Functional Reactive Programming (FRP)

- Abstraction to Match a Domain
- Dataflow Languages

} Prehistory

- Denotational Semantic Design
- What is FRP?

} Early Design choices

- Goodbye denotational semantics
- Integrate with widespread languages
- Web GUI domain

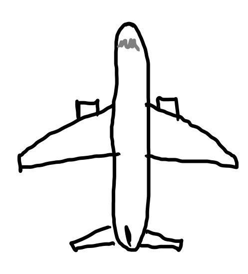
} Modern FRP Solutions

Prehistory: Reactive Systems

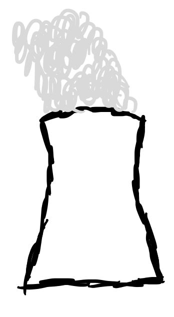
1968 — 1972 — ... — 1988

Analog Flight: Apollo 8 fly-by-wire → Military Planes ... → Airbus A320

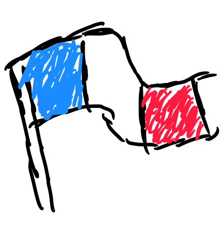
Joystick → [Levers / Software] → Rudder Control



Temp → [Circuits / Software] → Control Rods



1980s: Domain Specific Languages For Reactive Systems

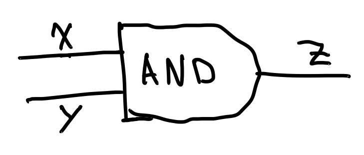


Esterel (Berry & Gonthier), Lustre (Halbwachs et.al.), Signal (Benveniste et.al.) → industry

Lucid (Wadge & Ashcroft) "Stream Programming"

Lustre Language: (Caspi, Pilaud, Halbwachs, Plaice) 1987 POPL

- Equations to relate variables for all of time



$z = x \text{ and } y \implies \forall t.\ z_t = x_t \text{ and } y_t$

- Discrete & implicit time

$$eqns ::= (id = e)^+$$

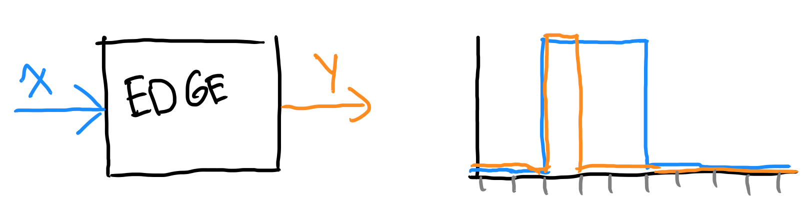
→ Recursive Equations!

$$\begin{aligned} e ::= &\ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\ &\mid e_1 + e_2 \\ &\ \vdots \\ &\mid c \\ &\mid id \end{aligned}$$

- Time Manipulation Primitives

$$\begin{aligned} e ::= \cdots &\mid e_1 \to e_2 \\ &\mid \text{pre}(e_1) \\ &\mid e_1 \text{ when } e_2 \\ &\mid \text{current}(e_1) \end{aligned}$$

Ex 1: Rising Edge of a signal ($\to$ and pre)




```
node EDGE (x: bool) returns (y: bool);
let
  y = false -> x and not pre(x);
tel
```

x	x_1	x_2	x_3	x_4 ...
pre(x)		x_1	x_2	x_3
y	F	$x_2 \wedge (\neg x_1)$	$x_3 \wedge (\neg x_2)$	$x_4 \wedge (\neg x_3)$

Ex 2: Sum of a signal (recursion)

```
node SUM(x: int) returns (y: int);
let
  y = x -> pre(y) + x;
tel
```

x	2	3	1	4	...
y	2	5	6	10	...
pre(y)		2	5	6	...

Ex 3: Sampling

when

x	x_0	x_1	x_2	x_3	x_4 ...
b	T	F	T	F	T ...
e = x when b	$e_0 = x_0$		$e_1 = x_2$		$e_2 = x_4$ ...

current

x	x_0		x_1		x_2
current(x)	x_0	x_0	x_1	x_1	x_2

What can go wrong?

