AP - Assignment 1

dvg554,
pgw622

September 2021

1 Design and implementation

*** Questions

In evalFull we decided to stick with default Haskell lazy-evaluation, such that an error only is thrown when the expression causing it is used/called. Thereby the expression "Let "x" (Div (Cst 4) (Cst 0)) (Cst 5)" would evaluate to 5. This also simplifies the implementation nicely. In evalErr we use monads in our implementation and want to evaluate each expression to unpack it if it doesn not return an error. It's still possible to align evalFull and evalErr error handling in this case, but we choose differnet versions for simplicity's sake and to demonstrate different design patterns.

Design choices

We noticed that when evaluating e.g. `Pow (Div (Cst 5) (Cst 0) (Cst 0)` both `evalFull` and `evalErr` returned 1. We believe that this is because the compiler optimizes the expression $base^{exp}$. When seeing that $exp = 0$, the $base$ is not evaluated. And in our case the base might err. We solved this by using the BangPattern language extension to force the evaluation of the base, making the function behave as intended.

In `showExp` we introduce an auxiliary function `showExpSub` that handles subexpressions, to avoid too much repetition in `showExp`.

In `evalErr` we use the fact the `Either a b` is an instance of the monad class. We are thus able to compress the functionality nicely with do-notation. Making the function much more concise and readable.

2 Assessment

Completeness

To the best of our knowledge the specifications asked for has been implemented (except the optional tasks).

Correctness

Consulting the *OnlineTA* and running our own fairly thorough test suite, we have not discovered any bugs or errors. Note that the test suite is just an extension of the test suite that was handed out. It can be run by `$ stack test` in the folder `../code/part2/`. Note that checking if a function returns a runtime error is not possible (without some deep GHC magic) and it is therefore not tested.
Because of the above we believe that our program works as intended and is indeed correct.

Efficiency

All the implemented functions are recursive in nature. We have tried to limit the execution time and space usage by 'caching' repetitive recursive calls (e.g. by using a let-binding for a recursive call if it is used multiple times in the same in-body). The running time of all the recursive functions is linear in the number of expressions.
Within what can be expected we believe that the running time and space usage of our program is acceptable.

Maintainability

We have tried to reuse as much code as possible within the functions, but it has been a hard task. In showExpr we made a short subfunction limiting the repetitive part of the code.

Among evalSimple, evalFull, evalErr, most of the functionality is similar, and we have tried to refactor it out to higher order functions, but since they all have different input- and return-types, we did not succeed in doing so.

Using monads in `evalErr` makes the functionality easy to extend and maintain.

The code is also properly layed out with no lines exceeding 80 characters, a comment for every function and no dead/unused code.