Modal FRP

Neel Krishnaswami
University of Cambridge

FLOLAC 2024
Taipei, Taiwan

Interactive Programs

Many forms of computation are <u>functions</u>:

- Compilers : Source ⟶ Object Code
- LaTeX : Source ⟶ PDF
- gzip : File ⟶ File

Interactive Programs

Other forms of computation are <u>interactive</u>:

- IDEs
- Word Processors
- Web browsers.

Interaction is Historical

Interactive Programs mix I and O

1. The user issues a command
2. The tool gives feedback
3. The user issues an updated command
4. The tool gives further feedback

User (or computer) actions are
<u>history-sensitive</u>!

How are GUIs currently built?

- The current state of the art is the event-based programing model

How are GUIs currently built?

- The current state of the art is the event-based programing model

- It dates back to the 1970s with the work on Smalltalk

Event-Based Programming

t:

type	callbacks
key	$[\lambda x. e_1 ; \lambda x. e_2 ; \lambda x. e_3 \ldots]$
click	$[\lambda x. t_1 ; \lambda x. t_2]$
touch	$[\lambda x. e]$
⋮	

Event-Based Programming

```
while (true) {
    let e = nextEvent();
    let fs = table[e.type];
    foreach (f in fs) {
        f(e.data)
    }
}
```

t:

type	callbacks
key	[λx.e_1 ; λx.e_2 ; λx.e_3 ...]
click	[λx.t_1 ; λx.t_2]
touch	[λx.e]
⋮	

Event-Based Programming

```
while (true) {
    let e = nextEvent();
    let fs = table[e.type];
    foreach (f in fs) {
        f(e.data)
    }
}
```

1. We wait for an event.

t:

type	callbacks
key	[$\lambda x.e_1$; $\lambda x.e_2$; $\lambda x.e_3$ ...]
click	[$\lambda x.t_1$; $\lambda x.t_2$]
touch	[$\lambda x.e$]
⋮	

Event-Based Programming

```
while (true) {
    let e = nextEvent();
    let fs = table[e.type];
    foreach (f in fs) {
        f(e.data)
    }
}
```

1. We wait for an event.

2. We look in the table for all callbacks of that type.

t:

type	callbacks
key	[$\lambda x.e_1$; $\lambda x.e_2$; $\lambda x.e_3$ ...]
click	[$\lambda x.t_1$; $\lambda x.t_2$]
touch	[$\lambda x.e$]
⋮	

Event-Based Programming

```
while (true) {
    let e = nextEvent();
    let fs = table[e.type];
    foreach (f in fs) {
        f(e.data)
    }
}
```

1. We wait for an event.

2. We look in the table for all callbacks of that type.

3. We execute the callbacks.

t:

type	callbacks
key	[$\lambda x.e_1$; $\lambda x.e_2$; $\lambda x.e_3$ ...]
click	[$\lambda x.t_1$; $\lambda x.t_2$]
touch	[$\lambda x.e$]
⋮	

Event-Based Programming

```
while (true) {
    let e = nextEvent();
    let fs = table[e.type];
    foreach (f in fs) {
        f(e.data)
    }
}
```

1. We wait for an event.
2. We look in the table for all callbacks of that type.
3. We execute the callbacks.
4. We wait for the next event.

t:

type	callbacks
key	[$\lambda x.e_1$; $\lambda x.e_2$; $\lambda x.e_3$ ...]
click	[$\lambda x.t_1$; $\lambda x.t_2$]
touch	[$\lambda x.e$]
⋮	

