Functional Programming 795 Assignment 2

Simeon Boshoff, 22546510, 22546510@sun.ac.za

October 19, 2022

1 Extending the data type *Prog* and the *Virtual Machine* to additionally support *Doubles*, *Int* lists, and *Double* lists

Extending the existing implementation of the monadic compiler to include doubles, integer lists and double lists, required the use of creating a new data type, as well as modifying all procedures which previously made use exclusively of integers. I created a new data type named "Generic" (A), which included the definition of the above-mentioned attributes.

```
data Generic = Int Int | Double Double | Ints [Int] | Doubles [Double]
             | deriving (Show, Eq)
```

(A) Generic data type

As we can see, a Generic type can either be an int, double, int list, or double list. This lead to modifying the "Expr" type as the value of an expression was now of type Generic, instead of being type integer. The modifications to the virtual machine (B), included changing the types used in *Stack*, *Mem*, and *Inst*.

```
type Stack = [Either Generic String]

type Mem   = [(Name,Either Generic String)]

type Code  = [Inst]

data Inst  = PUSH (Either Generic String)
           | PUSHV Name
           | POP Name
           | DO Op
           | JUMP Label
           | JUMPZ Label
           | LABEL Label
           | deriving Show

type Label = Int
```

(B) *Virtual Machine* modifications

When switching from a fundamental datatype such as *Int* to a generic type, the biggest difference is the application of operators to expressions. Integer addition, subtraction, multiplication as well as division is already built into Haskell, however we now have to redefine what those operations mean to our new generic datatype. This involves specifying what happens when we attempt to apply an operation to each of our four different types included in "Generic". Initially, the *execOp* function used to take as input an operator (*Add*, *Sub*, *Mul*, *Div*), two integers, and return a single integer as output. It physically applied the built in operations for integers, such as (+, -, *, and /). This was replaced by my function calls, seen in (C), where instead of applying an operator directly, I have written functions which can handle any of the types included in "Generic". I have also modified *fac* to take a generic as input.

```haskell
execOp :: Op -> Either Generic String -> Either Generic String -> Either Generic String
execOp _ (Right x) (Left y) = (Right "Error. Attempting to divide by zero.")
execOp _ (Left x) (Right y) = (Right "Error. Attempting to divide by zero.")

execOp Add x y = genericAdd x y
execOp Sub x y = genericSub x y
execOp Mul x y = generticMul x y

execOp Div x (Left (Double 0)) = (Right "Error. Attempting to divide by zero.")
execOp Div x (Left (Int 0)) = (Right "Error. Attempting to divide by zero.")
execOp Div x y = genericDiv x y
```

(C) execOp implementation

In order to simplify my work, I will rather describe my functions than put a large screenshot of the code, as I have decided to account for additional operations such as adding an integer and a double together, as well as adding an integer value to a list of doubles. I will include the code in an appendix if needed. (D) shows the definition of such a function, as it takes as input two generic variables, and provides as output a single generic variable. As can be observed, the cases for when the two generic types are both integers, and both doubles are shown. As we know that if this case is satisfied, we are working with fundamental types in Haskell, we can apply the base operator accordingly.

```haskell
genericAdd :: Either Generic String -> Either Generic String -> Either Generic String
genericAdd (Left(Int x)) (Left(Int y)) = (Left(Int (x + y)))
genericAdd (Left(Double x)) (Left(Double y)) = (Left(Double (x + y)))
```

(D) genericAdd snippet

How different types work in addition (genericAdd)

- Int + Int = Int
- Double + Double = Double
- Int + Double = Double (converts the int to a double using *fromIntegral*) (works in reverse)
- Int + [Int] = [Int] (adding the int value to all elements of the list using *map*)
- [Int] + Int = [Int] ++ Int (adding the int to the list as a new element)
- [Int] + [Int] = [Int] (adds the values of one list to the respective elements of the other list using *zipWith*)
- [Double] + [Double] = [Double] (same as above)

All of the other functions *genericSub*, *genericMul*, and *genericDiv* work more or less the same. The only case which I did not implement is adding a double to an integer list, because this would require converting all elements of that list to doubles.

In the *exec'* function, the variable h is checked to be zero, and is initially expected to be an integer. This required the creation of another function which returns a boolean for each of the cases where h is an integer or a double. This is then correctly implemented in *exec'*.

```haskell
genericIsZero :: Either Generic String -> Bool
genericIsZero (Left(Int h)) = h == 0
genericIsZero (Left(Double h)) = h == 0
```

(E) genericIsZero

The last part of changing the type from integer to generic was that in each function when we use a hard-coded value (such as integer 1), then we need to declare it as "(Int 1)" so that the type is known. This had to be implemented for all static values used in the functions such as *fac*.

