Introduction to state-based FRP

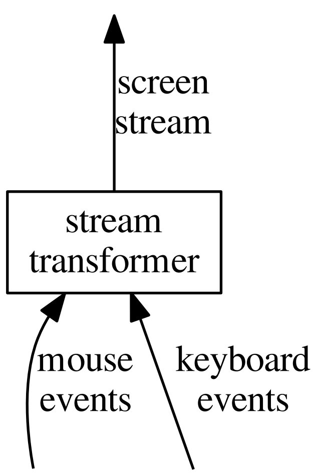
Draft

July 14, 2014

1 Motivation

1.1 Stream-based FRP

Interactive programs can be described as stream transformers. For example, interactive programs with a graphical user interface (GUI) can be described as stream transformers with keyboard and mouse events as input streams and a continuously changing screen as an output stream:

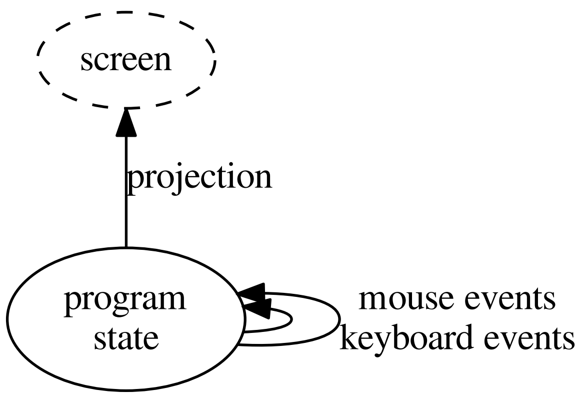


The goal of *functional reactive programming* (FRP) is to declaratively describe interactive programs. One key aspect of declarative descriptions that they are composable. This means that FRP descriptions of interactive programs can be decomposed into FRP descriptions of simpler interactive programs.

One possible composable description of interactive programs are stream transformers equipped with different combinators like horizontal and vertical composition. Let us call this system *stream-based FRP*.

1.2 State-based FRP

Stream-based FRP is not the only possible declarative description of interactive programs with a GUI. Another possibility is to describe the program state, to describe how mouse and keyboard events alter the program state and to project the program state onto the screen:



It is not obvious to generalize this description to arbitrary interactive programs, and to decompose this description into simpler parts. Let us call *state-based FRP* a system characterized by this goal.

State-based FRP is an alternative to stream-based FRP. In many cases, state-based FRP decomposition of interactive programs is simpler than their stream-based FRP decomposition.

This document gives an introduction to state-based FRP as implemented in the `lensref` package[1]. The interface of `lensref` is built around the reference data type. We start with an interface of basic reference operations and we add new operations step-by-step to the interface.

[1] http://hackage.haskell.org/package/lensref

2 Basic operations on references

2.1 The reference data type

A *reference* is an editable view of the program state.

A reference has an associated *value* for each program state.

A reference has also a *context*. The context tells what kind of effects may happen during reference write, for example. It also helps to distinguish references created in different regions. Not every reference context is valid; the `RefContext` type class classifies valid reference contexts:

```
class Monad m => RefContext m    -- reference contexts

instance RefContext IO
instance RefContext (ST s)
instance RefContext m => RefContext (ReaderT r m)
instance (RefContext m, Monoid w) => RefContext (WriterT w m)
...
```

The type of a reference is determined by its context type and the type of its possible values:

```
data RefContext m => Ref m a   -- abstract data type of references
```

2.2 Interface of the basic operations

The basic reference operations are *reference reading*, *reference writing* and *reference creation*. The interface of the basic reference operations is the following:

```
-- reference read action
readRef  :: RefContext m =>  Ref m a -> RefReader m a
-- reference write action
writeRef :: RefContext m =>  Ref m a -> a -> RefWriter m ()
-- new reference creation action
newRef   :: RefContext m =>  a -> RefCreator m (Ref m a)

data RefReader  m a  -- reference reader computation
data RefWriter  m a  -- reference writer computation
data RefCreator m a  -- reference creator computation

instance RefContext m => Monad (RefReader m)
instance RefContext m => Monad (RefWriter m)
```

```haskell
instance RefContext m => Monad (RefCreator m)

instance MonadTrans RefWriter
instance MonadTrans RefCreator

readerToWriter  :: RefContext m => RefReader m a -> RefWriter  m a
readerToCreator :: RefContext m => RefReader m a -> RefCreator m a
```

Reference reading returns the value of a reference for any program state. Given a reference `(r :: Ref m a)` and a value `(x :: a)`, reference writing changes the program state such that the value of `r` in the changed state will be `x`.[2] Given a value `(x :: a)`, new reference creation extends the program state with a new `a`-typed field initialized with `x` and returns a reference whose value is always the value of the new field in the program state.

