AP Assignment 4:
Free Monads

Nikolaj Gade (qhp695), Sebastian Larsen Prehn (vpz655), Alba Dekens (dmb904)

1 Introduction

The code submitted alongside this report compiles and was expensively tested. Thus, we believe it behaves as expected. Although most of the code performs efficiently, we do have some parts of task 2 and 3 that do not work optimally, which will be highlighted in the report.

2 The TryCatchOp Effect

We extend `EvalOp` and its Functor with `TryCatchOp` by implementing the effect stated in the assignment with `m1` and `m2` and implement it so that the function *f* applies to both.

Afterwards, we implement catch using `TryCatchOp` on the inputs `m1` and `m2` utilizing `Free` since `EvalM` is of a free monad structure.

We then add support for `TryCatchOp` by writing a function that recursively evaluates `TryCathchOp m l`, evaluating the *m* that might throw an error *l*. In the case of the `runEvalIO` version, we also have to utilize `Right` and `pure` to match the Either Error a IO.

For the tests, we ran the same tests for both the IO- and Pure-based functions and had them pass for both `runEval'` and `runEvalIO'`.

3 Key-value Store Effects

We first extend `EvalOp`'s `Functor` instance with `KvGetOp` and `KvPutOp`.

For `KvGetOp`, we define `fmap f (KvGetOp v k)` by calling the function in argument that takes a `Val` to return a result of type `a`, and then apply function `f` on that result. This is wrapped along with the key in the constructor `KvGetOp`.

For `KvPutOp`, we put both the key and associated value in the `KvPutOp` constructor along with the application of function `f` on the result of the `KvPutOp` operation `m`.

Then, we define `evalKvGetOp` and `evalKvPutOp`.

`evalKvGet` takes a key `v` that is passed to the `KvGetOp` constructor along with the function `\w -> pure w` that wraps `w` in `pure`. A new `Free` monad is created with that `KvGetOp`.

`evalKvPut` takes a key `v1` and a value `v2` that are passed in the `KvPutOp` constructor along with `pure ()` which is needed to indicate that the result is `()` once the computation is done. This is all then encompassed in the creation of a `Free` monad.

Next, we extend `runEval'` to accommodate `KvGetOp`. The first step is to `lookup` the key in the state `s` that contains all the key-value pairs already put beforehand. If the key exists within the state, we execute with `runEval'` the next step of the computation. Conversely, if the key does not exist, we return an error that will stop the computation.

We extend `runEval'` further with `KvPutOp` by calling `runEval'` in order to compute the next step given as the third argument of `KvPutOp` with a new state containing the new key-value pair. This is easily done by adding the new pair to the current state as the `lookup` function used to get a value returns the first match found. However, this means that the state will continue expanding without ever freeing memory, which could lead to issues.

3.1 Using a Database File for the Key-Value Store

The `runEvalIO'` implementations of `KvPutOp` and `KvGetOp` are very similar to the `runEval'` implementations. The main diference is the use of `readDB` and `writeDB`, and the introduction of error handling for `readDB`. This is done with a simple case statement that propagates any error.

3.2 Missing Keys

Getting a value from a user is as simple as using the `prompt` and `readVal` functions in conjunction with each other. We then use a case statment to test if readVal gave an error. If not, `runEvalIO'` is run on `KvPutOp` again with the new key. This means that if continuously given valid but non-existant keys, the code will continue to ask. We believe this is the intended behaviour.

4 TransactionOp Effect

The `runEval'` implementation of `TransactionOp` uses `runEval'` on `do l >> getState`. This returns the printed strings and the state (or error) after `l` has been run. After that, we check if the output was a state or an error with a case statement: If it's a state, we run `m` with that state. If it's an error, we run `m` with the original state. We also add the printed strings from the transaction onto the list of printed strings from after the transaction.

The `runEvalIO'` implementation is similar. We use `withTempDB` to save a copy of the original database. If the function returns an error, we simply restore that copy.

Our testing tests that bare transactions have the correct return value (`TransactionOp 1` and `TransactionOp Propagation`), that the state can be read after a successful transaction (`TransactionOp 2`), that the transaction uses and adds to the original state (`TransactionOp 3`), that transactions fail correctly (`TransactionOp Fail`), that printing works properly (`TransactionOp Printing`), and that transactions can be nested correctly (`TransactionOp Nested`).

5 Questions

1. For our implementation, there is no difference between the interpreters in regards to them both letting effects made by `m1` be visible in `m2`. In order to change that so that it would be invisible, one would have to save the state before running `m1`, and use that saved state on `m2`, similar to how we handled `TransactionOp`.

2. The payload should be used for `put`'ing into the state, which returns `()`. Any meaningful computation should be done outside of transactions, so the payload should really only be allowed to return `()`.