Functional Thinking for Fun & Profit

A **tech_tonic** talk by Srikumar K. S.
Chennai, 23 July 2015
Imaginea Technologies Inc.

Agenda

- Functional Thinking versus Functional Programming
- Introduce notation that's becoming common
- Present examples of thinking with the notation
- Present key engineering patterns
- ….First, on something I want to get out of the way.

Object Oriented Programming

Encapsulation

Inheritance

Object

Metaclass

Method

Class

Member

Message

Property

Virtual method

Prototype

Dependency injection

Functional programming

Closure

Function

Referential transparency

Immutable

Currying

Pure

Applicative

Monad

Type classes

Lazy

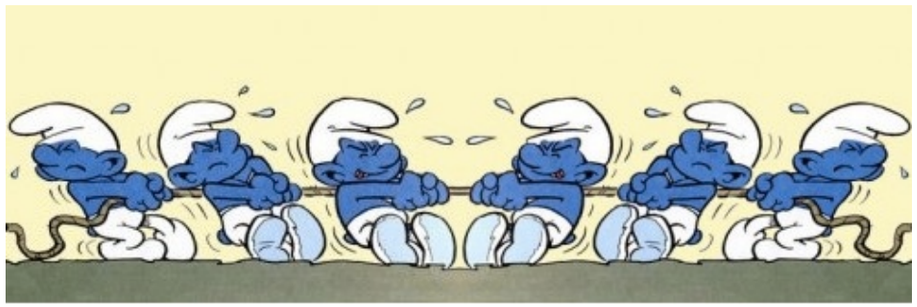
Functor

Declarative

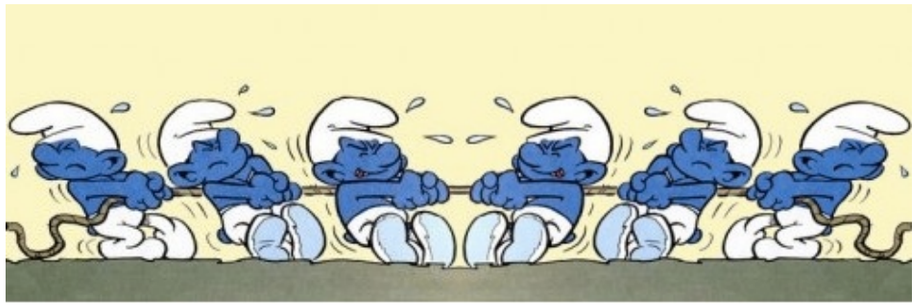
Monoid

Arrows

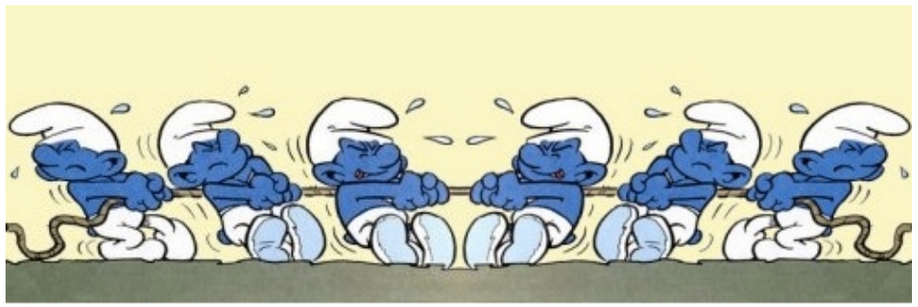
Category



FP vs. OOP



Functions vs. Classes



Closures vs. Objects

Koan

Anton: Student

Qc Na: Master

originally by Anton Van Straaten
in a conversation with Guy Steele in 2003.

Originally by Anton Van Straaten, in 2003, in a conversation with Guy Steele, famous for Scheme.

The venerable master Qc Na was walking with his student, Anton. Hoping to prompt the master into a discussion, Anton said "Master, I have heard that objects are a very good thing - is this true?" Qc Na looked pityingly at his student and replied, "Foolish pupil - objects are merely a poor man's closures."

Chastised, Anton took his leave from his master and returned to his cell, intent on studying closures. He carefully read the entire "Lambda: The Ultimate..." series of papers and its cousins, and implemented a small Scheme interpreter with a closure-based object system. He learned much, and looked forward to informing his master of his progress.

On his next walk with Qc Na, Anton attempted to impress his master by saying "Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures." Qc Na responded by hitting Anton with his stick, saying "When will you learn? Closures are a poor man's object." At that moment, Anton became enlightened.

Anton: Master, I've heard that objects are a very good thing - is this true?

Qc Na: Foolish pupil - objects are merely a poor man's closures.

