

# **React,** **F**unctional **R**eactive **P**rogramming for OCaml

Daniel Bünzli, freelance programmer

*<http://erratique.ch>*

*[daniel.buenzli@erratique.ch](mailto:daniel.buenzli@erratique.ch)*

**1**

**FRP**

**What & Why**

**Reactive**

**vs.**

**Transforma-  
tional**

**Functional**  
**vs.**  
**Imperative**

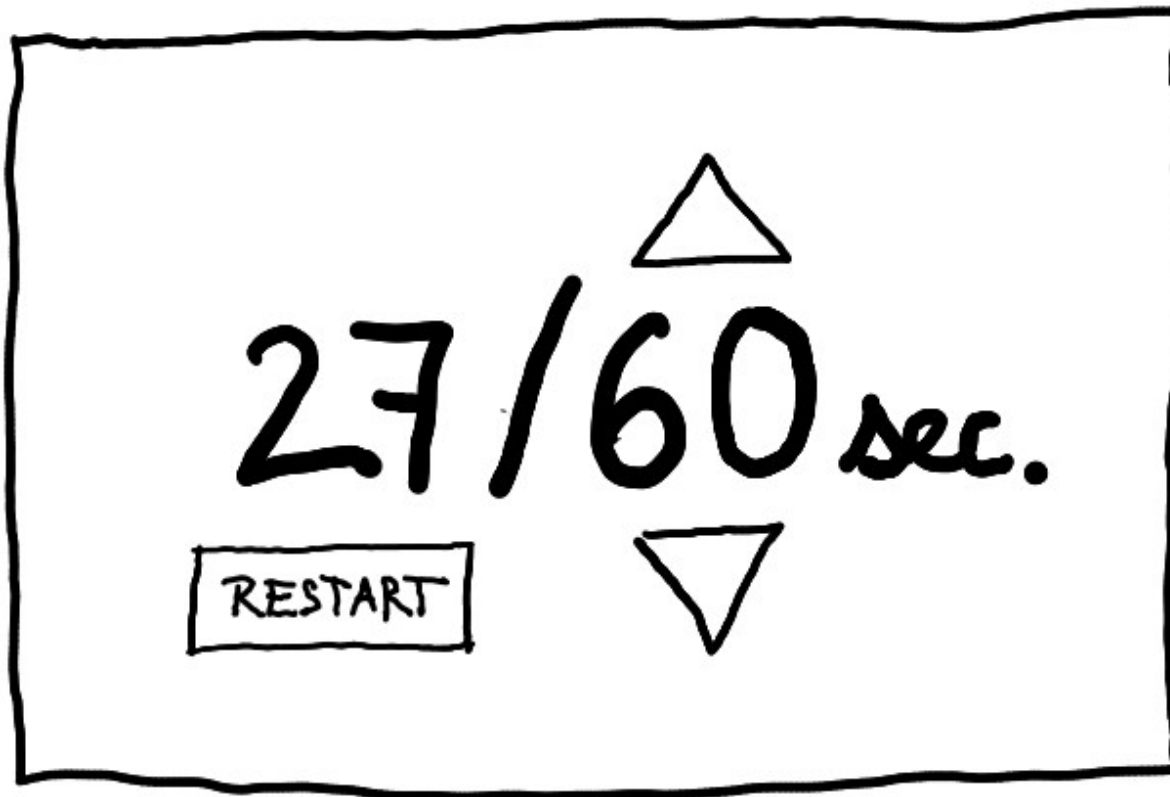
**Reactive**

**with**

**callbacks and**

**side-effects**

# A counter example



# Callback joy

```
let duration = ref 60
let elapsed = ref 0
```

```
let () =
  let on_trigger () =
    elapsed := min (!elapsed + 1) !duration;
    notify_elapsed_changed ()
  in
  Timer.each_second ~on_trigger
```

```
let duration_stepper =
  let on_change v =
    duration := v;
    elapsed := min !elapsed !duration;
    notify_duration_changed ();
    notify_elapsed_changed ();
  in
  Stepper.create ~value:!duration ~on_change
```

```
let restart_button =
  let on_click () = elapsed := 0; notify_elapsed_changed () in
  Button.create ~on_click
```

# Functional

Functional Reactive  
Animation, ICFP'97

Conal Elliott & Paul Hudak

# Signals & Events

```
type 'a signal = 'a S.t ≈ time -> 'a
```

```
type 'a event = 'a E.t ≈ time -> 'a option
```

# Combinators !

```
val E.map : ('a -> 'b) -> 'a E.t -> 'b E.t
```

```
val S.map : ('a -> 'b) -> 'a S.t -> 'b S.t
```

```
val S.hold : 'a -> 'a E.t -> 'a S.t
```

...

# FRP joy

```
let restart : unit E.t = RButton.create ()
let duration : int S.t = RStepper.create ~value:60
let secs : int S.t = RTimer.seconds ()

let elapsed : int S.t =
  let restart_t : int E.t = S.sample (fun _ t -> t) restart secs in
  let t0 : int S.t = S.hold (S.value secs) restart_t in
  S.l3 (fun d t t0 -> min d (t - t0)) duration secs t0
```

**2**

**React**

**FRP engine**

# Combinators

**Create**

**primitives**

**Updates**

# Create primitives

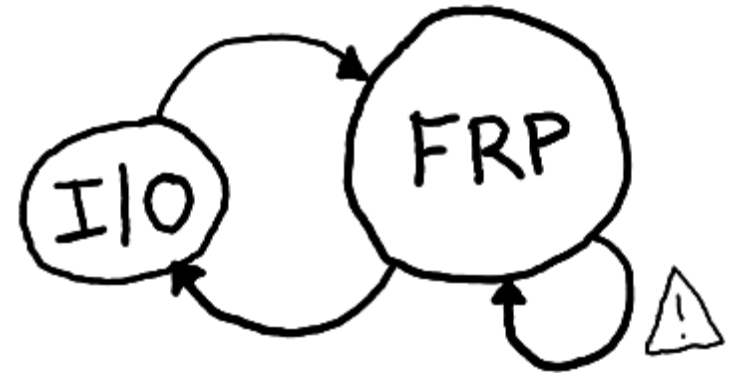
```
val E.create : unit -> 'a E.t * ('a -> unit)
```

```
val S.create : 'a -> 'a S.t * ('a -> unit)
```

# Callbacks ► FRP

```
let value = 60
let s_val : int S.t, set = S.create value in
let s = Stepper.create ~value ~on_change:set
```

# Program **structure**



```
(* create primitives *)
```

```
...
```

```
(* define derived signals & events *)
```

```
...
```

```
let () =
```

```
  while true do update_primitives () done
```

**Try it !**

*<http://erratique.ch/software/react>*

*<http://erratique.ch/talks/ocamlum-2010>*

# OCaml & FRP

## Froc

*<http://github.com/jaked/froc>*

## Concurrent cell

*<http://ccell.forge.ocamlcore.org/>*

# Other & FRP

## Haskell

*[http://haskell.org/haskellwiki/Functional\\_Reactive\\_Programming](http://haskell.org/haskellwiki/Functional_Reactive_Programming)*

## Javascript

*<http://flapjax-lang.org/>*

## Scala

*<http://lamp.epfl.ch/~imaier/>*

## Scheme

*<http://docs.plt-scheme.org/frtime/index.html>*

**?**

# Leaks

**Recursive values**

**Side-effects (eq.)**

**Runtime costs**

**Program struct.**