Introduction to state-based FRP

June 6, 2014

Have you ever wondered about applying a lens to an `IORef`?

```
lensMap :: Lens' a b -> IORef a -> IORef b
```

If this operation makes sense to you, read on to get the full story. We will add five operations as interesting as `lensMap` one-by-one. At the end we arrive to a fascinating FRP system which I call *state-based FRP*. Why did I choose this name? It will turn out at the end.

1 Applying a lens to a reference

1.1 Lenses

First I would like to clarify that we use Edward Kmett's lens notations. However, we need just a tiny fraction of the `lens` library. The needed definitions in a simplified form are the following:

```
type Lens' a b = (a -> b, a -> b -> a)

set :: Lens' a b -> a -> b -> a
set = snd

(^.) :: a -> Lens' a b -> b
(^.) = flip fst

united :: Lens' a ()
united = (const (), const)

_1 :: Lens' (x, y) x
_1 = (fst, \(_, y) x -> (x, y))
```

```
_2 :: Lens' (x, y) y
_2 = (snd, \(x, _) y -> (x, y))

(.) :: Lens' a b -> Lens' b c -> Lens' a c
-- I cheat here, see [?] to understand why function
-- composition can be used for lens compostion
```

1.2 How to apply a lens to an `IORef`?

Application of a lens to an `IORef` is only possible with a modified `IORef` definition:

```
type IORef' a = (IO a, a -> IO ())
```

`IORef'`s still can be created, read and written:

```
readIORef' :: IORef' a -> IO a
readIORef' = fst

writeIORef' :: IORef' a -> a -> IO ()
writeIORef' = snd

newIORef' :: a -> IO (IORef' a)
newIORef' a = do
    r <- newIORef a
    return (readIORef r, writeIORef r)
```

Lens application is now also possible:

```
lensMap :: Lens' a b -> IORef' a -> IORef' b
lensMap (get, set) (read, write) =
    ( fmap get read
    , \b -> do
        a <- read
        write $ set a b
    )
```

1.3 Usage example

A simple example how to use `lensMap`:

```
main = do
    r <- newIORef' ((1,"a"),True)
    let r' = lensMap (_1 . _2) r
    writeIORef' r' "b"
```

The values of `r` and `r'` are connected: whenever `r` is written `r'` changes and whenever `r'` is written `r` changes. At any time the following holds:

```
rv' = rv ^. _1 . _2
```

where `rv` and `rv'` are the actual values of `r` and `r'`, respectively.

1.4 What is `lensMap` good for?

It seems natural that if we have a reference to a state, we can build a reference to a substate of the state. I claim that `lensMap` allows to write code easier to compose. I give a try to verify my claim in the summary section.

2 Joining a reference

By joining a reference I mean the following operation:

```
joinRef :: IO (IORef' a) -> IORef' a
```

2.1 What is `joinRef` good for?

Suppose we have `mb :: IO Bool` and `r1, r2 :: IORef' Int`. With `joinRef` we can make a reference `r` which acts like `r1` or `r2` depending on the *actual* value of `mb`.

```
r :: IORef' Int
r = joinRef $ do
    b <- mb
    return $ if b then r1 else r2
```

`joinRef` allows more than just switching between two references dynamically. One can build a network of references with `lensMap` and make this network fully dynamic with `joinRef`.

2.2 Why is it called `joinRef`?

I call it `joinRef` because with another definition of `IORef'`, `Control.Monad.join` acts like `joinRef`!

```
type IORef'' a = IO (a, a -> IO ())
```

`IORef"` is isomorphic to `IORef'`:

```
convTo :: IORef' a -> IORef'' a
convTo (read, write) = do
    a <- read
    return (a, write)

convFrom :: IORef'' a -> IORef' a
convFrom r =
    ( fmap fst r
    , \a -> do
        (_, write) <- r
        write a
    )
```

`joinRef` is `join`:

```
joinRef :: IO (IORef'' a) -> IORef'' a
joinRef = join
```

3 Backward application of a lens to a reference

By backward lens application I mean the following operation:

```
extRef :: IORef' b -> Lens a b -> a -> IO (IORef' a)
```

3.1 Why is it called `extRef`? How is it backward lens application?

It is called `extRef` because an existing program state can be *extended* with it. Suppose that `r :: IORef' Int`. Suppose that we would like to double the possible values of `r`, i.e. we would like to extend the state referenced by `r` with a `Bool` value. We can do it with the following definition:

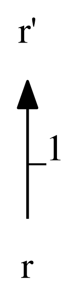
```
r :: IORef' Int
r = ...

do
    (r' :: IORef' (Int, Bool)) <- extRef r _1 (0, False)
```

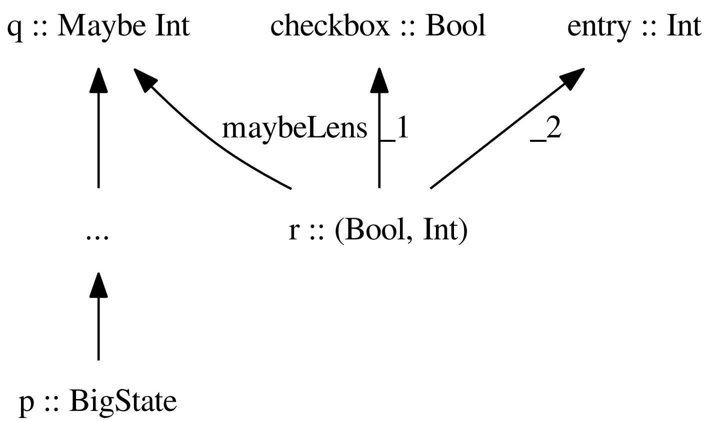
The third parameter of `extRef` determines the initial value of `r'`. If the value of `r` is `15` at the creation time of `r'` then the initial value of `r'` is `(15, False)`.

