Functional Reactive Programming

Heinrich Apfelmus

Graphical User Interface

1968 – Douglas Engelbart
“Mother of all Demos”

mouse, hyperlinks, videoconferencing, shared-screen editing, …

custom programming languages

Object-Oriented Programming (OOP)

1973 – Xerox Alto Computer

Graphical User Interface on a desk

first object-oriented programming language: SmallTalk

Functional Reactive Programming (FRP)

1997 – Conal Elliott, Paul Hudak: "Functional Reactive Animation"

functional reactive programming
=
declarative programming with data that changes over time

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Scienc
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. Fran has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia anima-

- capturing and handling s
  even though motion inpu
- time slicing to update eac
  rameter, even though th
  vary in parallel; and

By allowing programmers
interactive animation, one ca
"how" of its presentation. Wit
not be surprising that a set
data types, combined with a
guage, serves comfortably for
trast with the common pract
guages to program in the co
presentation style. Moreover,
semantics, higher-order functi
ing, and systematic overloadin
erties for supporting modeled a
Fran provides these data types
Haskell [9].

Advantages of Modeli
The benefits of a modeling app
to those in favor of a function
gramming paradigm, and incl
tion, composability, and clean

Functional Programming

Data:
Functions

function

```
odd :: Int -> Bool
odd n = (n `mod` 2) == 1
```

function with
  function argument

```
filter :: (Int -> Bool)
       -> [Int] -> [Int]
```

example

```
filter odd  [1,2,3,4] = [1,3]
filter even [1,2,3,4] = [2,4]
```

Data:
Functions

function

```
inc :: Int -> Int
inc n = n + 1
```

function composition

```
f . g = \x -> f (g x)
```

example

```
inc2 = inc . inc
inc3 = inc . inc2
```

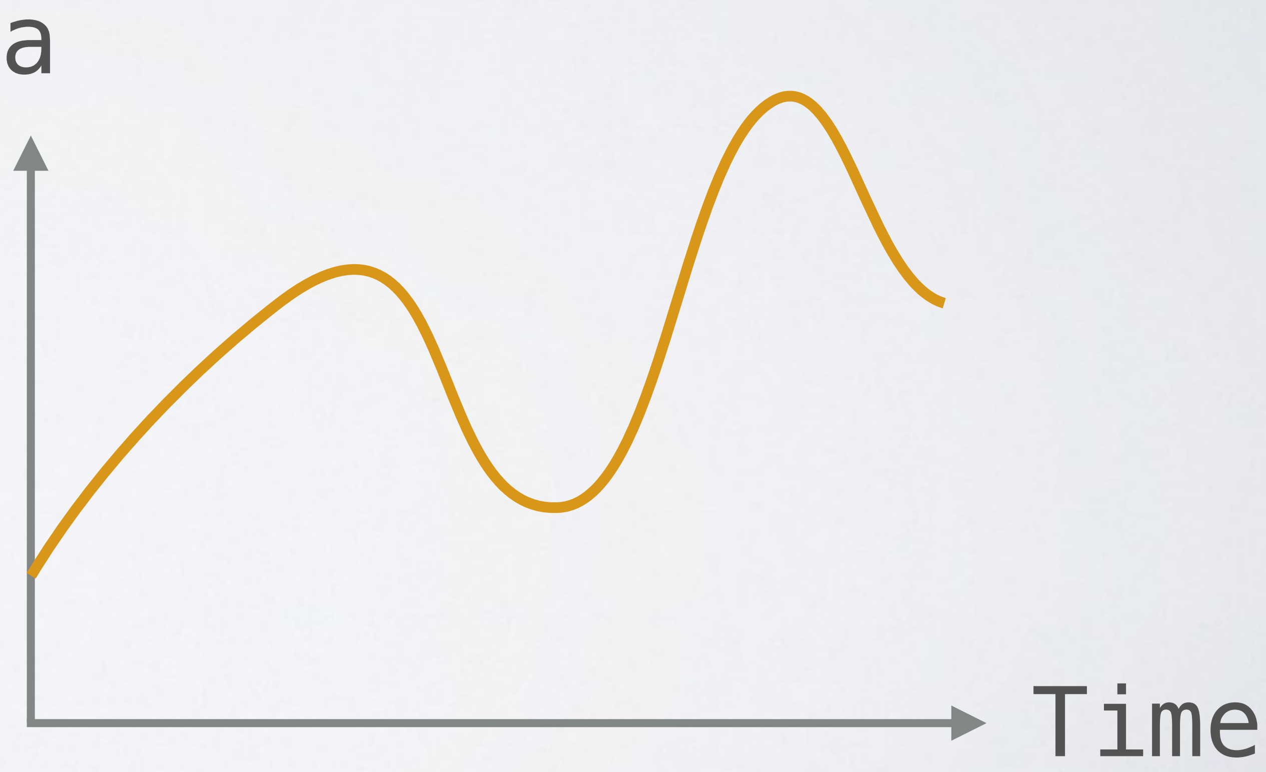
Functional Reactive Programming

Behavior

```
type Behavior a = Time -> a
```

“value that changes over time”

