Advanced Programming - Assignment 2

KUnet-ID: vsf505

September 2020

1 Design and Implementation

The instantiation of the monad and the implementation of the basic functions ('abort','look','withBinding','output' and 'truthy') is pretty much straight-forward.
The 'operate' function works a lot with pattern matching in order to recognize wrong input data types (mostly not IntVal) and returns an error message if this is the case. Also, division by zero in Div and Mod lead to an error.
The 'apply' function is implemented using the conditional pattern matching. It returns an error if range gets 0 or >3 parameters or if the called function is unknown (aka not range or print). The range function just uses the list comprehension feature of Haskell.
The print function uses an auxiliary function 'toStr' that returns the String representation of a single value and applies this helper function to the whole list of parameters.
The evaluation of expressions is again very straight-forward except of list comprehensions:

```
...
eval (Compr e xs) = do
  ys <- evalCompr e xs
  return (ListVal ys)

evalCompr :: Exp -> [CClause] -> Comp [Value]
    evalCompr e [] = sequence [eval e]
    evalCompr e ((CCIf f):xs) = do
      x <- eval f
      if truthy x
        then evalCompr e xs
        else return []
    evalCompr e ((CCFor v f):xs) = do
      x <- eval f
      case x of
        (ListVal ys) -> do
```

```
17          zs <- sequence [withBinding v y (evalCompr e xs
              )| y <- ys]
18          return (concat zs)
19        _ -> sequence [abort (EBadArg "CCFor needs a
             ListVal as parameter")]
```

The evaluation of list comprehensions makes use of the helper function 'evalCompr', which computes all the values that are being generated by the comprehension and puts them in a 'Comp [Value]'. The reason for using the Comp Monad here is the recursive call of evalCompr in line 17. By using this structure, the code for generating lists is quite simple and short and falsified CCIf-clauses can just return an empty list.
The 'exec' and 'execute' functions are again pretty straight-forward.

2 Quality assessment

When it comes to the assessment of the quality of the code, it is a little harder to show the correctness than in Assignment 0.
In my automated tests, i used Tasty and QuickCheck to check some properties and do a lot of unit tests of the code.
The unit tests are first of all checking all different types of Errors. Especially, the EBadArg error can be returned in quite a lot of cases and therefore is being checked for all of its possible occurrences. The next unit tests are the misc.ast that was already given as an example and a lot of tests regarding list comprehensions in the file testListCompr.ast:
I tried to test the basic functionality of list comprehensions, running a list comprehension with more than one CCFor clause and checking whether using the same variable inside a list comprehension overwrites the value of the global variable. I also tested the behavior of the list comprehension when there are two CCFor clauses with the same variable name "x" and what happens if the CCFor gets an empty list as parameter. In addition to these tests on list comprehensions, i also did some testing of the range-function. I tested the range function with 1, 2 and 3 parameters, negative steps and whether range(a,b) produces an empty list when $a > b$. I also tried to write the abstract code in a better readable format (aka .boa) but i am not sure whether it is in the exact format that our boa-language has.
The last unit-test (aka additionalTests.ast) tests the List-expression and a related test using the In operand.
In order to run QuickCheck (property) tests, i first had to instantiate Arbitrary and since I thought that i might need more than one instantiation for the different tests, i ended up creating newtypes of Exp (ArithExp), Oper (ArithOper) and Value (ArithValue). Instantiating these is important to test the commutativity and associativity of Plus and Times.
propCommutativityAdd then tests whether $a + b = b + a$ (similarly with Mul).
propAssociativityAdd tests whether $(a+b)+c = a+(b+c)$ (similarly with Mul).
In addition to that, there are two more property tests that test the functionality

of the order operators: The first test tests whether $(a < b) \oplus ((a = b) \vee (a > b))$ ($\oplus$ is the logical xor, if a is smaller than b, then it mustn't be equal or greater than b at the same time). The other test checks whether $a < b$ implies $b > a$: $(a < b) \Rightarrow (b > a)$ (in the code, i made use of $x \Rightarrow y \equiv \neg x \vee y$).
As you might have noticed, the creation of a newtype for Exp etc. was not necessary in the end, since every property test worked on arithmetical Expressions. But i see it as a good practice, there will be for sure some time where i need to different instantiations of a data type in order to test different properties.
The code passes all of my tests and all test cases in the OnlineTA. That is why i am convinced that the code has a good quality and that the code is ready to be used as a base for next weeks assignment.

