Lettier

549 Followers About Follow

Your Easy Guide to Functional Reactive Programming (FRP)

Lettier Oct 23, 2018 · 8 min read

Managing state. It's the bane of programmers everywhere. The more stateful processes, subroutines, objects, components, widgets, doodads, etc. there are, the harder it is to

keep your app from becoming a giant mud pit.

"What, this? No this is nothing. You should see my other code base."

But fear not! You can use functional reactive programming (FRP) in your quest to manage state.

Imagine you have three entities called `red`, `blue`, and `purple`. Your job is to wire them all together such that `red + blue = purple`. Adjusting `red` and/or `blue` should automatically adjust `purple`.

Let's try solving this imperatively and then we'll switch to FRP.

First, create some objects capable of retaining and manipulating their own state.

```
Color = function() {
  this.color = 0;

  this.onChange = function() {};

  this.set = function(color) {
    this.color = color;
    this.onChange(this.color);};

  this.get = function() {return this.color;};

  this.setOnChange = function(fun) {
    this.onChange = fun;}}

red    = new Color();
blue   = new Color();
purple = new Color();
```

Next, set up the callbacks, wiring them together.

```
red.setOnChange(function(color){
  this.purple.set(color + this.blue.get());}.bind(this));

blue.setOnChange(function(color){
  this.purple.set(color + this.red.get());}.bind(this));
```

To see what's going on, set up a logging callback for `purple`.

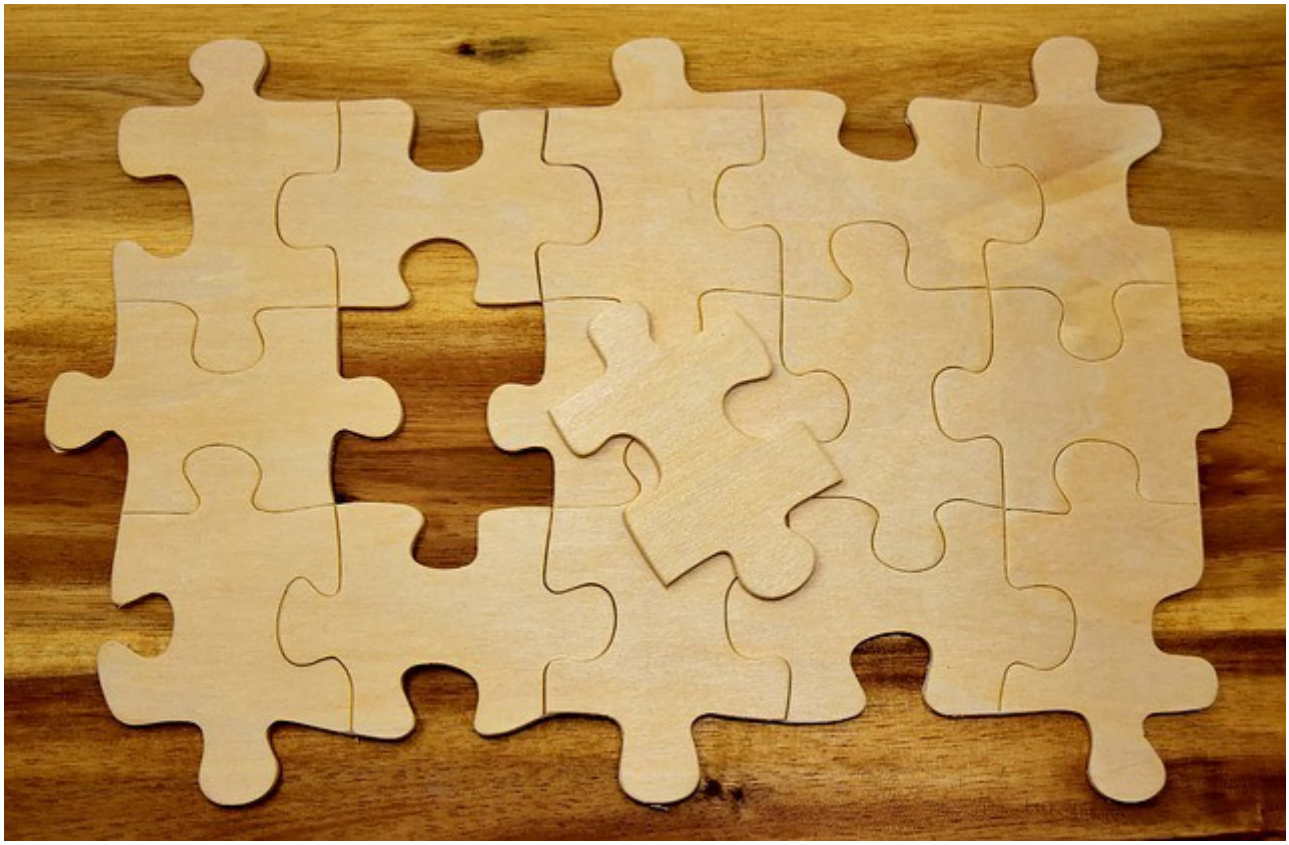
```
purple.setOnChange(function(color){
  console.log("Purple: " + color);});
```

And then test it out by changing the state of `red` and `blue`.

```
red.set(2);
blue.set(4);
```

```
> red.set(2);
  Purple: 2
<- undefined
> blue.set(4);
  Purple: 6
<- undefined
>
```

Now let's switch to FRP.



