Advanced Programming 2025

Assignment Four

Free Monads

Due: 2025-10-5

Synopsis: Implementing free monad-based interpreters for a small arithmetic language.

1 Introduction

In this assignment you will implement an interpreter for APL, based on free monads. In particular, you will extend what you developed in the exercises with new effects and new ways of interpreting them. Since the handout for the assignment differs slightly from the solution to the exercises, do **not** base the assignment on your own solutions to the exercises. You **must** use the assignment handout. In particular, the assignment handout removes the `StateGetOp` and `StatePutOp` effects in favour of a more fine grained representation.

For this assignment, you'll be implementing the following:

- Try-catch effects.
- A pure and IO-based interpretation of key-value store effects.
- Interactive missing key recovery.
- A transactional computation effect.

For all tasks you are expected to add appropriate tests to the file `Interp_Tests.hs`. For information on how to test the IO-based effects, please refer to the week 4 exercises.[1] Do not rename any of the definitions already present in the handout and do not change their types unless the assignment text explicitly instructs you to do so.

To make testing easier, we recommend that you **replace the definition of `eval`** in `APL.Eval` with your version of `eval` from your solution to

[1] `github.com/diku-dk/ap-e2025-pub/tree/main/week4#testing-runevalio`

assignment 2. A complete version of `eval` is not needed to fully solve the assignment; all functionality in this assignment can be tested without using `eval` by constructing appropriate `EvalM` values (directly or by using the interface functions in `APL.Monad`).

2 Tasks

Task 1: The `TryCatchOp` Effect

The `TryCatchOp` effect is for exception handling. A `TryCatchOp m1 m2 k` effect says to interpret `m1`; if it fails (i.e., returns a `Left` value), then interpret `m2`. Finally, pass the value resulting from interpreting `m1` or `m2` to the continuation `k`. For example,

```
> runEval $ Free $ TryCatchOp (failure "Oh no!") (pure $ ValInt 1) pure
([], Right (ValInt 1))
> divZero = CstInt 1 ‘Div‘ CstInt 0
> runEval $ eval $ TryCatch (CstInt 5) divZero
([], Right $ ValInt 5)
> badEql = CstInt 0 ‘Eql‘ CstBool True
> runEvalIO $ eval $ TryCatch badEql divZero
Left "Division by zero"
```

To start, extend `EvalOp` with a `TryCatchOp` constructor:

```
-- APL.Monad
data EvalOp a
  = ...
  | TryCatchOp (EvalM Val) (EvalM Val) (Val -> a)
```

and extend `EvalOp`'s `Functor` instance appropriately. Also complete the definition of `catch`.

Lastly, add support for `TryCatchOp` effects to `runEval'` in `APL.InterpPure` and to `runEvalIO'` in `APL.InterpIO`; `runEval` and `runEvalIO` do not need to treat `TryCatchOp` identically; for example, when interpreting `TryCatchOp m1 m2 k` it's permissible for the effects of `m1` to be visible in `m2` in `runEvalIO` but not visible in `runEval` (or vice versa).

2.1 Task 2: Key-value Store Effects

In this task, you'll add `KvGetOp` and `KvPutOp` operations for reading and storing values from the key-value store, respectively. For example,

```
> put0 m = KvPutOp (ValInt 0) (ValInt 1) m
> get0 = Free $ KvGetOp (ValInt 0) $ \val -> pure val
> runEval $ Free $ put0 get0
([],Right (ValInt 1))
```

Start by extending the `EvalOp` type with the `KvGetOp` and `KvPutOp` constructors:

```
-- APL.Monad
data EvalOp a
  = ...
  | KvGetOp Val (Val -> a)
  | KvPutOp Val Val a
```

then, extend `EvalOp`'s `Functor` instance appropriately. Next, complete the definitions of `evalKvGet` and `evalKvPut` using `KvGetOp` and `KvPutOp`, respectively.

