Assignement 1 Advanced Programing

ØYVIN KONGLEVOLL MOXNESS & ASKE VALDEMAR PETERSEN, DIKU, University of Copenhagen, Denmark

1 INTRODUCTION

This report will describe the non-trivial parts and design choices of our Arithmetic Expression handler. Three different levels of evaluation functions were made that is: `evalSimple, evalFull` and `evalErr`. The latter will be explained more in depth than the two others as they are more trivial and built on the same principles as `evalErr`.

2 IMPLEMENTATIONOF EVALSIMPLE AND EVALFULL

Throughout all of the functions we extensively use pattern matching to recursively solve every different arithmetic expression given by the algebraic data type `Exp`. This also enables us to evaluate nested expressions such as `Pow ((Div (Cst 4) (Cst 2)) ((Add (Cst 3) (Cst 4)))` which is $\frac{4}{2}^{3+4} = 2^7 = 128$. The function `evalSimple` is declared as `evalSimple :: Exp -> Integer` which means that it takes an expression and returns an integer. For the function we have the base case: `evalErr (Cst e1) = e1` this means that the expression is a constant and we return it as an integer. This enables us to recursively solve the expressions by deconstructing the expression until they return either an `Int` or an error. To do addition for example we do the following: `evalSimple (Add e1 e2) = evalSimple e1 + evalSimple e2`. Here we recursively deconstruct `e1 : Exp` and `e2 : Exp` and solve with the two integers returned. Due to this recursion we can evaluate nested expressions as stated earlier.

For `evalFull` we follow the same logic, however, it also takes an environment to write into and lookup values. Therefore `evalFull` is declared as: `evalFull :: Exp -> Env -> Integer`. `evalFull` behaves similar to evalSimple, but it is also solves the more complex expressions such as `If, Var, Let, Sum`, but still by pattern matching on expressions and deconstructing those expressions to integers using recursion.

2.1 Strict error-propagation in exponentiation

When Haskell computes the expression a^b it will by default return 1 as long as $b = 0$ even though a is undefined or an error. Therefore in the case of calculating $\frac{4}{0}^0$ Haskell will return 1 instead of a division by zero error unless this is explicitly handled. In the expression `Pow e1 e2` if e1 is evaluated to an `error` we want the entire expression to return this error instead of returning 1, when $e2 = 0$. To do this we handled the `Pow` expression as:

```
evalFull (Pow a b) e =  let x = evalFull a e in
                            x `seq` x ^ evalFull b
```

Here we use the built-in function **seq : a -> b -> b** this ensure that both a and b are evalutaed before **seq** returns a value. This way we ensure that $e1$ will be evaluated even though $e2 = 0$ and that way the expression will result a potential error instead of just 1.

2.2 extendEnv

To keep track of variable bindings we create *environments* which maps variable names to their respective values, if it exists. The type `Env :: VName -> Maybe Integer` is a mapping that takes a variable name and return either `Just a` if the variable name has a binding in the environment and if not it returns `Nothing`. In order to bind values to variable names we use the function `extendEnv :: VName -> Integer -> Env -> Env`. This function takes a variable name , integer and an environment and returns a new extended environment with a binding from the variable name to the integer. If there were previous bindings these will still be intact. This is done by using a lambda which binds the new variable name to the integer and then proceeds to update with all the existing bindings from the environment.

2.3 evalErr

`evalErr` required that the following errors were handled correctly: division by zero, negative exponent as we do not handle floats and use of undefined variable. Monads seemed like an obvious choice to handle this scenario as the functions could have the side effect of returning an error from an expression. While we didn't have a full understanding of how monads work, we figured that we could forward the result of our expressions to the final return statement. In case one of the expressions were erroneous the return function managed to report one of the found errors. The results from each expression were forwarded using the bind operator >>= followed by an anonymous lambda function. Thus we can define our `Add` arithmetic as:

```
evalErr e1 r >>= \x -> evalErr e2 r >>= \y ->
    return $ x + y
```

When handling division we check whether the divisor is 0 through an if statement and in case the expression is evaluated to the Either ArithError `Left EDivZero`, and same pattern is used when handling the power expression, that is if the exponent is below zero it evaluates to `Left ENegPow`.

2.4 Deal with errors in unneeded parts of Let-expressions

The implementation of `Let` doesn't handle nor care about errors in irrelevant parts of the binding. This would seems unnecessarily strict and would require a longer and less readable implementation as it would need to search the body of the `Let` expression for the unused variable.

3 ASSESSMENT

3.1 Completeness

Most of the code seems to function as intended however there is an issue when the function handles the `Sum` expression. While we are unsure of how to solve it we believe it might be due to how the environment is updated but this is explained further in the correctness section.

3.2 Correctness

To evaluate the correctness of the code we created a series of tests designed to extensively examine our implementation in different edge cases and error handling especially for `evalErr` and`evalFull`. The tests were designed to ensure that our implementation would be tested against a variety of scenarios. In addition to this we ran our functions through the *Online TA* test at find.incorrectness.dk to discover flaws in our implementation. This is where we realized that our implementation of `Sum` were faulty to some extent. There seems to be a problem when we sum from n to a variable whereas the variable is updated within the body of the sum expression. We defined the sum call recursively and evaluate the `to` variable at every iteration. Thus if the variable gets updated within the body, so does the range of the sum expression. This could lead to infinite or shortened summation loops.

3.3 Other

During the development of the algorithm we had just begun coding in Haskell thus we chose the first path that seemed to solve our problem, without taking into account (or even knowing of) other solutions that might have been better. Thus we made some hastened decisions and ended up using some monads which we might not fully comprehend.

4 CONCLUSION

This report has described a somewhat successful arithmetic expression handler, that struggles with evaluating deep sum expressions. Some of the design choices and the reflections that were made were reported as well as the flaws that the implementation includes.