Prof. Dr. Peter Thiemann
Janek Spaderna
janek.spaderna@pluto.uni-freiburg.de

Winter Term 2022/23
Issued 2023-01-23
Discussion on 2023-01-27

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exercise Sheet 11

Please take 10 minutes to fill in the evaluation.

1 Monad Transformers

In this part of the exercise we will use various monads and monad transformers from the transformers library. Below is a handy table relating the "standard version" of the monads you know with the transformer versions.

Standard Monad	Transformer	Base Type	Combined Type
`Maybe`	`MaybeT`	`Maybe a`	`m (Maybe a)`
`Either`	`ExceptT`	`Either e a`	`m (Either e a)`
`Reader`	`ReaderT`	`r -> a`	`r -> m a`
`Writer`	`WriterT`	`(a, w)`	`m (a, w)`
`State`	`StateT`	`s -> (a, s)`	`s -> m (a, s)`
	`RWST`	combines `Reader`, `Writer`, `State`	

It is a very common occurrence with monad stacks that you have to embed an `IO` computation. It is possible to nest multiple calls to `lift` but the `MonadIO` class from `base` offers a more ergonomic interface. A single call to `liftIO` lifts an `IO` action into any transformer stack.

Exercise 1 (Computations with protected data)

We want to write a program that manipulates password protected data. In order to access the data, the user has to provide their password. Of course, access can fail if the password is wrong.

Assume the following interface to the data. (In practice, passwords should *never* be stored in plain text but only salted and hashed!)

```
type Password = String

newtype ProtectedData a = ProtectedData (Password -> Maybe a)

accessData :: Password -> ProtectedData a -> Maybe a
accessData pw (ProtectedData tryAccess) = tryAccess pw
```

1. Implement `run` and `access` for the monad stack `Protected`.

   ```
   type Protected s = MaybeT (Reader (ProtectedData s))

   run :: ProtectedData s -> Protected s a -> Maybe a

   access :: Password -> Protected s s
   ```

2. Instead of having to provide the password to every call of `access` we want to integrate with the `IO` monad to ask the user for their password and remember it so that the user is asked at most once.

   Write a new type alias `ProtectedIO` and implement `runIO` for the adjust transformer stack.

3. Write a function `embed` to turn a `Protected s a` computation into a `ProtectedIO s a` computation.

4. Implement `accessIO` with the behaviour specified above.

Exercise 2 (Structured Logging)

With the `Writer` monad and its transformer actions can emit output. Output from multiple actions is automatically combined using the `Monoid` instance.

In this exercise we will use the `Writer` monad to accumulate log messages. However, we will not only collect a list of strings but provide a functions to structure them into sections. In a second step we will extend our logging functions to include time stamps. For this you will have to include the time library in your package dependencies.

We start from this basic structure of our log. It is generic over the type of messages and how we label sections.

```
data Item s m = Msg m | Section s [Item s m]
  deriving (Show)

type Log s m = [Item s m]

type Logging s m = Writer (Log s m)
```

1. Implement these functions:

   ```
   -- Log a single message.
   log :: m -> Logging s m ()

   -- Group the nested messages in a section.
   section :: s -> Logging s m a -> Logging s m a

   -- Extract the final result with the log messages.
   runLogging :: Logging s m a -> (a, Log s m)
   ```

2. To annotate all messages with time stamps we use

   ```
   type StampedLog s m = [Item (UTCTime, s, UTCTime) (UTCTime, m)]
   ```

   Import `Data.Time` to get access to all time-related functionality required for this exercise. The current time can be retrieved using `getCurrentTime`.

   To be able to use `getCurrentTime` we have to adjust the monad stack because `getCurrentTime` runs in the `IO` monad. Define the type synonym `StampedLogging` with a suitable adjusted monad stack.

   Implement versions of `log`, `section` and `runLog` using time stamps. The adjusted version of `section` should include two time stamps: one taken before executing nested computation, one after. You can test your implementation by using `threadDelay`.

2 GADTs

In this second part we will work with GADTs. This requires the language extension of the same name. Add this line at the top of your file:

```
{-# LANGUAGE GADTs #-}
```

Exercise 3 (Safe Lists)

Using `head` with lists in Haskell can lead to hard-to-debug errors. If the argument is a empty list the function will throw an exception at run time.

1. Define a list-like data structure called **`SafeList`** which supports a "safe" `head` operation. "Safe" in this context means that invalid inputs don't crash the program. Instead, the type checker should reject calls to `safeHead` with an empty **`SafeList`** as the argument. Additionally, write a corresponding `safeLast` function.

   ```
   safeHead (Cons 2023 Nil)  -- ok
   safeHead Nil              -- type error
   ```

2. Write a function `safeAppend`. Ensure that applying `safeHead` to the result of appending two empty lists gives a type error. But what other troubles are not able to resolve?