A SubScript Interpreter

Assignment 1

Kai Arne S. Myklebust, Silvan Adrian

Handed in: October 3, 2018

Contents

1 Design/Implementation

Overall in this assignment our goal was to not use helper functions, where not specificaly needed. This to make readability easier and making the code less complex since many of our helper functions from the previous assignment were unnecessary. We started by making a helper function for each arithmetic operation from ”initial context”. This was to get the first and second element from the list. After a while we got really annoyed doing that and found out that you can just get the first and second element by changing to [] list-brackets. We used head and tail in equality, but onlineTA says that we should not use them. We check for empty lists and a list of different length, so we check for possible errors which we found while testing from head and tail. In evalExpr we used the do notation to make it more readable when we have multiple actions in the same statement.

EDIT RESUBMISSION: We proof that our monad instance for SubsM satisfies the monad laws by testing each of the three monad laws individually. For the second Law we wrote a Test which we can’t test for equality therefore the test is commented out (see comment in Test.hs) but we were able to reproduce that we got the same result back by running it in ghci.

- return v >>= f == f v
- m >>= (\a -> return a) == m
- (m >>= f) >>= g == m >>= (\a -> (f a >>= g))

We worked our way through the monads, by doing many tests and figuring out what return values where needed. In addition we read a lot of monad introductions and articles and used more than 20 hours just to partly understand this assignment. Also we got help in another TA class. In the monad return we knew that we needed the right value of SubsM and worked our way from there. In fail we knew that we needed the left side value (Error). For the bind we used the slides from the lecture and the haskell wiki to work our way forward. Underway we did testing to see that we got the right return values.

2 Code Assessment

According to our own tests and onlineTa, everything except array compression works. Array compression was the hardest part and is only partly working. ACFor only works for arrays with numbers. If you have a String it sees the string as only one element and not multiple characters. ACFor does not work with nested for's. We weren't able to make nested for's working. We use putVar and know that it works. So the ACFor can see the variable, but one problem is that only the body should see the variable but now the whole ACFor sees the new variable. In ACIf we have the problem when the if clause evaluates to false it has to return a ´Value´, but the assignment says it should return nothing. IF ACIf is inside a ACFor our solution does not work, but if there's a single ACIf then it works.

We ran our own tests to show these failures. These can be run by ´stack test´ (the 6 out of 86 tests fail, which we also described in the assessment above).

One place where our test cases were able to help us find errors was in equality and having arrays of different lengths. We fixed it by checking for empty arrays and for different array lengths then we return a FalseVal.