3 Code

3.1 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]) }

--basic instantiation of Monad, append the 'output' at the end of the list and
--return an error if one of the computations returns an error
instance Monad Comp where
  return a = Comp $ \_ -> (Right a,mempty)
```

```haskell
  (>>=) m f = Comp $ \e -> case runComp m e of
    (Left err,xs) -> (Left err, xs)
    (Right v,xs) -> case runComp (f v) e of
      (Left err,ys) -> (Left err, xs `mappend` ys)
      (Right n, ys) -> (Right n, xs `mappend` ys)

-- You shouldn't need to modify these
instance Functor Comp where
  fmap = liftM
instance Applicative Comp where
  pure = return; (<*>) = ap

-- Operations of the monad

--just return an error
abort :: RunError -> Comp a
abort err = Comp $ \_ -> (Left err, mempty)

--if n could not be found, return error
look :: VName -> Comp Value
look n = Comp $ \e -> case lookup n e of
  Nothing -> (Left (EBadVar n), mempty)
  (Just v) -> (Right v, mempty)

--prepend binding to environment because 'lookup' returns first occurence in list
withBinding :: VName -> Value -> Comp a -> Comp a
withBinding n v m = Comp $ \e -> runComp m ((n,v):e)

--only add string to string array
output :: String -> Comp ()
output s = Comp $ \_ -> (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

--return error if the wrong types are used for any operation or division by zero
--is attempted.
operate :: Op -> Value -> Value -> Either String Value
```

```haskell
operate Plus (IntVal a) (IntVal b) = Right (IntVal (a+b))
operate Plus _ _ = Left "Plus only works with values of type IntVal"
operate Minus (IntVal a) (IntVal b) = Right (IntVal (a-b))
operate Minus _ _ = Left "Minus only works with values of type IntVal"
operate Times (IntVal a) (IntVal b) = Right (IntVal (a*b))
operate Times _ _ = Left "Times only works with values of type IntVal"
operate Div (IntVal a) (IntVal b) = if b == 0
  then Left "Division by zero"
  else Right (IntVal (a `div` b))
operate Div _ _ = Left "Div only works with values of type IntVal"
operate Mod (IntVal a) (IntVal b) = if b == 0
  then Left "Division by zero"
  else Right (IntVal (a `mod` b))
operate Mod _ _ = Left "Mod only works with values of type IntVal"
operate Eq a b = if a == b
  then Right TrueVal
  else Right FalseVal
operate Less (IntVal a) (IntVal b) = if a < b
  then Right TrueVal
  else Right FalseVal
operate Less _ _ = Left "Less only works with values of type IntVal"
operate Greater (IntVal a) (IntVal b) = if a > b
  then Right TrueVal
  else Right FalseVal
operate Greater _ _ = Left "Greater only works with values of type IntVal"
operate In v (ListVal xs) = if v `elem` xs
  then Right TrueVal
  else Right FalseVal
operate In _ _ = Left "The second operand of In has to ba a ListVal"

--return error if range gets 0 or >3 Values in the array, one of the parameters
--of range is not an IntVal or an unknown function is called.
apply :: FName -> [Value] -> Comp Value
apply f xs
```