First, create the `red` and `blue` events.

```
red  <- create
blue <- create
```

From those, create some `red` and `blue` behaviors.

```
let red'  = step(0)(red.event)
let blue' = step(0)(blue.event)
```

And from those behaviors, create the `purple` behavior.

```
let plus b1 b2 = apply(map(+)(b1))(b2)

let purple = red' `plus` blue'
```

Like before, set up some logging.

```
let log' = log `compose` append("Purple: ") `compose` show
void(animate(purple)(log'))
```

And finally test it out by pushing some values for both `red` and `blue`.

```
red.push(2)
blue.push(4)
```

```
Purple: 6
Purple: 6
Purple: 6
Purple: 6
Purple: 6
Purple: 6
Purple: 6
Purple: 6
Purple: 6
5918 Purple: 6
>
```

⚠️ If the FRP version seems completely alien 👽 — don't worry. By the end you'll understand it. 😃

Looking at the two, the FRP version is declarative. You declared or expressed the relationship that purple = red + blue. You gave the what, but not the how. You didn't have to write any of the plumbing with callbacks. In both cases, whenever `red` or `blue` changes — `purple` changes. So we didn't lose any functionality by using FRP.

What's functional reactive programming?

> Functional Reactive Programming, or FRP, is a general framework for programming hybrid systems in a high-level, declarative manner. The key ideas in FRP are its notions of behaviors and events. — Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles.

It wasn't always called "Functional Reactive Programming," though. Initially it was called, "Functional Reactive Animation."

> Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of behaviors and events. — Conal Elliott and Paul Hudak. 1997. Functional reactive animation.

Over the years the term has been adopted and used to describe implementations that do not necessarily adhere to the precise mathematical specification or "denotational semantics" as laid out by Conal Elliott and Paul Hudak.

In any case, true (or what was originally) FRP deals with continuous (versus discrete) time, behaviors, and events. If something is called FRP but doesn't model time as continuous, doesn't use behaviors, and/or doesn't use events then you might consider it, "Functional *and* Reactive Programming (F&RP)."

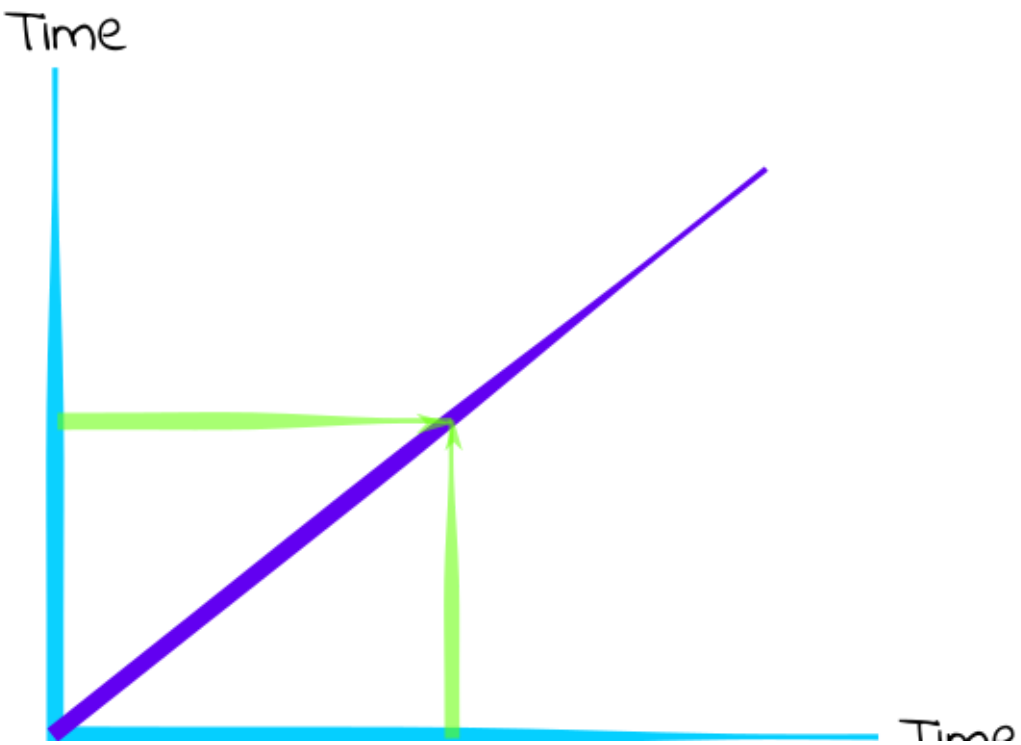
GIF created with Gifcurry.

