Advanced Programming
Home Assignment 4

Carlo Rosso, Chinar Shah

Contents

1 Introduction

This assignment was much more difficult than the previous one. The concept of monads is a tricky one, in particular Free monads and its ability to handle side effects. Hence, working through the exercise and the assignment took much longer than usual. Task One was not time consuming or that difficult to implement as we could utilise previously defined functions from the exercise. The following two tasks however, were relatively difficult and much more time consuming to do.

The following describes how to run the tests

1. unzipping `a4-handout.zip`;
2. entering the `a4-handout` directory through the terminal;
3. running `cabal test` in the terminal.

2 Task - The TryCatchOp effect

The implementation of this task is functionally correct. After the previous assignments, our understanding of TryCatch and its implementation was sound enough to effectively and correctly finish this task. Overall it was very mechanical.

3 Task - Key-value Store Effects

This task was fairly tricky and time consuming but overall we believe the functionality of all three subtasks are correct. This is substantiated by the tests

```
testCase "Example3.4" $ do
        (_, res) <-
          captureIO ["ValBool True"] $
            runEvalIO $
              Free $
                KvGetOp (ValBool True) $
                  \val -> pure val
        res @?= Right (ValBool True),
      testCase "Example3.3" $ do
        (_, res) <-
          captureIO ["lol"] $
            runEvalIO $
              Free $
                KvGetOp (ValBool True) $
                  \val -> pure val
        res @?= Left "Invalid value input: lol",
      testCase "Example3.4" $ do
        (_, res) <-
          captureIO ["ValInt 1"] $
            runEvalIO $
              Free $
                KvGetOp (ValInt 0) $
                  \val -> pure val
        res @?= Right (ValInt 1),
```

Here we ensure that

1. If a key is missing in the database, instead of failing, prompt the user to enter a replacement value.

2. The input should be either ValInt or ValBool. Invalid input should return an error. (lol case)
3. If a key exists in the database and all inputs are correct then we get the correct key back

4 Task - TransactionOp effect

This task was of a similar difficulty to the previous one - however we implemented it slightly easier considering that the familiarity with the code has increased. This is also fully functional with the code performing the key parts well:

1. Handling all transactions effectively
2. Ensuring atomic writes to the key-value store
3. Correctly rolling back state changes upon failure
4. Supporting nested transactions without issue.

5 Asnwers to Questions

1. Consider interpreting a TryCatchOp m1 m2 effect where m1 fails after performing some key-value store effects.

(a) Is there a difference between your pure interpreter and your IO-based interpreter in terms of whether the key- value store effects that m1 performed before it failed are visible when interpreting m2? If so, why?

Yes there is a difference between the pure interpreter and the IO based interpreter. This difference is to do with the visibility of any key-value store effects that m1 performs before failing. In the pure interpreter (runEval) they are visible because it is directly manipulating the in-memory state and because there is no rollback logic after the failure, when m2 is interpreted it is interpreted with the state modified by m1. This differs to the IO-based interpreter where the visibility occurs because the key-value store is written to a database file, and m1's changes are persisted in the file even if m1 fails and the interpreter continues interpreting m2 with the state reflecting the modifications made by m1. (the state changes are similar)

(b) Suppose you've implemented your interpreters such that the key-value store effects that m1 performed before it failed are always visible when interpreting m2. Without chang- ing the interpreters, is it possible to have different behav- ior where the key-value store effects in m1 are invisible in m2? If so, how? If not, why not?

Yes it is possible and it is possible using the functions implemented in TransactionsOP. If we wrap the key-value store effects of m1 inside a TransactionOp, the key-value modifications in m1 are only committed if m1 succeeds. And similarly if m1 fails, the changes to the key-value store are rolled back, making them invisible to m2.

Why does the computation payload in the TransactionOp (EvalM ()) a constructor return a () value? Do any other return types make sense? Justify your answer

The computation payload in the TransactionOp (EvalM ()) a constructor returns a () value because the primary purpose of a transaction is to manage side effects and we do not need to return the result. The () type signals that the transaction's main concern is the success or failure of the side effects and not the result of the computation.

Whilst we can technically return an EvalM a type - so returning the result. It would be syntactically incorrect because we want to manage the side effects rather than focus on the computation result and () allows the function to maintain its focus on that. Hence () is the only type that makes sense.