Anton: Master, I have diligently studied the matter, and now understand that objects are truly a poor man's closures.

Qc Na: (hitting Anton with his stick) When will you learn. Closures are a poor man’s object.

At that very moment, Anton was enlightened.

Functional Thinking is thinking about **systems**,
based on a **mathematical** foundation.

Functional Thinking

- Spotlight on functions
- New notation for functions and type description
- Application to systems at all levels

Why is notation important?

$$cxxi \times xi = mcccxxxi$$

$$121 \times 11 = 1331$$

1. What were you doing in your head?
2. Which one do you think you'll bother to learn for actual use?

Why is notation important?

$$cxxviii \times cclvi = mmm\text{...xxxii times...}mmdcclxviii$$

$$128 \times 256 = 32768$$

1. The notation limits the magnitude of numbers available to our thinking. At most a few thousands.
2. The place value system places no limits on the magnitude of the numbers we can think about.

Why is notation important?

$$32768 = 3 \times 10^4 + 2 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$

1. Good notation hides a lot of regular behaviour under the hood so you can take it for granted.

Why is notation important?

$$m\frac{d^2\vec{r}}{dt^2} = \frac{-GMm\vec{r}}{|\vec{r}|^3}$$

1. More complex example - calculus as invented by Newton for gravity.
2. The mass “m” of a planet is the same value on both the sides (ignore a subtlety here), so cancel it out.

Why is notation important?

$$\frac{d^2\vec{r}}{dt^2} = \frac{-GM\vec{r}}{|\vec{r}|^3}$$

1. What does this mean?
2. It means that a tennis ball and a nearby feather at roughly the same distance from the sun and moving with the same velocity will always move together (ignoring attraction between them).
3. They will appear to be at rest relative to each other.

Why is notation important?

$$\vec{r} = \rho \vec{r'}$$
$$t = \tau t'$$

1. Now let's change the scale of a planet's orbit and the time it takes to go around the sun.
2. "Rho" is the spatial scaling factor
3. "Tau" is the temporal scaling factor

Why is notation important?

$$\frac{\rho}{\tau^2}\frac{d^2\vec{r'}}{dt'^2} = \frac{-\rho G M \vec{r'}}{\rho^3 |\vec{r'}|^3}$$

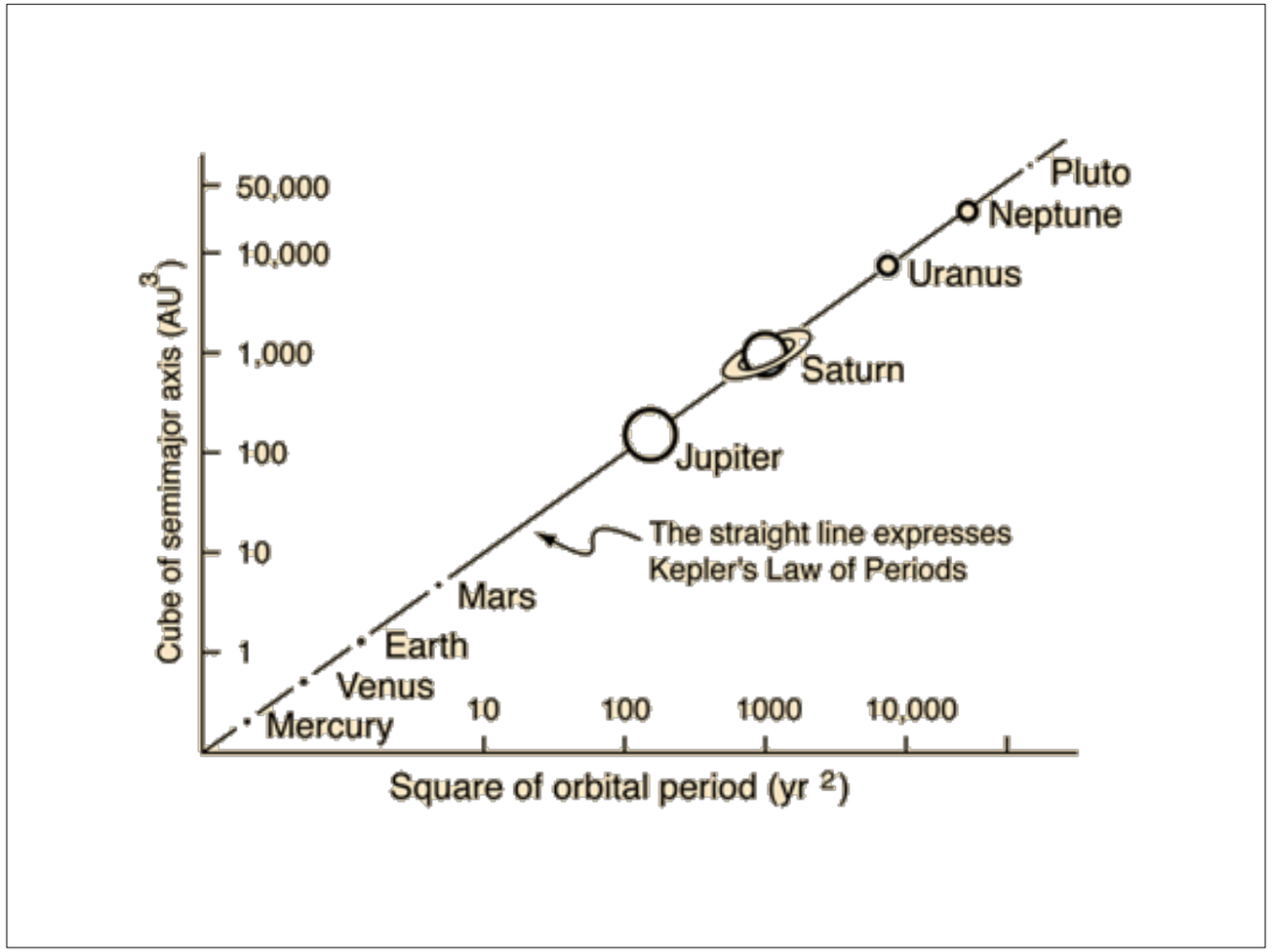
1. The “rho” at the top cancels out.
2.

