**Design and implementation**

1. Operations of the monad

In the look function we used the function lookup to search for the value in the list of tuples and because it returns a maybe value, we used just and nothing.

1. Helper functions for interpreter

In function operate we treated also the cases when there is a division and mod by zero, and returned and error. Also, we treated the case when the values are invalid data type. For the apply function we created some helping functions, “rangeFunction”, “isRight”, “wrapComp”, “getRight”, “getLeft”, “printF”. We used this function to remove redundant code structures an to make the code more readeable.

1. Main functions of interpreter

For the main functions of the interpreter, we added to the list of helper function “extractList”.

**Assessment of the code**

We tried to asses as much as possible, and to test all function. In total in the test suite there are 31 tests with different function and values. The assessment of the code was mainly done with the online tool as we didn’t know how to test them any better with the test suite.

To run the test suite, you need to run in terminal stack test.

**Completeness**

The code is not 100% completed. All the other functions except “eval” works fine. The eval function is working for almost all cases, except the case when Exp is “Compr Exp [CClause]”. That part is partially done, it’s implemented only for the case where the list ([CClause]) is empty. We tried implementing it for other cases, but it wasn’t successful. The attempt is commented below the function.

**Correctness**

There aren’t any known bugs. The code that is implemented works fine. Beside the missing implementation that was stated in the Completeness part, everything works fine.

The correctness of the code was mainly tested with the online tool (<https://find.incorrectness.dk/>), and in many cases the tests were passed. Out of 110 tests we failed only 13, and all of them were failed due to now having an implementation for eval for “Compr Exp [CClause]” case.



Beside this method we also tested the code with a suite we wrote. We tried to address most of the possible cases.

**Efficiency**

The overall space complexity of the code is O(n). About time complexity we are not sure how to measure it, but from what we saw from the tests and from the online tool they run fairly fast, so we think the time complexity should be fine. As for improving the code, we are not sure, what it can be done, as we tried to write the code in the most efficient way.

**Robustness**

From the point of view of robustness, the code is robust, in situations where the input data may be correct from the Haskell point of view, but are not good as data for functions, then an error is returned. Many of the functions in the code handle possible exceptions. The divide-by-zero exception is handled for both the div and the mod.

**Maintainability**

The code is in good shape, all the repetitive code was transformed into functions, so instead of having a lot of copy pasted code, there are functions, that are easy to maintain and modify. From the point of view of monadic abstraction, the code respects this, all functions after "abort", "look", "withBinding" and "output" functions use these four functions instead of relying on Comp's implementation. As for the test suits, most of the cases are copy pasted segments with changes.

From the point of view of comments, the code doesn’t contain so many of them because we didn’t know if we should add them. The layout of the code is right all, the indentation is respected.

**Other**

Warmup.hs

-- Edit all the definitions with "undefined"

module Warmup where

import Control.Monad

type ReadData = Int

type WriteData = String  -- must be an instance of Monoid

type StateData = Double

-- Plain version of RWS monad

newtype RWSP a = RWSP {runRWSP :: ReadData -> StateData ->

                                    (a, WriteData, StateData)}

-- complete the definitions

--  Monad m => RWSP a -> (a -> RWSP b) -> RWSP b

instance Monad RWSP where

  return a = RWSP (\\_ s -> (a,mempty,s))

  m >>= f = RWSP $ \r0 s0 ->

    let (a, w1, s1) = runRWSP m r0 s0

    in let (b,w2, s2) = runRWSP (f a) r0 s1

    in (b, w1 <> w2 , s2)

-- No need to touch these

instance Functor RWSP where

  fmap = liftM

instance Applicative RWSP where

  pure = return; (<\*>) = ap

-- returns current read data

askP :: RWSP ReadData

askP = RWSP (\r s -> (r, mempty, s))  -- freebie

-- runs computation with new read data

withP :: ReadData -> RWSP a -> RWSP a

withP r' m = RWSP (\\_ s -> runRWSP m r' s)