1. "Nonsense": $x = 3x + 1$
   - ↳ Does not denote [Kahn 1974] & later
   - ↳ Cycles must have pre
2. Sensical, but "bad". Ex:

x	x_0	x_1	x_2	x_3	x_4
e	$e_0 = x_0$		$e_1 = x_2$		$e_2 = x_4$
z = x + e	$z_0 = x_0 + x_0$		$z_1 = x_1 + x_2$		$z_2 = x_2 + x_4$

   - ↳ Unbounded Memory!
   - ↳ Domain-Specific!
   - ↳ Semantics-Specific!
   - ↳ "clock calculus" ensures compatible clocks

Semantics: Labelled Transition Systems + external stimuli (Recall: π-calculus)

$$P_0 \xrightarrow{R_0} P_1 \xrightarrow{R_1} P_2 \xrightarrow{R_2} \cdots$$

with inputs I_0, I_1, I_2 into P_0, P_1, P_2

Equations:

$$eqn \xrightarrow[\sigma]{} eqn'$$

$\vdots$

Expressions:

$$\sigma \vdash e \xrightarrow{v} e'$$

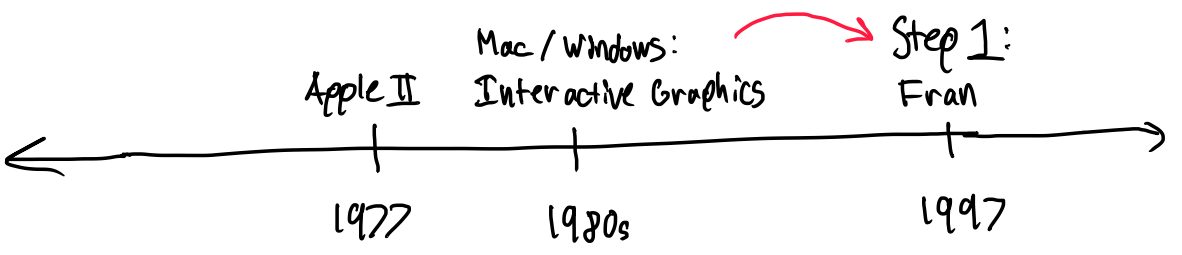
$$\frac{e_1 \xrightarrow{v} e_1'}{\sigma \vdash e_1 \to e_2 \xrightarrow{v} e_2}$$

$\vdots$

Implementation: Compile Lustre, Esterel, Signal → Finite Automaton in C, then call from driver C code

Prehistory Recap:

1. Declaratively state reactions to the world
2. Time is implicit and discrete
3. Time manipulation primitives
4. Rule-out terms: nonsense vs. undesirable — Subjectively Depends on Domain!
5. Semantics by LTS
6. Compiles to finite automata



Step 1: Functional Reactive Animation (Fran)

Elliott and Hudak, 1997

- Question: How to make interactive graphics functional language?
- Elliott: Seems a lot like reactive systems!
  - Continuous time rather than discrete
    → Composability for modelling animations
    → Hopefully avoid clock calculus mess
  - Embedded DSL in Haskell
  - Denotations first, implementation second

Fran Types:

data Behavior a = (--- hidden ---)

$T = \mathbb{R}$ (double)

data Event a = (--- hidden ---)

Ex: Behavior Shape is an animation

Fran Denotation & Primitives ("Combinators")

$$at : Behavior\ a \to (T \to a)$$

$$occ : Event\ a \to List\ (T, a)$$

time :: Behavior T
at[[time]] = λt. t

$lift_0$ :: a → Behavior a
$at[[lift_0\ x\ b]] = \lambda t.\ x$

$lift_1$ :: $(a_1 \to a_2) \to$ Behavior $a_1 \to$ Behavior a_2
$at[[lift_1\ f\ b]] = \lambda t.\ f\ (at[[b]]\ t)$

⋮

$(\Longrightarrow)$ :: Event a → $(T \to a \to b) \to$ Event b

predicate :: Behavior Bool → Event ()

snapshot :: Event a → Behavior b → Event (a, b)

until :: Behavior a → Event (Behavior a) → Behavior a

⋮

Fran Example

```
wiggle :: Behavior Double
wiggle = (lift1 sin) ((lift1 (*)) (lift0 pi) time)
wiggle = sin (pi * time)
```

Haskell Typeclass Magic

```
ball = move (integral wiggle, 0) circle
```

```
cycle :: Color → Color → T → Behavior Color
cycle x y t0 = x `until` (lbp t0 ⟹
                λt. λ_. cycle y x t)
```

