Haskell Assignment 1

Philip Rajani Lassen (vgh804)
Nicolas Ringsmose Larsen (vgn209)

September 2018

1 evalExpr

For the resubmission, we worked on using monadic style more, and finishing up `evalExpr`. We ended up with a working implementation for nearly all cases. The most troublesome cases was when we were given an array of expressions. This is done in expressions of type

`Array [Expr] and Call FunName [Expr]`

The simplest way for us to evalute a list of expressions seemed to be to map `evalExpr` over the list. However, this returns a `[SubsM Value]`. For both our original submission and our resubmission, we spent a lot of time figuring out how to take each `SubsM Value` and append the inner values on to each other.

Enter our helper function `concatSubsM`:

```
    concatSubsM :: [SubsM Value] -> SubsM [Value]
```

The function works by folding the `liftM2` function (with the const function inside) over the `[SubsM Value]` list. Thus creating a list of the values inside each `SubsM`, and returning them inside a single `SubsM [Value]`.

For the array comprehensions, we had some issues with the finer points. These will be addressed in the assessment section, this section will go over our design choices instead.
For `ACFor` we used a strategy similar to the `Array` and `Call` case earlier. This time, we mapped a helper function over the list contained in the `Expr` part of `ACFor Ident Expr ArrayCompr`. The helper function esentially binds the `Ident` to a value (each value in the list) and then evaluates the `ArrayCompr` with this binding.
For `ACIf` we simply evaluated the expression and moved on to evaluating the following array comprehension if the first evaluation returned true. However, if it returned false, we decided to return an `UndefinedVal`, even though we really just wanted to return nothing. This would make for a slight problem, as we will see later.

2 Monad Laws

2.1 Left Identity

We need to show that

```
do {x' <- return x
    f x'
    }
is the same as
do {f x}
In our Monad the result of x' in x' <- return x is indeed x. Thus
f x' = f x
Thus we have shown left Identity
```

2.2 Right Identity

We need to show that

```
  m >>= return = m
```

This follows from the fact that

```
  m >>= return = SubsM $ \(c1,c2) -> case m of SubsM x ->
                    let var = x (c1,c2) in
                        case var of
                        Right (a, env) -> case (return a) of
                            SubsM y -> y (env,c2)
                            Left er -> Left er
```

And it follows from the definition of `return a` that `m >> return` reduces to `m` in the above function

3 Process

In order to implement the code we spent a large portion of the time reading about Monads. We hoped that this would help speed our ability to produce the code, however we still lacked some understanding. When we wrote the code we had difficulties getting our implementations to pass the type-checker. This was due to our lack of understanding of why functions and types were chosen to be what they were.

4 Assessment

Our implementation works for most cases, and we feel like we have gotten a better grasp on monads for the resubmission. However, there are still three

issues (two of them somewhat resolved) with the list comprehensions.
The first one happened if an `ACFor` had an `ACIf` inside. Since our implementation of `ACIf` returns an `UndefinedVal` if the test is false, the resulting array would contain such values instead of just throwing them away.
We were able to put a bandaid on this problem by filtering out `UndefinedVal` in the `ACFor` case. However, this means that there can never be an UndefinedVal in the resulting array. We are not sure if this matters for list comprehensions, since we already have the `Array(e)` function.

The second problem is that we do not know how to "reset" the context after evaluating the list comprehensions. Thus, all bindings made inside a list comprehensions persists outside it as well.

Lastly, for nested `ACFor` expressions, we returned an `ArrayVal [ArrayVal [Value]]`, which was not the intention of the nested for loops. We patched this problem by checking explicitly for lists of ArrayVals inside `ACFor`. In this case, we would use the helper function `concatArrayVals` to concatinate all of the inner lists to a single `ArrayVal [Value]`

Overall, we feel more comfortable using monads, but confusions still arise for these special cases.

5 Testing

We were able to pass all but two of the twenty nine tests of the online TA (27/29 were passed). We discussed why our implementation was unable to pass these two tests earlier. For our own testing we used the Tasty Testing framework, and built our Test Suite in Test.hs. We tried to cover some edge cases. We wrote 8 tests. In some of the cases a test may be comparing a list to another list, which means that there are more than 8 tests. The tests can be run by running the command

```
stack test
```

The results should look something like

```
Progress 1/2: subscript-interpreter-0.0.0Tests
  Boolean Tests:   OK
  Number Tests:    OK
  String Tests:    OK
  Undefined Tests: OK
  Array Tests:     OK
  Comma Tests:     OK
  Functions calls: OK
  Other Tests:     OK

All 8 tests passed
```