Finally, extend `runEval'` in `APL.InterpPure` to support `KvGetOp` and `KvPutOp` effects as follows:

- On `KvGetOp key k` effects, `runEval'` should lookup the key in the state (the function `lookup` will be useful). If the key is contained in the state with value `val`, continue interpreting on `k val`. Otherwise, fail by returning a `Left` with an appropriate error message.
- On `KvPutOp key val m` effects, `runEval'` should should insert the association `(key, val)` into the state. If the key already exists in the state, it should be replaced by the new association.

Using a Database File for the Key-Value Store

Rather than storing the key-value store in a pure manner and passing state around as a parameter during interpretation (as you did in `runEval`), in `runEvalIO` we can instead write the key-value to a database file on disk.

To keep things simple, the database will just be a simple text file. We need a way to read and write data from the database—the following functions are provided to do so:

```
-- APL.InterpIO
writeDB :: FilePath -> State -> IO ()
readDB :: FilePath -> IO (Either Error State)
```

Note that `readDB` returns an error in the form of a `Left`-expression when trying to read an invalid database; if this happens, your interpreter should simply propagate the error and return the `Left`-expression.

For simplicity, the database can only store `ValInt` and `ValBool` values; function values are **not** supported. Your implementation is **not** expected to handle storing/reading of `ValFun` values to/from the database.

Your task is to modify `runEvalIO'` so that it stores state (i.e., the key-value store) in a database file and to add support for `KvGetOP` and `KvPutOp` effects to `runEvalIO'`.

Specifically, extend `runEvalIO` to support `KvGetOp` and `KvPutOp` effects as follows:

- On `KvGetOp key k` effects, `runEvalIO'` should read the database in from `db` and then lookup the key in the database. As with `runEval`, return `Left` if the key doesn't exist in the database. Otherwise, pass the associated value to `k` and continue.

- On `KvPutOp key val m` effects, `runEvalIO'` should read the database in from the `db` file to get a value of type `State`; let's call this value `dbState`. It should then insert the association `(key, val)` into `dbState` to construct a new state `dbState'`. As with `runEval`, if the key `key` already exists, the corresponding association should be replaced by the new one. Finally, overwrite the `db` database file with `dbState'` by using `writeDB`.[2]

For your testing, remember that the database only supports `ValInt` and `ValBool` values; you do **not** need to add tests for storing/reading `ValFun` values. Also note that `runEvalIO` clears the database on each execution, so database values will **not** persist between invocations of `runEvalIO`.

[2]This isn't a particularly efficient way to insert a key-value pair—we do it this way for simplicity.

Missing keys

Having a database is great and all, but our computations still fail if we look for a key that isn't in the database. Since `runEvalIO` already uses `IO`, instead of failing, we can interactively prompt the user to specify the value of a missing key during interpretation.

Here's an example of how this should work:

```
> runEvalIO $ eval $ KvGet $ CstInt 0
Invalid key: ValInt 0. Enter a replacement: ValInt 5
Right (ValInt 5)
> runEvalIO $ evalKvGet $ ValInt 0
Invalid key: ValInt 0. Enter a replacement: ValBool True
Right (ValBool True)
```

Keys entered this way should *not* be added to the database.

Add support for this functionality to `runEvalIO'`. To do so, you'll need to modify how `runEvalIO'` interprets `KvGetOp key k` effects. When the key `key` doesn't exist in the database file, your implementation must print a message to the terminal saying that the given key is invalid and then prompt the user for a replacement. The entered replacement should either be a `ValInt` or a `ValBool`. All other inputs (notably, `ValFun` values) are **not** supported. To convert the input string into a `Val`, use the provided `readVal :: String -> Maybe Val` function in `APL.InterpIO`. When the input is invalid (i.e., not of the form specified above), `readVal` will return `Nothing`; in this case, your interpreter should fail with an appropriate message:

```
> runEvalIO $ eval $ KvGet $ CstInt 0
Invalid key: ValInt 0. Enter a replacement: lol
Left "Invalid value input: lol"
```