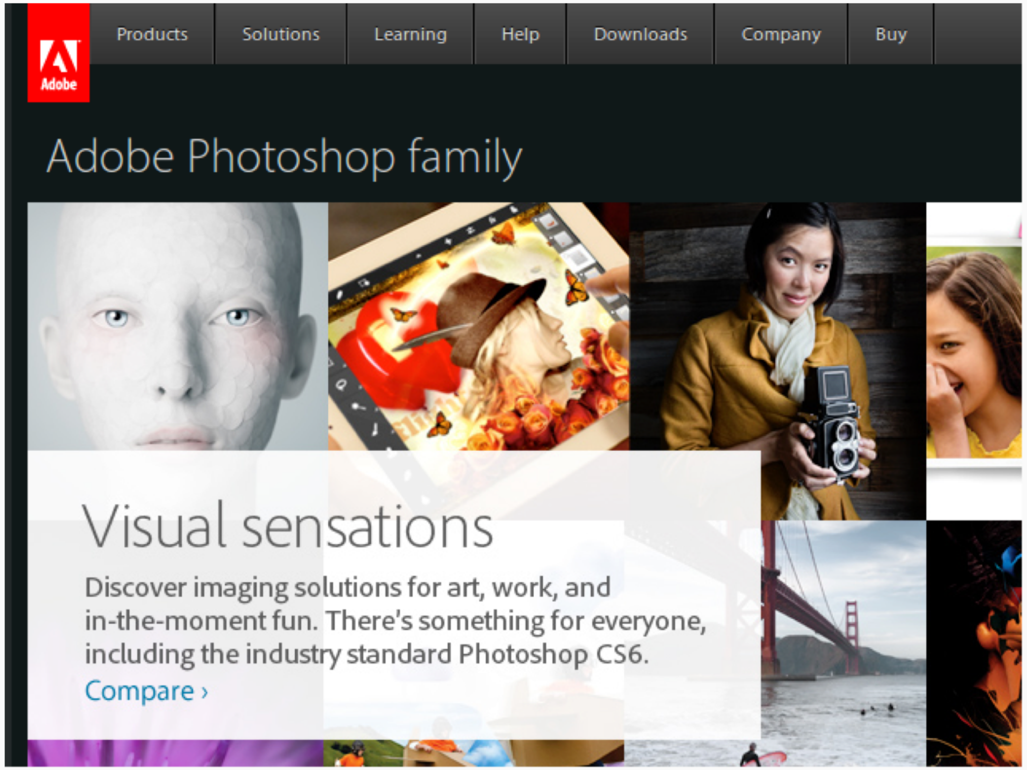
Event-Based Programming

This code is:

- Higher-Order
- Imperative
- Concurrent

This is <u>very</u> difficult!

GUIs are Hard



- UI code < 1/3 codebase
- But majority of bugs
- GUIs are harder to write than optimized image processing code!

FRP

In 1997, Hudak and Elliot proposed

Functional Reactive Programming

FRP

Idea: replace state with streams

State:	h	e	l	l	o	w	o	r	l	d	...
Time:	-1	0	1	2	3	4	5	6	7	8	...

Then an interactive program is a function

$$f : \text{Stream}(\text{Input}) \to \text{Stream}(\text{Output})$$

FRP

Streams have a clear API:

$\text{head} : \text{Stream}(A) \to A$

$\text{tail} : \text{Stream}(A) \to \text{Stream}(A)$

$\text{cons} : A \times \text{Stream}(A) \to \text{Stream}(A)$

$\text{map} : (A \to B) \to \text{Stream}(A) \to \text{Stream}(B)$

$\text{fix} : (A \to A) \to A$

FRP

Much state can be replaced with recursively-defined streams

$$\text{count} : \mathbb{N} \to S(\mathbb{N})$$

$$\text{count } n = \text{cons}(n, \text{count}(n+1))$$

$$\text{count}(0) = [0, 1, 2, 3, 4, \ldots]$$

FRP

Streams can be manipulated with ordinary functional programming:

map (fun n → n * 2) (count 0)

= [0, 2, 4, 6, 8, . . .]

Problems with FRP

- When programs are correct
  FRP programs are beautiful
- When programs are wrong
  FRP programs are very
  hard to debug

Problem #1: Causality

```
trade : S(Price) → S(Trade)

trade ps =
    let today = head ps
    let tomorrow = head (tail ps)
    let order = if today < tomorrow then Buy else Sell
    cons (order, trade (tail ps))
```

- This mathematically well-defined
- But it is not <u>causal</u>

Making Streams Causal

Introduce $\bullet A$ "later an A". Then

$$\text{head} : S(A) \to A$$
$$\text{tail} : S(A) \to \bullet S(A)$$
$$\text{Cons} : A \times \bullet S(A) \to S(A)$$
$$\text{map} : (A \to B) \to S(A) \to S(B)$$
$$\text{fix} : (\bullet A \to A) \to A$$

