ(succ(0))))

Here are the test logs in the same order:

Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
app (abs (f:arr (Nat,Nat).abs (y:Nat.app (f,app (f,y)))),abs (x:Nat.x))
–Type:–
arr (Nat,Nat)
–Normal Form:–
abs (y:Nat.app (abs (x:Nat.x),app (abs (x:Nat.x),y)))

Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
app (abs (f:arr (Nat,Nat).f),abs (x:Nat.x))
–Type:–
arr (Nat,Nat)
–Normal Form:–
abs (x:Nat.x)
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
app (abs (x:Nat.x),0)
–Type:–
Nat
–Normal Form:–
0
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
abs (this:Bool.If this then 0 else 0)
–Type:–
arr (Bool,Nat)
–Normal Form:–
abs (this:Bool.If this then 0 else 0)
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
app (app (fix(abs (f:arr (Nat,arr (Nat,Nat)).abs (x:Nat.abs (y:Nat.If iszero x then y else app (app (f,(pred x)),(succ y)))))),2),3)
–Type:–
Nat
–Normal Form:–
5

Tests for Let follow. Look at the ending examples for **Let POLYMORPHISM** such as double, triple, and the identity function. These are placed in test.TLBN. We will test the following examples:

let x = 0 in succ(x)

let x = 0 in let y = false in if y then succ(x) else succ(succ(x)) fi

let x = true in if x then x else x fi

let x = app (abs (true:Bool.true),false) in x

let x = false in succ(x)

let x = (if true then succ (0) else pred (0) fi) in if true then x else x fi

let x = (app (abs (x:Bool.x),false)) in if x then x else x fi

let double = abs(f.abs(y.app(f,app(f,y)))) in let a = app(app(double,abs(x:Nat.succ(succ(x)))),succ(0)) in let b = app(app(double,abs(x:Bool.x)),false) in if b then a else a fi

let id = abs(f.f) in let blah = app(id,0) in let blahblah = app (id,false) in if blahblah then 0 else 0 fi

let triple = abs(f.abs(y.app(f,app(f,app(f,y))))) in let a = app(app(triple,abs(x:Nat.succ(succ(x)))),succ(0)) in let b = app(app(triple,abs(x:Bool.x)),false) in if b then a else a fi

Here are the test logs:

Cynthias-MacBook-Pro-2:homework6 cynthiawu $ ./main test.TLBN
—Term:—
1
–Type:–
Nat
–Normal Form:–
1
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If False then 1 else 2
–Type:–
Nat
–Normal Form:–
2
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If True then True else True
–Type:–
Bool
–Normal Form:–

True
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
app (abs (true:Bool.True),False)
–Type:–
Bool
–Normal Form:–
True
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
(succ False)
–Type:–
main: cannot have succ of non Nat type
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If True then If True then 1 else (pred 0) else If True then 1 else (pred 0)
–Type:–
Nat
–Normal Form:–
1
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If app (abs (x:Bool.x),False) then app (abs (x:Bool.x),False) else app (abs (x:Bool.x),False)
–Type:–
Bool
–Normal Form:–
False
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If app (app (abs (f:arr (Bool,Bool).abs (y:Bool.app (f,app (f,y)))),abs (x:Bool.x)),False) then app (app (abs (f:arr (Nat,Nat).abs (y:Nat.app (f,app (f,y)))),abs (x:Nat.(succ (succ x)))),1) else app (app (abs (f:arr (Nat,Nat).abs (y:Nat.app (f,app (f,y)))),abs (x:Nat.(succ (succ x)))),1)
–Type:–
Nat
–Normal Form:–
5
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If app (abs (f:Bool.f),False) then 0 else 0

–Type:–
Nat
–Normal Form:–
0
Cynthias-MacBook-Pro-2:homework6 cynthiawu$ ./main test.TLBN
—Term:—
If app (app (abs (f:arr (Bool,Bool).abs (y:Bool.app (f,app (f,app (f,y)))))),abs (x:Bool.x)),False) then app (app (abs (f:arr (Nat,Nat).abs (y:Nat.app (f,app (f,app (f,y)))))),abs (x:Nat.(succ (succ x)))),1) else app (app (abs (f:arr (Nat,Nat).abs (y:Nat.app (f,app (f,app (f,y)))))),abs (x:Nat.(succ (succ x)))),1)
–Type:–
Nat
–Normal Form:–
7