Assignment 4

Advanced Programming
Department of Computer Science
University of Copenhagen

Magnus/mfh144 | Mathilde/qfh987

06/10/2024

Contents

1. Introduction

In this assignment we kept on working on the existing interpreter from previous assignments. We made it more generic by exchanging the instance of *Monad* for *EvalM*. Instead we added a new data type *Free* with defined monad properties. This will make our interpreter more generic as all types of functors can now be wrapped in the Free monad. We then define the *EvalOp* functor and define how these operations should be interpreted in the pure and the IO based implementation.

We have made test cases for all new functionality and all tests pass. We believe all functionality is correctly implemented.

The tests can be run from the a4-handout directory with the following command:

```
a4-handout $ cabal test
```

2. Task: The TryCatchOp Effect

We believe this functionality works as intended and it passes all tests.
In this task we implemented the *TryCatchOp*. In this operation, we evaluate the first *EvalM* and check if it succeeded. If it does, we just return the result. If not, we evaluate the second *EvalM* and return the result of that. In the pure implementation, we made sure that all *PrintOp*s in a failing evaluation of the try block is still reflected in the result of the catch block. We do this by appending the string list from the catch block to the string list from the failed try block. That way the order of prints are preserved. We also use the initial state to evaluate the catch *EvalM*, so that state changes from a failing try is not reflected in catch. In the IO implementation we do not need to keep track of prints since they are actually printed properly. To keep track of the state, we need to read the initial state using *getState* before evaluation. If the try evaluation fails, we can then overwrite the db file with the initial state, as to reset the state to what it was before evaluation of the failed try block. We then evaluate the catch block after. Again, success in the try evaluation will just return the result.

3. Task:Key-Value Store Effects

We believe this functionality works as intended and it passes all tests.
In this task we implemented the *KvGetOp* and *KvPutOp* to get and put values in the key-value store.

KvGetOp

In the pure implementation of *KvGetOp* we were asked to look up the key in the state and interpret on the continuation function if the lookup was succesfull otherwise fail with a Left. We do exactly that with a call to *lookup key s* matching on *Just v* when succesfull and *Nothing* when failing.
In the IO implementation of *KvGetOp* we first extended *runEvalIO′* with support for state effects, *StateGetOp* which is called through *getState*. *StateGetOp* reads from the database to get the state. If there is a state *runEvalIO′* is run with the given state otherwise an error is thrown. We then implemented *KvGetOp* which uses *getState* to check for a state. If one exists the key is looked up in the state and if a value is found *runEvalIO′* is run on the continuation function with the value found passed into it. Otherwise the user is prompted using *prompt* to enter a replacement key. If the new key successfully reads as a Val r*runEvalIO′* is run on the continuation function with the replacement key. Otherwise an error is thrown with the message of an invalid value input.

KvPutOp

In the pure implementation of *KvPutOp* the association must be insterted into the state and replaced if it already exists. We do that simply by calling *runEval′* on the environment, a filtered state and m. The filtered state inserts the key value pair in front of a potentially altered state with an existing association to the key removed.

Int the IO implementation of *KvPutOp* we first extended *runEvalIO′* with *StatePut* which is called through *putState*. *StatePut* uses *writeDB* to write (or overwrite) the state to the file *db*. Then *KvPutOp* uses *getState* to check for a state. If one exists it uses *runEvalIO′* and putState on a filtered state as in the pure implementation. FInally it runs *runEvalIO′* on the continuation function.

4. Task: TransactionOp Effect

We believe this functionality works as intended and it passes all tests.
In this task we implemented the *TransactionOp*. This operation runs a transaction and then continues evaluation, regardless of the outcome of the transaction. The interesting part of the transaction operation is that a failing transactions state changes should be lost. In the pure iterpretation, we do this by evaluating the transaction followed by a *getState*. If the transaction succeeded, we can then pass the new state to the continued evaluation. If not, we use the initial state. We also make sure that prints are always propagated, so we append printing lists regardless of success or failure. In the IO interpretation we use the *withTempDB* helper function provided. We then copy the initial db into the temporary db and use this to evaluate the transaction. If the transaction succeeded, we copy the temporary db into the original db, if not we just continue evaluation with the initial db, since this has not been modified. We do not need to keep track of prints, since they are printed properly in the IO interpretation.

5. Questions

1. *TryCatch m1 m2*
   (a) No, there is no difference. In the pure interpreter it is simple to just pass the initial state to the evaluation of m2, but in the IO interpreter we make sure to read the initial state and overwrite the db file with the initial state before evaluating m2.
   (b) No, we don't see how this could be possible. If this would be possible, we would be able to construct an input that would circumvent the implementation of the interpreter, since we cannot modify the interpreter. We do not see how this can happen. We might have misunderstood the question.
2. The *TransactionOp* returns () because the return resembles failure or success only. If we were to return a value, we would need it to be meaningful in some way. A meaningful value could be the modified state that results from the transaction. This would be meaningful in our pure implementation especially, since we already evaluate the transaction using *m » getState*, so we read the state immediately after the transaction. In the IO implementation it does not matter, since we operate on files, so we don't read the state here.