Advanced Programming - Fall 2019

Assignment 1

Till Grenzdörffer, Michal Pikulski

Handed in: September 18, 2019

Contents

Design considerations and encountered problems

This assignment was very well described and did not leave as much decisions for us to take as the previous one (like handling errors). It did, however, include some interesting places which we've spent a good amount of time implementing:

- Implementation of list comprehension - the way it is done right now is very explicit and perhaps a bit bloated. We could've delegated some parts of the code to external functions, however the concept was difficult enough to comprehend, let alone read the implementation. That's why we preferred to keep this way, for the sake of clarity.
- There are some places in code that, by design and implementation, should not be reachable, as the eval function should always return the expected type. However, in case our implementation was faulty and we would need to debug it, we decided to throw explicit errors there.
- We chose not to implement printing strings nested within lists with quotation marks. This was mentioned to be optional and we did not feel like the time spent on implementation, combined with the unreadability of escaping complicated sequences within the string, justified the time we would have to spend on it.
- We had some problems when implementing test cases. It's not exactly implementation related, but we've spent a good amount of time on it and we feel like our current solution might use some work. So, we're not sure how to best test for equality when we perform computations, as results are usually kept in a `Comp` monad. Right now we compare two do block (each of yields a monad) and that required us to define an equality operator for `Comp`. Our idea was to use the

applicative property, so something along the lines of : `(==) <*> (val1) <*> (val2).` Where the values are within a monad and the result would too.

Assessment of code quality

Quality assessment

We believe that the amount of tests we have prepared, at minimum ensures a well running code, that achieves its purpose without any bugs and runtime errors. Some areas we have tested more than others, but it was rather due to our lack of ideas for proper, non-redundant test-cases. We feel that some parts of the implementation could definitely use some work (we have mentioned which ones in the problems section), but overall we are pleased with the effort we have invested into this assignment and are confident in the resulting program.

Testing strategy

We have divided our testing strategy into four stages:

- Manual testing during development
- Online TA test cases
- Unit testing
- Property testing

Unit testing

For some of the simple functions, which were not that complicated and didn't seem to require extensive testing we have prepared a set of unit tests. This applies to functions like `truthy`, `apply`, `print`, `output` and some of equality operators. In some cases, we also tested the aforementioned functions for some properties, but we still included unit testing for certain edge cases. We have prepared a total of 27 such tests, divided into 3 test groups.

Property testing

Just as it was suggested in the lectures, we tried to derive a set of properties for each function, that we felt should always hold true. We later created appropriate test suites that covered those properties. In some cases we had even more ideas, but lacked the time to implement all of property tests for a given function. We'll mention those in this section, with a note that it's just an idea.

- Testing our implementation of arithmetic operators, for which we prepared arbitrary types for random generation.
    - Commutative property
    - Associative property
    - *We did consider testing distributive property, however function operate only accepted values and not other expressions*

    - *Another interesting property might be identity elements for each operator, but some of that is included in testing our range function so we didn't prepare separate test cases*
- Testing out implementation of range function
    - The length of a generated list should be equal to the number of elements between start and end, divided by step. (One exception is, when step is bigger than the number of elements, we then only get one element).
    - *We tried implementing reversing the lists, so if we called the range function with a given step and then called range again on the result, with negative step, we should get the original input list. Implementation, however, prevented us from doing that, as the original result is actually a ListVal.*
- List comprehensions - using a recursive, arbitrary type for automatic generation of our `Values`, including `ListVal`
    - If we apply an operation with its identity element, the evaluation of that comprehension should return the same list. We prepared that for `Plus` and `Times`.
    - *We tried to implement a property, where applying opposing operators with the same element (e.g. Plus 5, Minus 5) one after another should return the original input list. Unfortunately, when we called evaluation inside a do block, the second call to eval messed up our environment and we weren't able to troubleshoot that problem.*
- `In` operator
    - The result of calling this operator on a list and argument should always yield equal results to just calling `` `elem` ``.
- `Look` and `withBiding`
    - We tested those two functions together assuming, that whenever we called `withBinding`, a result of looking up a variable with `look` in the same environment, should yield our original value.

