Why (Functional) Reactive Programming?

Stephen Blackheath, Dec 2013
Stephen Blackheath

- Author of o/s Sodium reactive library for Haskell, C++ and Java

- http://reactiveprogramming.org/
Q: What is reactive programming? 1/4

A composable/modular way to code event-driven logic
Q: What is reactive programming? 2/4

A different way of thinking

*events are conceptually 'pulled'*
Q: What is reactive programming? 3/4

Observer pattern that doesn't suck

(drop-in replacement)
Q: What is reactive programming? 4/4

A technique to help rescue commercial projects that have hit the complexity wall.

- time to add a feature/fix a bug has got out of control
Applications

User interfaces
Games
Control/monitoring applications
Configuration

...anything event-heavy
...anywhere Observer is used
Interesting potential
Observer pattern

```java
void addListener(Listener l) {
    listeners.add(l)
}

void removeListener(Listener l) {
    listeners.remove(l)
}

private void notify(Event ev) {
    for (l <- listeners) l(ev)
}
```
Observer pattern

- Inverts dependency
  => consumer depends on source
  => separates concerns
  => extensibility
  => better than polling

- Simple, effective, widely used

- *What could possibly go wrong?*
Observer pattern - problems

1. Unpredictable order
2. Missed first event
3. Messy state
4. Threading issues
5. Leaking listeners
6. Accidental recursion

=> Compounding complexity (not scalable)
Observer – 1. Unpredictable order

- B and C both listen to A
- C listens to B

- How do we decide what order the events should arrive at C?
Observer – 2. Missed first event

Main() {
    Source s = new Source();
    Consumer c = new Consumer(s);
    c.setValue(s.getValue());
}

Observer – 3. Messy state

- Callbacks -> flow of control is inverted
- Brittle, non-composable state machines
- Messy state in object fields
- Coding mistakes => Illegal states

```csharp
private bool dragging;
private Item draggedItem;
void handle(MouseEvent ev) {
    if (ev.isMouseDown()) dragging = true; else
    if (ev.isMouseMove()) {if (dragging) ..;} else
    if (ev.isMouseUp())   {if (dragging) ..;}
}
```
Observer – 4. Threading issues

- Multiple threads?
  - Race conditions?
  - How can we ensure we don’t get callbacks after removeEventListener returns?

- Typical solution in industry: Use one thread
  - Not tenable in the multi-core age
  - Event-driven code is naturally parallel: Why throw it away?
Observer – 5. Leaking listeners

- We have to remember to deregister our listener
  ...or we get memory leaks

- What happened to the safety that garbage collection was supposed to give us?
Observer – 6. Accidental recursion

- Take care with notify() - don’t know what knock-on effect it might have:

```c
void processState() {  // State machine
    if (state == INIT_STATE) {
        ...
        notify(READY);    *
        state = READY_STATE;
    }
}
```

* Did `processState()` get called recursively by one of our subscribers?

=> Processing is out of order
Observer pattern

- Ill-defined semantics
  => Compounding complexity
  => Unpredictable behaviour
  => Angry customers
RP has *compositionality*

Observer does not.

(My definition)

- A and B are compositional if
  - The complexity of A + B is sum of the complexities of A and B
Observer doesn't scale

Program complexity increases exponentially with size

=>

Lack of scalability
Existing projects 1/2

Commercial projects often hit a SCALABILITY WALL after a year or two

RP => help break through

(event-based code only, but this is often where problems are)
Existing projects 2/2

- Drop-in replacement for Observer
  1. hard to stuff up
  2. easy to read (clarity of intent)
  3. composable / modular

- Not "all or nothing"
  - Can introduce in stages
Why not address issues as they arise?

- Complete solution is hard to implement
  - I think it’s because the problem itself is difficult
  - Observer = hire purchase: code now, pay later
    - (over and over again)
  - RP: Pay once up front

- Sodium: 900 lines of code
Scalability of RP?

Theoretically...

Compositionality

=>

Complexity vs. size ~ closer to linear
Scalability of RP?

Anecdotally...

"We kept waiting for the wheels to fall off, but they never did" - Anthony Jones

5 years of RP development
Functional language needed?

No.

(But static typing is desirable.)

Sodium => Haskell, Java, C++ ... more to come
Something fundamental?

