Introduction to state-based FRP

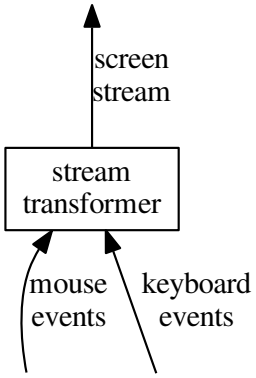
Draft

July 22, 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:



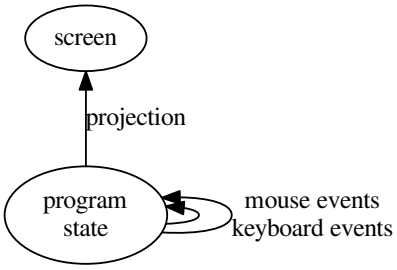
The goal of *functional reactive programming* (FRP) is to declaratively describe interactive programs. Declarative descriptions are composable which means that FRP descriptions of interactive programs can be decomposed into FRP descriptions of simpler interactive programs.

Most declarative descriptions of interactive programs is built around event streams and stream transformers equipped with different combinators like horizontal and vertical composition. Let us call these systems *stream-based FRP* systems.

1.2 State-based FRP

Stream-based FRP is not the only possible way to declaratively describe interactive programs.

In case of interactive programs with a GUI it is possible 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 scheme to arbitrary interactive programs, and to decompose it into simpler similar parts. Let us call *state-based FRP* a system characterized by this goal.

State-based FRP is an alternative to stream-based FRP systems. 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].

2 Basic operations on references

The interface of `lensref` is built around the reference data type. This section defines an interface of basic reference operations. We add new operations step-by-step to this interface in the following sections.

2.1 The reference data type

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

Each reference has 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:

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

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

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

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

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

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

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)
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 (declaration of `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.

2.2.1 On reference reader computations

Any `RefReader` computation has an associated value for each program state, its return value. Moreover, `RefReader` is isomorphic to the `Reader` monad (see later) so a `RefReader` computation is characterized by its associated values. Therefore a `RefReader` computation is a non-editable view of the program state i.e. a *read-only reference*.

Non-editable views (reference reader computations) and editable views (references) of the program state are called *views*.

A reference reader computation can also be seen as a dynamic value. For example, an `Int` value is a static integer whilst a `(RefReader Int)` value is a dynamically changing integer. Dynamic values contain static values: `(pure 3 :: RefReader Int)` is a static dynamically changing integer.

Reference reader computations are closely related to signals in stream-based FRP. Notice the difference. A signal is a time-varying value whilst a reference reader computation is a state-varying value. Although the program state varies by time, a state-varying value is not the same as a time-varying value because the program may have the same state in different time positions.

2.3 Laws

The following laws are part of the interface.

[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.

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]

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

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**:

```
readerToWriter (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:

[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).

```
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]

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 unlike `RefReader`, `RefCreator` is not idempotent. For example, `(liftM2 (,) (newRef 14) (newRef 14))` and `(liftM (\a -> (a, a)) (newRef 14))` has different behaviour since the former creates two distinct references whilst the latter creates two entangled references.

2.4 Derived constructs: `writeRefDyn`, `modRef`

`writeRef` can be generalized such that the written value 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.

Another derived construct, `modRef` can be expressed in terms of `writeRefDyn`:

```
modRef :: RefContext m => Ref m a -> (a -> a) -> RefWriter m ()
modRef r f = writeRefDyn r $ fmap f $ readRef r
```

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

2.5 Running `RefCreator`

`lensref` exposes the following function:

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

`runRefCreator` turns a `(RefCreator m a)` computation into an `(m a)` computation. During the creation of the `(RefCreator m a)` value it is possible to turn any `(RefWriter m b)` computation into an `(m b)` computation.