2 Functionality to allow for division by zero to fail gracefully

In order to code the implementation for allowing division by zero operations to fail gracefully, a fundamental change had to occur in how variables were used. A wrapper was added to allow for the program to work with "Either Generic String", which means that any variable used was either of type generic or type string. In this case, *Left* meant that the variable was of type generic, and *Right* meant that the variable was of type string, implying that it was an error message.

When a division by zero occurred, it would be caught in the *execOp* function shown in (C). When the *DIV* operator was called with two values, and the second value (either int or double) was a zero, then the function would pattern match it, and return the *Right*-string "Error. Attempting to divide by zero.".

```
// execOp function snippet
execOp _ (Right x) (Left y) = (Right "Error. Attempting to divide by zero.")
execOp _ (Left x) (Right y) = (Right "Error. Attempting to divide by zero.")
execOp Div x (Left (Double 0)) = (Right "Error. Attempting to divide by zero.")
execOp Div x (Left (Int 0)) = (Right "Error. Attempting to divide by zero.")
```

When we encounter the case where we attempt to execute an operation where one of the variables is already a "Right"-string, we return the same message. This could be changed to not be hard-coded, and the variable "x" could be what is returned, as it is the specific error message.

The rest of the implementation changes can be seen in the screenshots (B), (D) and (E). In every function call where a variable was used, we now have to denote it as the appropriate side of the *Either*, such as "(Left (Int 5))".

This allows the program to fail gracefully, as any computations that involve divide by zero, will result in a string containing an error message, instead of a generic value. Hench no crashes occur, but the user does not receive the wanted output.

Operation executed: execOp Div (Left (Int 5)) (Left (Int 0))
Output received: Right "Error. Attempting to divide by zero."

3 Implementing Newton Raphson for square roots and the Collatz sequence

My implementations for finding the root of a number through using the Newton Raphson method, as well as creating the Collatz sequence for a given integer, builds on the same structure as the given *fac* method. These functions all take a single generic variable as input, and returns a *prog* variable that the Virtual Machine can execute. These new methods have also been tested by using Hspec, in order to validate their results.

3.1 Newton Raphson

The Newton Raphson method for finding the square root of a number, is conducted via the following steps:

1. Initialize the *root* variable and N as the input variable provided.
2. Calculate the root by the following function: $root = 0.5 * (root + N / root)$.
3. Repeat 2. until some stopping condition is met.

For my implementation, as shown in (F), I required the number of iterations of the function to be provided as input. This variable is then decremented by 1 for every loop in the *While*, and ensures that the user can predetermine the accuracy of the value returned. This acts as the stopping condition, when *"S"* is zero. The correct way of doing this is to create a threshold variable that can be specified by

the user, however this required the absolute value of the difference between the value and the previous value to be calculated, which was deemed unnecessarily difficult to program.

```
root :: Generic -> Generic -> Prog
root x n = Seqn [Assign "R" (Val x),
                 Assign "N" (Val x),
                 Assign "S" (Val n),
                 While (Var "S") (Seqn
                   [
                      Assign "R" (App Mul (Val (Double 0.5)) (App Add (Var "R") (App Div (Var "N") (Var "R")))),
                      Assign "S" (App Sub (Var "S") (Val (Int 1)))
                   ])]
```

(F) Implementation of Newton Raphson to find square root.

Operation executed: exec(comp(root (Int 15) (Int 100)))

Output received: [("R",Left (Double 3.872983346207417)),("N",Left (Int 15)),("S",Left (Int 0))]

As we can see, we obtained the square root of 15 after 100 iterations to be 3.87, in the "R" variable.

3.2 Collatz sequence

To find the Collatz sequence of an integer input, the following pseudo-code was used:

1. Create an empty list, and initialize the stopping condition variables.
2. Add the input integer to the list as the first element.
3. Assign the input integer to variable A.
4. If A is even: calculate new value $A = A/2$, and add it to the list.
5. If A is odd: calculate new value $A = A * 3 + 1$, and add it to the list.
6. Repeat from 4. until $A = 1$.

For my implementation, I calculate if a value is even or odd, by using integer division. I divide the value by 2, and the multiply it by two. For example, if we divide 5 by 2, we obtain the integer value 2. If we multiply this by 2, we obtain 4. If we subtract 4 from the original value, we get 1, which allows for the *odd If*-statement to execute. Otherwise we get 0, and the *Else* part executes. The addition operator allows for adding an integer to an integer list with the correct syntax, highlighted in (G). To calculate the stopping criteria, we subtract 1 from A, and if A is 1, then it will result in 0, stopping the *While*-loop.