Surprising similarity between independent RP library implementations

=> Boiled down to essence?
What I want to achieve

- Show problems with observer ✓
- Claim that RP fixes them all ✓
- Explain why RP fixes them ✓
- Prove that RP fixes them ✗
  - Experience => proof
## Reactive vs. Erlang Actor

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Observer</th>
<th>Actor</th>
<th>Reactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separation of concerns</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>1. Unpredictable order</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>2. Missed first event</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>3. Messy state</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>4. Threading issues</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>5. Leaking listeners</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>6. Accidental recursion</td>
<td>✗</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>Compositionality</td>
<td>✗</td>
<td>✗</td>
<td>✔️</td>
</tr>
<tr>
<td>Distributed processing</td>
<td>✗</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>

Reactive: strong consistency guarantee
Performance / Parallelism

- RP design is performance friendly
  - But unrealized

- Can code now and work on engine later
  - IF small performance hit OK now

- Investment to date: near $0
Performance

- Consistent
  - Linear processing cost
  - Observer can be exponentially bad

- Great unrealized potential
  - Many potential optimisations
  - Can code now + fix engine later
    - IF a small performance hit is OK now
Parallellism / multi-core

- Strong consistency guarantee
  - Complicates parallellism

- Currently: not that great
  - Relax consistency

- Huge potential in optimization
  - Can reason about parallelism
Testability / Refactorability

Excellent

- Dependencies are explicit
  - Very loose coupling
- Compiler detects breakage
Debugability 1/2

- Abstract so a little hard to inspect
  - Traces do work
  - Execution order is unpredictable

- Standard debugger annoying
  - Jumps between app & RP library
  - Specialized debugger would help
Debugability 2/2

- But **bugs are rare**

- "Our predominant source of bugs was not understanding the problem we were trying to solve" - Anthony Jones
Anecdote

- In a hurry
- Knocked together a very general "mouse drag" implementation
- Thought, "That'll work"
- Ran the program
- It worked

http://reactiveprogramming.org/
See Gesture.hs in Sodium game engine
Reactive system

- I/O typically done outside the reactive system
Reactive system

EIGHT PRIMITIVES

never
map
filter
merge
hold
snapshot
lift
switch

TWO DATA TYPES

Event
Behaviour
Event

data Event a
class Event<a>

- Event represents a *stream of events*.
  - Manipulate the **whole stream**, not individual **occurrences**.
- Has a type parameter, e.g. `Event<MouseEvent>`
Event

data Event a
class Event a

- No explicit add/removeListener()
  - Implied by merely referencing the Event
- No explicit notify()
  - The only “push” operation is the original input event
Notions of time

```haskell
data Event a
class Event a
```

- Two main views on time
  1. Classic FRP: events are stamped with time, behaviours are functions of time.
    - Behaviours vary continuously
  2. Sodium/others: Events and behaviour state changes are ordered. Time is just another behaviour.
    - Behaviours change discretely
Behaviour

```
data Behaviour a
class Behaviour a
```

- Time-varying value, a.k.a. “Signal”
- e.g. Space ship position, Vehicle speed
- Behaviours are like events that remember their last value, BUT...
Behaviour

```haskell
data Behaviour a
class Behaviour a
```

- Continuous or discrete?
  - Some argue the *changes* should not be observable
  - (not so in *Sodium* – chose convenience over purism)
Reactive system

EIGHT PRIMITIVES

never
map
filter
merge
hold
snapshot
lift
switch

TWO DATA TYPES

Event
Behaviour
Event primitives

-- Event that never fires

\[
\text{never} :: \text{Event } a \\
\text{Event}\langle A \rangle \text{ never}()
\]

-- Modify contained value

\[
\text{map} :: (a \rightarrow b) \rightarrow \text{Event } a \rightarrow \text{Event } b \\
\text{Event}\langle B \rangle \text{ map(Function}\langle B(A) \rangle \ f, \text{Event}\langle A \rangle \ a)
\]

-- Optionally don't propagate

\[
\text{filterJust} :: \text{Event } (\text{Maybe } a) \rightarrow \text{Event } a \\
\text{Event}\langle A \rangle \text{ filterOptional(Event}\langle \text{Optional}\langle A \rangle \rangle \ \text{eoa})
\]

\[
\text{merge} :: \text{Event } a \rightarrow \text{Event } a \rightarrow \text{Event } a \\
\text{Event}\langle A \rangle \text{ merge(Event}\langle A \rangle \ a, \text{Event}\langle A \rangle \ b)
\]
Event simultaneity

If you split an event into two, are they...

