Home Assignment 1
Advanced Programming

qhz731

September 15, 2024

1 Introduction

I add the supports for the `functions`, `try-catch`, `pretty-printer` to the project. Wrote some test cases into the AST_Tests.hs for the `pretty-printer` functions and some test cases into the Eval_Tests.hs for the `try-catch` and `function` functions. To run the tests, please run the command `cabal test` in the command line under the same directory as the a1.cabal.

2 Design and Implementation

2.1 Functions

2.1.1 Lambda

In the task descriptions, When evaluated the `Lambda` procuces a function value represented by `ValFun`. The `ValFun`'s type definition is given by the task descriptions, which is

```
data Val
  = ...
  | ValFun Env VName Exp
```

So we simply implement the evaluation of the `Lambda` in a patter matching flavor as below.

```
eval env (Lambda vname body) =
  Right $ ValFun env vname body
```

Where we use `body` to represent the function expression. I extend the original `eval` function to define the evaluation of `Lambda`, maintain the program in a functional way.

2.1.2 Apply

The `Apply` applies a function expression to an argument expression. The evaluation of a function expression must be a `ValFun`, so we first evaluate the first argument of the `Apply` constructor to check whether it is a `ValFun`.

```
eval env (Apply fun arg) = case eval env fun of
  Left err -> Left err
  Right (ValFun env' vname body) ->
    case eval env arg of
      Left err -> Left err
      Right argVal -> eval (envExtend vname argVal env') body
  Right _ -> Left "Non-function applied."
```

the input arguments are current environment and the `Apply` type and the I first evaluate the first argument `fun` of the `Apply`, if there is an error I return a `Left err`, else if the function expression is a `ValFun` I evaluate the second argument of the `Apply`. The evaluate success only if the second argument expression does not raise an error. Then, as the task description shows, the application will store the argument's name and argument's value into the environment then using the new environment to evaluate the body. Finally, I check if the first argument of `Apply` is a function. I think the implementation of the extension of the function `eval` of the `Apply` part is functional. I extend the function of `eval` and recursively implement the evaluation function of the `Apply`, and using the previous defined function to extend the `eval` function. Basically, I use the function to extend the function, and maintain the function only with functions, so it is functional.

2.2 Try-catch

I follow the task description that first check whether the first expression has error, if it passes the evaluation, then I do the evaluation on the second expression. I use the pattern matching flavor to the `eval` function, with 2 arguments. The first argument is a environment and the second argument is a `TryCatch` expression. Then the function will evaluate the first expression. If the first expression has error, then it will evaluate the second expression and return the result of the second expression(value or error). In other case, if the first expression pass the evaluation, it returns the first expression's value. I think this implementation is functional because it extends the definition of the `eval` function, and it evaluate the `TryCatch` recursively.

```
eval env (TryCatch e1 e2) = case eval env e1 of
  Left err -> eval env e2
  Right v -> Right v
```

2.3 Pretty-printer

I define the function begin with the type declaration `printExp :: Exp -> String`. Then I start define the function body by defining the const type print function.

```
printExp :: Exp -> String
printExp (CstInt n) = show n
printExp (CstBool b) = show b
```

Then based on those basic definition, I define the `Add, Sub, Mul, Div, Pow, Eql`. As an example the `Add`'s `printEXp` defined like

```
printExp (Add e1 e2) = printExp e1 ++ "+" ++ printExp e2
```

I defined the expression print function recursively like the `Add`'s print function above. The print function for the `Var x` input will return the `x` simply. And for print function with the `Apply` as input, I parenthesize the function part and the argument part. The other expressions' print function are defined recursively and using pattern matching to extend the `printExp` function. The implementation of the `printExp` is also functional.

3 Question Answers

3.1 Apply's evaluation order

The order of the evaluation of the `Apply` really matters. It should first input a function and then an expression. Then it should apply the function to the argument. The order matters because if we catch the error in the evaluation of the function or the argument, then we won't apply the function. The corresponding test cases are:

```
testCase "Apply␣(wrong␣order)" $
  eval envEmpty (Apply (CstInt 2)
    (Lambda "x" (Add (Var "x") (CstInt 1)))) @?
    = Left "Non-function␣applied.",

testCase "Apply␣(non-function)" $
  eval envEmpty (Apply (CstInt 2) (CstInt 3)) @?
  = Left "Non-function␣applied."
```

3.2 TryCatch

The `TryCatch` evaluation cannot be implemented by evaluating the both subexpression first together. For example below

```
testCase "TryCatch␣(exception)" $
  eval envEmpty (TryCatch (Div (CstInt 2)
  (CstInt 0)) (CstInt 1)) @?= Right (ValInt 1)
```

if the two subexpression are evaluated together, it will return an error when we evaluate the `Div (CstInt 2) (CstInt 0)`, rather than return the 1.

But if the both expression are pure and we are in a lazy evaluation context, the catch expression wouldn't be actually executed unless it was needed.

3.2.1 Infinite loop

It is possible for my implementation of eval to go into an infinite loop. As the test case below,

```
Apply (Lambda "x" (Apply (Var "x") (Var "x")))
  (Lambda "x" (Apply (Var "x") (Var "x")))
```

In this condition, the `Apply` will apply a lambda function as the input to another same function. This lambda function will always call itself and will never end. When the `cabal test` runs, it will recurse infinitely.

4 Assessment of Code

4.1 Completeness

All the asked-for functionalities(except for optional) are implemented and necessarily fully working.

4.2 Correctness

All implemented functionalities are work correctly and pass all tests designed ourselves.

4.3 Maintainability

For the auxiliary definitions in the code, you can tell the functionality from one function name.So for the maintainability, it is easy for other to maintain the code with the auxiliary definitions. Also, the code is well commented. For the test suite, there is not so many auxiliary as the source code, as copying code is easy to understand.

4.4 reference

I used the AI tool Cursor to generate some of the test cases for the Eval_Tests.hs and the AST_Tests.hs.