Reference reader, writer and creator computations are abstract data types with `Functor`, `Applicative` and `Monad` instances (`Functor` and `Applicative` instances was left implicit for brevity). `RefWriter` and `RefCreator` are monad transformers. `RefReader` is not a monad transformer because no side effect is allowed during reference reading.

Reference writer and creator computations may involve reference reader computations: `readerToWriter` lifts reference reader computations to reference writer computations; `readerToCreator` lifts reference reader computations to reference creator computations.

The distinction between `RefReader`, `RefWriter` and `RefCeator` is necessary for operations introduced later.

2.3 Laws

The following laws are part of the interface.

2.3.1 Law 1: write-read

Let `(r :: Ref m a)` be a reference and `(x :: a)` a value. Reading `r` after writing `x` into it returns `x`, i.e. the following expressions have the same behaviour:[3]

```haskell
writeRef r x >> readerToWriter (readRef r)
    ~        -- write-read
writeRef r x >> return x
```

[2] This is an incomplete definition of reference writing because it does not define how reference writing changes the values of other references. We leave this question open for now.

[3] We say that two expressions has the same behaviour if they are replaceable in any context without changing the functional properties of the program (difference in resource usage is possible).

The write-read law is analogue to the set-get law for lenses. The following laws which are analogue to the get-set and set-set lens laws are **not required** in the `lensref` library.

The read-write law is **not required**:

```
readRef r >>= writeRef r
    ~        -- read-write
return ()
```

The write-write law is **not required**:

```
writeRef r x' >> writeRef r x
    ~        -- write-write
writeRef r x
```

2.3.2 Law 2: `RefReader` has no side effects

Let `(m :: RefReader m a)`. `m` has no side effects, i.e. the following expressions have the same behaviour:

```
m >> return ()
    ~        -- RefReader-no-side-effect
return ()
```

2.3.3 Law 3: `RefReader` is idempotent

Let `(m :: RefReader m a)`. Multiple execution of `m` is the same as one execution of `m`, i.e. the following expressions have the same behaviour:

```
liftM2 (,) m m
    ~        -- RefReader-idempotent
liftM (\a -> (a, a)) m
```

Laws 2 and 3 together implies that `RefReader` has no effects, i.e. it is isomorphic to the `Reader` monad.[4]

[4] http://stackoverflow.com/questions/16123588/what-is-this-special-functor-structure-called

2.3.4 Law 4: `RefCreator` has no extra side effects

Let `(c :: RefCreator m a)`. `c` has no side effects if `m` has no side effects, i.e. if

```
m >> return ()
    ~
return ()
```

holds for all `(m :: m)` then

```
c >> return ()
    ~       -- RefCreator-no-side-effect
return ()
```

Law 4 is similar to law 2 but stated for the `RefCreator` monad instead of `RefReader` and with and extra condition for the reference context.

Note that there is no law similar to law 3 for `RefCreator`, because `RefCreator` is not idempotent. For example, `(liftM2 (,) (newRef 14) (newRef 14))` and `(liftM (\a -> (a, a)) (newRef 14))` has different behaviour because the former creates two distinct references whilst the latter creates two entangled references.

2.4 Running `RefCreator`

```
runRefCreator
    :: RefContext m
    => ((forall b . RefWriter m b -> m b) -> RefCreator m a)
    -> m a
```