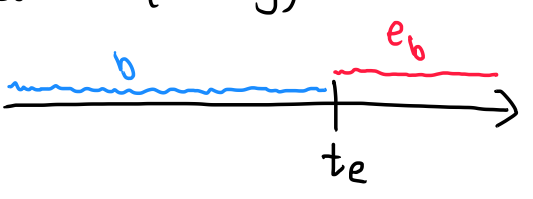
```
ballAnim t0 = withColor (cycle red blue t0) ball
```

Implementation

Strategy: 1. Choose a concrete (efficient) representation for Behavior a and Event a

2. Pull-based (polling)

b `until` e:



Reflection:

- Potentially good abstraction?
- Pull-based implementation ☹
- Denotational semantics is nice, but continuous functions $\mathbb{R} \to a$ hide a lot of complexity:

→ Contains many crazy terms

- Implementation ≠ Semantics
- Space leaks, etc.

Step 1.1: Are Continuous Time Semantics Really a good idea?

Functional Reactive Programming from First Principles
Wan and Hudak, 2000

- Compare continuous vs. discrete sampling semantics
- Take limit as sampling interval $\to 0$

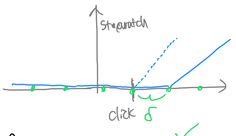
$\mathcal{at}$: Behavior $a \to \mathbb{R} \to \mathbb{R} \to a$ (start time, sample time) — Continuous Semantics ↪ Elliott

VS.

$\tilde{\mathcal{at}}^*$: Behavior $a \to \mathbb{R} \to \mathbb{R} \to a$ — Limit of Sampling Implementation

$\tilde{\mathcal{at}}^*[\![b]\!]\ t_0\ t = \lim_{\delta \to 0}$ ⟨Sampling implementation⟩

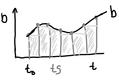
Ex 1: stopwatch = 0 `until` click $\Rightarrow \lambda t_c.$ $time - t_c$ — Time Capture Primitive



$\lim_{\delta \to 0} \tilde{\mathcal{at}}^*[\![\text{stopwatch}]\!] = \mathcal{at}[\![\text{stopwatch}]\!]$ ✓

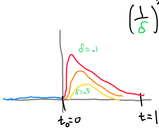
Ex 2: Integral Primitive

integral b = $\lambda t_s.$ cumulative sum of rectangles at sample points



$\mathcal{at}[\![\text{integral } b]\!]\ t_0\ t = \int_{t_0}^{t} (\mathcal{at}[\![b]\!]\ t_0\ t)\, dt$

bizarre = 0 `until` $(time > 0) \Rightarrow \lambda \delta.$ $\left(\frac{1}{\delta}\right)^2 time(1 - time)^{1/\delta}$



bad = integral bizarre

$\tilde{\mathcal{at}}^*[\![\text{integral bizarre}]\!]\ 0\ 1 = \lim_{\delta \to 0} \int_0^1 \left(\frac{1}{\delta}\right)^2 t(1-t)^{1/\delta}\, dt = \lim_{\delta \to 0} \frac{(1/\delta)^2}{(1/\delta + 1)(1/\delta + 2)} = 1$

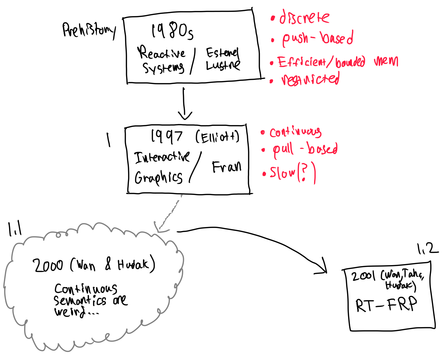
$\neq$

$\int_0^1 (\tilde{\mathcal{at}}^*[\![\text{bizarre}]\!]\ 0\ t)\, dt = \int_0^1 \lim_{\delta \to 0} \left(\frac{1}{\delta}\right)^2 t(1-t)^{1/\delta}\, dt = \int_0^1 0\, dt = 0$

$\Rightarrow$ Sampling $\not\to$ Semantics!