Code listing

BoaInterp.hs

```haskell
-- 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 (\_e -> (Right a, []))
  m >>= f  = Comp (\e -> let (firstA, firstSL) = (runComp m) e in
                            case firstA of
                              (Left err) -> ((Left err), firstSL) -- error occured in m, pass it on!
                              -- apply f to val if no error ocurred , then evaluate the returned a (secondA)
                              (Right val) -> let (secondA, secondSL) = runComp (f val) e in
                                (secondA, firstSL ++ secondSL))

-- 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 e = Comp (\_env -> (Left e, []))

look :: VName -> Comp Value
look name = Comp (\env -> case (lookup name env) of
                            (Just a) -> (Right a, [])
                            Nothing -> (Left (EBadVar name), []))

withBinding :: VName -> Value -> Comp a -> Comp a
withBinding name val m = Comp (\env -> (runComp m) ((name, val) : env))

output :: String -> Comp ()
```

```haskell
output s = Comp (\_env -> (Right (), [s]))

-- Helper functions for interpreter
truthy :: Value -> Bool
truthy NoneVal = False
truthy FalseVal = False
truthy (IntVal 0) = False
truthy (StringVal "") = False
truthy (ListVal []) = False
truthy _ = True

operate :: Op -> Value -> Value -> Either String Value
-- arithmetic ops
operate Plus (IntVal v1) (IntVal v2) = Right $ IntVal $ v1 + v2
operate Plus _ _ = Left "Plus operation only defined on two IntVals"
operate Minus (IntVal v1) (IntVal v2) = Right $ IntVal $ v1 - v2
operate Minus _ _ = Left "Minus operation only defined on two IntVals"
-- higher precedence
operate Times (IntVal v1) (IntVal v2) = Right $ IntVal $ v1 * v2
operate Times _ _ = Left "Times operation only defined on two IntVals"
operate Div (IntVal v1) (IntVal v2)
  | v2 /= 0 = Right $ IntVal $ v1 `div` v2
  | otherwise = Left "Attempted division by zero"
operate Div _ _ = Left "Div operation only defined on two IntVals"
operate Mod (IntVal v1) (IntVal v2)
  | v2 /= 0 = Right $ IntVal $ v1 `mod` v2
  | otherwise = Left "Attempted to execute x modulo zero"
operate Mod _ _ = Left "Mod operation only defined on two IntVals"
-- equality ops
operate Eq v1 v2 = Right $ if (v1 == v2) then TrueVal else FalseVal
operate Less (IntVal v1) (IntVal v2) = Right $ if (v1 < v2) then TrueVal else FalseVal
operate Less _ _ = Left "Less operation only defined on two Intvals"
operate Greater (IntVal v1) (IntVal v2) = Right $ if (v1 > v2) then TrueVal else FalseVal
operate Greater _ _ = Left "Greater operation only defined on two Intvals"
-- In op
operate In v1 (ListVal v2) = Right $ if v1 `elem` v2 then TrueVal else FalseVal
operate In _ _ = Left "In operator takes only Lists as second argument!"

apply :: FName -> [Value] -> Comp Value
apply "range" ((IntVal a):(IntVal b):(IntVal step):rest)
  | rest == [] = return (ListVal [(IntVal x) | x <- [a, (a+step)..(b-(signum step))]]) -- if step is negative,
last possible value is b+1
  | step == 0 = abort (EBadArg "range function called with zero step")
  | otherwise = abort (EBadArg "range function called with >3 arguments.")
apply "range" ((IntVal a):(IntVal b):rest)
  | rest == [] = return (ListVal [(IntVal x) | x <- [a, (a+1)..(b-1)]])
  | otherwise = abort (EBadArg "range called with non-integer arguments.")
apply "range" ((IntVal b):rest)
  | rest == [] = return (ListVal [(IntVal x) | x <- [0, 1..(b-1)]])
  | otherwise = abort (EBadArg "range called with non-integer arguments.")
apply "range" [] = abort (EBadArg "range called with zero arguments."  )
apply "range" _ = abort (EBadArg "range called with non-integer arguments.")
apply "print" x = do output (print' x False)
```

```
                     return NoneVal
apply fname _ = abort (EBadFun fname)

-- Bool indicates if value is in list
print' :: [Value] -> Bool -> String
print' [] _ = ""
print' (x:xs) il
  | xs == [] = (printVal x)
  | otherwise = (if il then (printVal x) ++ ", " else (printVal x) ++ " ") ++ (print' xs il)

printVal :: Value -> String
printVal NoneVal = "None"
printVal TrueVal = "True"
printVal FalseVal = "False"
printVal (IntVal x) = show x
printVal (StringVal x) = x
printVal (ListVal x) = "[" ++ (print' x True) ++ "]" -- this should not happen though

-- Main functions of interpreter
eval :: Exp -> Comp Value
eval (Const val) = return val
eval (Var vname) = look vname
eval (Oper op e1 e2) = do v1 <- eval e1
                          v2 <- eval e2
                          case (operate op v1 v2) of
                            (Left err) -> abort (EBadArg err)
                            (Right res) -> return res
eval (Not e) = do v <- eval e
                  if truthy v then return FalseVal else return TrueVal
eval (Call fname arge) = do argv <- eval (List arge)
                            case argv of
                              (ListVal l) -> apply fname l
                              _ -> error "this code should be unreachable." -- eval(List should always evaluate
to a listval)
eval (List []) = return (ListVal [])
eval (List (e1:es)) = do v1 <- eval e1
                         vs <- eval (List es)
                         case vs of
                          (ListVal l) -> return (ListVal (v1 : l))
                          _ -> error "this code should be unreachable." -- eval(List should always evaluate to
a listval)

eval (Compr e []) = do lv <- eval e
                       return (ListVal [lv])
eval (Compr _ ((QFor _ (Const (ListVal []))):_)) = return (ListVal [])
eval (Compr e ((QFor vname (Const (ListVal (l:ls)))):qs)) = do res1 <- withBinding vname l (eval (Compr e qs))
                                                               res2 <- eval (Compr e ((QFor vname (Const
(ListVal ls))):qs))
                                                               case res2 of
                                                                (ListVal rl2) -> case res1 of
                                                                                  (ListVal rl1) -> return
(ListVal (rl1 ++ rl2))
                                                                                  _ -> error "Compr did not
```

```
return a list. Impl error!" -- return (ListVal (res1 : rl2))
                                                                  _ -> error "Compr did not return a list. Impl
error!" -- QFor always returns a ListVal
-- evaluate list and call the eval with const listval, see above
eval (Compr e ((QFor vname fe):qs)) = do fv <- eval fe
                                         case fv of
                                           (ListVal l) -> eval (Compr e ((QFor vname (Const (ListVal l))):qs))
                                           _ -> abort (EBadArg "2nd. argument of Compr should be a list.")
eval (Compr e ((QIf ie):qs)) = do iv <- eval ie
                                  if (truthy iv) then eval (Compr e qs) else return (ListVal [])

exec :: Program -> Comp ()
exec [] = return ()
exec ((SDef vname e):sts) = do v <- eval e
                               withBinding vname v (exec sts)
exec ((SExp e):sts) =       do eval e
                               exec sts

execute :: Program -> ([String], Maybe RunError)
execute p = case err of
             (Left err) -> (res, Just err)
             (Right _) -> (res, Nothing)
  where (err, res) = runComp (exec p) []
```

