Matt Parsons
CSCI 4550
December 1st, 2015
HW #7

1 Unification

This document is a Literate Haskell file. To execute it, install the Haskell build tool `stack` and run the following commands in the project directory:

```
stack setup
stack build
stack runghc Unification.lhs
```

to execute the `main` function.

As usual, we'll begin with the customary list of imports and a language pragma.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module Unification where

import Control.Monad.State
import Control.Monad.Except
import Data.Map (Map)
import qualified Data.Map as Map
```

1.1 Representing the Problem

The expression representation presented here corresponds exactly to the specification.

```
data Expr
  = Lit String
  | Var String
  | List [Expr]
  | Expr :=> Expr
  deriving (Eq, Ord, Show)

infixr 4 :=>
```

The `:=>` data constructor represents a compound expression, and is read as "Operator :=> Arguments." We can specify the association direction (to the right) and precedence. This means that the expression `foo :=> bar :=> baz` has the implied parentheses `foo :=> (bar :=> baz)`.

The pseudo code algorithm relies on early returns and failures, so we'll model that with the `ExceptT` monad transformer. We'll keep track of the θ variable using the State monad. The following represents the definition of our problem solving monad stack:

```haskell
newtype Unification a
  = Unification
  { unUnification :: ExceptT UnificationError (State Theta) a
  } deriving (Functor, Applicative, Monad,
              MonadState Theta, MonadError UnificationError)

data UnificationError
  = VariableOccursCheck
  | IncompatibleUnification Expr Expr
  | InexplicableFailure
  | OccursCheckWithNotVar
  | UnifyVarWithNotVar
  deriving Show

type Theta = Map Expr Expr

runUnify :: Expr -> Expr -> Either UnificationError Theta
runUnify a b = evalState (runExceptT (unUnification (unify a b))) mempty
```

1.2 Implementation

We're able to pattern match on the constructors used, which makes it quite easy to implement the algorithm. We want to return the final state value that we build up, so we use the `get` function to return the current state value.

```haskell
unify :: Expr -> Expr -> Unification Theta
unify a b    | a == b = get
unify (Var a) b       = unifyVar (Var a) b
unify a       (Var b) = unifyVar (Var b) a
```

The first three cases account for two equal expressions, or a variable occurring in one of the expressions being compared.

```haskell
unify (opX :=> argsX) (opY :=> argsY) = do
    unify opX opY
    unify argsX argsY
```

If we have a compound expression, we unify the operators, and then the arguments.

```haskell
unify (List as) (List bs) = do
  zipWithM unify as bs
  get
```

The `zipWithM` function is a higher order combinator that takes a function that accepts two arguments and returns a monadic value, and two lists of values. It zips the lists together pairwise, and applies the function to the elements. The monadic context gets carried through, so at each point, the function will be updating the state and may possibly throw an error. If we finish that, then we return the result state.

```
unify a b =
  throwError $ IncompatibleUnification a b
```

Finally, we throw an error if we unify two incompatible expressions.

Unifying a variable follows the pseudocode closely. Two lookups are done. If neither variable nor expression are in θ, then we'll do an occurs-check. If that succeeds, we take the target expression and replace every variable in the expression with their currently known substitutions, if any exist. After that, we update the existing variable set with our new substitution, and return the current state.

```
unifyVar :: Expr -> Expr -> Unification Theta
unifyVar var@(Var _) x = do
  theta <- get
  case Map.lookup var theta of
       Just val -> unify val x
       Nothing ->
         case Map.lookup x theta of
              Just val -> unify var val
              Nothing -> do
                occursCheck var x
                let x' = Map.foldrWithKey replace x theta
                modify (Map.insert var x')
                updateVariables var x'
                get
unifyVar _ _ = throwError UnifyVarWithNotVar
```

The `occursCheck` function recursively descends into the possible expression structure, ensuring that the `Var a` is not present in `expr`.

```
occursCheck :: Expr -> Expr -> Unification ()
occursCheck var@(Var a) expr = do
  case expr of
       Var b -> do
         if a == b
            then throwOccurs
            else gets (Map.lookup expr) >>= mapM_ (occursCheck var)
```

If `expr` is a variable, we'll immediately throw if the two variables are equal. Otherwise, we'll lookup the expression in the θ state. We then take advantage of the `Maybe` functor – if it returned a value, then we'll do an `occursCheck` with the variable and the returned value. Otherwise, we'll do nothing.

```
       Lit _ -> return ()
       List xs -> mapM_ (occursCheck var) xs
       op :=> arg -> do
         occursCheck var op
         occursCheck var arg
```

```
  where
    throwOccurs = throwError VariableOccursCheck
occursCheck _ _ = throwError OccursCheckWithNotVar
```

A literal can't contain a variable, so we're fine. For a list, we'll map over the list with the `occursCheck`, and for a compound expression, we'll check to see if the variable occurs in either operator or arguments. Finally, if we tried to do an `occursCheck` with the wrong type of expression, then we'll throw an error.

```
updateVariables :: Expr -> Expr -> Unification ()
updateVariables var@(Var a) replacement =
  modify (Map.map (replace var replacement))
```

For updating the variables, we map over the θ state and replace the occurences of the `var` variable with its replacement.

```
replace :: Expr -> Expr -> Expr -> Expr
replace source replacement target
  | source == target = replacement
  | otherwise =
    case target of
         List xs ->
           List (map (replace source replacement) xs)
         op :=> arg ->
           replace source replacement op :=> replace source replacement arg
         a -> a
```

If `source` and `target` are equal, then we return the `source`. Otherwise, we'll descend recursively into the expression structure. If the target is a `List` of expressions, then we map replacement over the list. If it's a compound expression, then we replace the operator and the arguments.

1.3 Solve it

We represent the given problem examples:

1. $P(X)$ and $P(a)$,
2. $P(f(Y, g(Y)))$ and $P(f(a, X))$,
3. $P(Y, Y)$ and $P(f(a), a)$
4. $P(Y, f(Y))$ and $P(X, X)$.

```
ex :: Int -> (Expr, Expr)
ex 1 =
  ( Var "P" :=> Var "X"
  , Var "P" :=> Lit "a"
  )
```

```
ex 2 =
  ( Var "P" :=> Lit "f" :=> List [ Var "Y", Lit "g" :=> Var "Y"]
  , Var "P" :=> Lit "f" :=> List [ Lit "a", Var "X" ]
  )
ex 3 =
  ( Var "P" :=> List [ Var "Y", Var "Y" ]
  , Var "P" :=> List [ Lit "f" :=> Lit "a", Lit "a" ]
  )
ex 4 =
  ( Var "P" :=> List [ Var "Y", Lit "f" :=> Var "Y" ]
  , Var "P" :=> List [ Var "X", Var "X" ]
  )
```

And we execute the unification function for each example:

```
main :: IO ()
main = forM_ [1..4] (print . uncurry runUnify . ex)
```

The output from running the code is:

```
$ stack runghc Unification.hs
Right (fromList [(Var "X",Lit "a")])
Right (fromList [(Var "X",Lit "g" :=> Lit "a"),(Var "Y",Lit "a")])
Left (IncompatibleUnification (Lit "f" :=> Lit "a") (Lit "a"))
Left VariableOccursCheck
```

`Right` indicates that a unification was successful. The θ map is converted into a list of pairs, where the first element in the pair is the key, and the second element is the expression that it was unified to. The algorithm correctly unified $\{X/a\}$ in the first example and $\{X/g(a), Y/a\}$ in the second.