Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 8

In this exercise we are going to look at monad transformers. For this purpose we need two libraries:

1. the `transformers` library contains the `MonadTrans` type class and instances for certain monads, e.g. `StateT`, `ReaderT`, and `WriterT`.
2. the `mtl` library builds on top of the `transformers` package and provides type classes for each monad transformer, which automatically handles `lift`ing and allows working with monad transformers in a more abstract style.

In the first exercise, we will work with the `transformers` library, which behaves in the same way as we saw in the lecture. In the second exercise, we will use the `mtl` library to rewrite the programs from the first exercise in a more idiomatic way.

We recommend using separate files for the two exercises, as the `mtl` library is supposed to be used instead of `transformers` and hence there is a certain amount of name overlap between the modules of both libraries.

Exercise 1 (The `transformers` package)

The `transformers` library behaves as we have seen in the lecture. If we have a monad transformer stack, e.g. `StateT Int IO`, then we can use `StateT` computations directly, but to use `IO` computations, we need to inject them into `StateT Int IO` by using `lift`.

1. Make yourself familiar with the `transformers` library by checking out the hackage documentation: https://hackage.haskell.org/package/transformers

   Of particular interest to us are

   - `Control.Monad.Trans.Class`, which contains the `MonadTrans` type class;
   - `Control.Monad.Trans.Except`, which contains a transformer for the `Either e` monad;
   - `Control.Monad.Trans.State`, which contains a transformer for the `State s` monad. The `StateT` monad transformer comes in two flavors: lazy and strict, which determines whether lazy evaluation should be used to thread the state through the computations or not. Most of the time, you want to use the strict version, where `put` computations will force their argument to be evaluated, as otherwise it can be easy to run into memory leaks.

2. Rewrite the todo-list program from sheet 4 exercise 2.3 such that it uses the `StateT [String] IO` monad instead of explicitly threading the todo-list `[String]` through the recursion.

   You can base your code on the solution for sheet 4, and for example start the implementation as:

   ```
   todo :: IO ()
   todo = evalStateT run [] where
     run :: StateT [String] IO ()
     run = ...
   ```

3. In the code supplement you can find a variation of the implementation of the while-language that we saw in exercise sheet 6.

   It differs in two ways from the original presentation:

   - the original presentation had a `VError` constructor for values, which was used to signal runtime errors, e.g. if the program tried to add an integer and a bool. The new presentation does not model runtime errors as values, but instead the interpreter can fail with an exception by returning `Either RuntimeError Val`.
   - the new presentation additionally has a `print` expression `EPrint`, which can print values to the terminal, so `eval` also needs `IO`.

   The interpreter in the code supplement is written directly in the `IO` monad and deals with exceptions and state manually, i.e. without using them as monads. Your job is to rewrite the interpreter such that a single monad transformer is used, which supports state, exceptions, and IO, e.g.

   ```
   type EvalM a = ExceptT RuntimeError (StateT Env IO) a

   eval :: Expr -> EvalM Val
   ```

Exercise 2 (The `mtl` package)

Working with the `transformers` library is annoying for two reasons:

1. Only the computations of the outer monad can be used directly, and the actions of the inner monads need to be injected via `lift`. For example, to use an `IO a` computation as a `ExceptT RuntimeError (StateT Env IO) a` computation, we need to `lift` it first under the exception monad and then we need to lift it again under the state monad. This creates a lot of syntactic noise and requires counting.
2. The monad transformer stack used for a function tends to also infect helper functions. For example, if we have a function using the `ExceptT RuntimeError (StateT Env IO) a` monad, but a helper function only requires to change the `Env` state, but does not need to throw exceptions or perform IO, then we have two suboptimal solutions: Either we also use `ExceptT RuntimeError (StateT Env IO) a` for the helper function, which allows to use it easily in the outer function, but fails to capture that the helper function doesn't need IO or exceptions, or we could use `State Env` for the helper function, but then we would need to write extra boilerplate code to inject it into the `ExceptT RuntimeError (StateT Env IO) a` computation of the outer function.

The `mtl` library solves both these problems by defining a separate type class for each monad transformer. For example, for the state monad there is the class

```
class Monad m => MonadState s m where
    get   :: m s
    put   :: s -> m ()
    ...
```

The idea is that this class is instantiated for each monad transformer stack which contains a `StateT`, and then functions like `get` and `put` can be used directly and are automatically lifted to the right position in the monad transformer stack, which solves the first problem of the `transformers` library.

Furthermore, it allows writing functions more abstractly without fixing the exact monad transformer stack. For example, instead of writing

```
printState :: StateT Int IO ()
printState = do
  s <- get
  lift $ print s
```

we can write

```
printState :: (MonadState Int m, MonadIO m) => m ()
printState = do
  s <- get
  print s
```

The second function can be used with any monad which supports an `Int` state and `IO`. This implies, that we can use it in a `StateT Int IO ()` monad, but also in larger monad transformer stacks like `ExceptT String (StateT Int (ReaderT Bool IO)) ()`.

Your tasks are

1. Make yourself familiar with the `mtl` library by checking out the hackage documentation: https://hackage.haskell.org/package/mtl

   Of particular interest are again

   - `Control.Monad.State.Strict`, which contains the `StateT` monad and the `MonadState` type class;
   - `Control.Monad.State.Except`, which contains the `ExceptT` monad and the `MonadError` type class;
   - `Control.Monad.IO.Class`, which contains the `MonadIO` type class. This module is actually defined in the standard library (the `base` library) but conceptionally also belongs to `mtl`. As the standardlibrary was not written with `MonadIO` in mind, you still need to use the `liftIO` function from the `MonadIO` type class, to lift `IO` actions into a monad supporting `IO`. We recommend to write wrapper functions for the required regular `IO` functions, e.g.:

     ```
     putStrLn' :: MonadIO m => String -> m ()
     putStrLn' = liftIO . putStrLn
     ```

2. Rewrite the todo-list program and the interpreter from the first exercise using the `mtl` approach. Eliminate all `lifts` by using the functions provided by the monad type classes, and work with type class constraints instead of fixing a particular monad transformer stack as shown for `printState` above.