Test.hs

```haskell
import BoaAST
import BoaInterp
import TestTypes

import Test.Tasty
import Test.Tasty.HUnit
import qualified Test.QuickCheck.Monadic as QCM
import qualified Test.Tasty.QuickCheck as QC
import Data.Either

-- Remove later
import Data.List
import Data.Ord
import System.Environment

-- To enable verbose options:
-- setEnv "TASTY_QUICKCHECK_VERBOSE" "TRUE"

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests

tests :: TestTree
tests = testGroup "All tests" [unitTst, propertyTst]

unitTst :: TestTree
unitTst = testGroup "UnitTests" [truthyTst, eqTst, outputTest]

-- Test truthy
truthyTst = testGroup "Testing truth values"
    [testCase "truthy Nothing"
        (assertBool "" (False == truthy(NoneVal))),
    testCase "truthy False "
        (assertBool "" (False == truthy(FalseVal))),
    testCase "truthy 9999 "
        (assertBool "" (True == truthy(IntVal 9999))),
    testCase "truthy 0 "
        (assertBool "" (False == truthy(IntVal 0))),
    testCase "truthy \"Looooooong string\" "
        (assertBool "" (True == truthy(StringVal "Looooooong string"))),
    testCase "truthy \"\""
        (assertBool "" (False == truthy(StringVal ""))),
    testCase "truthy []"
        (assertBool "" (False == truthy(ListVal []))),
    testCase "truthy [1, 2]"
        (assertBool "" (True == truthy(ListVal [IntVal 1, IntVal 2]))),
    testCase "truthy [False]"
        (assertBool "" (True == truthy(ListVal [FalseVal]))),
    testCase "truthy [True, False]]"
        (assertBool "" (True == truthy(ListVal [TrueVal, FalseVal])))]
```

```haskell
-- Teste edge cases foor equlality operators
eqTst = testGroup "Test comparison opeartors in operate"
    [testCase "True == True"
        (assertBool "" (TrueVal == (Data.Either.fromRight (FalseVal) (operate Eq TrueVal TrueVal)))),
    testCase "False == False"
        (assertBool "" (TrueVal == (Data.Either.fromRight (FalseVal) (operate Eq FalseVal FalseVal)))),
    testCase "True == False"
        (assertBool "" (FalseVal == (Data.Either.fromRight (TrueVal) (operate Eq TrueVal FalseVal)))),
    testCase "[] == False"
        (assertBool "" (FalseVal == (Data.Either.fromRight (TrueVal) (operate Eq (ListVal []) FalseVal)))),
    testCase "[] == []"
        (assertBool "" (TrueVal == (Data.Either.fromRight (FalseVal) (operate Eq (ListVal []) (ListVal
[]))))),
    testCase "-10 < 0 "
        (assertBool "" (TrueVal == (Data.Either.fromRight (FalseVal) (operate Less (IntVal (-10)) (IntVal
0))))),
    testCase "-100 < -10"
        (assertBool "" (TrueVal == (Data.Either.fromRight (FalseVal) (operate Less (IntVal (-100)) (IntVal
(-10)))))),
    testCase "-100 > 100"
        (assertBool "" (FalseVal == (Data.Either.fromRight (TrueVal) (operate Greater (IntVal (-100)) (IntVal
100))))),
    testCase "0 > 0"
        (assertBool "" (FalseVal == (Data.Either.fromRight (TrueVal) (operate Greater (IntVal 0) (IntVal
0)))))
    ]

-- Test apply -> print -> output
outputTest = testGroup "a"
    [testCase "output NoneVal"
        (assertBool "" (["None"] == snd (runComp (apply "print" [NoneVal]) []))),
    testCase "output TrueVal"
        (assertBool "" (["True"] == snd (runComp (apply "print" [TrueVal]) []))),
    testCase "output FalseVal"
        (assertBool "" (["False"] == snd (runComp (apply "print" [FalseVal]) []))),
    testCase "output int 0"
        (assertBool "" (["0"] == snd (runComp (apply "print" [IntVal 0]) []))),
    testCase "output int -100"
        (assertBool "" (["-100"] == snd (runComp (apply "print" [IntVal (-100)]) []))),
    testCase "output []"
        (assertBool "" (["[]"] == snd (runComp (apply "print" [ListVal []]) []))),
    testCase "output [1, 2, 3]"
        (assertBool "" (["[1, 2, 3]"] == snd (runComp (apply "print" [ListVal [IntVal 1, IntVal 2, IntVal 3]])
[]))),
    testCase "output [NoneVal]"
        (assertBool "" (["None"] == snd (runComp (apply "print" [NoneVal]) [])))
    ]

-- Proprety tests
propertyTst :: TestTree
propertyTst = testGroup "Property Tests" [prop_com,
                                          prop_ass,
                                          prop_apply_range,
```

```haskell
                                          prop_in_op,
                                          prop_look_withBinding,
                                          prop_list_compr]

prop_com = testGroup "Test commutative property"
    [ QC.testProperty "Commutative property of Plus, Mul and Eq" $