- position in animation
- text value in GUI
- volume in music



Example: Behavior

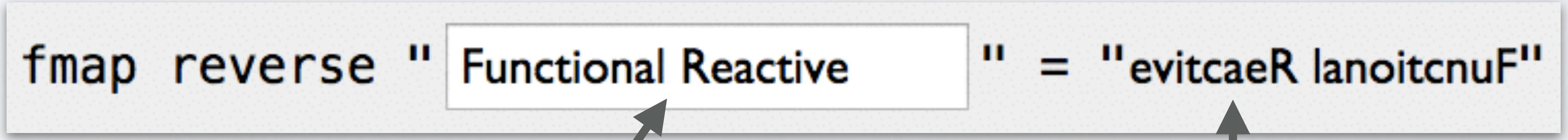
Pendulum

Behavior API

```
fmap :: (a -> b)
     -> Behavior a -> Behavior b
```

“apply function at every moment in time”

example



```
fmap reverse " Functional Reactive " = "evitcaeR lanoitcnuF"
```

`Behavior String` `Behavior String`

Example: Behavior

Text box

Data:
Infinite Lists

infinite list

```
[1..]
```

"never print everything!"

take first elements

```
take 4 [1..] = [1,2,3,4]
take 7 [1..] = [1,2,3,4,5,6,7]
```

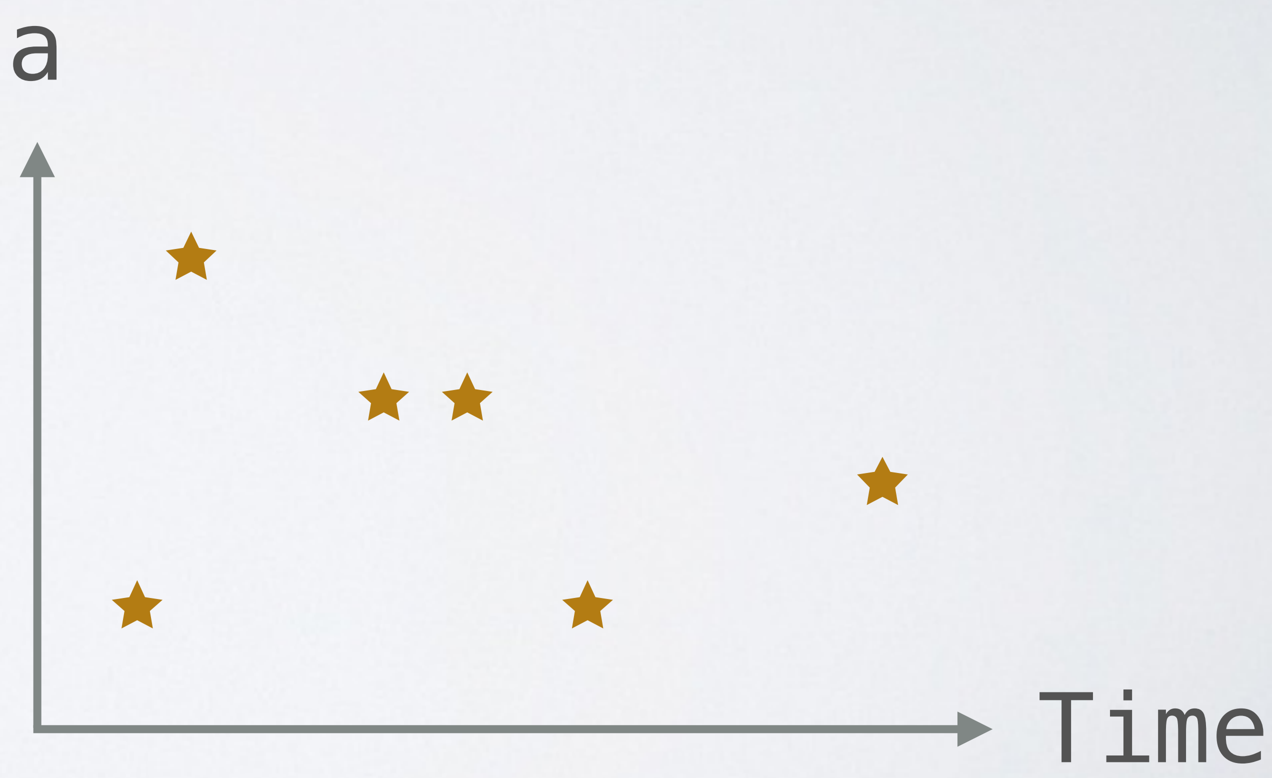
"potentially infinite"