```
  | f == "range" && (length xs == 0) = abort (EBadArg "range needs at least one parameter")
  | f == "range" && (length xs > 3) = abort (EBadArg "range takes at most three parameters")
  | f == "range" && (length xs == 1) = case xs !! 0 of
    (IntVal i) -> return (ListVal [IntVal n | n <- [0..i-1]])
    _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
  | f == "range" && (length xs == 2) = case xs !! 0 of
    (IntVal a) -> case xs !! 1 of
      (IntVal b) -> return (ListVal [IntVal n | n <- [a..b-1]])
      _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
    _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
  | f == "range" && (length xs == 3) = case xs !! 0 of
    (IntVal a) -> case xs !! 1 of
      (IntVal b) -> case xs !! 2 of
        (IntVal c) -> if c == 0
          then abort (EBadArg "step must not be 0")
          else if c > 0
            then return (ListVal [IntVal n | n <- [a..b-1],(n-a) `mod` c == 0])
            else return $ ListVal $ reverse [IntVal n | n <- [b+1..a],(a-n) `mod` (abs c) == 0]
        _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
      _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
    _ -> abort (EBadArg "range only takes values of type IntVal as parameter")
  --use auxiliary function to pretty print the single values and cut the last ' '
  | f == "print" = let str = concatMap (\x -> toStr x ++ " ") xs in output (take (length str - 1) str) >> (return NoneVal)
  | otherwise = abort (EBadFun f)
    where
      --auxiliary function to pretty print values
      toStr :: Value -> String
      toStr NoneVal = "None"
      toStr TrueVal = "True"
      toStr FalseVal = "False"
      toStr (IntVal i) = show i
```

```
      toStr (StringVal s) = s
      toStr (ListVal xs) = let str = concatMap (\x-> toStr x ++ ", ") xs in "[" ++ (take (length str - 2) str) ++ "]"


-- Main functions of interpreter
eval :: Exp -> Comp Value
eval (Const v) = return v
eval (Var v) = look v
eval (Oper op a b) = do
  aR <- eval a
  bR <- eval b
  case operate op aR bR of
    (Left s) -> abort (EBadArg s)
    (Right v) -> return v
eval (Not e) = do
  eR <- eval e
  if truthy eR
    then return FalseVal
    else return TrueVal
eval (Call f xs) = do
  ys <- mapM eval xs
  apply f ys
eval (List xs) = do
  ys <- mapM eval xs
  return (ListVal ys)
eval (Compr e xs) = do
  ys <- evalCompr e xs
  return $ ListVal ys

  --auxiliary function that evaluates all expressions of the List comprehension
  --uses Comp [Value] to be able to return "nothing" when CCIf fails, uses
  --Comp [Value] to be able to do the recursive call with withBinding
  where
    evalCompr :: Exp -> [CClause] -> Comp [Value]
    evalCompr e [] = sequence [eval e]
    evalCompr e ((CCIf f):xs) = do
      x <- eval f
      if truthy x
        then evalCompr e xs
        else return []
    evalCompr e ((CCFor v f):xs) = do
```

```haskell
        x <- eval f
        case x of
          (ListVal ys) -> do
            zs <- sequence [withBinding v y (evalCompr e xs)| y <- ys]
            return $ concat zs
          _ -> sequence [abort $ EBadArg "CCFor needs a ListVal as parameter"]

--executes program but returns Comp
exec :: Program -> Comp ()
exec [] = return ()
exec ((SExp e):xs) = (eval e) >> (exec xs)
exec ((SDef v e):xs) = do
  x <- eval e
  withBinding v x (exec xs)

--executes program and explicitly return output list and RunError if there was one
execute :: Program -> ([String], Maybe RunError)
execute p = let (a,b) = runComp (exec p) [] in (b,case a of
  Left e -> Just e
  Right _ -> Nothing)
```

3.2 Test.hs

```haskell
import BoaAST
import BoaInterp

import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck

--I first thought of making tests with other instantiations of Arbitrary as well and therefore created these newtypes.
--After some testing, i came to the conclusion that property testing is pretty difficult for other expressions and did only unit tests for these cases.

newtype ArithExp = ArithExp Exp
  deriving (Show, Eq)
newtype ArithOper = ArithOper Op
  deriving (Show, Eq)
newtype ArithValue = ArithValue Value
```

```
  deriving (Show, Eq)

instance Arbitrary ArithValue where
  arbitrary = do
    i <- arbitrary
    return $ ArithValue $ IntVal i

instance Arbitrary ArithOper where
  arbitrary = oneof $ map return $ map ArithOper [Plus, Minus, Times, Div, Mod]

instance Arbitrary ArithExp where
  arbitrary = sized arb
    where
      arb 0 = do
        (ArithValue i) <- arbitrary
        return $ ArithExp $ Const i
      arb n = do
        (ArithOper op) <- arbitrary
        (ArithExp e) <- arb $ n`div`2
        (ArithExp f) <- arb $ n`div`2
        return $ ArithExp $ Oper op e f