2.5 Examples

TODO

3 References connected by lenses

3.1 Lenses summary

We use Edward Kmett's lens notation. The needed definitions from the `lens` package are the following:

```
-- data type for lenses (simplified form)
type Lens' a b

-- lens construction with get+set parts
lens :: (a -> b) -> (a -> b -> a) -> Lens' a b

-- the get part of a lens, arguments flipped
(^.) :: a -> Lens' a b -> b

-- the set part of a lens, arguments flipped
set :: Lens' a b -> b -> a -> a

-- data type for isomorphisms (simplified form)
type Iso' a b

-- iso construction with to+from parts
iso :: (a -> b) -> (b -> a) -> Iso' a b

-- lens from anything to unit
united :: Lens' a ()

-- ad-hoc polymorphic tuple element lenses
_1 :: Lens' (x, y) x
_1 :: Lens' (x, y, z) x
_1 :: ...
_2 :: Lens' (x, y) y
_2 :: Lens' (x, y, z) y
_2 :: ...
_3 :: Lens' (x, y, z) z
_3 :: Lens' (x, y, z, v) z
_3 :: ...
...

-- function composition can be used for lens composition
(.) :: Lens' a b -> Lens' b c -> Lens' a c

-- conversion from isomorphisms to lenses is implicit
id :: Iso' a b -> Lens' a b

-- id is the identity isomorphism (and a lens too)
id :: Iso' a a
```

Utility functions used:

```
-- flipped fmap
<&> :: Functor f => f a -> (a -> b) -> f b
```

3.1.1 Use of improper lenses

Let `(k :: Lens' A B)`, `(a :: A)`, `(b :: B)` and `(b' :: B)`. Edward Kmett's three common sense lens laws are the following:

```
set k b a ^. k          ==  b               -- set-get
set k (a ^. k) a        ==  a               -- get-set
set k b (set k b' a)    ==  set k b a       -- set-set
```

The `lensref` library can deal with lenses which do not satisfy the get-set or the set-set laws. `lensref` calls these lenses improper lenses and uses the same `Lens'` type for them.

3.2 Lens connections

Let `(a :: Ref m A)` and `(b :: Ref m B)`. We say that `a` and `b` are connected by `(k :: Lens' A B)` iff the following holds:

- ```
  readRef b
      ~        -- lensMap-read
  readRef a <&> (^. k)
  ```

- For all `(y :: B)`

  ```
  writeRef b y
      ~        -- lensMap-write
  readerToWriter (readRef a) >>= writeRef a . flip (set k) y
  ```

3.3 The `Ref` category

Lens connection is a transitive relation, and every reference is connected to itself by the `id` lens, so references as objects and lens connections as morphisms form a category. Let us call this category `Ref`.

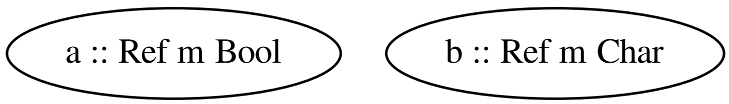
Some properties of the `Ref` category:

- Terminal object: `(r :: Ref m ())`
- The program state is an initial object. It can not be typed in Haskell because new reference creation changes the type of the program state.
- Pullbacks up to isomorphism?
  ```

4 Reference diagrams

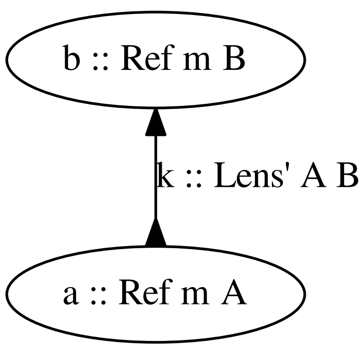
4.1 References

Ellipses denote references. The following diagram shows two independent references:



4.2 Lens connection

An arrow with and inverted arrow tail denotes a lens connection between references:

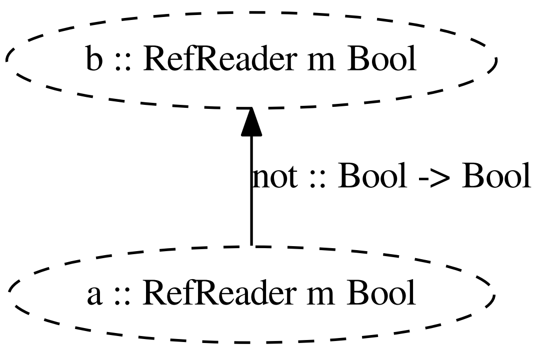


The layout of the nodes is not significant, but it tries to reflect the information dependency between them. Usually the value of node `b` is determined by the value of node `a` if there is an undirected path between them and `b` is not below `a`.

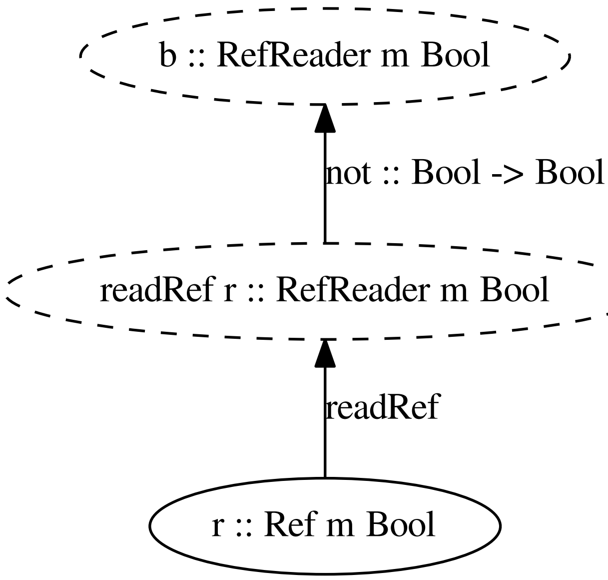
Connection by an isomorphism between references is a special case of connection by lenses. For better information dependency visualization, in case of connection by an isomorphism the references will be shown at the same level:

4.3 `RefReader` computations

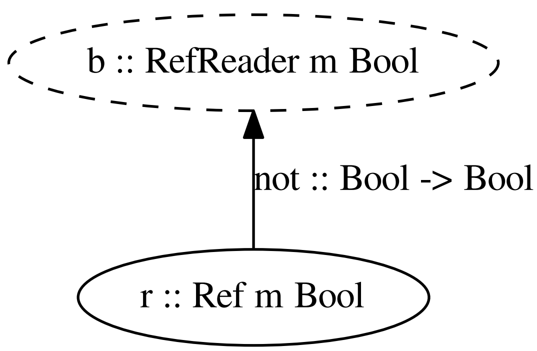
Dashed ellipses denote reference reader computations. Arrows between reference reader computations denote functors in the `RefReader` category:



The special `readRef` arrow connects `(r :: Ref m a)` to `(readRef r :: RefReader m a)`:



`readRef` may be left implicit, so the previous diagram can be simplified like this:



4.4 `RefWriter` computations

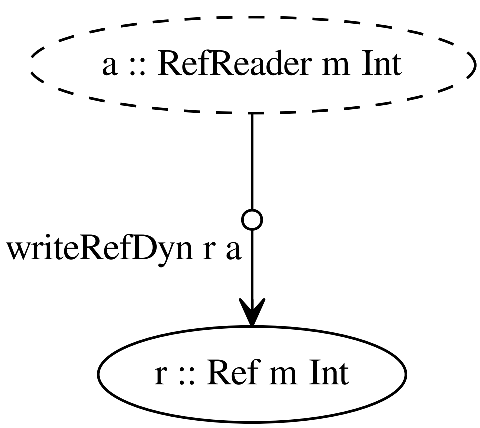
Recall the type of `writeRef`:

```
writeRef :: Ref m a -> a -> WriteRef m ()
```

`writeRef` can be generalized such that the written value is "dynamic", i.e. it is a `RefReader` value:

```
writeRefDyn :: Ref m a -> RefReader m a -> RefWriter m ()
writeRefDyn r m = readerToWriter m >>= writeRef r
```

`writeRefDyn` alone is enough to construct the needed `RefWriter` computations in many cases. `writeRefDyn` is denoted by an arrow from the `RefReader` computation to the reference:



4.5 Widget decorations

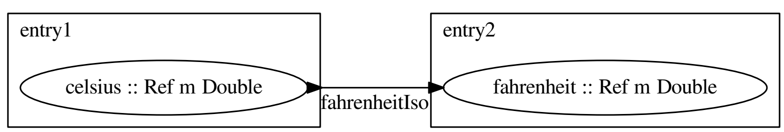
Groups of `Ref`, `RefReader` and `RefWriter` values can be connected to GUI widgets.[5] This connections may be shown as decorations on reference diagrams.

Some possible connetions are the following:

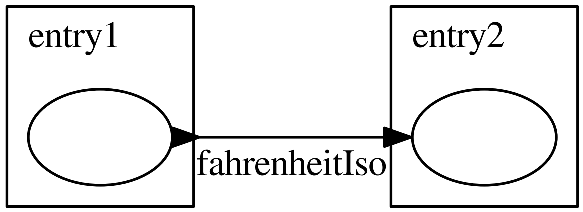
[5]See the `lgtk` library which is based on `lensref`.

- `(RefReader m String)` computations can be connected to dynamic labels. The label shows the actual return value of the computation.
- `(Ref m Bool)` values can be connected to checkboxes. Iff the value of the reference is `True`, the checkbox is checked. Note that this describes a two-way connection between the checkbox and the program state (changing the program state may alter the checkbox and vice-versa).
- References with basic types like `Int`, `Double` or `String` can be connected to entries.
- `(RefWriter m ())` computations can be connected to buttons. When the button is pressed, the computation is executed.
- `(RefReader m Bool)` computations can be attached to checkboxes, entries or buttons. The widgets are dynamically activated or deactivated whenever the computation returns `True` or `False`, respectively.
- `(RefReader m String)` computations can be attached to buttons. The return value of the computation is shown as a dynamically changing button label.

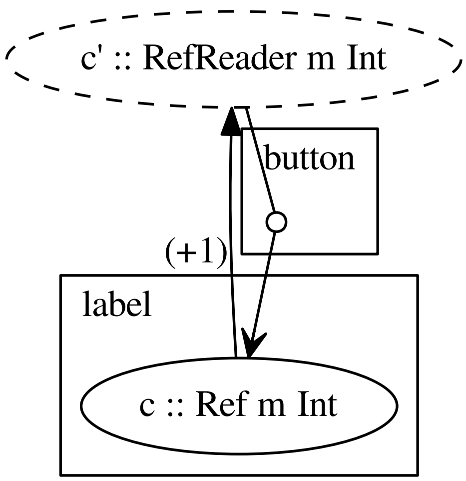
For example, a Celsius-Fahrenheit converter has two entangled `Double` value entries:



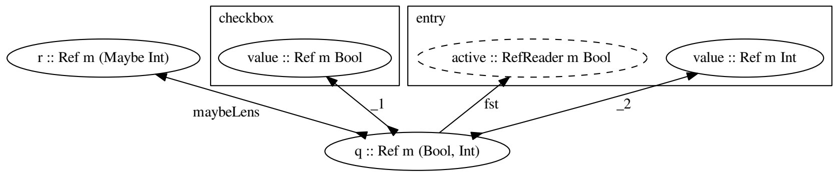
Note that the diagram contains superfluous information to improve readability. The following stripped diagram contains just enough information to describe the same interactive program:



A simple counter has an integer label and a button (here the reference `c` was converted implicitly to a `RefReader` value by `readRef`):

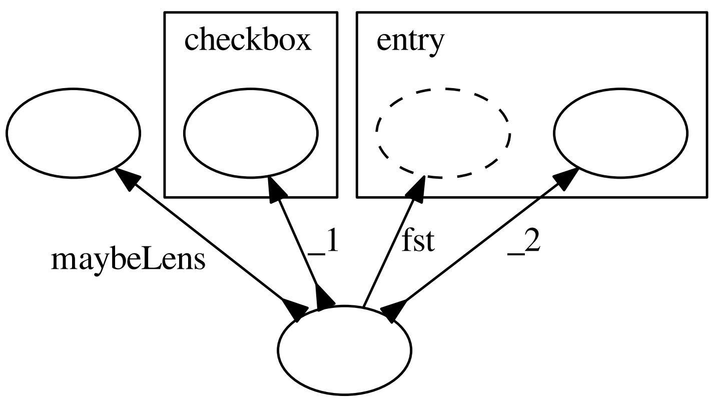


In a bit more complex example a `(Maybe Int)` value editor is shown. Note that the primary reference is here `r`; the reference `q` is needed to remember the entry value when the user deactivates and re-activates the entry by clicking the checkbox twice.




```
-- improper lens: set-set law is violated
maybeLens :: Lens' (Bool, a) (Maybe a)
maybeLens = lens get set
  where
    get (True, a) = Just a
    get _ = Nothing
    set (_, a) = maybe (False, a) ((,) True)
```

The same diagram stripped:



5 Reference-network creation

Arbitrary tree-shaped reference-network can be created with the following two operations:

```
-- forward lens-application
lensMap :: Lens' a b -> Ref m a -> Ref m b

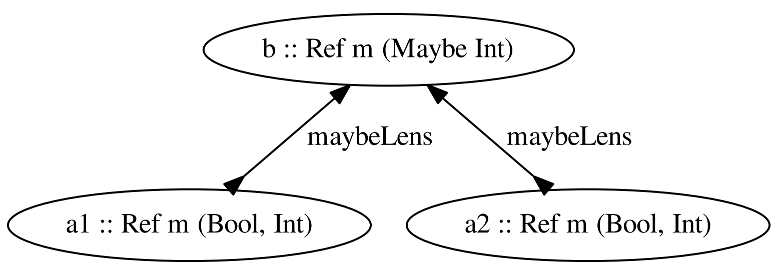
-- backward lens-application (reference extension)
extendRef :: Ref m b -> Lens' a b -> a -> RefCreator m (Ref m a)
```

Let `(k :: Lens' a b)` and `(r :: Ref m a)`. Then `r` and `(lensMap k r)` are references connected by `k`.

Let `(q :: Ref m b)`, `(k :: Lens' a b)` and `(x :: a)`. Let `(r :: Ref m a)` the return value of `(extendRef q k x)`. Let *s* the program state before the creation of `r` and *s'* the program state after the creation of `r`. Then the following hold:

- `r` and `q` are connected by `k`.
- All references have the same value in *s'* as in *s* (with the exception of `r` whose value is not defined in *s*).
- The value of `r` in *s'* is `(set k y x)` where `y` is the value of `q` in *s* (or in *s'*). Note that this is the most meaningful value for `r` such that the previous two statements hold.

Note that `extendRef` is needed for the creation of reference-networks like the following:



`extendRef` is called reference extension because the result reference may contain more information than the original reference. For example, considering the previous diagram, `a1` and `a2` may have different values in some program states, so they are not completely entangled and their values are not determined by the value of `b`.

6 Tracking changes in the reference-network

6.1 Motivation

6.2 Semantics

- currentValue
- instance Monad RefReader

7 Dynamic networks

- RefReader m (Ref m a)
- onChange
- joinRef

8 Resource handling

- onRegionStatusChange

9 Old stuff

9.1 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
    )
```

9.2 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.

9.3 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.

10 Joining a reference

By joining a reference I mean the following operation:

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

10.1 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`.

10.2 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
```

11 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)
```

11.1 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.

11.2 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 ...

11.3 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.

11.4 `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 ()
```

11.5 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).

12 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)
```

12.1 Semantics with examples

TODO

12.2 Variations

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

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

TODO

13 Bindings to outer actions

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

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

13.1 Resource handling

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

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

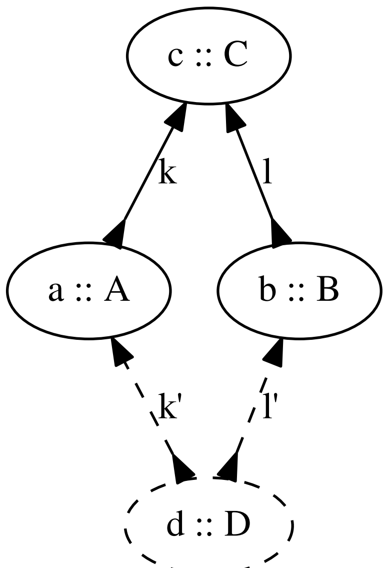
14 Summary

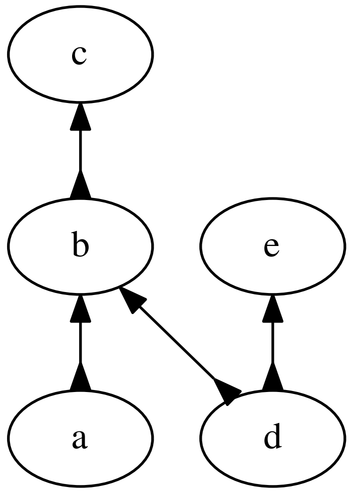
TODO

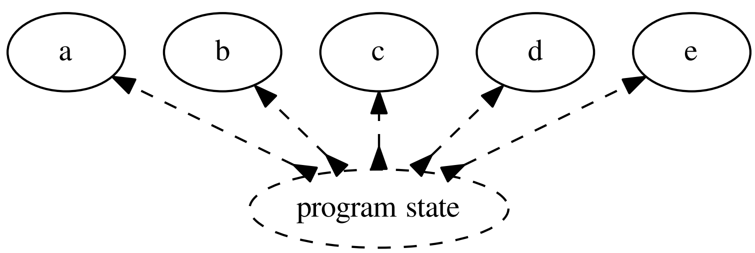
14.1 Comparison to existing Haskell FRP frameworks

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

14.2 Extra diagrams




c
b
e
a
d


a
b
c
d
e
program state