6 Code

```
module SubsInterpreter
       (
         Value(..)
       , runExpr
       -- You may include additional exports here, if you want to
       -- write unit tests for them.
       )
       where

import SubsAst

-- You might need the following imports
import Control.Monad
import qualified Data.Map as Map
import Data.Map(Map)

-- | A value is either an integer, the special constant undefined,
--   true, false, a string, or an array of values.
-- Expressions are evaluated to values.
data Value = IntVal Int
           | UndefinedVal
           | TrueVal | FalseVal
           | StringVal String
           | ArrayVal [Value]
           deriving (Eq, Show)

type Error = String
type Env = Map Ident Value
type Primitive = [Value] -> Either Error Value
type PEnv = Map FunName Primitive
type Context = (Env, PEnv)

initialContext :: Context
initialContext = (Map.empty, initialPEnv)
  where initialPEnv =
          Map.fromList [ ("===", equals)
                       , ("<", lt)
                       , ("+", plus)
                       , ("*", mul)
                       , ("-", sub)
                       , ("%", modulo)
                       , ("Array", mkArray)
                       ]
```

```
newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a, Env)}

instance Monad SubsM where
  return x = SubsM $ \c -> case c of
                    (env, _) -> Right (x, env)
  m >>= f = SubsM $ \(env, penv) -> case runSubsM m (env,penv) of
                          Right (a,newEnv)  -> runSubsM (f a) (newEnv,penv)
                          Left err          -> Left err
  fail s =  SubsM $ \_-> Left s

-- You may modify these if you want, but it shouldn't be necessary
instance Functor SubsM where
  fmap = liftM
instance Applicative SubsM where
  pure = return
  (<*>) = ap

equals :: Primitive
equals [x, y]
  | x == y = Right TrueVal
  | otherwise = Right FalseVal
equals _ = Left "Values can not be compared"

lt :: Primitive
lt [IntVal x, IntVal y] = if x < y then Right TrueVal else Right FalseVal
lt [StringVal x, StringVal y] = if x < y then Right TrueVal else Right FalseVal
lt _ = Left "Values can not be compared"

plus :: Primitive
plus [(IntVal x), (IntVal y)] = Right $ IntVal $ x+y
plus [(StringVal x), (StringVal y)] = Right $ StringVal $ x ++ y
plus [(IntVal x), (StringVal y)] = Right $ StringVal $ show x ++ y
plus [(StringVal x), (IntVal y)] = Right $ StringVal $  x ++ show y
plus _ = Left "Values can not be added"

mul :: Primitive
mul [(IntVal x), (IntVal y)] = Right $ IntVal (x*y)
mul _ = Left "Values can not be multiplied"

sub :: Primitive
sub [(IntVal x), (IntVal y)] = Right $ IntVal (x-y)
sub _ = Left "Values can not be subtracted"

modulo :: Primitive
modulo [(IntVal x), (IntVal y)] = Right $ IntVal (x `mod` y)
modulo _ = Left "Values can not be modded"
```

```
mkArray :: Primitive
mkArray [IntVal n] | n >= 0 = return $ ArrayVal (replicate n UndefinedVal)
mkArray _ = Left "Array() called with wrong number or type of arguments"

modifyEnv :: (Env -> Env) -> SubsM ()
modifyEnv f = SubsM $ \(env, _) -> Right ((), f env)

putVar :: Ident -> Value -> SubsM ()
putVar name val = modifyEnv $ Map.insert name val

getVar :: Ident -> SubsM Value

getVar name = SubsM $ \(env, _) -> case Map.lookup name env of
                                        Just a  -> Right (a, env)
                                        Nothing -> Left "Error"

getFunction :: FunName -> SubsM Primitive
getFunction name = SubsM $ \(env, penv) -> case Map.lookup name penv of
                                        Just a  -> Right (a, env)
                                        Nothing -> Left "Error"

evalExpr :: Expr -> SubsM Value
evalExpr expr = case expr of
                  Number i -> return $ IntVal i
                  String s -> return $ StringVal s
                  Array exps -> do
                          vals <- let subsList = map evalExpr exps in
                                        concatSubsM subsList
                          return $ ArrayVal vals
                  Var ident -> getVar ident
                  Assign ident exp1 -> do
                          value <- evalExpr exp1
                          putVar ident value
                          getVar ident
                  Call fun exps -> do
                          f <- getFunction fun
                          vals <- let subsMList = map evalExpr exps in
                                        concatSubsM subsMList
                          case f vals of
                            Left err -> fail err
                            Right val -> return val
                  Comma exp1 exp2 -> evalExpr exp1 >> evalExpr exp2
                  Compr (ACBody exp1) -> evalExpr exp1
                  Compr (ACFor ident exp1 arrCmp) -> do
                          list <- evalExpr exp1
```

```
              case list of
               ArrayVal arr -> do
                vals <- let subsMList = map mapHelp arr
                    in concatSubsM subsMList
                -- The filter is just used to remove undefined values,
                -- since we return them in the ACIf if the expression
                -- returns false.
                filtered <- return $ filter (/= UndefinedVal) vals
                return $ ArrayVal $ case filtered of
                 -- For nested loops, we get a list of ArrayVals,
                 -- we just want a single list.
                 (ArrayVal _ : _)  -> concatArrayVals filtered
                 _                 -> filtered
               _ -> fail "Expression must be an ArrayVal"
           -- binds the id to a variable x then evaluates exp1 with this new
           where mapHelp x = putVar ident x >> evalExpr (Compr arrCmp)
          Compr (ACIf exp1 arrCmp) -> do
              bool <- evalExpr exp1
              case bool of
               TrueVal   -> evalExpr (Compr arrCmp)
               FalseVal  -> return UndefinedVal
               _         -> fail "Not a boolean"
          TrueConst -> return TrueVal
          FalseConst -> return FalseVal
          Undefined -> return UndefinedVal

-- "Concatinates" an entire list of SubsM Values into a single SubsM [Value]
concatSubsM :: [SubsM Value] -> SubsM [Value]
concatSubsM = foldr (liftM2 (:)) (return [])

-- Concatinates an ArrayVal of ArrayVal xs to a single ArrayVal xs
concatArrayVals :: [Value] -> [Value]
concatArrayVals = concatMap valueList
 where valueList (ArrayVal xs) = xs
    valueList ys = [ys]

runExpr :: Expr -> Either Error Value
runExpr exp1 = case runSubsM (evalExpr exp1) initialContext of
               Left er -> Left er
               Right (a, _) -> Right a
```