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

--tests commutativity of Add
propCommutativityAdd :: ArithExp -> ArithExp -> Bool
propCommutativityAdd (ArithExp e) (ArithExp f) = execute [SExp (Call "print" [Oper Plus e f])] == execute [SExp (Call "print" [Oper Plus f e])]

--tests commutativity of Mul
propCommutativityMul :: ArithExp -> ArithExp -> Bool
propCommutativityMul (ArithExp e) (ArithExp f) = execute [SExp (Call "print" [Oper Times e f])] == execute [SExp (Call "print" [Oper Times f e])]

--tests associativity of Add
propAssociativityAdd :: ArithExp -> ArithExp -> ArithExp -> Bool
propAssociativityAdd (ArithExp e) (ArithExp f) (ArithExp g) =
  execute [SExp (Call "print" [Oper Plus (Oper Plus e f) g])] == execute [SExp (Call "print" [Oper Plus e (
```

```
Oper Plus f g)])]

--tests associativity of Mul
propAssociativityMul :: ArithExp -> ArithExp -> ArithExp -> Bool
propAssociativityMul (ArithExp e) (ArithExp f) (ArithExp g) =
  execute [SExp (Call "print" [Oper Times (Oper Times e f) g])] == execute [SExp (Call "print" [Oper Times e (Oper Times f g)])]

--tests whether only one of the following statements holds: e < f , (e = f || e > f)
propIfLessThenNotGreaterOrEqual :: ArithExp -> ArithExp -> Bool
propIfLessThenNotGreaterOrEqual (ArithExp e) (ArithExp f) =
  (fst (execute [SExp $ Call "print" [Oper Less e f]]) == []) ||
    ((fst (execute [SExp $ Call "print" [Oper Less e f]]) == ["True"])
    /= (fst (execute [SExp $ Call "print" [Oper Eq e f]]) == ["True"])
      || (fst (execute [SExp $ Call "print" [Oper Greater e f]]) == ["True"]))

--tests whether f > e, if e < f
propIfLessThenGreaterReversed :: ArithExp -> ArithExp -> Bool
propIfLessThenGreaterReversed (ArithExp e) (ArithExp f) =
  (fst (execute [SExp $ Call "print" [Oper Less e f]]) == []) ||
    (not(fst (execute [SExp $ Call "print" [Oper Less e f]]) == ["True"])
    || (fst (execute [SExp $ Call "print" [Oper Greater f e]]) == ["True"]))

tests :: TestTree
tests = testGroup "Tests:" [unitTests, propertyTests]
  where
    unitTests = testGroup "Unit Tests:" [errorTests, fileTests]
    --tests all possible errors
    errorTests = testGroup "Error tests:" [
      testCase "test EBadVar" $
        execute [SExp (Call "print" [Oper Plus (Const (
```

```
IntVal 2)) (Const (IntVal 2))]), SExp (Var "hello")] @?= (["4"], Just (EBadVar "hello")),
    testCase "test EBadFun" $
      execute [SExp (Call "bla" [])] @?= ([],Just $ EBadFun "bla"),
    testCase "CCFor clause without ListVal as parameter" $
      execute [SExp $ Compr (Var "x") [CCFor "x" (Const NoneVal)]] @?= ([],Just $ EBadArg "CCFor needs a ListVal as parameter"),
    testCase "step = 0 when calling range" $
      execute [SExp $ Call "range" [Const $ IntVal 1, Const $ IntVal 10, Const $ IntVal 0]] @?= ([], Just $ EBadArg "step must not be 0"),
    testCase "range with non-IntVal" $
      execute [SExp $ Call "range" [Const $ StringVal "a"]] @?= ([],Just $ EBadArg "range only takes values of type IntVal as parameter"),
    testCase "division by zero" $
      execute [SExp $ Oper Div (Const (IntVal 1)) (Const (IntVal 0))] @?= ([],Just $ EBadArg "Division by zero"),
    testCase "not a ListVal as second parameter of 'Oper In'" $
      execute [SExp $ Oper In (Const NoneVal) (Const (StringVal "s"))] @?= ([],Just $ EBadArg "The second operand of In has to ba a ListVal")]
  --tests the .ast files in the example folder. For information about the specific test cases, please refer to the report.
  fileTests = testGroup ".ast-files Tests:" [
    testCase "List comprehension tests from testListCompr.ast" $ do
      pgm <- read <$> readFile "examples/testListCompr.ast"
      out <- readFile "examples/testListCompr.out"
      execute pgm @?= (lines out,Just (EBadArg "Less only works with values of type IntVal")),
    testCase "misc.ast from handout" $ do
      pgm <- read <$> readFile "examples/misc.ast"
      out <- readFile "examples/misc.out"
      execute pgm @?= (lines out, Nothing),
    testCase "Additional tests with 'List [Exp]' and 'Oper In Exp Exp'" $ do
      pgm <- read <$> readFile "examples/additionalTests.ast"
```