- Simultaneous?
  - No concept of order
  - (ideally any ordering in the implementation is undetectable)
- Non-simultaneous (as in observer pattern)?
  - Processed in a specific order
Reactive system

EIGHT PRIMITIVES

never
map
filter
merge
hold
snapshot
lift
switch

TWO DATA TYPES

Event
Behaviour
Behaviour primitives

\[
\text{hold} :: \text{a} \rightarrow \text{Event a} \rightarrow \text{Behaviour a}\\
\text{Behaviour}<\text{A}> \text{ hold}(\text{A a}, \text{Event}<\text{A}> \text{ ea})
\]

\[
\text{snapshot} :: \text{Behaviour a} \rightarrow \text{Event b} \\
\rightarrow \text{Event (a, b)} \\
\text{Event}<\text{(A,B)> snapshot(\text{Behaviour}<\text{A}> \text{ a},} \\
\text{Event}<\text{B}> \text{ b})
\]

- **hold** turns a stream of events into a behaviour [a.k.a. **stepper**]
  - a = initial value

- **snapshot** ‘samples’ a behaviour’s value as at Event occurrence
Reactive system

EIGHT PRIMITIVES
never
map
filter
merge
hold
snapshot

TWO DATA TYPES
Event
Behaviour

lift
switch
Lift – any arity – 1/3

- Lift N-ary function (n >= 2) into behaviours
  - e.g. +

TX packets + RX packets = Total packets
Lift – any arity – 2/3

\[ \text{LiftA2} :: (A \to B \to C) \to \]
  Behaviour A \to
  Behaviour B \to
  Behaviour C

Behaviour\langle C \rangle \text{ lift2(}
  Function\langle C(A,B) \rangle f,
  Behaviour\langle A \rangle ba,
  Behaviour\langle B \rangle bb) \]
Lift – any arity - 3/3

Function on values

→

Function on behaviours
Reactive system

EIGHT PRIMITIVES
never
map
filter
merge
hold
snapshot
lift
switch

TWO DATA TYPES
Event
Behaviour
switch

- Allows the reactive network to change dynamically
- Use reactive logic to modify reactive logic
switch (behaviour)

switch :: Behaviour (Behaviour A) -> Behaviour A

Behaviour<A> switch(
    Behavior<Behaviour<A>> bba)
switch (behaviour)

- e.g. switch between video streams

```
fuzz :: Behaviour Image
channel1 :: Behaviour Image
channel2 :: Behaviour Image

let eChan1 = map (const channel1) eButton1
let eChan2 = map (const channel2) eButton2
let sel = hold fuzz (merge eChan1 eChan2)
let screen = switch sel
```
switch (behaviour)
switch (behaviour)
switch (behaviour)
switch (behaviour)

- e.g. switch between video streams

```
fuzz :: Behaviour Image
channel1 :: Behaviour Image
channel2 :: Behaviour Image

let eChan1 = map (const channel1) eButton1
let eChan2 = map (const channel2) eButton2
let sel = hold fuzz (merge eChan1 eChan2)
let screen = switch sel
```
switch (event)

switchE :: Behaviour (Event A) → Event A

Event<A> switchE(Behaviour<Event<A>> bba)
State 1/3

Behaviours are the ONLY way to keep state

Mutable variables
Behaviours hold state: can only change in the presence of events

Events are stateless, but they can sample the value of a behaviour:

\[
\text{snapshot} :: \text{Behaviour}\, a \rightarrow \text{Event}\, b \\
\rightarrow \text{Event}\, (a, b)
\]

\[
\text{Event}<(A,B)\rangle\, \text{snapshot}(\text{Behaviour}<A\rangle\, a, \\
\text{Event}<B\rangle\, b)
\]

Each is stateless by itself: Interaction $\Rightarrow$ state
ATOMICITY

- All state changes caused by a single event occurrence are atomic

- "Just don't have to care about..."
Example: Crate crush
Inputs & outputs

game :: Drawable p  -- ^ Crate
  -> Drawable p  -- ^ Grass
  -> Sound p  -- ^ Create
  -> Sound p  -- ^ Smash
  -> GameInput p
  -> Reactive (GameOutput p)