Operation executed: exec(comp(collatz (Int 11)))

Output received: [("A",Left (Int 1)),("list",Left (Ints [11,34,17,52,26,13,40,20,10,5,16,8,4,2,1])),("isRunning",Left (Int 0)),("rem",Left (Int 2)),("compare",Left (Int 0))]

As we can see, the "list" variable contains the correct Collatz sequence for the number 11.

```haskell
collatz :: Generic -> Prog
collatz n = Seqn [Assign "A" (Val n),
                  Assign "list" (Val (Ints [])),
                  Assign "isRunning" (Val (Int 1)),
                  Assign "list" (App Add (Var "list") (Var "A")),
                  While (Var "isRunning") (Seqn
                    [
                      -- Divide the number by 2 and multiply it by 2, to find even or odd.
                      Assign "rem" (App Div (Var "A") (Val (Int 2))),
                      Assign "rem" (App Mul (Var "rem") (Val (Int 2))),
                      Assign "compare" (App Sub (Var "A") (Var "rem")),

                      -- If the difference is 1, then A is odd.
                      If (Var "compare") (Seqn
                        [
                          Assign "A" (App Add (App Mul (Val (Int 3)) (Var "A")) (Val (Int 1))),
                          Assign "list" (App Add (Var "list") (Var "A"))
                        ]
                      )
                      -- Else, then A is even.
                      (Seqn
                        [
                          Assign "A" (App Div (Var "A") (Val (Int 2))),
                          Assign "list" (App Add (Var "list") (Var "A"))
                        ]
                      ),

                      Assign "isRunning" (App Sub (Var "A") (Val (Int 1)))
                    ])]
```

(G) Implementation of Collatz sequence.

3.3 Testing in Hspec

For testing, I implemented 7 tests in Spec.hs to ensure that the functions operate as they should, and provide correct output. This includes testing the factorial, newton, and collatz functions as well as attempting division by zero. The testing also includes checking for if addition is attempted to a variable which is already an error. For example, we test if the root method works if we provide an non-integer number as input, such as 64.8. We then know that it should return in a Double, and that the value of the double should be 8.0498.

```haskell
main :: IO ()
main = hspec $ do
  describe "Factorial example 1" $ do
    it "Attempts to find the factorial of 5." $ do
      exec(comp(fac (Int 5)))!!0 `shouldBe` ("A",Left (Int 120))

  describe "Collatz example 1" $ do
    it "Attempts to find the Collatz sequence of Int 5." $ do
      (exec(comp (collatz (Int 5))))!!1 `shouldBe` ("list",Left (Ints [5,16,8,4,2,1]))

  describe "Collatz example 2" $ do
    it "Attempts to find the Collatz sequence of Double 11." $ do
      (exec(comp (collatz (Int 11))))!!1 `shouldBe` ("list",Left (Ints [11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]))

  describe "Newton's method example 1" $ do
    it "Attempts to find the square root of Int 13." $ do
      (exec(comp (root (Int 13) (Int 100))))!!0 `shouldBe` ("R",Left (Double 3.6055512754639896))

  describe "Newton's method example 2" $ do
    it "Attempts to find the square root of Double 64.8." $ do
      (exec(comp (root (Double 64.8) (Int 100))))!!0 `shouldBe` ("R",Left (Double 8.049844718999243))

  describe "Divide by zero example 1" $ do
    it "Attempts to divide by zero with execOp." $ do
      (execOp Div (Left (Int 5)) (Left (Int 0))) `shouldBe` (Right "Error. Attempting to divide by zero.")

  describe "Divide by zero example 2" $ do
    it "Attempts to add a value to an error variable." $ do
      (execOp Add (Right ("BlaBla")) (Left (Int 21))) `shouldBe` (Right "Error. Attempting to divide by zero.")
```

(H) Hspec tests

In order to run these tests, we run "stack test", and it automatically conducts these tests and

provides the results (I):

```
Factorial example 1
  Attempts to find the factorial of 5.
Collatz example 1
  Attempts to find the Collatz sequence of Int 5.
Collatz example 2
  Attempts to find the Collatz sequence of Double 11.
Newton's method example 1
  Attempts to find the square root of Int 13.
Newton's method example 2
  Attempts to find the square root of Double 64.8.
Divide by zero example 1
  Attempts to divide by zero with execOp.
Divide by zero example 2
  Attempts to add a value to an error variable.

Finished in 0.0020 seconds
7 examples, 0 failures
```

(I) Hspec test results

Overall, all of the test pass, and I am happy with the operation of my implementations. This was a cool project ;).

References

[1] Brink van der Merwe's classes and course slides. Stellenbosch University 2022.

[2] Graham Hutton's Youtube Haskell course. Available [https://www.youtube.com/playlist?list=PLF1Z-APd9zK7usPMx3LGMZEHrECUGodd3]

[3] Various classmates (Specifically Troy) who have helped to explain certain parts of the assignment.