Functional Reactive Rust

Building applications with streams and signals using Rust & Carboxyl

by Eduard Bopp at Lambda Days 2016

Motivation

1. Systems programming, trouble with event handling
2. Sold on FRP, but due to performance can't use Haskell, Clojure, Elixir, Elm…

Carboxyl

- FRP library written in safe Rust
- Originally built for game development
- On GitHub: aepsil0n/carboxyl
- On crates.io: carboxyl 0.2


Rust

Rust — Overview

- Systems programming language
- Near C performance
- Statically prevents segfaults
- Guarantees thread-safety
- Zero-cost abstractions
- Lots of functional programming features

Rust — Hello, World!

```rust
fn main() {
    println!("Hello, World!");
}
```

Rust — Let Bindings

Immutable

```
let x: i32 = 1;
```

Mutable

```
let mut y: i32 = 5;
y = 4;
println!("{}", y); // --> 4
```

Rust — Ownership

```
fn sum(v: Vec<i32>) -> i32 {
    v.into_iter()
        .fold(0, |s, x| s + x)
}

let v = vec![1, 2, 3];
let result = sum(v);
```

cannot use `v` *any longer!*

Rust — Borrowing

```
fn sum(v: &[i32]) -> i32 {
    v.iter()
        .fold(0, |s, x| s + x)
}

let v = vec![1, 2, 3];
let result = sum(&v);
```

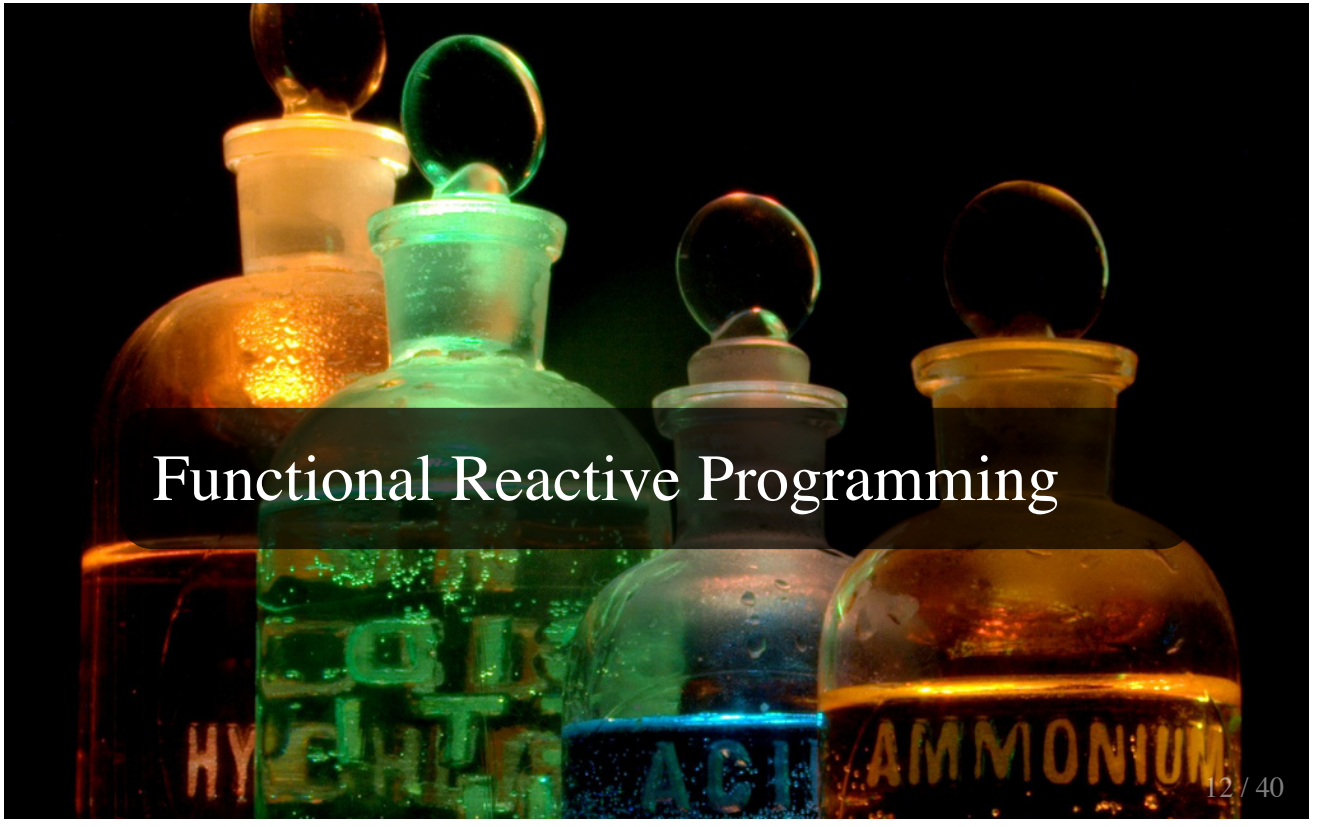
Rust — Mutable borrows

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

Rust — Structs & Enums

```rust
struct Point {
    x: f64,
    y: f64,
}

enum Option<T> {
    Some(T),
    None
}
```


