AP Assignment 1

szw935, grc196

September 2021

1 Introduction

Note that both of us are retaking this course, so we may have some reused code from last year. We asked whether this was acceptable, and we were told it was if we make it clear in the report, so here we are. This is also why it says it took zero minutes to do the installation and reading in the time-sheet.

2 Design and Implementation

2.1 Lazy vs. Eager

Our implementation of Let in `evalFull` uses lazy evaluation, so variable binding is only done if and when it appears in the expression. This way, no errors are thrown if the binding expression has errors in it, since `evalFull` has no convenient way to handle errors except for throwing the standard error function. `evalErr` uses eager evaluation since it can handle such errors, as well as it coincidentally being the implementation we figured out. We allow for let expressions where the unbound variable is not used in both.

Since our `evalErr` implementation is already eager, we just call it directly from `evalEager`. Our `evalFull` implementation is already lazy, so the implementation of `evalLazy` uses `evalFull` in the Let-statement as `evalLazy (Let v e1 e2) r = evalLazy e2 (extendEnv v (evalFull e1 r) r)` so as to avoid type miss-matching. Unfortunately this is not a perfect implementation, as `evalFull` cannot handle exceptions if there are any present in the expression. We theorize that we could create a separate environment type that instead has a type of `VName -> Either ArithError Integer`, and an auxiliary function of type `Env -> NewEnv`, so that we could fix the type miss-match issue.

2.2 Pow Error-propagation

For the `Pow` expression type in `evalSimple` and `evalFull` we use `seq` to make sure any errors in `e1` for `(Pow e1 e2)` gets propagated, even if the value is not needed for the final result.

2.3 Show Compact

We did not quite get `showCompact` working; what we ended up with is `showExp` without the parentheses. We theorize that we could use an auxiliary function called `needParenthese`, with type signature `Exp -> Bool -> String`, to wrap around `showCompact`. This would check whether the given expression needs parentheses around it. Our attempt at it gave a type miss-match, since `showCompact :: Exp -> String` defines no boolean. We left the method in our code, albeit unused.

2.4 Testing

We tackled our testing by unit testing as extensively as we could. With each of the functions we brainstormed as many scenarios as possible, and had each test do the smallest thing possible. With all our unit tests passing we can be sure that any larger executions of nested and combined functions would work as intended.

For `evalFull` we test all the same functionality as `evalSimple` with extended functionality, and the same goes for `evalErr`, `evalEager` and `evalLazy` with `evalFull`. We made sure to test eagerness and laziness worked in all the methods as we intended it to. We did not find a suitable way to test the basic functionality of `extendEnv` without using it along with `evalFull` or `evalErr`, which we made sure to highlight in a comment.

Since `evalSimple` and `evalFull` cannot handle errors, and neither could our implemented test suite, errors could only be tested in the methods returning `Either ArithError Integer`. We also found no way to test that expressions are evaluated from left to right, and not evaluated after an error, ie. in `(Add (Var "v") e2)`, `e2` is never evaluated as `v` is unbound. That being said, with our understand of the `do` notation we used, these tests would pass.

3 Assessment

The submitted code are much alike last year, but this time we had a much better understanding of Haskell and had an easier time figuring out how and why the code works as it does. Besides our stated issues with `showCompact` and `evalLazy`, our code works against the specifications, although there still could be improvements. With the depth with went into with our tests we are confident in saying this completeness and correctness is accurate, however we have not tested for efficiency at all. For maintainability, we probably could find some more auxiliary functions to reuse code (especially for `evalErr` and `evalLazy`), but we have remove some duplication, for example with `evalEager = evalErr`