TDA342
Report for Replay Monad - Part II

QUFEI WANG
JACOB J. NILSSON

March 20, 2020

Task 1

The 'web form' functionality is implemented in *ReplayWeb.hs* under the directory *src*, and an executable web application based on it is defined in *Main.hs* under the directory *app*.

Type *Web* has a signature

$$type\ Web\ a = Replay\ Form\ Answer\ a$$

which is a *Replay monad* with type *Form* playing the role of the question, *Answers* playing the role of the answer and a specific type *a* being the type of the result of this *Replay monad* computation.

Type *Form* is a list of *Question*, and type *Question* is an abstraction for the *input* element from html.

```
data Question = Q {desc :: String, qseq :: Int, verify ::
    [String -> Maybe String]}

type Form = [Question]
```

The meanings of the fields of *Questoin* are explained as follows:

- *desc*: The *string* representation of the question itself.
- *qseq*: The sequence of the question, which is used to generate an identity for its corresponding *input* element.

- *verify*: A list of functions used to verify the input data, which can be seen as an answer to the question provided by the user.

We define *Form* in this way such that we can categorize the questions into different groups. We display one group of questions each time, so that our application can be seen as proceeding in several phases. Due to the limit of time, we consider only element $< input\ type = "text" >$ as the way of interaction, other types of form elements, such as *select*, *textarea*, different types of *input*, *etc.*, can be added to our language by specifying proper value constructors and fields. To illustrate this possibility, a piece of code is shown here:

```
type Form = [FormElement]

data FormElement = InputText {desc :: String, qseq :: Int
    , verify :: [String -> Maybe String]}
  | InputCheckBox {desc :: String, qseq :: Int, val ::
      String}
  | Textarea {qseq :: Int, row :: Int, col :: Int}
```

As opposed to *Form*, *Answers* is list of *QAnswer* where a *QAnswer* is just an answer to a *Question*. *QAnswer* takes the form of a tuple to keep track of the corresponding *Question*.

```
type Answers = [QAnswer]

type QAnswer = (Int, String)
```

runWeb is the function that serves as the 'server' part of our application. It takes a *Web PersonRecord* monad and returns the user with a group of questions and gathers the answers from the user to the *Web* monad. Here, a *PersonRecord* is just a data structure to hold the verifed answers we get from the user. We just give the signature of the function *runWeb* here, a detailed explanation will be provided later.

```
data PersonRecord = PR {
  prName :: String,
  fstInt :: Int,
  favLang :: String,
  prPet  :: String,
  sndInt :: Int } deriving (Show, Read)

runWeb :: Web PersonRecord -> ActionM ()
```

Task 2

The *cut* function is implemented in *SReplay.hs*. Module *SReplay* is an upgraded version of *Replay*, with the semantics of all existing types and functions unchanged plus the functionality of *cut*. The modifications are listed as follows:

- The notion of 'step'(which is also a kind of *state*) was added into the semantic of *ReplayT*. The idea is that, if every computation is tagged with a sequence number, then by marking out a range of computations with a final result, we can cut the trace behaviour in this range of computations by simply providing the tagged result.

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

-- | State of the monad, used to identity the current
    step of computation.
type Step = Int
```

- Type *Trace* was enhanced with two other fields, *tags* and *step*. *tags* is used to mark a range of computations that could be cut, while *step* just indicates the current computation step.

```
data Trace r = Trace {items :: [(Step, Item r)], tags
    :: Map Step (Step, String), step :: Step}
  deriving (Show, Read)
```

- Implementations of functions *liftR*, *return'*, *bind*, *iot*, *askt* are modified to adjust to the semantics of *step*.

```
liftR :: (Monad m, Show a, Read a) => m a -> ReplayT
   m q r a
liftR m = ReplayT $ \step before after ->
    do a <- m
       return $ (Right (a, before, after), step)

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

bind :: Monad m => ReplayT m q r a -> (a -> ReplayT m
    q r b) -> ReplayT m q r b
bind r1 f = ReplayT $ \step before after -> do
  (a1, step') <- replayT r1 step before after
```

```
  case a1 of
    Left l -> return (Left l, step')
    Right (a, t1, t2) -> replayT (f a) (step' + 1) (
        stepIncr t1) (stepIncr t2)

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

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

- An implementation of *cut* was added. Code of this function and comments showing how it works can be found at line 145 in the source file.
- Several auxiliary functions, such as *createTag*, *deleteTag*, *shiftTag*, *etc.*, are added, which are quite easy to understand and we won't elaborate here.