Functional Reactive Programming
AMMONIUM

FRP — The basic idea

- Functional approach to handle how events affect application state
- `map`, `filter`, `fold`, etc. over time-varying data structures
- Comes in a million different flavours

Carboxyl's flavour of FRP

- Two types:
  - `Stream` is a sequence of discrete events
  - `Signal` is a value that varies *continuously* over time
- Time is implicit via transactions

Overview

Building streams & signals

```
extern crate carboxyl;
use carboxyl::{Sink, Stream, Signal};

let sink = Sink::new();
let stream = sink.stream();
let signal = stream.hold(3);

assert_eq!(signal.sample(), 3);

sink.send(5);
assert_eq!(signal.sample(), 5);
```

Iterate over stream

```
let sink = Sink::new();
let stream = sink.stream();

let mut events = stream.events();
sink.send(4);
assert_eq!(events.next(), Some(4));
```

Map

```
stream.map(|x| x * x)
```

Filter

```
stream.filter(|&x| x < 0)
stream.filter_map(|&x| if x > 2 { Some(x - 2) } else { None })
option_stream.filter_some()
```

Merge

```
stream_a.merge(&stream_b)
```

Fold

```
stream().fold(0, |a, b| a + b)
```

Snapshot

```
fn func(t: Time, value: Event) -> NewThing { /* ... */ }

let time: Signal<Time> = ...;
time.snapshot(&stream, func)
```

Lift

```
fn f(a: A, b: B) -> C { /* ... */ }

lift!(f, &as, &bs)
```

Only for up to arity 4, because of macros…

More

- Dynamic switching of streams and signals
- Coalesce to resolve events from the same transaction

Building applications

Crate ecosystem

- carboxyl-xyz for command line interface, system time, windowing
- Elmesque: port of Elm graphics API to Rust
- Piston: modular game engine
- Gfx, Glium: 3D graphics
- Glutin: windowing context
- lots more…

Demo time!

Application structure

```
fn app<W: StreamingWindow>(window: &W) -> Signal<View> {
    let context = context(window);
    let actions = context
        .snapshot(&events(window), intent)
        .filter_some();
    let state = actions.fold(init(), update);
    lift!(view, &context, &state)
}
```

adapted from Cycle.js & Elm architecture for continuous time semantics

Context

signal part of the input

```
#[derive(Clone)]
enum Context { Hover, Free }

fn centered(size: Dimension, position: Position) -> Position { /* ... */ }

fn hovers(position: Position) -> bool { /* ... */ }

fn context<W: StreamingWindow>(window: &W) -> Signal<Context> {
    lift!(
        |size, cursor|
            if hovers(centered(size, cursor)) { Context::Hover }
            else { Context::Free },
        &window.size(),
        &window.cursor()
    )
}
```

Events

discrete part of the input

```
#[derive(Clone)]
enum Event { Click }

fn clicks(event: ButtonEvent) -> Option<Event> { /* ... */ }

fn events<W: StreamingWindow>(window: &W) -> Stream<Event> {
    window.buttons()
        .filter_map(clicks)
}
```

Actions

```
#[derive(Clone)]
enum Action { Toggle }

fn intent(context: Context, _: Event) -> Option<Action> {
    match context {
        Context::Hover => Some(Action::Toggle),
        Context::Free => None
    }
}
```

...

```
let actions = context
    .snapshot(&events(window), intent)
    .filter_some();
```

State

```
type State = bool;

fn init() -> bool { false }

fn update(current: State, _: Action) -> State { !current }
```

...

```
let state = actions.fold(init(), update);
```

View

```
type View = Vec<Form>;

fn hello() -> Form { /* ... */ }
fn button(color: Color) -> Form { /* ... */ }

fn view(context: Context, state: State) -> View {
    let color = match context {
        Context::Hover =>
            if state { light_blue() } else { light_orange() },
        Context::Free =>
            if state { blue() } else { orange() }
    };
    vec![button(color), hello()]
}
```

...

```
let output = lift!(view, &context, &state);
```

More…

- Composition
- Effects

Implementation

Inspirations

- Originally similar to Sodium
- Later looked at Push-Pull FRP by C. Elliott
- *But:* Purely functional approach is not feasible with strict evaluation and lifetimes

Implementation strategy

- Use observer pattern internally
- Make discrete changes atomic using transactions
- Signals are impure functions (system time, analog instruments, etc.)

Current pain points

- Lots of atomic reference counting
- Lots of heap allocation and pointer indirection
- Transactions are pretty dumb
  - Global mutex prevents parallel event processing

Ressources

Rust

- https://users.rust-lang.org/
- https://www.reddit.com/r/rust/

Carboxyl

- https://crates.io/crates/carboxyl
- https://github.com/aepsil0n/carboxyl

Thank you!

- Twitter: @aepsil0n
- Email: eduard.bopp (at) aepsil0n.de