-- adds some write data to accumulator

tellP :: WriteData -> RWSP ()

tellP w = RWSP (\\_ s-> ((),w,s))

-- returns current state data

getP :: RWSP StateData

getP = RWSP (\\_ s -> (s, mempty,s))

-- overwrites the state data

putP :: StateData -> RWSP ()

putP s' = RWSP (\\_ \_ -> ((),mempty,s'))

-- sample computation using all features

type Answer = String

sampleP :: RWSP Answer

sampleP =

  do r1 <- askP

     r2 <- withP 5 askP

     tellP "Hello, "

     s1 <- getP

     putP (s1 + 1.0)

     tellP "world!"

     return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

type Result = (Answer, WriteData, StateData)

expected :: Result

expected = ("r1 = 4, r2 = 5, s1 = 3.5", "Hello, world!", 4.5)

testP = runRWSP sampleP 4 3.5 == expected

-- Version of RWS monad with errors

type ErrorData = String

newtype RWSE a = RWSE {runRWSE :: ReadData -> StateData ->

                                    Either ErrorData (a, WriteData, StateData)}

-- Hint: here you may want to exploit that "Either ErrorData" is itself a monad

instance Monad RWSE where

  return a = RWSE $ \\_ s -> Right (a,mempty,s)

  m >>= f = RWSE $(\r s -> case runRWSE m r s of

    Right (a, w1, s1 ) -> case runRWSE (f a) r s1 of

        (Right (b, w2, s2)) -> Right (b, w1<>w2, s2)

        (Left e) -> Left e

    (Left e) -> Left e)

instance Functor RWSE where

  fmap = liftM

instance Applicative RWSE where

  pure = return; (<\*>) = ap

askE :: RWSE ReadData

askE = RWSE (\r s -> Right (r, mempty, s))

withE :: ReadData -> RWSE a -> RWSE a

withE r' m = RWSE (\\_ s -> runRWSE m r' s)

tellE :: WriteData -> RWSE ()

tellE w = RWSE (\\_ s-> Right ((),w,s))

getE :: RWSE StateData

getE = RWSE $ \\_ s -> Right(s,mempty,s)

putE :: StateData -> RWSE ()

putE s' = RWSE $ \\_ \_ -> Right((),mempty,s')

throwE :: ErrorData -> RWSE a

throwE e = RWSE (\\_ \_ -> Left e)

sampleE :: RWSE Answer

sampleE =

  do r1 <- askE

     r2 <- withE 5 askE

     tellE "Hello, "

     s1 <- getE

     putE (s1 + 1.0)

     tellE "world!"

     return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

-- sample computation that may throw an error

sampleE2 :: RWSE Answer

sampleE2 =

  do r1 <- askE

     x <- if r1 > 3 then throwE "oops" else return 6

     tellE "Blah"

     return $ "r1 = " ++ show r1 ++ ", x = " ++ show x

testE = runRWSE sampleE 4 3.5 == Right expected

testE2 = runRWSE sampleE2 4 3.5 == Left "oops"

-- Generic formulations (nothing further to add/modify)

-- The class of monads that support the core RWS operations

class Monad rws => RWSMonad rws where

  ask :: rws ReadData

  with :: ReadData -> rws a -> rws a

  tell :: WriteData -> rws ()

  get :: rws StateData

  put :: StateData -> rws ()

-- And those that additionally support throwing errors

class RWSMonad rwse => RWSEMonad rwse where

  throw :: ErrorData -> rwse a

-- RWSP is an RWS monad

instance RWSMonad RWSP where

  ask = askP; with = withP; tell = tellP; get = getP; put = putP

-- So is RWSE

instance RWSMonad RWSE where

  ask = askE; with = withE; tell = tellE; get = getE; put = putE

-- But RWSE also supports errors

instance RWSEMonad RWSE where

  throw = throwE

-- Generic sample computation, works in any RWS monad

sample :: RWSMonad rws => rws Answer

sample =

  do r1 <- ask

     r2 <- with 5 ask

     tell "Hello, "

     s1 <- get

     put (s1 + 1.0)

     tell "world!"

