Home Assignment 2
Advanced Programming

qhz731

September 22, 2024

1 Introduction

I extended the `EvalM` monad by defining the `State`, this can make the `EvalM` store a list of print history. I implemented the `Print` and made it available to print the list of strings stored in the type `State`. I modified the `runEval` and every other thing to fit the new `EvalM`. I also implemented the Key-Value in the ASL and made it can be stored and fetched in the `State`. I also added the implementation of check in the Check.hs file. Finally, I added some test cases in both Check_Tests.hs and Eval_Tests.hs to ensure the correctness of the program.

To run the tests, please run the command `cabal test --test-show-details=always` in the command line under the same directory as the a2.cabal.

2 Task: Printing

My implementation of `Print` is functional and handles basic cases correctly. I defined the type of `State` to contain a list of string as the log. And I modified the `EvalM` monad to be

```haskell
newtype EvalM a = EvalM (Env -> State -> (State, Either Error a))
```

The `return`, `>>=` and `runEval` were implemented as

```haskell
instance Monad EvalM where
  return x = EvalM $ \_env state -> (state, Right x)
  EvalM x >>= f = EvalM $ \env state ->
    case x env state of
      (s, Left err) -> (s, Left err)
      (s, Right x') -> case f x' of
        EvalM m -> m env s

runEval :: EvalM a -> ([String], Either Error a)
runEval (EvalM m) = case m envEmpty stateEmpty of
  ((s, _), Left err) -> (s, Left err)
  ((s, _), Right x) -> (s, Right x)
```

I did not explicitly describe how the state is modified in `>>=` and `runEval`. All state modifications are done by other auxiliary functions. The `evalPrint` helper function appends a new string to the end of log part of the state while keeping the key-value store unchanged.

My implementation of `Print` should be functionally correct, since it correctly evaluates expressions before printing, handles different value types and the log to print appropriately, and returns the evaluated value after printing.

My implementation does not handle the case where evaluating `e` results in an error. If `eval e` fails, the entire `Print` operation will fail instead of printing an error message. It only supports three types of value: `ValInt, ValBool` and `ValFun`.

3 Task: Key-value store

My implementation of key-value store is overall functionally correct. As shown in the previous section, the EvalM monad accepts a new state parameter in addition to env. It handles basic insert, update, and get operations correctly. Keys and values are correctly evaluated before storing, updating existing keys is properly handled by replacing the old value, and it integrates well with the EvalM monad to keep the state consistent.

Potential issues with my implementation are that there is no type checking for keys or values, allowing any `Val` type to be used as a key, and the error handling in `KvGet` may not be detailed enough.

If the user try to get a non-existing key, it returns an error instead of `Nothing` of the type `Maybe`. This is inconsistent with the behavior of the `lookup` function in Haskell. Additionally, it may have problems if the value used as a key is not comparable (such as a function).

4 Task: Type checking

I defined a new Environment type `type Env = [VName]`, I did not actually evaluate an `Exp` in the implementation, I just added the `VName` that bounded in the `Let` and `Lambda Exp` to the `Env`. For its main purpose (checking variable scope), this implementation is functionally correct. It can correctly identify unbound variables and maintain the correct variable scope. Even without evaluation, incorrect recursive definitions can be detected, `Exp` like `Let "x" (Var "x") e2` will not pass type checking.

But without real evaluation, runtime errors may cause potential issue, such as the expression `e1` may contain runtime errors that cannot be caught during type checking. For example, `Let "x" (Div (CstInt 1) (CstInt 0)) e2` will pass type checking but will fail at runtime.

5 Question Answers

5.1 Complexity of Print Implementation

The time complexity of my implementation of `Print` is $O(n + m)$, where the n is the number of the nodes in the expression tree. In the case of `eval (Print str e)`, the execution happens both in `evalPrint` and the `eval e`. In the previous case, the complexity is $O(m)$, where m is the total length of all previously printed strings. In the latter case, the complexity is $O(|e|)$ the execution time depends on the length of the expression linearly.

When considering the times of the program call the `Print`, I let the p be the times of calling `Print`, and the case becomes more complex. The complexity of the `value <- eval e` is $O(pn)$ where the n is the number of the nodes in the expression tree. The complexity of the `evalPrint` depends on the history of the printing list. ++ in Haskell takes linear time so the total time of appending the list is $O(1 + 2 + ... + p) = O(p^2)$. The total complexity of the p print operations is $O(pn + p^2)$

The complexity of the `Print` implementation can be improved by implementing an $O(1)$ append operation or delaying the string concatenation until it is needed. Using `Data.Seq` may help.

5.2 Memory Usage of KvPut

If a program keeps updating the same key with `KvPut`, the memory usage of the program should remain constant. In my implementation, `KvPut` first check if the key is already stored in the key-value store, if not, just append the new pair, if already stored, the existing old key-value pair is filtered out, the new key-value pair is added.

```
evalKvPut :: Val -> Val -> EvalM ()
evalKvPut k v = EvalM $ \_env (s, d) ->
  case lookup k d of
    Nothing -> ((s, (k, v) : d), Right ())
    Just _ -> ((s, (k, v) : filter (\(key, _) -> key /= k) d),
      Right ())
```

5.3 Effects visibility in Trycatch

In my implementation of `TryCatch`, the effects performed by `e1` will not be visible to `e2`. I write some cases to test this behavior.

```
  --
  testCase "TryCatch" $
    eval' (TryCatch
            (Let "x" (KvPut (CstInt 1) (CstInt 5))
              (Div (CstInt 1) (CstInt 0)))
            (KvGet (CstInt 1)))
      @?= ([], Left "Invalid␣key:␣ValInt␣1"),
  --
  testCase "TryCatch" $
    eval' (TryCatch
            (TryCatch
              (Let "x"
                (Print "Inner" (Div (CstInt 1) (CstInt 0)))
                (CstInt 1))
```

```
                    (Print "Caught␣inner" (CstInt 2)))
                  (CstInt 3))
            @?= (["Caught␣inner:␣2"], Right (ValInt 2)),
```

To change the behavior and make the effects caused by `e1` to be visible to `e2`, I can change the `catch`'s implementation to:

```
catch :: EvalM a -> EvalM a -> EvalM a
catch (EvalM m1) (EvalM m2) = EvalM $ \env state ->
  case m1 env state of
    (s', Left _) -> case m2 env s' of
        (s'', Right x) -> (s'', Right x)
        (s'', Left err) -> (s'', Left err)
    (s', Right x) -> (s', Right x)
```

In this case, I use the updated state `s'` to evaluate the `e2`.

6 Reference

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