GameInput { giMouse = eMouse, giTime = time } =

data GameInput p = GameInput {
  giMouse :: Event (MouseEvent p),
  giTime :: Behavior Double
}

data GameOutput p = GameOutput {
  goSprite :: Behavior (Sprite p),
  goEffects :: Event (Sound p)
}
New crate 1/2

-- Event Point
let eNewPoint = filterJust $ snapshotWith (\mev rects ->
    case mev of
        MouseDown _ pt | not (any (pt `inside`) rects) -> Just pt
        _ -> Nothing
    ) eMouse rects

-- Event (Int, Point)
eIdNewPoint <- identify eNewPoint

-- | Pair each event occurrence with a unique id
identify :: Event a -> Reactive (Event (Int, a))
identify ea = do
    ident <- count ea
    return $ snapshotWith (\a ident ->
        (ident, a)) ea ident
New crate 2/2

-- Event (Int, Point)
eIdNewPoint <- identify eNewPoint

-- Event (Int, Behavior Rect)
let eIdNewCrate = execute $ fmap ($ (i, pt) -> do
  rect <- crate i pt crates time
  return (i, rect)
) eIdNewPoint

let eCreate = fmap ($ (id, crate) -> M.insert id crate) eIdNewCrate

-- Behavior (Map Int (Behavior Rect))
crateMap <- accum M.empty (merge eCreate eDestroy)
let eDestroy = filterJust $ snapshotWith ($mev crates ->
  case mev of
    MouseDown _ pt ->
      case filter (\(ident, rect) -> pt `inside` rect)
               crates of
          ((ident, rect):_) -> Just $ M.delete ident
          _ -> Nothing
      _ -> Nothing
) eMouse crates

-- Behavior (Map Int (Behavior Rect))
crateMap <- accum M.empty (merge eCreate eDestroy)
Switch to flatten beh. of beh.

```haskell
-- Behavior (Map Int (Behavior Rect))
crateMap <- accum M.empty (merge eCreate eDestroy)

-- Behavior [Behavior (Int, Rect)]
let crates0 = fmap (map (\(i, b) ->
  fmap (i, ) b) . M.toList) crateMap

-- Behavior (Behavior [(Int, Rect)])
let crates1 = fmap sequenceA crates0

-- Behavior [(Int, Rect)]
crates <- switch crates1

-- Behavior [Rect]
let rects = fmap (map snd) crates
```
let eSound = merge
  (fmap (const playCreate) eCreate)
  (fmap (const playDestroy) eDestroy)
sprites =
  fmap (mconcat . (drawGrass grassRect:) .
         map drawCrate) rects

return $ def { goSprite = sprites,
               goEffects = eSound }
Crate falling/stacking logic 1/3

crate :: Int
  -> Point
  -> Behavior [(Int, Rect)]
  -> Behavior Double
  -> Reactive (Behavior Rect)

crate me pt0 crates time =
Crate falling/stacking logic 2/3

let obstacles = fmap (crates ->
    -- add the grass to the list of obstacles
    [grassRect] ++
    -- exclude myself
    mapMaybe ((i, rect) ->
        if i /= me then Just rect
        else Nothing) crates
) crates

-- delta time
eDeltaT = fmap realToFrac (delta (flip subtract) time)

delta :: (a -> a -> a) -> Behavior a -> Event a a
delta minus a = snapshotWith minus (updates a) a
Crate falling/stacking logic 3/3

rec
  let eAccel = snapshotWith (\dt ((vel, sitting, _), pos, obstacles) ->
    ... SNIP (physics stuff) ...)

  ) eDeltaT (liftA3 (,,) velocity position obstacles)
  velocity <- integrate velPlus ((0, 0), False, 0) eAccel
  position <- integrate plus pt0 $ fmap (\(vel,_,dt) ->
    scale (1/dt) vel) (updates velocity)
  return (fmap (,,crateSize) position)
Stephen Blackheath

- Author of o/s **Sodium** reactive system for Haskell, C++ and Java

- [http://reactiveprogramming.org/](http://reactiveprogramming.org/)

- Other RP systems: Reactive banana, Flapjax, Elm, Scala.React, Elerea, Netwire