     return $ "r1 = " ++ show r1 ++ ", r2 = " ++ show r2 ++ ", s1 = " ++ show s1

-- Generic sample computation, works in any RWS monad supporting errors

sample2 :: RWSEMonad rwse => rwse Answer

sample2 =

  do r1 <- ask

     x <- if r1 > 3 then throw "oops" else return 6

     tell "Blah"

     return $ "r1 = " ++ show r1 ++ ", x = " ++ show x

testP' = runRWSP sample 4 3.5 == expected

testE' = runRWSE sample 4 3.5 == Right expected

testE2' = runRWSE sample2 4 3.5 == Left "oops"

allTests = [testP, testE, testE2, testP', testE', testE2']

BoaInterp.hs

-- Skeleton file for Boa Interpreter. Edit only definitions with 'undefined'

module BoaInterp

  (Env, RunError(..), Comp(..),

   abort, look, withBinding, output,

   truthy, operate, apply,

   eval, exec, execute)

  where

import BoaAST

import Control.Monad

type Env = [(VName, Value)]

data RunError = EBadVar VName | EBadFun FName | EBadArg String

  deriving (Eq, Show)

newtype Comp a = Comp {runComp :: Env -> (Either RunError a, [String]) }

instance Monad Comp where

  return a = Comp $ \\_ -> (Right a, mempty)

  m >>= f = Comp $ \env -> case runComp m env of

      (Right a, \_) ->

        let (\_, s1) = runComp m env

        in let (b, s2) = runComp (f a) env

        in (b, s1 <> s2)

      (Left error, s) -> (Left error, s)

-- You shouldn't need to modify these

instance Functor Comp where

  fmap = liftM

instance Applicative Comp where

  pure = return; (<\*>) = ap

-- Operations of the monad

abort :: RunError -> Comp a

abort re = Comp $ \\_ -> (Left re, mempty)

look :: VName -> Comp Value

look name = Comp $ \env -> case lookup name env of

  Just value-> (Right value, mempty)

  Nothing -> (Left (EBadVar name), mempty)

withBinding :: VName -> Value -> Comp a -> Comp a

withBinding v x m = Comp (\env -> runComp m ((v, x):env))

output :: String -> Comp ()

output s = Comp $ \\_ -> (Right (),[s])

-- Helper functions for interpreter

truthy :: Value -> Bool

truthy x

  | x == NoneVal = False

  | x == FalseVal = False

  | x == IntVal 0 = False

  | x == StringVal "" = False

  | x == ListVal [] = False

  | otherwise = True

operate :: Op -> Value -> Value -> Either String Value

operate Plus (IntVal v1) (IntVal v2) = Right (IntVal (v1 + v2))

operate Minus (IntVal v1) (IntVal v2) = Right (IntVal (v1 - v2))

operate Times (IntVal v1) (IntVal v2) = Right (IntVal (v1 \* v2))

operate Div (IntVal v1) (IntVal v2)

  | v2 == 0 = Left "error"

  | otherwise = Right (IntVal (v1 `div` v2))

operate Mod (IntVal v1) (IntVal v2)

  | v2 == 0 = Left "error"

  | otherwise = Right (IntVal (v1 `mod` v2))

operate Eq v1 v2

  | v1 == v2 = Right TrueVal

  | otherwise = Right FalseVal

operate Less (IntVal v1) (IntVal v2)

  | v1 < v2 = Right TrueVal

  | otherwise = Right FalseVal

operate Greater (IntVal v1) (IntVal v2)

  | v1 > v2 = Right TrueVal

  | otherwise = Right FalseVal

operate In \_ (ListVal []) = Right FalseVal

operate In v1 (ListVal (x:xs)) =

    if operate Eq v1 x == Right TrueVal

      then Right TrueVal

    else

      operate In v1 (ListVal xs)

operate \_ \_ \_ = Left "error"

listHelper :: [Value] -> String

listHelper [] = ""