A Code Listing

```
1 module SubsInterpreter
2         (
3           Value(..)
4         , runExpr
5         , equality
```

```
6         , smallerThen
7         , add
8         , mul
9         , sub
10        , modulo
11        , mkArray
12        , SubsM
13        -- You may include additional exports here, if you want to
14        -- write unit tests for them.
15        )
16        where
17
18  import SubsAst
19
20  -- You might need the following imports
21  import Control.Monad
22  import qualified Data.Map as Map
23  import Data.Map(Map)
24
25
26  -- | A value is either an integer, the special constant
    ↪  undefined,
27  --   true, false, a string, or an array of values.
28  -- Expressions are evaluated to values.
29  data Value = IntVal Int
30             | UndefinedVal
31             | TrueVal | FalseVal
32             | StringVal String
33             | ArrayVal [Value]
34             deriving (Eq, Show)
35
36
37  type Error = String
38  type Env = Map Ident Value
39  type Primitive = [Value] -> Either Error Value
40  type PEnv = Map FunName Primitive
41  type Context = (Env, PEnv)
42
43  initialContext :: Context
44  initialContext = (Map.empty, initialPEnv)
45    where initialPEnv =
46            Map.fromList [ ("===", equality)
47                         , ("<", smallerThen)
48                         , ("+", add)
```

```
                          , ("*", mul)
                          , ("-", sub)
                          , ("%", modulo)
                          , ("Array", mkArray)
                          ]

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

instance Monad SubsM where
  return x = SubsM $ \(e, _) -> Right (x,e)
  m >>= f = SubsM $ \c@(_, p) -> runSubsM m c >>= \(x, e') ->
   ↪  runSubsM (f x) (e', p)
  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

equality :: Primitive
equality [IntVal a, IntVal b] = if (a == b) then Right TrueVal
  ↪  else Right FalseVal
equality [UndefinedVal, UndefinedVal] = Right TrueVal
equality [StringVal a, StringVal b] = if a == b then Right
  ↪  TrueVal else Right FalseVal
equality [TrueVal, TrueVal] = Right TrueVal
equality [FalseVal, FalseVal] = Right TrueVal
equality [ArrayVal [], ArrayVal []] = Right TrueVal
equality [ArrayVal [], ArrayVal _] = Right FalseVal
equality [ArrayVal _, ArrayVal []] = Right FalseVal
equality [ArrayVal a, ArrayVal b] = if head a == head b
  then equality [ArrayVal (tail a), ArrayVal (tail b)]
  else Right FalseVal
equality [_, _] = Right FalseVal
equality _ = Left "Wrong number of arguments"

smallerThen :: Primitive
smallerThen [IntVal a, IntVal b] = if a < b then Right TrueVal
  ↪  else Right FalseVal
smallerThen [StringVal a, StringVal b] = if a < b then Right
  ↪  TrueVal else Right FalseVal
```

```haskell
smallerThen [_, _] = Right FalseVal
smallerThen _ = Left "Wrong number of arguments"

add :: Primitive
add [IntVal a, IntVal b] = Right (IntVal(a + b))
add [StringVal a, StringVal b] = Right (StringVal(a ++ b))
add [IntVal a, StringVal b] = Right(StringVal(show a ++ b))
add [StringVal a, IntVal b] = Right(StringVal(a ++ show b))
add [_, _] = Left "No Int or String"
add _ = Left "Wrong number of arguments"

mul :: Primitive
mul [IntVal a, IntVal b] = Right (IntVal(a*b))
mul [_, _] = Left "No Integer"
mul _ = Left "Wrong number of arguments"

sub :: Primitive
sub [IntVal a, IntVal b] = Right (IntVal(a-b))
sub [_, _] = Left "No Integer"
sub _ = Left "Wrong number of arguments"

modulo :: Primitive
modulo [IntVal a, IntVal b] = if b == 0 then Left "Division by
    ↪ Zero" else Right (IntVal(mod a b))
modulo [_, _] = Left "No Integer"
modulo _ = Left "Wrong number of arguments"

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 $ \(e, _) -> Right ((), f e)

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

getVar :: Ident -> SubsM Value
getVar name = SubsM $ \(e, _) -> case Map.lookup name e of
                                    Just v -> Right (v, e)
                                    Nothing -> Left "No value
                                     ↪ found in map"
```

```haskell

getFunction :: FunName -> SubsM Primitive
getFunction name = SubsM $ \(e, p) -> case Map.lookup name p of
                                        Just v -> Right (v, e)
                                        Nothing -> Left "No value
                                        ↪  found in map"

evalExpr :: Expr -> SubsM Value
evalExpr Undefined = return UndefinedVal
evalExpr TrueConst = return TrueVal
evalExpr FalseConst = return FalseVal
evalExpr (Number a) = return $ IntVal a
evalExpr (String a) = return $ StringVal a
evalExpr (Var a) = getVar a
evalExpr (Array []) = return (ArrayVal [])
evalExpr (Array (a:ax)) = do
  a <- evalExpr a
  ArrayVal ax <- evalExpr(Array ax)
  return (ArrayVal (a:ax))
evalExpr (Compr (ACBody e)) = evalExpr e
evalExpr (Compr (ACFor i e c)) = do
  a <- evalExpr e
  case a of
    ArrayVal xa -> do
      val <- mapM (\x -> do
        putVar i x
        evalExpr(Compr c)) xa
      return (ArrayVal val)
    StringVal xs -> do
      (StringVal s) <- (\_ -> evalExpr(Compr c)) xs
      return (StringVal s)
    _ -> fail "FOR needs an array or string"

evalExpr (Compr (ACIf e c)) = do
  a <- evalExpr e
  case a of
    TrueVal -> evalExpr (Compr c)
    FalseVal -> return (ArrayVal [])
    _ -> fail "IF needs a boolean"

evalExpr (Call a b) = do
  f <- getFunction a
  ArrayVal bv <- evalExpr (Array b)
  case f bv of
```

```
170       Right r -> return r
171       Left l -> fail l
172
173   evalExpr (Assign a b) = do
174     v <- evalExpr b
175     putVar a v
176     return v
177
178   evalExpr (Comma a b) = do
179     _ <- evalExpr a
180     evalExpr b
181
182   runExpr :: Expr -> Either Error Value
183   runExpr expr = case runSubsM (evalExpr expr) initialContext of
184                    Right r -> Right (fst r)
185                    Left l -> Left l
```