Home Assignment 4
Advanced Programming

qhz731

October 5, 2024

1 Introduction

In this assignment, I extended the EvalM free monad with TryCatchOp effect, Key-Value Store effect and TransactionOp effect. For each effect I first extended EvalOp with each effect, then extended the Functor instance for EvalOp to include the new state effects, then extended the code for the pure and IO interpreter to include support for the new state effects. Regarding testing, I wrote a series of unit tests to verify the correctness of each new effect. These tests cover basic functionality, edge cases, and error handling.

To run the tests, please run the command `cabal test --test-show-details=streaming` in the command line under the same directory as the a3.cabal using WSL. There are some test cases that require input, when they ask for input, the expected input will be press Enter.

2 Task: The TryCatchOp Effect

My TryCatchOp implementation is functional for most purposes. It successfully handles basic exception catching scenarios in both the pure interpreter and the IO interpreter.

There are significant differences in my TryCatchOp implementation in the pure interpreter and the IO interpreter, reflecting the two different exception handling strategies.

In pure interpreter, I used stricter isolation strategy. If m1 fails, all effects in m1 (including state modifications and printing operations) are not visible in m2. When m1 fails, m2 is run with the initial state, ensuring complete isolation. The print content in m1 is discarded when it fails and does not appear in the final output.

In IO interpreter, the visibility of the effects is different from that of the pure interpreter. The effects of m1 are visible in m2, including modifications to the database and printing operations. I did not use withTempDB to isolate the database changes caused by the failed m1. The print content in m1 is retained and output even if m1 fails.

3 Task: Key-value Store Effects

I implemented the required functions for Key-value store effects (KvGetOp and KvPutOp). Pure interpreter and IO interpreter exhibit different behavioral characteristics.

In the pure interpreter, KvGetOp returns a Left error for non-existent keys, which meets the expected exception handling mechanism. The IO interpreter handles exception through user interaction and provides an error recovery mechanism. In the pure interpreter, TryCatchOp completely isolates the effects of failed operations, which means that a key-value store operation that fails in a try block will not affect subsequent catch blocks. In the IO interpreter, since no temporary database is used, the key-value store operation in the try block will affect the database state even if it fails, which may cause inconsistencies.

4 Task: TransactionOp effect

The TransactionOp effect is functionally correct in both pure interpreter and IO interpreter and meets the specific requirements of the task. Even if the calculation inside the transaction fails, the error will not be propagated, and the transaction always returns Right(), regardless of whether the internal operation succeeds.

The pure interpreter fully meets the requirements and implements the expected transaction operation and error handling. Use `(ps, newState) = runEval'rs (em >> getState)` to get the updated state. If the internal operation succeeds, the external operation will use the updated state, otherwise the external operation will use the initial state.

The IO interpreter implements persistent transaction operations using the file system. Use withTempDB to create a temporary database to ensure the atomicity of the transaction. When the inner operation succeeds, the contents of the temporary database are copied to the main database. When the inner operation fails, the temporary database is discarded, the main database remains unchanged, and no exception is thrown.

5 Q&A

5.1 TryCatchOp m1 m2 effect

(a) There are differences between pure interpreters and IO interpreters in terms of whether the keyvalue store effects that m1 performed before it failed are visible when interpreting m2. In pure interpreter, the key-value storage effect after the failure of m1 is not visible when interpreting m2. This is because the pure interpreter uses the initial state before m1 is executed when executing m2. In this way, the pure interpreter maintains stricter isolation and consistency. In an IO interpreter, the key-value storage effect after the failure of m1 is visible when interpreting m2. This is because the IO interpreter directly operates the database file, and the operation of m1 directly modifies the database state. When m1 fails, these modifications have already taken effect and still exist when m2 is executed. The reason for this difference is that the two interpreters handle state differently. Pure interpreters use immutable state transfer, while IO interpreters directly modify external state (database files).

Without changing the interpreter, it is possible to make the key-value store effects in m1 invisible in m2. We can achieve this by using transactions (TransactionOp). The method is as follows:

```
TryCatchOp (transaction m1) (transaction m2)
```

By wrapping m1 and m2 in a transaction, m1 succeeds, the transaction will be committed, and all effects of m1 will take effect. If m1 fails, the transaction will be rolled back, and all effects of m1 (including key-value store operations) will be undone and invisible to m2. This method works in both pure interpreters and IO interpreters.

5.2 TransactionOp (EvalM ()) a

The computation payload return () value in the TransactionOp (EvalM ()) a constructor emphasizes the main purpose of transactions: to perform a series of operations atomically without focusing on the specific return value. Using the () type simplifies the concept of transactions, focusing on side effects (such as state changes) rather than return values. This approach also provides consistency, so that all transactions have the same return type, which simplifies the code that handles transactions.

Other return types may also make sense. For example, `TransactionOp (EvalM Bool) a` provides a simple way to check the success or failure of a transaction, while `TransactionOp (EvalM State) a` allows direct inspection of the resulting state of a transaction, but may expose too many internal details.

6 Reference

I used the AI tool Cursor to generate some of the test cases for the Interp_Tests.hs. The generated tests are carefully checked to ensure the reliability.