2.5.1 Safety of `runRefCreator`

`runRefCreator` is an unsafe function because it is possible to implement a restricted form of `unsafeCoerce` with it.[5] `lensref` exposes `runRefCreator` because of the following reasons:

- There is a trade-off between the safety of `runRefCreator` and the complexity of its interface.
- There are two easy rules to follow to keep the current `runRefCreator` safe:
  - Call `runRefCreator` only once in your program. This rule seems quite restrictive but `lensref` is designed such that one call of `runRefCreator` is enough in most use cases. A relaxed form of this rule is that do not allow to interact references created with different `runRefCreator` calls.
  - Use `runRefCreator` in a single-threaded environment.
- A thread-safe variant and/or a multiple-call-safe variant of `runRefCreator` can be defined in terms of the current `runRefCreator` function. See … TODO

2.6 Examples

For running basic tests we defined a specialized form of `runRefCreator`:

```
runRefTest :: (forall s . RefCreator (ST s) (RefWriter (ST s) a)) -> a
runRefTest m
    = runST $ join $ runRefCreator $ \runRW -> m <&> runRW
```

[5] TODO

Note that `runRefTest` is a safe function (unlike `runRefCreator`).

Some test cases (the desired result is shown in comments):

```
-- readTest == 3
readTest = runRefTest $ do
    r <- newRef (3 :: Int)
    return $ readerToWriter (readRef r)

-- writeTest == 4
writeTest = runRefTest $ do
    r <- newRef (3 :: Int)
    return $ do
        writeRef r 4
        readerToWriter (readRef r)
```

Nothing special so far.

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
```

```haskell
-- 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:

```haskell
-- 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:

```haskell
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 application on references

The `lensref` interface has a *lens application on references*:

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

If references would be implemented as lenses on the program state, lens application would be the same as function composition (with arguments flipped). It is not possible to implement references with lenses because of several reasons, but it is possible to use function composition instead of `lensMap`. In fact, the current implementation of `lensref` is function composition. However, this fact is not reflected in the interface because a newtype wrapper on reference values inhibits this. The newtype wrapper is needed only to keep the type of references monomorphic.

3.2.1 Semantics of lens application

Let `(r :: Ref m a)` and `(k :: Lens' a b)`. `(lensMap k r)` is characterized by the following laws:

- ```
  readRef (lensMap k r)
      ~       -- lensMap-read
  readRef r <&> (^. k)
  ```
- For all `(y :: B)`

  ```
  writeRef (lensMap k r) y
      ~       -- lensMap-write
  modRef r (flip (set k) y)
  ```

3.2.2 Examples

TODO

3.3 Reference extension

Let `(k :: Lens' a b)` and `(q :: Ref m b)`. *Reference extension*, or *backward lens application*, helps to create a reference `(r :: Ref m a)` such that `q ~ lensMap k r`:

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

3.3.1 Semantics of reference extension

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:

- `q ~ lensMap k r`
- All references with the exception of `r` have the same value in *s'* as 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.

`extendRef` is called reference extension because the invocation of `extendRef` extends the program state or the reference state if we think of references as substates of the program state.

3.3.2 Examples

TODO

3.4 The `Ref` category

Let `(a :: Ref m A)` and `(b :: Ref m B)`. We say that `a` and `b` are connected by `(k :: Lens' A B)` iff `b` ~ `(lensMap r a)`.

Lens connection is a transitive relation and every reference is connected to itself by the identity 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: `(unitRef :: Ref m ())` which is exposed in the `lensref` interface.
- 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 Diagrams

We draw reference diagrams to visualize different type of connections between views of the program state.

4.1 `Hask` diagrams

Reference diagrams are based on diagrams in the `Hask` category.

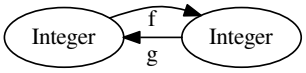
In `Hask` objects are types and arrows are functions.

4.1.1 Motivation

Why are `Hask` diagrams useful (compared to source code)?

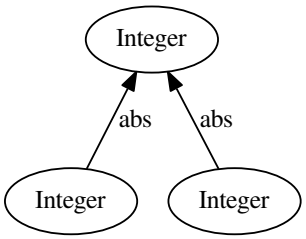
- Variable names are not needed.
- Diagrams may show constraints (see the following diagrams).
- We decorate `Hask` diagrams to show data flow between views in the next section.