Lesson:

- Simple on paper $\neq$ Simple in practice



Step 1.2: RT-FRP

Wan, Taha, Hudak, 2001

- Discrete time
- Fewer primitives:

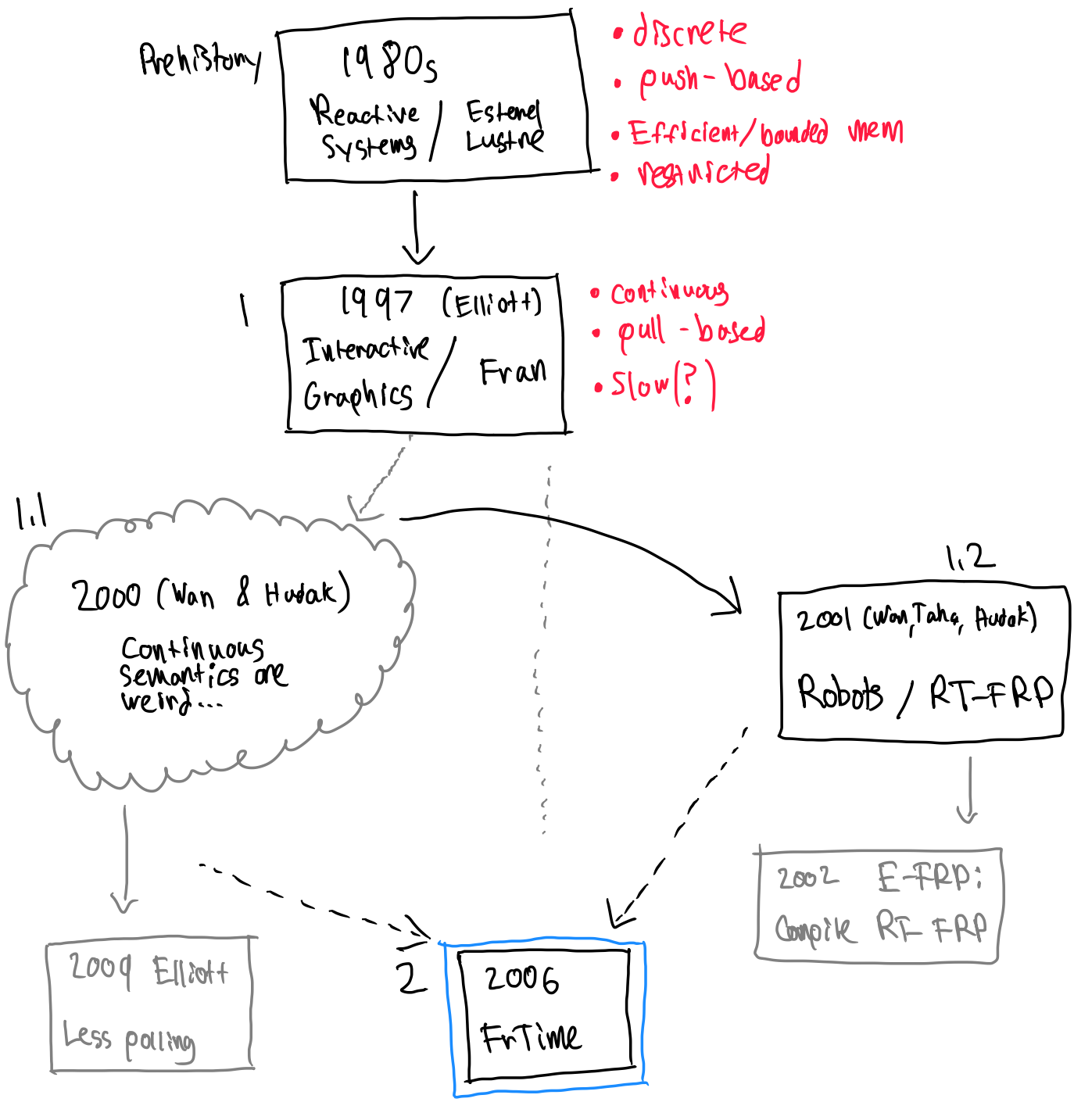
	Fran	Lustre
• time	time	x
• delay c s	x	c → pre(s)
• let snapshot x ← s_1 in s_2	lift	(implicit)
• s_1 switch on x ← e in s_2	until	if
• Recursion		

- Recursion:
  - Syntactically require tail calls during mutual recursion
    +
  - Wild, ad-hoc type system
    ⇓
  - Bounded Memory Usage → same guarantee as Esterel / Lustre
- Operational Semantics: Labelled Transition Systems!

$$\frac{\mathcal{E};k \vdash s \xrightarrow{c'}_{t,i} s'}{\mathcal{E};k \vdash \text{delay } c\ s \xrightarrow[c]{} \text{delay } c'\ s'}$$

→ Same as Lustre!