listHelper (x:[]) = printHelper x

listHelper (x:xs) = printHelper x ++ ", " ++ listHelper xs

printHelper :: Value -> String

printHelper NoneVal = "None"

printHelper TrueVal = "True"

printHelper FalseVal = "False"

printHelper (IntVal c) = show c

printHelper (StringVal x1) = x1

printHelper (ListVal x) = "[" ++ listHelper x ++ "]"

printF :: [Value] -> String

printF [] = ""

printF (x:[]) = printHelper x

printF (x:xs) = printHelper x ++ " " ++ printF xs

isRight :: Either a b -> Bool

isRight (Right \_) = True

isRight (Left \_) = False

getRight :: Either a b -> b

getRight (Right v) = v

getRight (Left \_) = undefined

getLeft :: Either a b -> a

getLeft (Left s) = s

getLeft (Right \_) = undefined

rangeFunction :: Value -> Value -> Value -> Either String [Value]

rangeFunction (IntVal x1) (IntVal x2) (IntVal x3)

  | x3 == 0 = Left "error"

  | x1 <= x2 && x3 < 0 = Right []

  | x1 >= x2 && x3 > 0 = Right []

  | otherwise = Right ((IntVal x1):(getRight (rangeFunction (IntVal (x1+x3)) (IntVal x2) (IntVal x3))))

rangeFunction \_ \_ \_ = Left "error"

apply :: FName -> [Value] -> Comp Value

apply "range" [e1, e2, e3] =

  let tmp = rangeFunction e1 e2 e3 in

    if isRight tmp

      then wrapComp (ListVal (getRight tmp))

    else

      abort (EBadArg (getLeft tmp))

apply "range" [e1, e2] = apply "range" [e1, e2, IntVal 1]

apply "range" [e2] = apply "range" [IntVal 0, e2, IntVal 1]

apply "range" \_ = abort (EBadArg "error")

apply "print" x =

  do

  {

    output (printF x)

    ;return NoneVal

  }

apply a \_ = abort (EBadFun a)

wrapComp :: Value -> Comp Value

wrapComp x = withBinding "\_" x (look "\_")

extractList :: Value -> [Value]

extractList (ListVal x) = x

extractList \_ = []

-- Main functions of interpreter

eval :: Exp -> Comp Value

eval (Const x) = wrapComp x

eval (Var v) = look v

eval (Oper op e1 e2) =

  do

  {

    x1 <- eval e1

    ;x2 <- eval e2

    ;let tmp = operate op x1 x2 in

      if isRight tmp

        then wrapComp (getRight tmp)

      else

        abort (EBadArg (getLeft tmp))

  }

eval (Not e) =

  do

  {

    tmp <- eval e

    ;

    if truthy tmp

      then wrapComp FalseVal

    else

      wrapComp TrueVal

  }

eval (Call f []) = apply f []

eval (Call f e) =

  do

  {

    x <- eval (List e)

    ;apply f (extractList x)

  }

eval (List []) = wrapComp (ListVal [])

eval (List (e:es)) =

  do

  {

    x <- eval e

    ;xs <- eval (List es)

    ;wrapComp (ListVal (x:(extractList xs)))

  }

eval (Compr \_ []) = wrapComp NoneVal

eval (Compr \_ \_) = undefined

-- eval (Compr e [CCFor name ex]) = undefined

-- eval (Compr e [CCIf ex]) = undefined

-- eval (Compr e cc) = case cc of

-- eval (Compr e ((name ex):es)) = undefined

-- eval (Compr e (ex:es))

  -- do

  -- {

  --   x <- eval e

  --   ;xs <- eval (CClause es)

  --   -- ;wrapComp (ListVal (x:(extractList xs)))

  -- }

-- eval (Compr e cc) = case cc of

  -- [CCFor name ex]  ->

  --   do

  --     {

  --       x <- eval ex

  --       ;xs <- eval (CClause e)

  --       ;wrapComp (ListVal (x:(extractList xs)))

  --       }

  -- [CCIf ex] -> ex

  -- (cc':xs) ->  xs

-- eval (Compr e [CCFor name ex]) =

--   do

--   {

--     x <- eval ex

--     ;xs <- eval (List es)

