Functional Reactive Programming

Maximilian Krome

Specification Languages for Verification

Mathematical Roots

- Formal *mathematical* description
- Provability
- ”What” versus ”How”

Concepts of Functional Programming

- Referential Transparency (No reassignment)
- Purity (No side effects)
- First Class or Higher Order Functions
- Recursion

Functional Programming Example

Quicksort

```
qsort :: (Ord a) => [a] -> [a]
qsort []       = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
        where   less = filter (<x)  xs
                more = filter (>=x) xs
```

Side Effects

So:

Side effects are against the programming paradigm.

But:

Side effects are required for a program to interact with its environment or users.

Functional Reactive Animation

- Authors: Paul Hudak and Conal Elliott
- First appearance: International Conference of Functional Programming 1997
- Purpose: Creation of (interactive) animation
- Signals: Behaviours and Events; both first class
- Implementation: Embedded in Haskell, running on HUGS (Haskell User Gofer System)

Features

- **Time**: Is a Behaviour
- **liftn** : Maps n signals to one other via a function parameter
- **timeTransform**: Accelerates/Delays behaviours
- **integrate**: Integrates a numeric behaviour
- **untilB**: Switching of signals

Reactivity

Red-Green-Cycle

```
colorCycle t0 = red 'untilB' lbp t0 *=>
        \t1 -> green 'untilB' lbp t1 *=>
        \t2 -> colorCycle t2
```

Integration

Mouse-Follower

```
followMouseRate im t0 = move offset im
      where offset = integral rate t0
            rate = mouse t0 .-. pos
            pos = origin2 .+^ offset
```

$$s(t) = \int v(t)dt + s_0$$

Behaviour attempts

1.
```
data Behavior a = Behavior (Time -> a)
```
   Not efficient enough

2.
```
data Behavior a = Behavior
            (Time -> (a, Behavior a))
```
   Sampling produces (simplified) new behaviour

3.
```
data Behavior a = Behavior
            (Time -> (a, Behavior a))
            ([Time] -> ([a], Behavior a))
```

Predicate

```
predicate (time * exp (4 * time) ==* 10) 0
```

Evaluates to true at an infinitely small time span

Interval Analysis

Remember:

```
(Ivl Time -> (Ivl a, Behavior a))
```

returns an interval of values the behaviour assumes in a certain time span.

Haskell is lazy

Only computes when the result is required

Advantages

- infinite data structures
- Spares unnecessary computation

Disadvantages

- time and space leaks
- Hard to predict resource requirements

Real Time FRP

- Authors: Zhanyong Wan, Walid Taha and Paul Hudak
- Purpose: Real Time Applications
- Implementation: As Intermediate Language

Properties for Real Time Development

- statically typed and type preserving
- signals are not first class
- bounded FRP term size
- constant time and space requirements for FRP commands

Features

1. **ext** is equivalent to **lift0** .
2. **let signal** x = s1 **in** s2 allows to access the current value of the first signal in an **ext** term forming the second signal
3. **delay** v s delays a signal by one tick. It displays v in the first tick.
4. s1 **switch on** x = ev **in** s2 switches from s1 to s2 whenever event ev occurs. Starts out as s1.

Syntax

Definition

$$e ::= x \mid c \mid () \mid (e_1,\ e_2) \mid e_\perp \mid \perp \mid \lambda x.e \mid e_1\ e_2 \mid fix\ x.e$$

$$v ::= c \mid () \mid (v_1,\ v_2) \mid v_\perp \mid \perp \mid \lambda x.e$$

$$\begin{aligned} s, ev ::= &\ input \mid time \mid ext\ e \mid delay\ v\ s \mid \\ &\ let\ signal\ x\ =\ s_1\ in\ s_2 \mid s_1\ switch\ on\ x\ =\ ev\ in\ s \end{aligned}$$

Compiling FRP into RT-FRP

Lift

lift1 e s ≡

```
let signal x = s in ext (e x)
```

lift2 e s1 s2 ≡

```
let signal x1 = s2 in
       let signal x2 = s2 in ext(e x1 x2)
```

Elm

- Author: Evan Czaplicki
- Purpose: GUIs for Web applications
- Implementation: Compiles into an intermediate language and then into JavaScript

Improvements over classic FRP

Classic FRP

- Continuous Signals
- Strict Event Ordering

Resulting Problems

- Needless Recomputation
- Global Delays

Solutions

- Only discrete Signals
- Memoization
- non Strict Event Ordering

Features

1. **lift** : Self explanatory
2. **async**: Marks independent code
3. **foldp**: It takes the current value of signal s and the accumulator a and feeds them to the function f. The result then replaces the accumulator. **foldp** f a s itself evaluates to a signal that contains all accumulator values.

Syntax

Definition

$$
\begin{aligned}
e ::= & () | n | x | \lambda x : \eta . e | e_1 \ e_2 | e_1 \oplus e_2 | \\
& \mathit{if} \ e_1 \ e_2 \ e_3 | \mathit{let} \ x \ = \ e_1 \ \mathit{in} \ e_2 | i | \\
& \mathit{lift}_n \ e \ e_1 \ , \ldots \ e_n | \mathit{foldp} \ e_1 \ e_2 \ e_3 | \\
& \mathit{async} \ e
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= & \mathit{unit} | \mathit{int} | \tau \to \tau' \\
\sigma ::= & \mathit{signal} \tau | \tau \to \sigma | \sigma \to \sigma' \\
\eta ::= & \tau | \sigma
\end{aligned}
$$

Signals of Signals

```
lift (foldp (+) 0) signalOfSignals
```

Evaluating < <1,2,3,4,...>, <5,6,7,8,...>, ...>

Remembering Everything

<<1,3,6,10,...>,
<5,11,18,26,...>, ...>

Evaluation only from the current time on

<<1,3>, <7, 15,...>, ...>

Graph representation

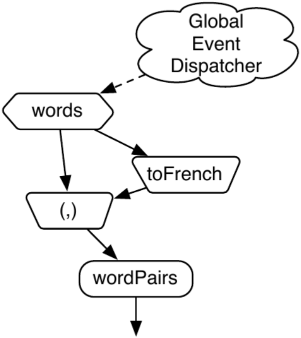

Figure: Program graph

Automaton

Definition

```
data Automaton a b =
        Step (a -> (Automaton a b, b))
```

Functions on Automatons

```
step : a -> Automaton a b
        -> (Automaton a b, b)
step input (Step f) = f input

run : Automaton a b -> b
        -> Signal a -> Signal b
run automaton base inputs =
        let step' input (Step f, _) = f input
        in lift snd (foldp step' (automaton,
           base) inputs)
```

"A Farewell to FRP"

Making signals unnecessary with The Elm Architecture

- Signals are hard to understand
- Signals are not used that much

Posted at the official website of Elm

Conclusion

Ideas of FRP are useful and it is flexible enough to suit a variety of applications, each approach to a different domain treats signals differently:

- Fran: First class behaviours and events
- RT-FRP: Only behaviours
- Elm: Only events

Event processing is key to most GUI frameworks

References I

- J. Backus.
  Can programming be liberated from the von neumann style?: A functional style and its algebra of programs.
  Commun. ACM, 21(8):613–641, Aug. 1978.
- C. Elliott and P. Hudak.
  Functional reactive animation.
  In *International Conference on Functional Programming*, pages 163–173, June 1997.
- M. Lipovača.
- J. M. Synder.

References II

Z. Wan, W. Taha, and P. Hudak.
Real-time FRP.
In *International Conference on Functional Programming (ICFP'01)*, 2001.