TDA342
Report for Replay Monad - Part I

QUFEI WANG
JACOB J. NILSSON

March 20, 2020

Task 1

File *Replay.hs* includes the implementation details of the *Replay* monad. The most important data type in this file is the 'replay monad' transformer, *ReplayT*, which is defined as a function. It accepts two *Traces*, where the first represents the accumulated *Results* and *Answers* and the second represents the unconsumed input *Results* or *Answers*. The result of the function is a value of type *Either* in the underlying monad. It is either a question with the currently accumulated answers and results, or a computation result with two traces that are prepared for further operations.

```
newtype ReplayT m q r a = ReplayT {replayT :: Trace r ->
    Trace r -> m (Either (q, Trace r) (a, Trace r, Trace r
    ))}
```

Type constructor *'ReplayT m q r'* is a monad, the basic functions *return*, (>>=) are defined as follows:

```
return' :: Monad m => a -> ReplayT m q r a
return' a = ReplayT $ \before after -> return $ Right (a,
     before, after)

bind :: Monad m => ReplayT m q r a -> (a -> ReplayT m q r
     b) -> ReplayT m q r b
bind r1 f = ReplayT $ \before after -> do
  a1 <- replayT r1 before after
  case a1 of
    Left l -> return (Left l)
```

```
    Right (a, t1, t2) -> replayT (f a) t1 t2
```

return' takes a value and makes it the result of the computation, leaving the input traces untouched. *bind* first runs the monad represented by its first parameter, and inspects the result. If the result is *Left l*, which means a question is raised without an answer, then the result is returned for the whole computation, ignoring the function that follows. If the result is $Right(a,\ t1,\ t2)$, then traces $t1$ and $t2$ are used as the input parameters to evaluate the function returned by $f\ a$.

The generalized version of the two important constructors *io*, *ask* are defined as follows:

```
iot :: (Show a, Read a, Monad m) => m a -> ReplayT m q r
    a
iot m = ReplayT $ \before after -> case after of
    [] -> do a <- m
             return $ Right (a, before ++ [Result (show a)
                 ], [])
    (x:xs) -> case x of
         Result s -> return $ Right (read s, before ++ [x],
              xs)
         _   -> error "type mismatch, expect: Result String,
             actual: Answer r"


askt :: Monad m => q -> ReplayT m q r r
askt q = ReplayT $ \before after -> do
    case after of
      [] -> return $ Left (q, before)
      (x:xs) -> case x of
                  Answer r -> return $ Right (r, before ++
                      [x], xs)
                  _        -> error "type mismatch, expect:
                      Answer r, actual: Result String"
```

Both of *iot*, *askt* pattern match on the second parameter $after$, which represents the upcoming *Results* or *Answers* that have not been consumed. For *iot*, if $after$ is an empty trace, then it performs the monad m, gets its result a, and makes it the part of the result of the computation. It also appends *Result* (*show* a) to the first parameter $before$, so that the following *ReplayT* computations (if any) could use it as an updated accumulated traces. If $after$ is of form $(x : xs)$, then a pattern match is made on x: If x is *Result* s, then it is converted to the target type by *read* s and returned. Otherwise there's a mismatch between the expected value and the actual value, an error occurs.

The same kind of reasoning can be applied on method *askt*, which we will not elaborate here.

What the generalized version of function *run*, which is *runt*, does is simply invoking the function of *ReplayT* with *emptyTrace* as the first paramter, and the acutal input as the second paramter. When the result is a *Left*, it is returned, otherwise it is curtailed to the appropriate type and returned.

```
runt :: Monad m => ReplayT m q r a -> Trace r -> m (
    Either (q, Trace r) a)
runt rt t = let m = replayT rt emptyTrace t in
  do a1 <- m
     case a1 of
       Left l -> return $ Left l
       Right (a2, t1, t2) -> return $ Right a2
```

Other types and functions in *Replay.hs* are either common auxiliary or derived from the functions above, therefore we will not give them explanations here, the commnets in the source file should be sufficient.

Task 2

> **Question: Why is it not possible to make your transformer an instance of MonadTrans ?**

A data type which is an instance of $MonadTrans$ should have the type of kind $(* \to *) \to * \to *$, which can be seen as accepting a *monad* and a concrete type to return a concrete type. Because the type parameters in the definition of our transformer is a *monad* type m followed by three concrete types q, r and a, it is impossible to make our transformer an instance of $MonadTrans$. However, one can verify that our transformer behaves exactly like a $'MonadTrans'$ given function $liftR$ as the implementation of $lift$ by safisfying the following laws:

- $lift\ .\ return = return$
- $lift\ (m >>= f) = lift\ m >>=\ (lift\ .\ f)$

Task 3

We included four test suites into our library, which can be found under the directory *test*. Case '$test - replay - io$' is just the original test file downloaded from Canvas. Case '$test - replay - state$' uses similiar testing code as $test - replay - io$, the difference is that it replaces the IO monad with a $State$ monad that is obtained by applying the type constructor $State$ (from $Control.Monad.State.Lazy$) on Int. The inner state is coded to record the number of invocations of method iot, so that the semantics of function $runProgram$ remains unchanged, even though its signature has changed to a non-monadic style $Program \rightarrow Input \rightarrow Result$.

Case '$test - replay - monadLaws$' just uses the same testing frame from '$test - replay - io$' to verify that $ReplayT$ abides by monad laws. Case '$test - replay - quickcheck$' uses $QuickCheck$ to generate random test cases to verify the monad laws. It does this by producing random values of needed types using custom generator, and verifying the monad laws as properties.