--     ;wrapComp (ListVal (x:(extractList xs)))

--   }

exec :: Program -> Comp ()

exec [] = return mempty

exec ((SDef v e):xs) =

  do {

    x <- eval e

    ;withBinding v x (exec xs)

  }

exec ((SExp e):xs) =

  do {

    eval e

    ;exec xs

  }

execute :: Program -> ([String], Maybe RunError)

execute x =

  if isRight (fst (runComp (exec x) []))

    then (snd (runComp (exec x) []), Nothing)

  else (snd (runComp (exec x) []), Just (getLeft (fst (runComp (exec x) []))))

Test.hs

-- Skeleton test suite using Tasty.

-- Fell free to modify or replace anything in this file

import BoaAST

import BoaInterp

import Test.Tasty

import Test.Tasty.HUnit

main :: IO ()

main = defaultMain $ localOption (mkTimeout 1000000) tests

tests :: TestTree

tests = testGroup "Stubby tests"

  [testCase "crash test" $

    execute [SExp (Call "print" [Oper Plus (Const (IntVal 2))(Const (IntVal 2))]),

    SExp (Var "hello")]

      @?= (["4"], Just (EBadVar "hello")),

   testCase "crash test negative" $

    execute [SExp (Call "print" [Oper Plus (Const (IntVal (-3)))(Const (IntVal (-2)))]),

    SExp (Var "hello")]

      @?= (["-5"], Just (EBadVar "hello")),

   testCase "test 1" $

    execute [SDef "x" (Const (IntVal 1)), SExp (Call "print" [(Var "x")])]

      @?= (["1"], Nothing),

   testCase "test 2" $

    execute [SDef "x" (Not (Const (IntVal 1))), SExp (Call "print" [(Var "x")])]

      @?= (["False"], Nothing),

   testCase "test 3" $

    execute [SExp (Call "print" [Oper Plus (Const (IntVal 2))(Const (IntVal 2))]),

    SExp (Var "hello")]

      @?= (["4"], Just (EBadVar "hello")),

   testCase "test 4" $

    execute [SExp (Call "print" [Oper Div (Const (IntVal 4)) (Const (IntVal 2))]),

    SExp (Call "print" [Oper Div (Const (IntVal 2)) (Const (IntVal 0))])]

      @?= (["2"], Just (EBadArg "error")),

   testCase "test 5" $

    execute [SExp (Call "print" [Oper Mod (Const (IntVal 3)) (Const (IntVal 2))]),

    SExp (Call "print" [Oper Mod (Const (IntVal 2)) (Const (IntVal 0))])]

      @?= (["1"], Just (EBadArg "error")),

   testCase "test 6" $

    execute [SExp (Call "print" [Oper Mod (Const TrueVal) (Const (IntVal 1))])]

      @?= ([], Just (EBadArg "error")),

   testCase "test 7" $

    execute [SExp (Call "print" [Oper Minus (Const (IntVal 6)) (Const (IntVal 3))])]

      @?= (["3"], Nothing),

   testCase "test 8" $

    execute [SExp (Call "print" [Oper Minus (Const (IntVal 3)) (Const (IntVal 4))])]

      @?= (["-1"], Nothing),

   testCase "test 9" $

    execute [SExp (Call "print" [Oper Minus (Const (IntVal 3)) (Const (IntVal 3))])]

      @?= (["0"], Nothing),

   testCase "test 10" $

    execute [SExp (Call "print" [Oper Minus (Const (IntVal 3)) (Const (IntVal (-3)))])]

      @?= (["6"], Nothing),

   testCase "test 11" $

    execute [SExp (Call "print" [Oper Minus (Const (IntVal (-3))) (Const (IntVal (-3)))])]

      @?= (["0"], Nothing),

   testCase "test 12" $

    execute [SExp (Call "print" [Oper Times (Const (IntVal 2)) (Const (IntVal 3))])]