Example for constraints shown in diagrams:



The diagram shows two `Integer`-typed variables in scope. Let us call them `x` and `y`. The corresponding source code may be `{y = f x}` or `{x = g y}` but both of them lacks a constraint shown on the diagram.

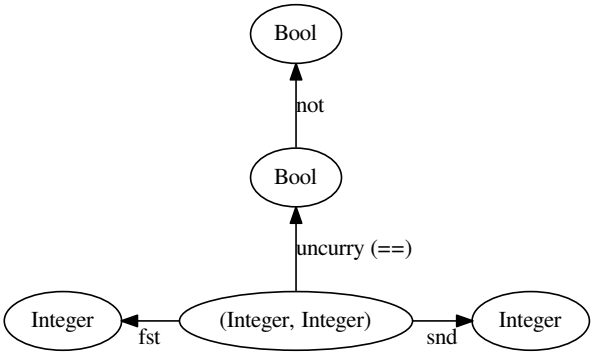
Another example:



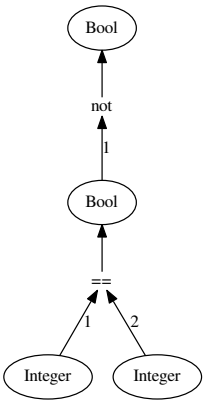
The diagram shows three `Integer`-typed variables in scope. Let us call them `a`, `x` and `y` from up to down and left to right. The diagram shows that `a ~ (abs x) ~ (abs y)` (the expressions have the same behaviour) which cannot be expressed with source code.

4.1.2 N-ary functions in diagrams

Usually n-ary functions on diagrams are presented in uncurried form with a product domain:



We allow a different presentation of n-ary functions. We use a new kind of nodes for this: there are type-nodes and function nodes in the diagram. The direct predecessors of function nodes are the function arguments. The order of arguments matter so the input arrows of function nodes are numbered. The previous diagram in this presentation:

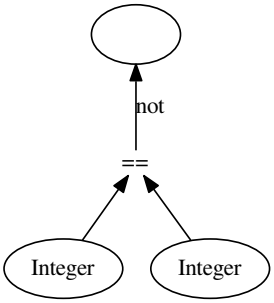


4.1.3 Simplifications

We use several simplification to make the diagrams smaller:

- A unary function may be presented as an arrow.
- An arrow number may be hidden if it is inferable or it does not matter.
- The label of a type node may be hidden if it is inferable.
- A type node with a hidden label and with one input arrow may be hidden.

The previous diagram with some of the simplifications applied (compare it also to the initial diagram):

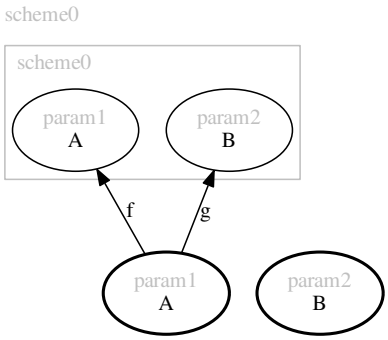


4.2 Diagram schemes

Diagram schemes help to modularize complex diagrams.

Diagram schemes are diagrams in which some nodes are marked as *parameters*. Diagram schemes may be *applied* to a cluster of nodes in a diagram which means that the diagram scheme is instantiated (copied) such that scheme parameters are identified with the clustered nodes. Diagram scheme parameters have identifiers to help to identify them in scheme applications.

Parameters of diagram schemes are shown with a bold outline. Diagram scheme application is shown with a clustering with a grey outline and the name of the scheme is show as the label of the clustering in the left-top corner. Parameter identifiers are shown with a grey label above the node label. An example:



The diagram shows a diagram scheme called “scheme0” applied to itself forming an infinite diagram (not very useful in this case).

4.2.1 Diagram schemes vs. functions

What are the advantages of diagram schemes compared to functions?

Diagrams schemes are more general than functions.