Making Streams Causal

```
trade : S(Price) → S(Trade)

trade ps =
    let today = head ps
    let tomorrow = head (tail ps)
    let order = if today < tomorrow then Buy else Sell
    cons (order, trade (tail ps))
```

Making Streams Causal

```
trade : S(Price) → S(Trade)

trade ps =
      let today = head ps
      let tomorrow = head (tail ps)
      let order = if today < tomorrow then Buy else Sell
      cons (order, trade (tail ps))
```

head : $S(\text{Price}) \to \text{Price}$

(tail ps) : $\bullet S(\text{Price})$

$\Big\}$ $S(\text{Price}) \neq \bullet S(\text{Price})$

Problem #2: Space Leaks

A Good Program

$\text{const} : \mathbb{N} \to S(\mathbb{N})$
$\text{const}\ n = \text{cons}(n, \text{const}(n))$

A BAD Program

$\text{const} : S(\mathbb{N}) \to S(S(\mathbb{N}))$
$\text{const}\ n = \text{cons}(n, \text{const}(n))$

These programs are identical

Only the types are different

Streams Abstract State

t = 0 a b c d e f

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1		b	c	d	e	f

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1		b	c	d	e	f
t = 2			c	d	e	f

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1		b	c	d	e	f
t = 2			c	d	e	f
t = 3				d	e	f

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1		b	c	d	e	f
t = 2			c	d	e	f
t = 3				d	e	f
t = 4					e	f

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1		b	c	d	e	f
t = 2			c	d	e	f
t = 3				d	e	f
t = 4					e	f

The const function has to save more and more state

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2			c	d	e	f
t = 3				d	e	f
t = 4					e	f

The const function has to save more and more state

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2	a	b	c	d	e	f
t = 3				d	e	f
t = 4					e	f

The const function has to save more and more state

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2	a	b	c	d	e	f
t = 3	a	b	c	d	e	f
t = 4					e	f

The const function has to save more and more state

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2	a	b	c	d	e	f
t = 3	a	b	c	d	e	f
t = 4	a	b	c	d	e	f

The const function has to save more and more state

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2	a	b	c	d	e	f
t = 3	a	b	c	d	e	f
t = 4	a	b	c	d	e	f

At time n, n values have to be buffered!

Streams Abstract State

t = 0	a	b	c	d	e	f
t = 1	a	b	c	d	e	f
t = 2	a	b	c	d	e	f
t = 3	a	b	c	d	e	f
t = 4	a	b	c	d	e	f

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	
t = 2	a	b	c	d	e	f	
t = 3	a	b	c	d	e	f	
t = 4	a	b	c	d	e	f	

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	
t = 3	a	b	c	d	e	f	
t = 4	a	b	c	d	e	f	

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	36
t = 3	a	b	c	d	e	f	
t = 4	a	b	c	d	e	f	

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	36
t = 3	a	b	c	d	e	f	36
t = 4	a	b	c	d	e	f	

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	36
t = 3	a	b	c	d	e	f	36
t = 4	a	b	c	d	e	f	36

Compare the stream to the number 36

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	36
t = 3	a	b	c	d	e	f	36
t = 4	a	b	c	d	e	f	36

Compare the stream to the number 36

36 never changes!

Streams Abstract State

t = 0	a	b	c	d	e	f	36
t = 1	a	b	c	d	e	f	36
t = 2	a	b	c	d	e	f	36
t = 3	a	b	c	d	e	f	36
t = 4	a	b	c	d	e	f	36

The stream changes over time

36 does not - it is <u>stable</u>

Making Streams

Introduce $\square A$ — the stable values of A

All	$\mathbb{N}$	stable
No	$S(\mathbb{N})$	Stable
Some	$A \to B$	Stable

head : $S(A) \to A$

tail : $S(A) \to \bullet S(A)$

Cons : $A \times \bullet S(A) \to S(A)$

map : $\square(A \to B) \to S(A) \to S(B)$

fix : $\square(\bullet A \to A) \to A$

Fixing Const

$$\text{const} : \Box A \to S(A)$$

$$\text{const}\ (\text{box}\ a) = \text{cons}(a, \text{const}\ a)$$

Now const is defined only for stable arguments!

Hey, That Looks Familiar...

- $\Box A$ and $\bullet B$ look <u>very</u> familiar
- FRP indeed needs multimodal types!