- Never gave examples demonstrating that it is more general than Esterel / Lustre!



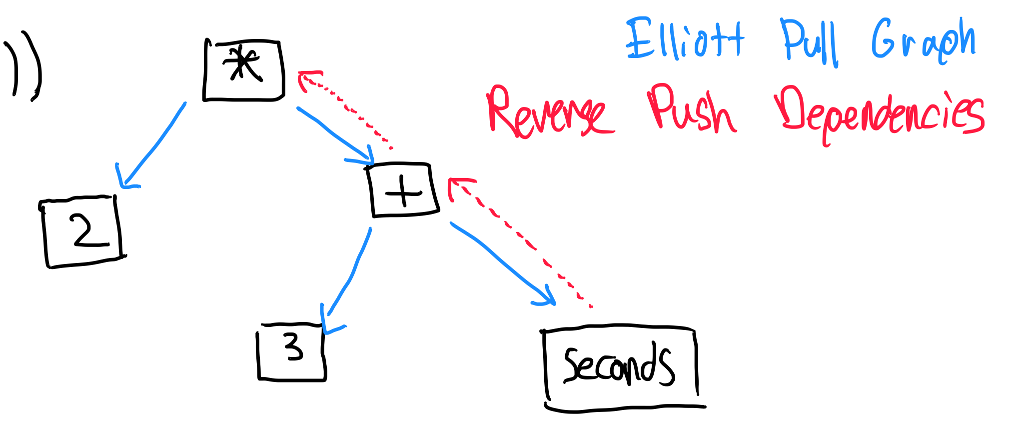
Step 2: Embedding Dynamic Dataflow in a Call-by-Value Language (FrTime)

Cooper & Krishnamurthi, 2006

- ~~Continuous semantics~~ discrete updates
- ~~Polling~~ push-based updates
- Higher-order ✓
- ~~Call-by-need~~ Scheme/Racket (use mutable state)
- Efficient ✓ (but weaken semantics)

Idea: Scheme Expression —eval→ Dataflow Graph (↓ Inputs, ↺ mutate state, ↓ outputs)

(* 2 (+ 3 seconds))



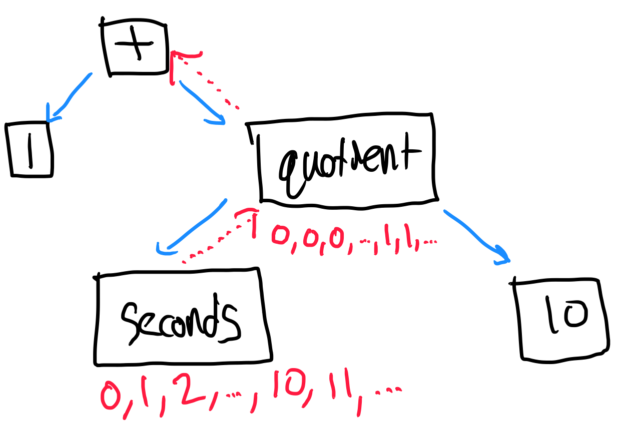
1. How to evaluate to a graph?

   Redefine primitives. Ex:

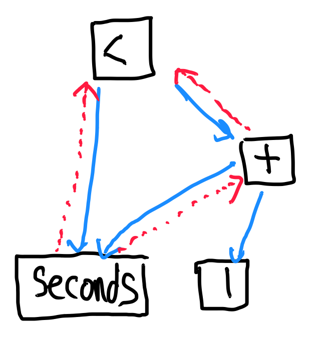
   (+ x y) ⟹ (if (or (signal? x) (signal? y))
   (make-signal (λ () (+ (current-value x)
   (current-value y))) x y)
   (+ x y))

2. Once graph is built, propagate changes as needed.

   Ex: (+ 1 (quotient seconds))



Ex: (< seconds (+ 1 seconds))