          \(CommOp o)  a b -> operate o (IntVal a) (IntVal b) == operate o (IntVal b) (IntVal a)]

prop_ass = testGroup "Test associative property"
    [ QC.testProperty "Associative property of Plus, Mul" $ testAssociative
    ]

prop_apply_range = testGroup "Test range function properties"
    [ QC.testProperty "Size of the generated list should be the same as (start-end)/step" $ testLength
    ]

prop_list_compr = testGroup "Test list conprehension properties"
    [ QC.testProperty "Identity comprehension with operator plus" $ testIdentityComprehensionPlus,
      QC.testProperty "Identity comprehension with operator minus" $ testIdentityComprehensionTimes
    ]

-- List returned list should always have the size equal to rouded (start-end)/step, except for a corner case
where we return a one element list - step > (end - start)
testLength start end (QC.NonZero step) = (do list <- apply "range" [IntVal start, IntVal end, (IntVal step)]
                                             case list of
                                                (ListVal listVal') -> return $ length listVal'
                                                _                   -> return (-1))
                                          ==
                                            if ((step > 0) && (start >= end)) || ((step < 0) && (start <=
end)) then createComp 0
                                                else (if (abs step > abs(end - start)) then createComp 1 else
createComp $  ceiling $ abs (fromIntegral (end - start)/ fromIntegral step))

-- Certain operators should have the associative property
testAssociative (AssOp o) a b c = (do ab <- (operate o (IntVal a) (IntVal b))
                                      abc <- (operate o ab (IntVal c))
                                      return abc)
                                  ==
                                    (do bc <- (operate o (IntVal b) (IntVal c))
                                        abc <- (operate o (IntVal a) bc)
                                        return abc)

-- Operate on operator In should always retun the same value as elem function from Haskell
prop_in_op = testGroup "Test In operator property"
                [ QC.testProperty "Test In operator property" $
                    (\a (LV list@(ListVal b)) -> let inVal = (operate In a list) in
                                                case inVal of
                                                    (Right v) -> truthy(v) == (a `elem` b)
                                                    _         -> False)
                ]

-- Binding a variable within an environment and then looking it up should provide the original value that was
bound to it
```

```haskell
prop_look_withBinding = testGroup "test look and withBinding"
                        [ QC.testProperty "test look and withBinding" $
                            (\vname val -> withBinding vname val (do a <- look vname
                                                                     return (a == val))


                                            ==
                                            (return True))
                        ]

-- Calling a comprehension with identity element of plus should result in the exact same list
testIdentityComprehensionPlus start end (QC.NonZero step) = do inputList <- eval (Call "range" [Const $ IntVal
start, Const $ IntVal end, Const $ IntVal step])
                                                               tmpList <- eval (Compr (Oper Plus (Const $
IntVal 0) (Var "x")) [QFor "x" (Const inputList)])
                                                               return (inputList == tmpList)
                                                            ==
                                                               (return True)

-- Calling a comprehension with identity element of times should result in the exact same list
testIdentityComprehensionTimes start end (QC.NonZero step) = do inputList <- eval (Call "range" [Const $
IntVal start, Const $ IntVal end, Const $ IntVal step])
                                                                tmpList <- eval (Compr (Oper Times (Const $
IntVal 1) (Var "x")) [QFor "x" (Const inputList)])
                                                                return (inputList == tmpList)
                                                             ==
                                                                (return True)
```