To make this concrete, I built a FRP demo called PureScript Pop 💥. Go ahead and have that open while you read along. When you get a chance, take a look at the source 📜. I made it very readable — even if you've never written PureScript before. 👍

What's a behavior?

Behaviors are "time-varying values." A behavior always has a value no matter what time it is and its value may change over time. Give me a point in time, a behavior, and I'll always have a value for you. Behaviors are continuous variables with respect to time.

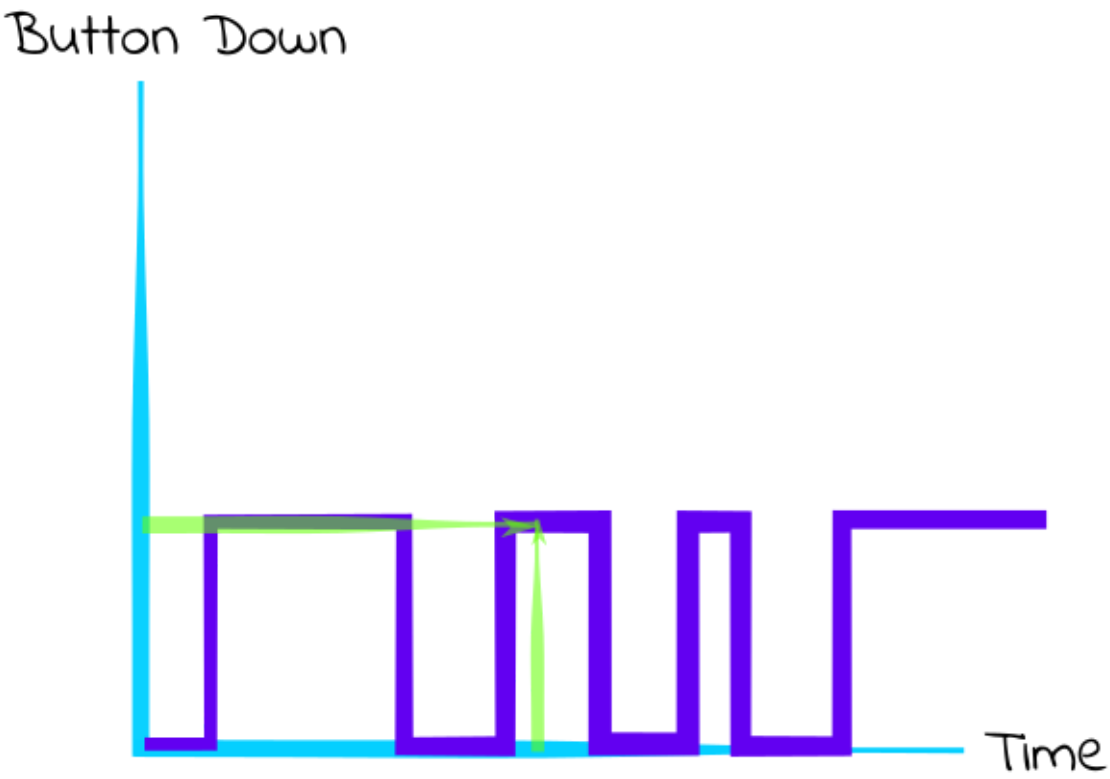
💡 The simplest behavior is just time itself. You give me a time and I give it right back to you.



Time

Or you can get more complicated like is the mouse button down or up? No matter what the time, the mouse button is either down or up (assuming there's no in-between transition).

```
let isMouseDown =
      step
        false
        (     (\ {value, buttons} ->
                value == 1 && member 0 buttons)
        `map` (withButtons
                mouse
                (
                      ((const 0) `map` up)
                `alt` ((const 1) `map` down))))
```



So as your application state moves through time, you can always count on behaviors having a value.

Behaviors get interesting when you combine them to create new behaviors — like combing red with blue to make purple.

```
unfold
  (\
    { isMouseDown'
    , isOverCoin'
    , isOverCoinSlot'
    }
    wasDragging
    ->
        not isOverCoinSlot'
    &&  (   (wasDragging  && isMouseDown')
        ||  (isMouseDown' && isOverCoin')
        ))
  (sample_
    (combine3Behaviors
      (Tuple
        (SProxy :: SProxy "isMouseDown'")
        isMouseDown)
      (Tuple
        (SProxy :: SProxy "isOverCoin'")
        isOverCoin)
      (Tuple
        (SProxy :: SProxy "isOverCoinSlot'")
        isOverCoinSlot))
    (interval 1))
  false
```

Here I combine three behaviors (`isMouseDown` , `isOverCoin` , and `isOverCoinSlot`) to make a new behavior (`isDragging`) that lets me know if the mouse is dragging a coin or not.

