MENRVA

@futurice freaklies

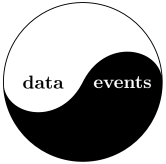
Oleg Grenrus

July 03, 2014

MENRVA– yet another reactive library

BACON.JS is (almost) non-glitchy, but not-so performant. RXJS is glitchy but performant.

I begin to bias towards using cold-observables of RXJS[1] (the good parts?) with data-flow propagation of something else (= MENRVA).



[1]Can use BACON.JS too, but I don't need glitch-free events

Semantics

Following definitions are from *Push-pull functional reactive programming*[2].

FRP – Behavior

Semantically, a *(reactive) behavior* is just a function of time. In MENRVA we call them *signals*.

$$\mathcal{S}\,\alpha = \mathcal{T} \to \alpha$$

Historically in FRP, $\mathcal{T} = \mathbb{R}$. As we'll see, however, the semantics of behaviors assumes only that $\mathcal{T}$ is totally ordered.

[2]Conal Elliott, *Push-pull functional reactive programming*, Haskell Symposium, 2009, `http://conal.net/papers/push-pull-frp/`

Applicative functor

Pure functions can be "lifted" to apply to signals. Classic FRP (CFRP) had a family of lifting combinators:

$$\texttt{lift}_n :: (\alpha_1 \to \cdots \alpha_n \to \beta) \to (\mathcal{S}\,\alpha_1 \to \cdots \mathcal{S}\,\alpha_n \to \mathcal{S}\,\beta)$$

Lifting is pointwise and synchronous:

$$\texttt{lift}_n\, f\, b_1 \cdots b_n = \lambda\, t \mapsto f\,(b_1\, t) \cdots (b_n\, t)$$

The *Functor* `map` is `lift`$_1$:

$$\texttt{map}\, f\, b = f \circ b$$

In MENRVA `map` is `map` and $\texttt{lift}_n$ is `combine` (with function parameter at the end):

```
var $sq = $source.map(function (x) {
  return x * x;
});

var $quad = menrva.combine($a, $b, $c, function (a, b, c) {
  return a * b + c;
});
```

And that's the whole (functional) API.

There is `onValue` to observe changes. And `set` and `modify` to initiate changes.

Monad

Although Signal is a semantic *Monad* as well, MENRVA doesn't have monadic `join` combinator.

Semantic of `join` is simple, but the correct implementation hard.

$$\texttt{join}\, s = t \mapsto s\, t\, t$$

Monad instance would make possible to make dynamic data flow networks. With applicative functor the shape of the network is static (and you can't make loops) $\equiv$ simple and easy.

Examples

`https://github.com/phadej/menrva/tree/master/examples`

- *counter* examples (pun not-intended). With data-first approach, you segregate *query* from *control*.
- *suggestions The introduction to Reactive Programming you've been missing*

What are plans for MENRVA?

Write more examples, to verify that MENRVA's simplistic approach is enough for the real world.

$\vdots$

Use in the real world application.

$\vdots$

PROFIT!

What are you going to add still?

Something to handle network destruction. Thus dynamically created (and destroyed) UI components create leaf-subnetwork which should be cleaned up. JavaScript doesn't have weak references (which might be enough to solve the problem). So we need some manual mechanism for cleanup.

One idea is to make own reference counting:

```
var $state = menrva.source(...);
var $derived = $state.map(...).retain();

var $componentData = $derived.map(...).retain();
// cleanup:
$componentData.release(); // also removes all callbacks etc.
```

What are you going to add still? – 2

Should we have monadic join? (= `flatMap`). Do we really need it? Can be worked around, but sometimes it is convenient:

```
$artistImage = $artist.flatmap(getArtistImageSignal);

function getArtistImageSignal(artist) {
  if (!cache[artist.id]) {
    cache[artist.id] = menrva.source(placeholderImage);

    // schedule carousel for images...
  }
  return cache[artist.id];
}
```

Here the cache (or artist image store) can be signal itself – no need for `flatMap`.

What are you going to add still? – 3

Borrow goodies from BACON.JS – Function construction rules:

```
$signal.map(".foo"); // shorter way of saying:
$signal.map(function (x) {
  return x.foo;
});
```

What are you going to add still? – 4

Lenses. Sub-value of source acting as source itself.

```
var $foobar = $source.lens("foo.bar");

$foobar.set(tx, value); // the same as
$source.modify(tx, function (source) {
  return assocIn(source, ["foo", "bar"], value);
});
```

assocIn – http://clojuredocs.org/clojure_core/clojure.core/assoc-in

And why the choice for transactions?

You able to update many source simultaneously, avoiding immediate propagation:

```
var $a = menrva.source(1);
var $b = menrva.source(2);

menrva.combine($a, $b, add).onValue(console.log);

var tx = menrva.transaction();
$a.set(tx, 2);
$b.set(tx, 3);
tx.commit();
```

And why the choice for transactions? – 2

Also to mention, transactions arise naturally, if you try to implement anything similar in HASKELL or CLOJURE (using STM).

There you will be forced to do all mutation inside STM transaction.

Changes initiated in transaction are applied only when you commit the transaction: you *always* have consistent data.

the end