Title

Brian Kuhns

April 2019

1 Modules

1.1 Types

```
module Types where

import BinTree
import Data.List
import qualified Data.Map as M
```

1.1.1 Types Used During Parsing

A Formula is Either a variable with a name a function one of the logical operators or a Quantified formula. At this stage both Predicates and Functions are stored as functions, and Both Variables are stored as Var. Show is Derived for Debugging purposes.

```
data Formula = Var String
             | Function String [Formula]
             | And Formula Formula
             | Or Formula Formula
             | Not Formula
             | Implies Formula Formula
             | Quantified Quantifier Formula
             deriving (Eq,Show)

data Quantifier = Forall String | Exists String deriving (Eq,Show)
```

1.1.2 Types Used in the Prover

The types used in the prover differentiate predicates from functions and constants from variables. They also use ID, (which is a type alias for Int) rather than Strings to represent names. This should improve performance, but also makes it easy to systematically generate new names.

```
type ID = Int
```

Predicates represent a possibly negated predicate, and all its arguments. The Bool represents rather the predicate is negated, the ID, is

```
data Predicate = P Bool ID [Term] deriving(Eq,Ord)
```

Terms are either variables, constants or functions. Inside the prover constants are treated as nularry functions. Variables only store their ID. Functions store their ID and

```
data Term = V ID | Symbol ID [Term] deriving(Eq,Ord)
```

```
termPo :: Term -> Term -> Ordering
termPo (V _) _ = EQ
termPo _ (V _) = EQ
termPo (Symbol li lts) (Symbol ri rts) = case compare li ri of
  LT -> LT
  GT -> GT
  EQ -> compare lts rts
```

1.2 BinTree

```
module BinTree where
import Control.Comonad
```

This bin-tree module is written to support derivation histories. For this reason each Node has exactly two children and the leaves which represent givens have no children.

```
data BinTree a = Node a (BinTree a) (BinTree a) | Leaf a
```

The functor instance is pretty standard, the function is mapped over each element in the tree.

```
instance Functor BinTree where
  fmap f (Leaf x) = Leaf $ f x
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

Much like the BinTree type can be viewed as a restriction of Tree to trees where each node has 0 or 2 children. The comonad instance restricts the comonad instance for Tree. Extract returns the root of the tree. Duplicate replaces each point in the tree with the subtree of which it is the root.

```
instance Comonad BinTree where
  extract (Node x _ _) = x
  extract (Leaf x )    = x
  duplicate t@(Leaf _) = Leaf t
  duplicate t@(Node _ l r)  = Node t (duplicate l) (duplicate r)
```

The Foldable instance is also pretty standard, It goes left children node right children

```
instance Foldable BinTree where
  foldMap f (Leaf x) = f x
  foldMap f (Node x l r) = (foldMap f l) <> (f x) <> (foldMap f r)
```

This function gets all the leaves of the tree as a list.

```
leaves :: BinTree a -> [a]
leaves (Leaf x) = [x]
leaves (Node _ l r) = leaves l ++ leaves r
```

The show instance shows the tree as structure with indentation putting each element on a different line. The leaves are prefixed with G to denote givens, and the nodes with D to denote derived.

"init" is used twice after unlines because unlines puts a newline at the end which is desirable in neither case here.

```
instance Show a => Show (BinTree a) where
  show (Leaf n) = "G " ++ show n
  show (Node n l r) = let
    indent = init . unlines . map ("  " ++) . lines
    in
    init . unlines $ [ "D " ++ (show n) , indent $ show l , indent $ show r]
```