TestTypes.hs

```haskell
module TestTypes where
import qualified Test.Tasty.QuickCheck as QC
import BoaAST
import BoaInterp
import Control.Monad

-- Arbitrary data type for commutative operators
newtype CommOperators = CommOp Op
    deriving (Eq, Show)
instance QC.Arbitrary CommOperators where
 arbitrary = fmap CommOp (QC.elements [Plus, Times, Eq])

-- Arbitrary data type for associative operators
newtype AssOperators = AssOp Op
 deriving (Eq, Show)
instance QC.Arbitrary AssOperators where
 arbitrary = fmap AssOp (QC.elements [Plus, Times])

-- Generators for value data type
newtype ListValue = LV Value deriving (Eq, Show)
instance QC.Arbitrary ListValue where
    arbitrary = QC.sized listVal

instance QC.Arbitrary Value where
    arbitrary = sizedVal

sizedVal = QC.sized valN
-- valN :: Int -> Gen a
valN 0 =  QC.oneof $ [ return NoneVal,
                return TrueVal,
                return FalseVal,
                liftM IntVal QC.arbitrary,
                liftM StringVal QC.arbitrary,
                return (ListVal [])
            ]
valN n = QC.oneof [return NoneVal,
                    return TrueVal,
                    return FalseVal,
                    liftM IntVal QC.arbitrary,
                    liftM StringVal QC.arbitrary,
                    do res <- subVal
                       return (ListVal [res])
                   ]
                where subVal = (valN (n `div` 2))
listVal n = do res <- subVal
               return $ LV (ListVal [res])
            where subVal = (valN (n `div` 2))

-- Helper functions for Moand coparison
createComp :: a -> Comp a
createComp a = Comp (\_e -> (Right a, []))
```

```haskell
instance (Eq a) => Eq (Comp a) where
    (Comp a) == (Comp b) = ((a []) == (b []))
```