The values of `r` and `r'` remain connected: whenever `r` is written `r'` changes and whenever `r'` is written `r` changes. The connection between `r` and `r'` is exactly the same as if `r'` was created first and `r` was defined by

```
r :: IORef' Int
r = lensMap _1 r'
```

In this sense `extRef` is the inverse of `lensMap` and this is why I call it backward lens application.





3.2 Implementation of `extRef`

It turns out that `extRef` cannot be defined with the previous definitions of `IORef'`. It can be defined on this modified `IORef'` data structure instead:

```
data IORef''' a = IORef''
    { readRef          :: IO a
    , writeRef         :: a -> IO ()
    , registerCallback :: IO () -> IO ()
    }
```

`registerCallback` takes and `IO` action and stores it (there is one store per reference). `writeRef` calls all the stored actions after setting the reference value. We do not go into further details here. The source code of a complete implementation can be found in ...

3.3 Multiple monads for reference operations

The return type of `extRef` is `IO (IORef' a)` which can be turned into `IORef' a` by `joinRef`:

```
joinedExtRef :: IORef' b -> Lens a b -> a -> IORef' a
joinedExtRef r k a = joinRef (extRef r k a)
```

In fact `joinedExtRef r k a` is quite useless and it behaves wrongly (setting its value me have no effect). We would like to disallow this combination of `extRef` and `joinRef`, therefore we introduce different monad layers in which different reference actions are allowed.

So far the following three monad layers turned out to be handy to work with:

monad layer	allowed actions
ReadRef	reference read
CreateRef	reference read and creation
WriteRef	reference read, creation and write

From now on, we replace `IO` by either `ReadRef` or `CreateRef` or `WriteRef`. We replace `IORef'` by `Ref` too.

In the new system `joinRef` has limited availability:

```
joinRef :: ReadRef (Ref a) -> Ref a
```

The result of `extRef` is in the `CreateRef` monad:

```
extRef :: Ref b -> Lens a b -> a -> CreateRef (Ref a)
```

Thus `joinRef` cannot be applied after `extRef` and the above puzzle is solved.

3.4 `newRef` as a special case of `extRef`

Before making a summay, notice that `extRef` is so strong that `newRef` can be expressed in terms of it:

```
newRef :: a -> CreateRef (Ref a)
newRef = extRef unitRef united
```

Here `unitRef` can be any reference which has type `Ref ()`.

We add `unitRef` to the set of basic reference operations (it is a constant):

```
unitRef :: Ref ()
```

3.5 Summary so far

The discussed data types and operations so far are the following:

```
Ref       :: * -> *
ReadRef   :: * -> *   -- instance of Monad
CreateRef :: * -> *   -- instance of Monad
WriteRef  :: * -> *   -- instance of Monad

liftReadRef   :: ReadRef a   -> CreateRef a
liftCreateRef :: CreateRef a -> WriteRef a

unitRef  :: Ref ()
lensMap  :: Lens' a b -> Ref a -> Ref b
readRef  :: Ref a -> ReadRef a
joinRef  :: ReadRef (Ref a) -> Ref a
extRef   :: Ref b -> Lens' a b -> a -> CreateRef (Ref a)
writeRef :: Ref a -> a -> WriteRef ()
```

`liftReadRef` is handy because during reference creation we can read existing references.

`liftCreateRef` is handy because during reference write we can create references (a use case is to create a new reference and give it as a value of a reference-reference).

4 Connecting events to reference change

This operation is the last big step into the direction of a fully working FRP system:

```
onChange :: Eq a => ReadRef a -> (a -> CreateRef b) -> CreateRef (ReadRef b)
```

```
onChange :: ReadRef (CreateRef b) -> CreateRef (ReadRef b)
```

4.1 Semantics with examples

TODO

4.2 Variations

```
onChangeMemo :: Eq a => ReadRef a -> (a -> CreateRef (CreateRef b)) -> CreateRef (ReadRef b)
```

```
onChangeAcc :: Eq a => b -> ReadRef a -> (b -> a -> CreateRef b) -> CreateRef (ReadRef b)
```

TODO

5 Bindings to outer actions

```
registerCallback :: Functor f => f (Modifier m ()) -> m (f (EffectM m ()))
```

```
liftEffect :: ... a -> CreateRef a
```

5.1 Resource handling

```
onRegionStatusChange :: (RegionStatusChange -> m ()) -> m ()
```

data RegionStatusChange = Kill | Block | Unblock deriving (Eq, Ord, Show)
TODO

6 Summary

TODO

6.1 Comparison to existing Haskell FRP frameworks

reactive reactive-banana Elerea FranTk Sodium Elm netwire yampa iTask
TODO