```
      out <- readFile "examples/additionalTests.out"
      execute pgm @?= (lines out,Nothing)]
  --tests basic properties of arithmetic operands
  propertyTests = testGroup "Property Tests:" [
    testProperty "propCommutativityAdd" propCommutativityAdd,
    testProperty "propCommutativityMul" propCommutativityMul,
    testProperty "propAssociativityAdd" propAssociativityAdd,
    testProperty "propAssociativityMul" propAssociativityMul,
    testProperty "(e < f) ^ (e == f || e > f) ?" propIfLessThenNotGreaterOrEqual,
    testProperty "(e<f) ==> (f>e) ?" propIfLessThenGreaterReversed]
```

3.3 testListCompr.ast

```
[SDef "squares" (Compr (Oper Times (Var "x") (Var "x")) [CCFor "x" (Call "range" [Const (IntVal 10)])]),
 SExp (Call "print" [Var "squares"]),
 SDef "x" (Const (IntVal 1)),
 SDef "overXandY" (Compr (Call "print" [Var "x", Var "y"]) [CCFor "x" (Call "range" [Const (IntVal 2),Const (IntVal 5)]), CCFor "y" (Const (ListVal [IntVal 2, StringVal "a",TrueVal]))]),
 SExp (Call "print" [Var "x"]),
 SDef "multipleX" (Compr (Var "x") [CCFor "x" (Const (ListVal [IntVal 1, IntVal 2, IntVal 3])),CCIf (Not(Oper Eq (Var "x") (Const (IntVal 2)))),CCFor "x" (Const (ListVal [IntVal 2]))]),
 SExp (Call "print" [Var "multipleX"]),
 SDef "emptyList" (Compr (Const (IntVal 1)) [CCFor "x" (Const (ListVal []))]),
 SExp (Call "print" [Var "emptyList"]),
 SDef "rangeTest" (Compr (Oper Plus (Var "x") (Var "y")) [CCFor "x" (Call "range" [Const (IntVal 0), Const (IntVal 11), Const (IntVal 5)]),CCFor "y" (Call "range" [Const (IntVal 5)])]),
 SDef "rangeTest2" (Call "range" [Const (IntVal (-20)), Const (IntVal (-31)),Const (IntVal (-2))]),
 SDef "rangeTest3" (Call "range" [Const (IntVal 2), Const (IntVal 1)]),
 SExp (Call "print" [Var "rangeTest", Var "rangeTest2", Var "rangeTest3"]),
```

```
SDef "errorAfter4" (Compr (Call "print" [Oper Less (Var "x") (Const (IntVal 3))]) [CCFor "x" (Const (ListVal [IntVal 1, IntVal 2, IntVal 3, IntVal 4, StringVal "oops"]))])]
```

3.4 additionalTests.ast

```
[
SDef "list" (List [Call "print" [Const (StringVal "first"),Const TrueVal], Compr (Oper Minus (Var "x") (Const (IntVal 1))) [CCFor "x" (Call "range" [Const (IntVal 11)])], Const (StringVal "second")]),
SExp (Call "print" [Var "list",List []]),
SExp (Call "print" [Oper In (Call "print" [Const (StringVal "third")]) (Var "list")])
]
```