AP - Assignment 2

dvg554,
pgw622

September 2021

Design and implementation

*** Questions

The monad instance of `Comp` is defined using `return a` which uses the `Comp` constructor to "inject" `a` into the simplest form of the monad, e.g. `(Right a, [])`. `(m >>= f)` projects the monad `m` with the current environment, i.e. `runComp m e`, where `e` is the current environment. If the projection results in an error, we abort and return the error message along with the current list of strings. If the first computation returns some result, we run the second computation, with `f` applied to the result of the first computation, i.e. `runComp (f a) e`. Note that the environment is passed along unchanged - we only have read access. Again we can abort with an error or return a result along with the concatenated list of strings from the first and second computation (this is also the case when an error occurs).

Boa's list comprehensions were implemented by constantly looking at the first clause in the list of clauses. If the list was empty, we would just evaluate the body expression one time in a list expression. If the clause-list contained clauses, the program looks at each. If it's a **CCFor clause**, it evaluates the list of values in the range. If it's empty, we return an empty **ListVal**.

We use this list of values in a map, that recursively calls eval with the tail of the clause list, since every clause in the tail should be used, for each of the elements in the current **CCFor clause**. If the current clause is a **CCIf clause**, it checks the expressions. If the expressions evaluates to true, we continue to recursively call eval with tails, otherwise we return an empty **ListVal**.

This procedure ends up with nested **ListVal's**, which are concatenated together.

Assessment

Completeness: According to the specifications of the assignment all the required functionality has been implemented. We back this claim up with a significant test suite and a flawless consultation with the *OnlineTA*. We have thoroughly read the specification and implemented all the details included.
To run our tests simply run `$ stack test` in the `../code/part2` folder. Our tests has been implemented using `Tasty` and are nicely organized by function or for small functions in meaningful groups (e.g. "monad operations"). We test both succeeding and failing conditions. No tests results in a fail. In addition to this the executable `boa` also returns correct results with all the provided examples.

Correctness: As stated in the previous subsection we believe our implementation is correct because of thorough testing. To the best of our knowledge no bugs exists (but can one ever be 100% sure?)

Efficiency: We have tried to make the code more efficient by quickly aborting when an error occurs. E.g. in `operate` where the function either takes correct input or immediately aborts with an error, this takes constant time and is very efficient. We have tried to implement this principle throughout the more advanced functions such as `apply` and `eval` (if the code results in an error, we do not want to spend time evaluating anything else).

Robustness: For all cases we know off an error or the like results in an error message being conveyed using `RunError` as described in the assignment. We make sure that our patterns always contain a wild card case resulting in aborting with an error (even though the pattern supposedly is unreachable). We provide informative messages that are relevant to the cause of the error and never abort with a Haskell runtime error.

Maintainability: Aside from commenting our code, leaving no dead/unused code and keeping our lines below 80 chars (where possible), we have separated the concerns by using helper function for e.g. `apply`, where both range and print rely heavily on helper functions, making the actual function rather simple. This makes the code easier to debug and maintain. We have tried to incorporate this principle throughout the implementation.
Even though it is required, we want to point out that `eval` and `exec` do not depend on the concrete implementation of the `Comp` monad, making it much easier to alter without breaking the mentioned functions (we use look, withBinding etc.). This also increase maintability.