TODO

4.3 Reference-diagrams

We decorate `Hask` diagrams to show data flow between views.

Three different kind of decorations are applied:

- Arrow colour will be red in case of some special functions.
- Arrow style of red arrows may not be plain.
- Layout of nodes is partially standardized.

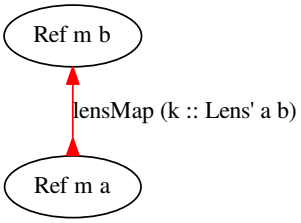
4.3.1 Arrow colouring

Arrows of data-flow functions are coloured red, other arrows are black. Data flow functions are the following:

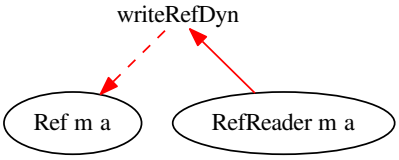
- `lensMap`
- `readRef`
- `fmap` for `RefReader` computations
- `writeRefDyn` (see later)

4.3.2 Arrow styles

Arrows representing lens application have an inverted arrow tail to reflect bidirectional data flow between references:



The arrow of first argument of `writeRefDyn` is drawn *backwards* and dotted to reflect that value of the reference is updated occasionally:



`writeRefDyn` may be seen as an occasional connection between views.

4.3.3 Layout of nodes

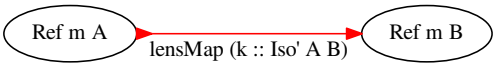
To help to read reference diagrams, we partially standardize the layout of nodes by the following layout rule:

Let `v` and `w` be views. If `v` determines `w` then `v` cannot be above `w` in the diagram.

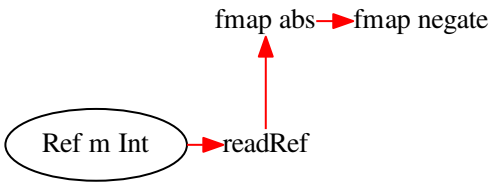
We say that the view `v` *determines* the view `w` if any of the following holds

- `w ~ (lensMap k v)` for some `k`
- `w ~ (fmap f v)` for some `f`
- `w ~ (fmap f $ readRef v)` for some `f`

Because of the layout rule, in case of lens-connection by an isomorphism the references are shown at the same level to reflect that they determines eachother:



Another example with `RefReader` computations:



4.4 Examples

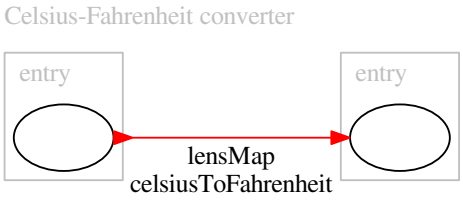
Groups of `Ref`, `RefReader` and `RefWriter` values can be connected to GUI widgets.[6] This connections are shown as diagram scheme applications in reference diagrams.

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

The possible connections are the following:

- `(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.

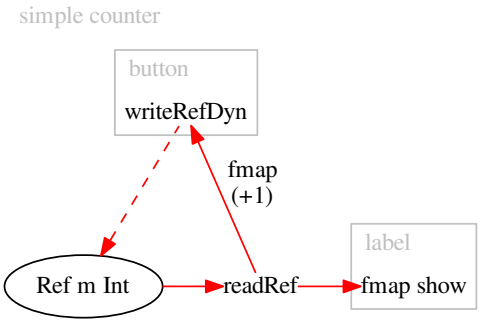
For example, a Celsius-Fahrenheit converter has two entangled `Double` value entries:




```
celsiusToFahrenheit :: Iso' Double Double
celsiusToFahrenheit = multiplying 1.8 . adding 32
```

```
> (click, put, get, delay) <- run' temperatureConverter
  |   0.00 0 Celsius =   32.00 1 Fahrenheit
> put 0 "32"
  |  32.00 0 Celsius =   89.60 1 Fahrenheit
> put 1 "3"
  | -16.11 0 Celsius =    3.00 1 Fahrenheit
```