      @?= (["6"], Nothing),

   testCase "test 13" $

    execute [SExp (Call "print" [Oper Times (Const (IntVal (-2))) (Const (IntVal 3))])]

      @?= (["-6"], Nothing),

   testCase "test 14" $

    execute [SExp (Call "print" [Oper Times (Const (IntVal (-4))) (Const (IntVal (-3)))])]

      @?= (["12"], Nothing),

   testCase "test 15" $

    execute [SExp (Call "print" [Oper Times (Const (IntVal 3)) (Const TrueVal)])]

      @?= ([], Just (EBadArg "error")),

   testCase "test 16" $

    execute [SExp (Call "print" [Oper Eq (Not (Const (IntVal 1))) (Const FalseVal),

     Oper Eq (Const (IntVal 1)) (Const FalseVal)])]

      @?= (["True False"], Nothing),

   testCase "test 17" $

    execute [SExp (Call "print" [Oper Eq (Not (Const (IntVal 1))) (Const (IntVal (-1))),

     Oper Eq (Const (IntVal 1)) (Const (IntVal 1))])]

      @?= (["False True"], Nothing),

   testCase "test 18" $

    execute [SExp (Call "print" [Oper Greater (Const (IntVal 1)) (Const (IntVal 3)),

     Oper Less (Const (IntVal 1)) (Const (IntVal 3))]),

     SExp (Call "print" [ Oper Less (Const (IntVal 0)) (Const TrueVal)])]

      @?= (["False True"], Just (EBadArg "error")),

   testCase "test 19" $

    execute [SExp (Call "print" [Oper In (Const (IntVal 2)) (Const (ListVal [IntVal 1, IntVal 2]))])]

      @?= (["True"], Nothing),

   testCase "test 20" $

    execute [SExp (Call "print" [Oper In (Const (IntVal 5)) (Const (ListVal [IntVal (-1), IntVal 2]))])]

      @?= (["False"], Nothing),

   testCase "test 21" $

    execute [SExp (Call "print" [Oper In (Const (IntVal 5)) (Const FalseVal)])]

      @?= ([], Just (EBadArg "error")),

   testCase "test 22" $

    execute [SExp (Call "a" [(Const TrueVal)])]

      @?= ([], Just (EBadFun "a")),

   testCase "test 23" $

    execute [SDef "x1" (Call "range" [(Const (IntVal 3))]),

    SDef "x2" (Call "range" [(Const (IntVal 1)), (Const (IntVal 0))]),

    SDef "x3" (Call "range" [(Const (IntVal 5)), (Const (IntVal 1)), (Const (IntVal (-2)))]),

    SExp (Call "print" [Var "x1", Var "x2", Var "x3"])]

      @?= (["[0, 1, 2] [] [5, 3]"], Nothing),

   testCase "test 24" $

    execute [SDef "x1" (Call "range" [(Const TrueVal)])]

      @?= ([], Just (EBadArg "error")),

   testCase "test 25" $

    execute [SDef "x3" (Call "range" [(Const (IntVal 5)), (Const (IntVal 1)), (Const (IntVal 0))])]

      @?= ([], Just (EBadArg "error")),

   testCase "test 26" $

    execute [SDef "x" (List [Oper Plus (Const (IntVal 10)) (Const (IntVal 25)), Const FalseVal]),

    SExp (Call "print" [Var "x"])]

      @?= (["[35, False]"], Nothing),

   testCase "test 27" $

    execute [SDef "x" (List [Const TrueVal, Const FalseVal]),

    SExp (Call "print" [Var "x"])]

      @?= (["[True, False]"], Nothing),

   testCase "test 28" $

    execute [SDef "x" (List [Const FalseVal, Const FalseVal]),

    SExp (Call "print" [Var "x"])]

      @?= (["[False, False]"], Nothing),

   testCase "test 29" $

    execute [SDef "x" (List [Const FalseVal, Const FalseVal, Const (StringVal "?")]),

    SExp (Call "print" [Var "x"])]

      @?= (["[False, False, ?]"], Nothing)]