Event

```
type Event a = [(Time, a)]
```

"occurrences that happen at particular times"

- mouse clicks in GUI
- notes in music

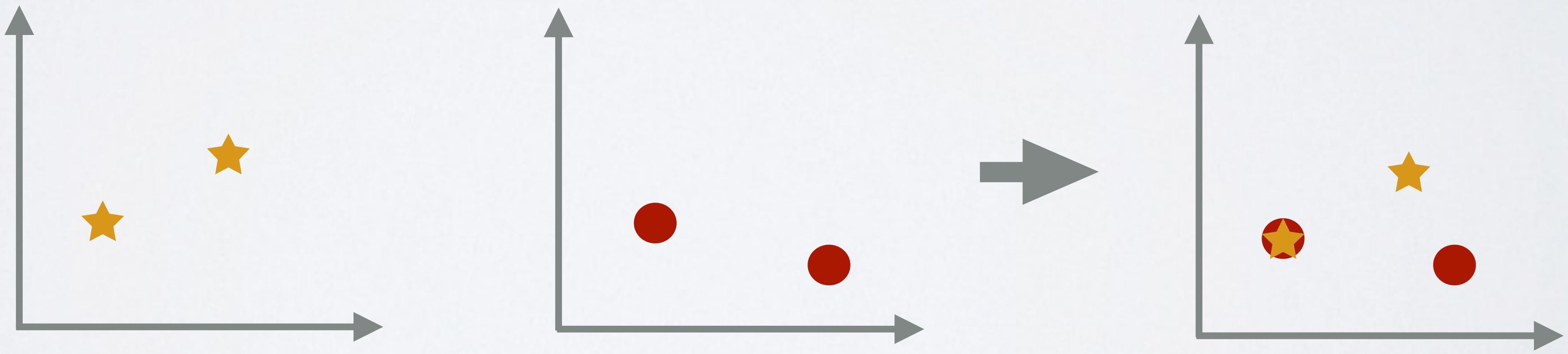


Event API

```
unionWith :: (a -> a -> a)
          -> Event a -> Event a -> Event a
```

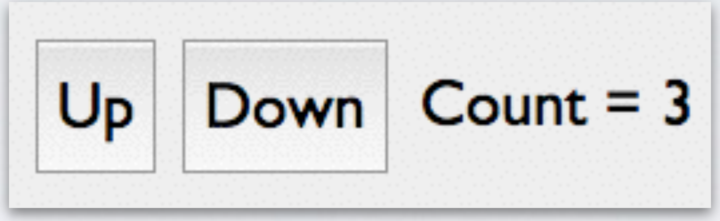
“merge event occurrences”

example



Event a Event a Event a

Why?
Traditional OOP

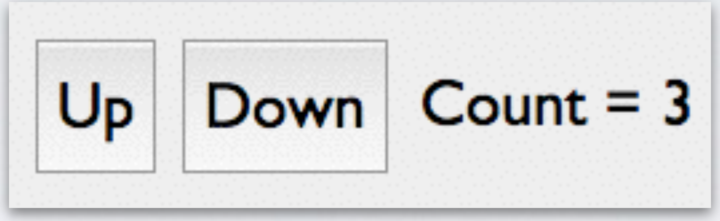


```
counter = Value(0)
```

```
on click up   do
    counter.update(\c -> c + 1)
```

```
on click down do
    counter.update(\c -> c - 1)
```

Why?
FRP



“specify all dependencies at declaration”

```
counter <- accumB 0 $ unionWith (.)
    ((\c -> c + 1) <$ click up  )
    ((\c -> c - 1) <$ click down)
```

Example: Event

Counter

FRP API

reactive-banana: 16 primitive functions

```
instance Functor     Behavior        -- fmap
instance Applicative Behavior        -- pure, (<*>)
instance Functor     Event           -- fmap
instance Monad       Moment          -- return, (>>=)
instance MonadFix    Moment          -- mfix

never     :: Event a
unionWith :: (a -> a -> a) -> Event a -> Event a -> Event a
filterE   :: (a -> Bool)   -> Event a -> Event a

(<@>)     :: Behavior (a -> b) -> Event a -> Event b
stepper   ::                 a -> Event a -> Moment (Behavior a)

valueB    :: Behavior a       -> Moment a
observeE  :: Event (Moment a) -> Event a
switchE   :: Event (Event  a) -> Moment (Event a)
switchB   :: Behavior a -> Event (Behavior a) -> Moment (Behavior a)
```

Languages & Libraries

- Haskell:
  - reactive-banana, threepenny-gui
  - reflex, reflex-dom
  - frpnow
- Java, Scala, C++, C#:
  - sodium
- Elm

Functional Reactive Programming

“specify all dependencies at declaration”