A test case for the new function *cut* is included in the file *TestCut.hs* under the directory *test*. The idea is to run a program twice, the first time we run it in a normal way, and the second time we annotate part of this program with function *cut* and run it again. We compare the running results with the expected correct ones to ensure that the desired effects and semantics of function *cut* are achieved.

```
type Program =  Replay String String Int
```

```
type Result  = ((Int, Int), Map.Map Int (Int, Int))

type Input   = [String]

-- | Checking a test case. Compares expected and actual
    results.
checkTestCase :: TestCase -> IO Bool
checkTestCase (TestCase name i r p) = do
  putStrLn $ name ++ ": "
  r' <- runProgram p i
  if r == r'
    then putStrLn "ok" >> return True
    else putStrLn ("FAIL: expected " ++ show r ++
                   " instead of " ++ show r')
         >> return False

-- | Running a program.
runProgram :: Program -> Input -> IO Result
runProgram prog inp = play emptyTrace inp
  where
    play t inp = do
      r <- runt' prog t
      case r of
        Right (x, t1, t2) -> case inp of
          [] -> let lenOfItems = length . items $ t1
                    transTag   = Map.map (\(step, str) ->
                                   (step, read str)) (tags t1)
                in return ((x, lenOfItems), transTag)
          _ -> error "too many inputs"
        Left (_, t') -> case inp of
          []       -> error "too few inputs"
          a : inp' -> play (addAnswer t' a) inp'

-- | List of interesting test cases.
testCases :: [TestCase]
testCases =
  [ TestCase
    { testName    = "test-without-using-cut"
    , testInput   = ["", "", "", ""]
    , testResult  = ((1, 10), Map.fromList [])
    , testProgram = do
        part1
        r2 <- part2
        part3
        return r2
    }
```

```
  , TestCase
    { testName    = "test-using-cut"
    , testInput   = ["", "", "", ""]
    , testResult  = ((1, 6), Map.fromList [(4, (8, 1))])
    , testProgram = do
        part1
        r2 <- cut part2
        part3
        return r2
    }
  ]

part1 :: Program
part1 = do
  io . putStrLn $ "begin part1"
  ask "question 1 in part 1"
  io . putStrLn $ "end part1"
  return 0

part2 :: Program
part2 = do
  io . putStrLn $ "begin part2"
  ask "question 1 in part 2"
  ask "question 2 in part 2"
  io . putStrLn $ "end part2"
  return 1

part3 :: Program
part3 = do
  io . putStrLn $ "begin part3"
  ask  "question 1 in part 3"
  io . putStrLn $ "end part3"
  return 0
```

Notice that the result of running a *Program* is presented in a way that the first part is a tuple of type (*Int*, *Int*), with the first *Int* being the final result of the computation, and the second being the number of items in the trace left after running the *Program*. The second part of the result is a transformed version of the value of the field *tags* in the trace after running the *Program*. It records the possible ranges of computations that could be cut and the final result of these ranges.

The expected result of the first case means that the final result of the computation is 1, which is returned by running *part*2, and the number of items left in the trace is 10, since each *io* or *ask* leaves an item in the trace. Because

there's no *cut* used, the *tags* returned is empty.

The expected result of the second case means that the final result of the computation remians the same with the first case, which is 1, but since *part*2 is annotated with *cut* this time, the items in the trace reduced from 10 to 6, *i.e.*, four items from *part*2 are reduced or *cut*, and the range of computations from step 4 to 8 are marked and the result of running *part*2 is saved.

Task 3

The interactive web program is backed by the function *runWeb*.

```
runWeb :: Web PersonRecord -> ActionM ()
runWeb web = do
  t <- getTraces
  play t True
  where play t first = do
          b <- getSubmit
          r <- liftIO (run web t)
          case r of
            Right pr -> finishPage pr
            Left (q, t) ->
              if b && first
                then do
                  ga <- gatherAnswers q
                  let as = Prelude.map fst ga
                      ss = Prelude.map snd ga
                      ss' = catMaybes ss
                  if length ss' == 0
                    then play (addAnswer t as) False
                    else questionPage q as ss' t
                else questionPage q [] [] t
        gatherAnswers = mapM findAnswer
```

The logic of this function is pretty straightforward.

1. We get the trace from the page and run our *Replay* monad with the trace.
2. If we reach to a final result, we present it on the final page.
3. Otherwise, we make a difference among the following cases: 1) we are dealing with a browser refresh(usually happens when the *url* of our server is first visited), 2) we have just supplemented the *Trace* with the answers

provided by the user and replay the program, 3) we are dealing with a user submission.

(a) In case 1) and 2), we just return the question to the page, waiting for a further interaction from the user.

(b) In case 3), we gather the answers from the user input and check the validity of these answers. If there's no scolding message present, we add the answers as an answer to this question and replay the program. Otherwise, we return the question together with our scolding messages to the user.

In file *Main.hs*, we designed a program that asks the user two group of questions in separate stages and display the information about the user in the final page. It's not a very complicated or fancy program but has all the features we talked above. We also annotated the *io* and *ask* actions with function *cut* so that you can see the items in the *Trace* are shrinked in the server log.

```
main :: IO ()
main = scotty 3000 $ do
    get "/" serve
    post "/" serve
  where
    serve :: ActionM ()
    serve = runWeb $ prog
    prog  = do
      cut . io $ putStrLn "[info] About to run questions
          in form-1"
      ans1 <- cut $ ask form1
      cut . io $ putStrLn "[info] About to run questions
          in form-2"
      ans2 <- cut $ ask form2
      cut . io $ putStrLn "[info] About to return the
          result"
      return $ toPersonRecord ans1 ans2
    form1 :: Form
    form1 = [(Q "What's your name?" 0 [checkIfEmpty,
        checkLength]),
             (Q "What's your favourite integer?" 1 [
                 checkIfInteger, checkLength]),
             (Q "What's your favourite programming
                 language?" 2 [checkIfEmpty, checkLength])
                 ]
    form2 :: Form
    form2 = [(Q "What's the name of your pet?" 3 [
        checkIfEmpty, checkLength]),
```

```
            (Q "What's your second favourite integer?" 4
                [checkIfInteger, checkLength]),
            (Q "Do you enjoying this programme?" 5 [
                checkAffirmative])]
toPersonRecord :: Answers -> Answers -> PersonRecord
toPersonRecord a1 a2 = PR (takeAnswer a1 0) (read $
    takeAnswer a1 1) (takeAnswer a1 2)
                          (takeAnswer a2 0) (read $
                              takeAnswer a2 1)
takeAnswer :: Answers -> Int -> String
takeAnswer a i = snd $ a !! i
```