GIF created with Gifcurry.

If your programming language has functional programming patterns like monad, applicative, and functor, your FRP library will typically create instances for monoid, applicative, semigroup, etc. allowing you to easily combine/compose behaviors declaratively.

Eventually you'll want to *react* to a behavior's value. Your FRP implementation should come with a function that allows you to perform some effectful action over time using the behavior's value. This effectful action could be updating a button label, moving a game character, logging some information, etc.

```
void $
  animate
    popMachineState
    (coinCursorAnimation
      [ coin1Img
      , coin2Img
      , coin3Img
      ])
```

Here I animate the cursor for a coin, depending on the pop machine's state.

GIF created with Gifcurry.

What's an event?

Events are time-stamped values or (time, value) pairs. You give me a time and I… may or may not have a value for you. In other words, events are discrete variables with respect to time.

💡 The simplest event is just a constant value at a constant point in time.

But you can get as complex as you like. For example, the location of the mouse at the time of a click.

Like behaviors, your FRP library should come with some function that allows you to perform some effectful action, with the event's value, whenever the event occurs.

```
void $
  subscribe
    event
    (log `compose` show)
```

How do I create a behavior from an event?

FRP gets really interesting when you create behaviors from events and vice versa.

```
let dispensingPopCountEvent = count dispensingPopEvent
let dispensedPopCount = step 0 dispensingPopCountEvent
```

From the `dispensingPopCountEvent`, I create a behavior that always has, as a value, the number of pops given out so far.

So I go from this…

To this…

Combining this behavior with others, I can determine which pop to animate on the page when the user inserts a coin.

```
case count of
  1 -> do
    void(goTo(popStop1Div)(functionalPopImg))
    void(goTo(popStop2Div)(reactivePopImg))
    result <- goTo(popStop3Div)(programmingPopImg)
    when result $
      popMachine.push(WaitingForPopRemoval)
  2 -> -- ...
  3 -> -- ...
  _ ->
    pure(unit)
```

GIF created with Gifcurry.

How do I create an event from a behavior?

Creating events from behaviors opens up a lot of opportunities . Your FRP library should come with functions for turning behaviors into events. One example is the `gateBy` function.

Let's say I wanted to go the other way. I have a behavior that holds how many pops have been dispensed so far. From that, I want an event that holds the pop number being dispensed, every time the machine starts dispensing a pop.

```
let dispensingPopNumber =
      gateBy
        (\ _ state -> state == DispensingPop)
        dispensedPopCount
        popMachine.event
```

Using the pop machine event as a trigger, I sample the `dispensedPopCount` behavior, emitting an event with the behavior's value whenever the pop machine's state goes to `DispensingPop`.

How do I combine behaviors?

Combining behaviors to make new behaviors is one of the great strengths of FRP.

Say you have two behaviors, each containing a list of numbers.

```
let behavior1 = step [] event1.event
let behavior2 = step [] event2.event

let behavior3 = behavior1 `append` behavior2

void $
  animate
    behavior3
    (log `compose` show)

event1.push [1]
event2.push [2]
```


`behavior3` has the value `[1,2]` and all I had to do was append `behavior1` to `behavior2`.

Say you have two Boolean behaviors…

```
let behavior3 = (||) `map` behavior1 `apply` behavior2
```

`behavior3` is `behavior1` or-ed with `behavior2` using functor's map and applicative's apply.

Say you have a string behavior and an integer behavior…

```
let behavior3 =
                    (\ string int ->
                      string `append` show int)
            `map`   behavior1
            `apply` behavior2
```

You can come up with all sorts of ways to combine behaviors into what you need.

How do I combine events?

Like behaviors, you can combine events in similar ways.

Say you have two events, one and two, and you want another event that occurs whenever one or two occurs.

```
event1 <- create
event2 <- create

let event3 = event1.event `alt` event2.event

void $
  subscribe
    event3
    (log `compose` show)

event1.push 1
event2.push 2
```


Using `alt`, I combined `event1` and `event2` into `event3`. When either `event1` or `event2` occurs, `event3` occurs.

FRP is cool.

FRP can be a really powerful way to architect your application using only functions! 🎉 With just behaviors and events, you can declaratively build up a highly stateful application.

For another FRP example, head over to Haskell to JavaScript where I build an on/off button using Haskell's Reflex.

JavaScript Functional Programming Web Development Programming Software Development