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



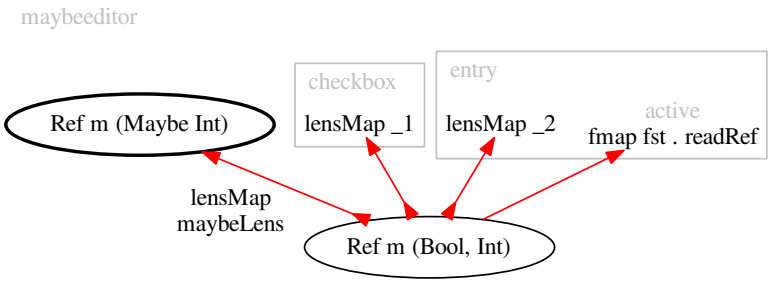

```
> (click, put, get, delay) <- run' counter
  |  0 0 Count1
> click 1
  |  1 0 Count1
> replicateM_ 3 (click 1)
  |  2 0 Count1
  |  3 0 Count1
  |  4 0 Count1
```

```
> (click, put, get, delay) <- run' booker
  | one-way flight0 return flight1
  |      0 2
  |      0 3
  | Book4
> click 1
  | one-way flight0 return flight1
  |      0 2
  |      0 3
  | Book4
> put 2 "blabla"
  | one-way flight0 return flight1
  | blabla 2
  |      0 3
  | Book4
> put 2 "10"
  | one-way flight0 return flight1
  |     10 2
  |      0 3
  | Book4
> put 3 "20"
  | one-way flight0 return flight1
```

```
  |     10 2
  |     20 3
  | Book4
> click 4
  |  You have booked a return flight on 10-20 5


> (click, put, get, delay) <- run' timer
  | Elapsed Time:  0.00% 0
  |  0.00s 1
  | Duration:  10.00 2
  | Reset3
> delay 1
  | Elapsed Time:  10.00% 0
  |  1.00s 1
  | Duration:  10.00 2
  | Reset3
> delay 100
  | Elapsed Time:  100.00% 0
  |  10.00s 1
  | Duration:  10.00 2
  | Reset3
```

In a bit more complex example a `(Maybe Int)` value editor is shown. Notice that the input reference is extended to remember the entry value when the user deactivates and re-activates the entry by clicking the checkbox twice.




```haskell
-- 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)
```

5 Reference join

5.1 Semantics

5.2 Examples

6 Dynamic network creation

6.1 Dependency relation

Dependency is binary relation between reference reading computations and references.

6.1.1 Motivation

Let `(w :: RefWriter m ())` and let `v` be a reference reader computation. Given a program state, we say that `v` is *sensitive* to `w` if executing `w` changes the value of `v`.

Sensitivity is not computable in general because equality check is not computable.[7] Dependency is a computable approximate sensitivity relation.

6.1.2 Semantics

Defining rules for dependency for a given program state s:

- `(readRef r)` depends on `r`
- `v` depends on `(lensMap k r)` if `v` depends on `r`
- `(v >>= f)` depends on `r` if either `v` depends on `r` or `(f a)` depends on `r` where `a` is the value of `r` in s

TODO: extendRef currentValue joinRef onChange

6.1.3 Examples

TODO

[7] Consider `(v :: RefReader (Integer -> Integer))`. It is not possible to check the equality of two values of `v`.

6.2 Reactions

6.2.1 Motivation

We want to bind reactions to changes of program state views.

What kind of reactions are appropriate?

- creation of new references
- outer effects
- reference write **should be avoided** because it could cause circles in the execution of reactions

=> Reference creation computations

`onChange`

Outer effects may not cause circles because …

6.2.2 Semantics

Register all onChange actions

Whenever `writeRef r a` happens, do all dependent actions in registration order

6.2.3 Examples

7 Resource handling

- onRegionStatusChange

8 Implementation / Cost semantics

9 Summary

TODO

9.1 Comparison to existing Haskell FRP frameworks

reactive reactive-banana Elerea FranTk Sodium Elm netwire yampa iTask

TODO