⟹ Update in Topological Order

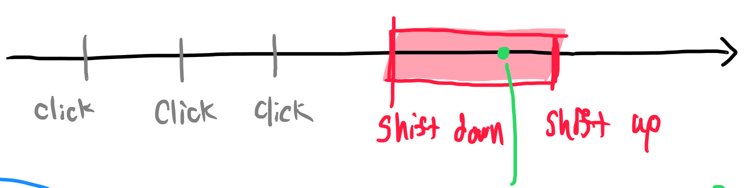
3. Complications:

   - Cycles?
     ↳ Add a special delay primitive
   - Dynamically Reconfigure the Graph
     ↳ Handle very carefully, avoid space leaks
     with some semantic "sacrifices"

Ex:

(define (count e) ...) : Event → Behavior Int
(define shift-down ...) : Behavior Bool
(define left-clicks ...) : Event

(define clicks-or-0
(if shift-down
(count left-clicks)
0))



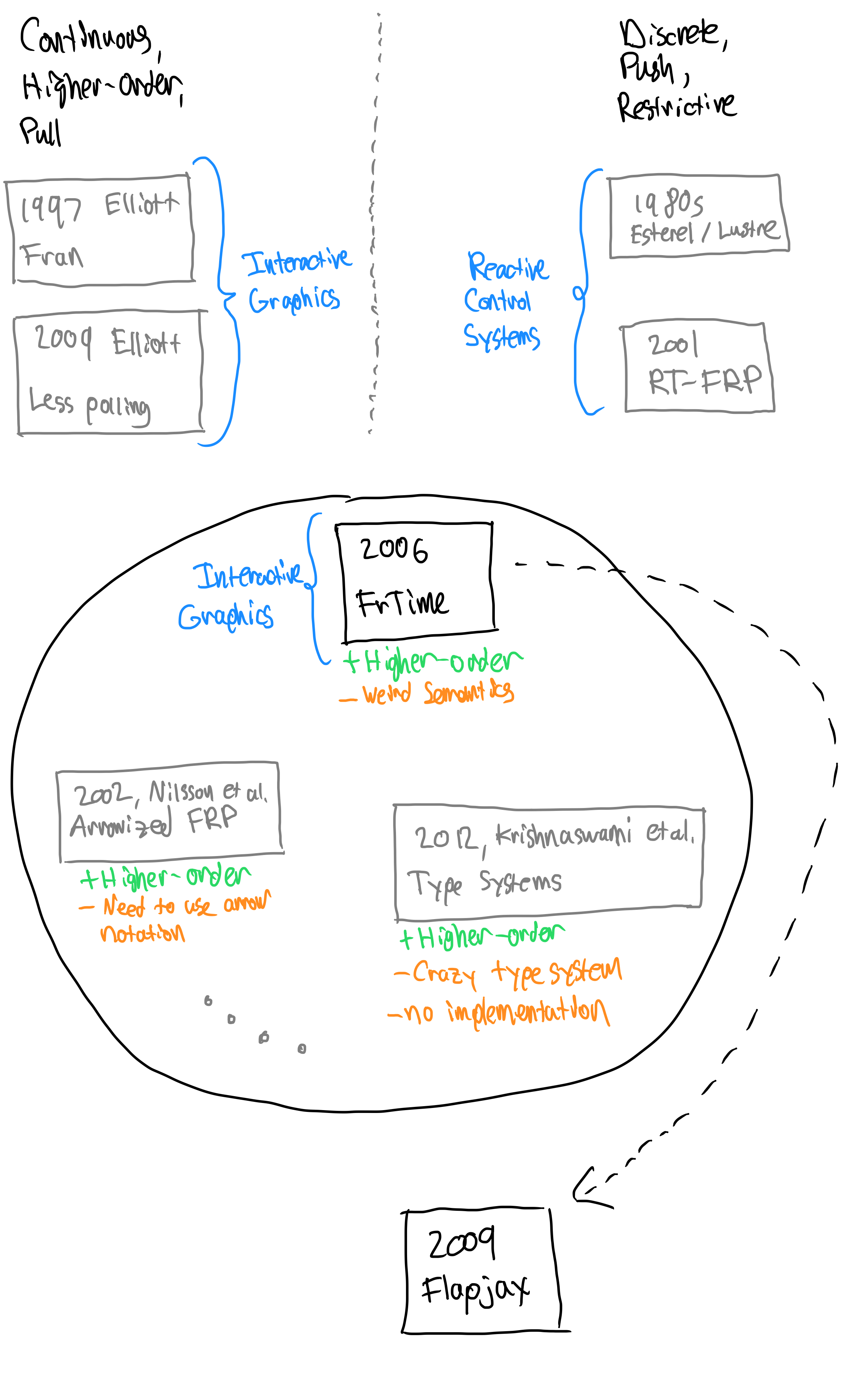
clicks-or-0 ?

