AFP Lab 2 – Replay Monad

Simon Robillard Inari Listenmaa

2015-02-25

The first section of this document describes the Replay monad. Second part describes the web DSL and briefly explains our example application.

1 The Replay Monad

Our Replay monad is a monad transformer of the following type:

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

The computation produces a trace that can be used as an argument for future replays. `q` is the type of question that can be asked of the user, `a` is the return type of the underlying monad `m` and `r` is the type of elements recorded in traces.

1.1 Traces

The trace contains two lists of elements, wrapped in type `Item`. When starting a replay with a non-empty string, all items are in the `todo` list. Whenever an item is consumed, it is moved to the `visited` list.

```
data Trace r = Trace { visited :: [Item r]
                     , todo    :: [Item r]
                     }
```

The datatype `Item r` has constructors for answers and results. `Answer r` is the answer of the type `r`, given by the user. `Result String` is the result of a computation in the underlying monad, stored as a string. The type `r` must have `Show` and `Read` instances, in order to be stored in the `Item r` type.

```
data Item r = Answer r | Result String
```

For handling traces, the module exports `emptyTrace`, the default value for an empty trace, and the function `addAnswer` to add answers to the trace.

```
emptyTrace :: Trace r
addAnswer  :: Trace r -> r -> Trace r
```

1.2 Replay API

The API provides the following functions:

```
ask   :: q -> ReplayT m q r r
run   :: ReplayT m q r a -> Trace r -> m (Either (q, Trace r) a)
liftR :: (Monad m, Show a, Read a) => m a  -> ReplayT m  q r a
io    :: (         Show a, Read a) => IO a -> ReplayT IO q r a
```

`ask` takes a value of type `q` and returns a Replay computation which expects an answer from the user or from the trace. `run` takes a Replay computation, a trace, and tries to run through the whole

The intended usage is that whenever a computation is stopped, the `run` function returns the latest question with the used trace. The application using the replay monad will prompt the user with the question, add the answer to the trace, and run the Replay monad again.

We implement the function `liftR` to lift a computation in the underlying monad and add it to the trace. `io` is a specialised version, for IO as the underlying monad. However, Replay monad cannot be made an instance of MonadTrans, because `liftR` doesn't satisfy the monad transformer laws:

```
lift . return == return
```

Our definition of return doesn't check the trace, but `liftR` does. Given e.g. a malformed trace, the resulting monads will have different behaviors.

```
lift (m >>= f) == lift m >>= (lift . f)
```

Here lift occurs once on the left, and twice on the right. Using the same reasoning, on the LHS the trace is modified once, and on the RHS twice.

To demonstrate the behaviour, we run some examples with the function `runningDebug` (in the file `Example.hs`), which prints the trace.

```
> runningDebug $ liftR $ return 3 >>= (\n -> return (n+1))
(4,Trace {visited = [Result "4"], todo = []})

> runningDebug $ liftR (return 3) >>= liftR . (\n -> return (n+1))
(4,Trace {visited = [Result "4",Result "3"], todo = []})
```

1.3 Cut

In order to optimise the replay monad, we implement a cut operation. The trace produced by that version will contain intermediary results only if the final result has not been computed.

We add a third constructor `Cut` to the type `Item` stored in trace. It stores the already computed result as a string.

```
data Item r = Answer r
            | Result String
            | Cut String
```

If we begin a computation with a cut, we check if the trace contains a `Cut` value to start with. If so, we know that the trace has been to the end of this computation before, and we can return that value safely, without calling `runReplayT`.

```
case todo t of
    --Cut s : return s, don't do computations
    _ -> do (qora, t') <- runReplayT ra t
            case qora of
              Right a -> return (Right a, addCut (show a) emptyTrace)
              Left q  -> return (Left q, t')
```

If the values in the trace are answers or results, we must first go through the computation the normal way. If we reach the end, we return the final value, with a trace that only contains that value. If we don't reach the end, we continue with the normal computation and accumulate answers and results to the trace.

Below is some demonstration of how this works:

First test, run `cut` with an incomplete trace. The function returns a normal trace, since it didn't reach the end.

```
> run (cut askAge) (addAnswer "1900" emptyTrace )
Left ("What year is it now?"
     ,Trace {visited = [Answer "1900"], todo = []})
```

Second test, run `cut` with a complete trace. (We use `runDebug` since we want to see the trace, even though the computation is ended.) It returns the final value, and a trace with only that value in it.

```
> runDebug (cut askAge) (addAnswer "1970" $
                         addAnswer "1900" emptyTrace )
(Right 70,Trace {visited = [Cut "70"], todo = []})
```

Third test, running `cut` with a cut trace. It returns the right value, and doesn't change the cut trace.

```
> runDebug (cut askAge) (addCut "70" emptyTrace)
(Right 70,Trace {visited = [Cut "70"], todo = []})
```

2 Web Programming

In the second part, we use the Replay monad to build web applications. We send HTML forms to the user using 'ask', and the answers are stored in the trace of the Replay monad.

2.1 A library for web forms

The type of our `Web` monad is `ReplayT` with `IO` as the underlying monad. We model the web forms with the type `Question`: a list of fields and a description.

```
type Web a = Replay Question Answer a

data Question = Question { par :: Text
                         , fields :: [Field]
                         }

type Answer = Map Text Text

data Field = Field { ident :: Text
                   , description :: Text
                   , visible :: Bool
                   }
```

We use the function `runWeb` to excract answers, using the Scotty function `param` and the identifiers of the fields. In case the user hasn't answered all the fields, `runWeb` sends the form again. In order to save the results for an application with many different web pages, we store the trace in a hidden field with the identifier `trace`.

2.2 An interesting web program

We have implemented a browser interface to execute some shell commands.

The main functionality happens in the function `coolShell`. It takes as an argument a piece of text and an integer, which works as a counter to produce unique field name. `coolShell` runs in a loop, and each time it outputs a HTML page with the argument text inserted in it, and increments the ID number. The generated forms asks the user to type in a unix command. We check that the user input is a legal command, and execute it with `runProcess`, showing the results to the user.