Hint: You can use the `prompt :: String -> IO String` function defined in `APL.InterpIO` to get input from the user.
Hint 2: To test your missing key handling, you can use `captureIO` to simulate input like so:

```
-- APL.Interp_Tests
testCase "Missing key test" $ do
  (_, res) <-
```

```
    captureIO ["ValInt 1"] $
      runEvalIO $
        Free $ KvGetOp (ValInt 0) $ \val -> pure val
  res @?= Right (ValInt 1),
```

Task 3: `TransactionOp` effect

Another thing we might want for our database is to have *transactional* or *atomic* writes to it. First, extend APL with a `Transaction` expression:

```
-- APL.AST
data Exp
  = ...
  | Transaction Exp
```

The semantics of evaluating `Transaction e` is to wrap evaluation of `e` within a transaction as defined below (and using `APL.Monad.transaction`). The result of evaluation is otherwise the result of evaluating `e`.

Next add the `TransactionOp` effect to the `EvalOp` type:

```
-- APL.Monad
data EvalOp a
  = ...
  | TransactionOp (EvalM Val) (Val -> a)
```

Notice that it has an `EvalM Val` payload—when this payload is interpreted, any effects that **change the state (i.e., the key-value store)** should be all-or-nothing: that is, they should only be manifested if the computation succeeds (i.e, it returns result `Right v`). If the computation fails (returns `Left e`), then the **state (i.e., the key-value store)** should be rolled back to the point it was at before the payload was executed. For example,

```
> goodPut = KvPut (CstInt 0) (CstInt 1)
> badPut = Let "_" (KvPut (CstInt 0) (CstBool False)) (Var "die")
> get0 = KvGet (CstInt 0)
> runEval $ eval $ Let "_" (Transaction goodPut) get0
([],Right (ValInt 1))
> runEval $ eval $ TryCatch (Transaction badPut) get0
([],Left "Invalid key: ValInt 0")
```

Note that if the enclosed computation fails, the error should be propagated. For example,

```
> runEval $ eval $ Transaction badPut
([],Left "Unknown variable: die")
```

Extend the `Functor` instance of `EvalOp` to support `TransactionOp` and, using `TransactionOp`, complete the definition of the `transaction` function.

As usual, extend `runEval'` with support for `TransactionOp` effects. To do so, you should only keep changes to the **state (i.e., the key-value store)** from executing the enclosed `EvalM Val` computation if it succeeds; otherwise continue execution with the prior state. You **must** include the result from any `PrintOp` effects that occurred before the failure (in the transactional computation) in the final output, regardless of whether or not the transactional computation succeeded:

```
> runEval $ transaction (evalPrint "weee" >> failure "oh shit")
(["weee"],Left "oh shit")
```

You must also correctly handle **nested transactions**. For example,

```
> runEval $ eval $ Let "_" (Transaction
                             (Let "_" goodPut
                              (TryCatch (Transaction badPut)
                                        (CstBool True))))
                  get0
([],Right (ValInt 1))
> runEval $ eval $ Let "_" (TryCatch (Transaction
                                       (Transaction badPut))
                                      (CstBool True))
                  get0
([],Left "Invalid key: ValInt 0")
```

Next, add support for `TransactionOp` to `runEvalIO'`. You should only manifest writes to the database if the whole transactional computation succeeds. To do this, before execution of the transactional computation, make a temporary copy of the database file. To make temporary databases, use the `withTempDB` function, which creates a fresh temporary database, passes it to a function returning an `IO`-computation, executes the computation, deletes the temporary database, and finally returns the result of the computation:

```
-- APL.InterpIO
withTempDB :: (FilePath -> IO a) -> IO a
withTempDB m = do
  tempDB <- newTempDB -- Create a new temp database file.
  res <- m tempDB     -- Run the computation with the new file.
  removeFile tempDB   -- Delete the temp database file.
  pure res            -- Return the result of the computation.
```

Note that `newTempDB` ensures that the database `tempDB` is fresh; i.e. that there are no other files named `tempDB`. You **must** use `withTempDB` to create temporary databases; **do not** use `newTempDB`. To copy a database, use the `copyDB` function:

```
-- APL.InterpIO
copyDB :: FilePath -> FilePath -> IO ()
```

During execution of the transactional computation, perform all writes and reads on the temporary database file. If the computation succeeds, subsequently copy the temporary database to the actual database and continue interpreting. If it fails, simply continue interpreting. As before, you must also correctly handle nested transactions. **Hint:** The function you pass `withTempDB` should call `copyDB` to copy the database to the temporary database. If the transactional computation succeeds, you will have to call `copyDB` again to copy the temporary database back.