⟹ 0

(define click-count (count left-clicks))
(define clicks-or-0-2
(if shift-down
click-count
0))

⟹ 3

→ Fran should give 3 for both



Step 3: Flapjax: A Programming Language for Ajax Applications

Meyerovich et al. 2009

— FrTime got the algorithm for push-update

— But for what domain?

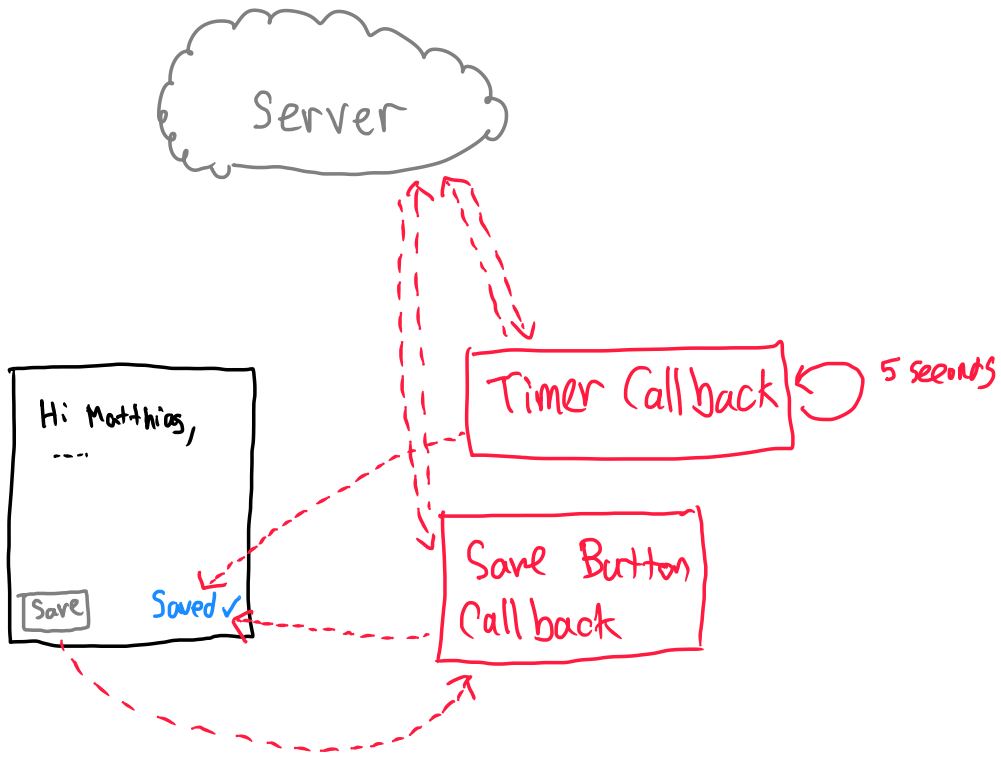
→ Interactive graphics? → cool for teaching!

→ GUIs?

→ Reactive control systems?

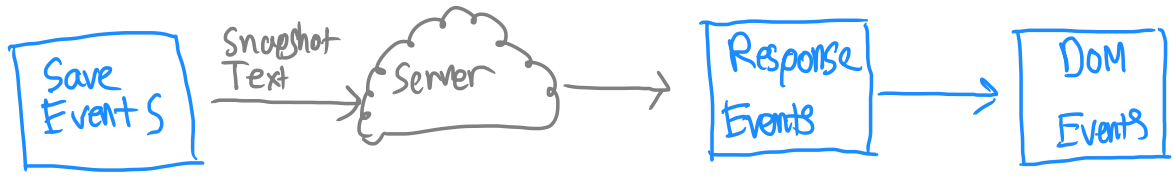
— Web GUIs! Callbacks = bad abstraction!

Ex:



- Complex policies for when to trigger save
- Policy defined by state mutation spread across multiple callbacks

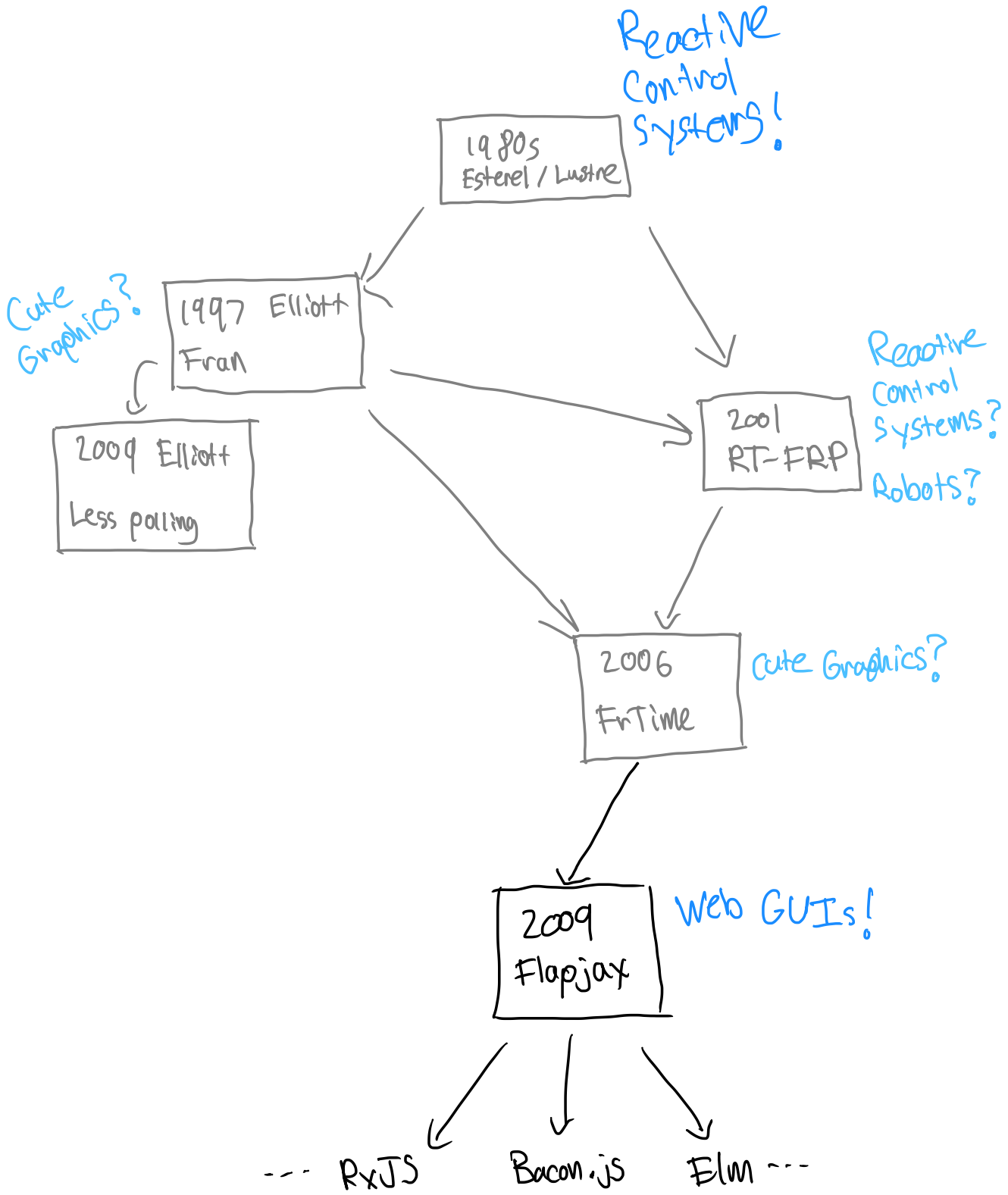
— Use Events / Behaviors as abstractions



e.g. merge(timer(5000), clicks(button))

— FrTime algorithm for Javascript

— Lots of Web-specific combinators!!



Reflections:

1. Don't be fooled by "simple" semantics

   → Implementation is what gets adopted into industry!

2. Design Space Explorations / Remove Assumptions

3. Abstractions follow Domains

   ↳ Take the domain seriously

4. What influences Language design?
   - Theory? Practice?