Task 4: Breaking out of loops

In this task you will implement support for breaking out of loops, similar to the `break` statement in C. In contrast with the previous tasks, the main challenge here is that you have to design the representation of the necessary effects yourself.

First, add the following constructor to `Exp`:

```
-- APL.AST
data Exp
  = ...
  | Break Exp
```

The intended semantics of evaluating `Break e` is to first evaluate `e` to a value v, then immediately return from the innermost enclosing loop with the value v.

To do this, extend the evaluation function in `APL.Eval`. You will need to add a new case for `Break` and to modify the case for `ForLoop`. You should make use of the functions `looping` and `breakLoop` from `APL.Monad`, which are intended to be the the entry points for the effects.

Then add corresponding new effects to `EvalOp` and implement `looping` and `breakLoop` to use them.

Finally, modify the interpretation functions in `APL.InterpPure` and `APL.InterpIO` to handle the new effects you have added. These implementations will likely be quite similar.

Hints

The operational behaviour of breaking is quite similar how errors are propagated and and handled by `TryCatch`, and your implementation can be fairly similar.

Examples

```
> runEval $ eval $
  ForLoop ("p", CstInt 0) ("i", CstInt 100) $
  Let "_" (Break (CstBool True)) (Var "i")
([],Right (ValBool True))
> runEval $ eval $ Break (CstBool True)
([],Left "Break␣outside␣loop")
```

3 Code handout

The code handout consists of the following nontrivial files.

- `a4.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test runner. **Do not modify this file.**
- `src/APL/AST.hs`: AST definition, which you will need to extend slightly.

- `src/APL/Eval.hs`: An incomplete evaluator corresponding to the solution to the week 2 exercises. You should **replace the definition of `eval`** with your complete version of `eval` from your solution to assignment 2.

- `src/APL/InterpIO.hs`: Contains the incomplete IO-based `runEvalIO` interpreter.

- `src/APL/InterpPure.hs`: Contains the incomplete pure `runEval` interpreter.

- `src/APL/Interp_Tests.hs`: An interpreter test suite where you will add plentiful tests.

- `src/APL/Monad.hs`: Contains all things related to the evaluation monad. Note that some definitions from assignment 2 have moved from `APL.Eval` to `APL.Monad` in this assignment; e.g. `Val` and definitions related to the environment.

- `src/APL/Util.hs`: Utility functions needed for serialization, testing IO, and making temporary database files. You can safely ignore this file. **Do not modify this file.**

4 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?

- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

4.1 The structure of your report

Your report must be structured exactly as follows:

Introduction: Briefly mention general concerns, any ambiguities in the problem text and how you resolved them, and your own estimation of the quality of your solution. Briefly mention whether your solution is functional, which test cases cover its functionality, which test cases it fails for (if any), and what you think might be wrong.

A section answering the following numbered questions:

1. Consider interpreting a `TryCatchOp m1 m2 k` 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?
   (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 changing the interpreters**, is it possible to have different behavior where the key-value store effects in `m1` are **invisible** in `m2`? If so, how? If not, why not?
2. Why is `TransactionOp` not defined as follows?

```
data EvalOp a
 = ...
 | TransactionOp (EvalM ()) a
```

   What problems might arise with this definition? Would it make 'TransactionOp' completely useless?
3. Why is `TryCatchOp` not defined as follows?

```
data EvalOp a
  = ...
  | TryCatchOp a a
```

   What problems might arise with this definition?

All else being equal, **a short report is a good report**.

5 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, structured as specified in the report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.

6 Assessment

You will get written qualitative feedback, and points from zero to four. There are no resubmissions, so please hand in what you managed to do, even if you have not solved the assignment completely.