Why is notation important?

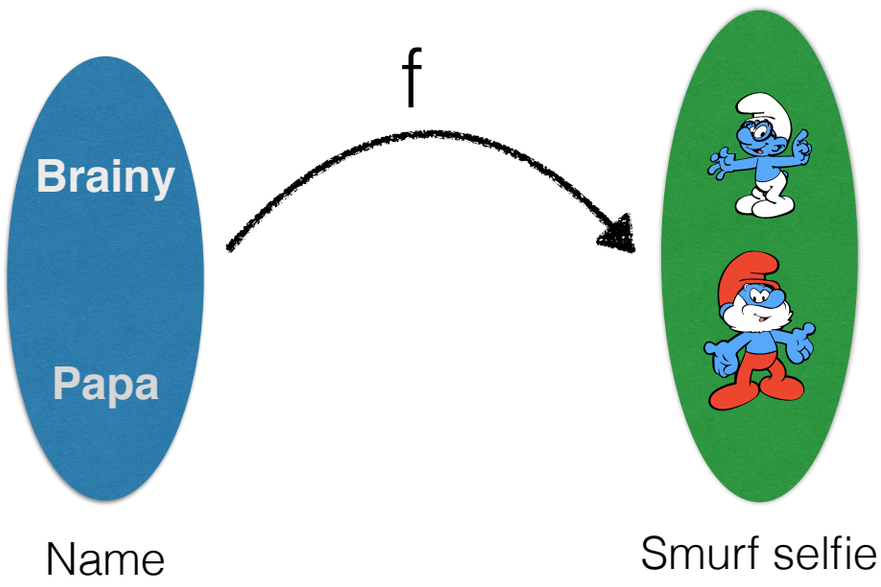
$$\rho^3 = \tau^2$$

$$\frac{d^2\vec{r'}}{dt'^2} = \frac{-GM\vec{r'}}{|\vec{r'}|^3}$$

1. If rho and tau satisfy the first equation, then the second equation has the same form as our original equation.
2. In other words, both r and r-prime are solutions of the gravity equation if rho^3 = tau^2.
3. That is Kepler's third law of planetary motion, which was determined empirically.
4. There is more gold in there, but we'll leave stop with that.
5. (Do I hear a whew!)

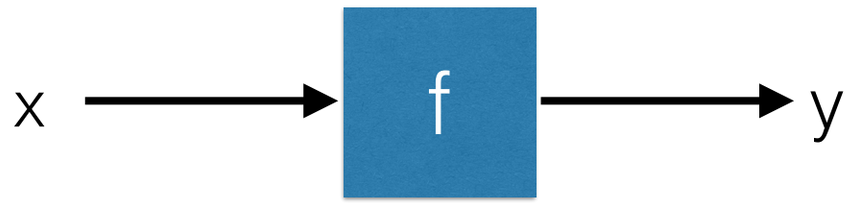

Pluto
Neptune
Uranus
Saturn
Jupiter
Mars
Earth
Venus
Mercury
50,000
10,000
1,000
100
10
1
Cube of semimajor axis (AU 3)
The straight line expresses
Kepler's Law of Periods
10
100
1000
10,000
Square of orbital period (yr 2)

Functions

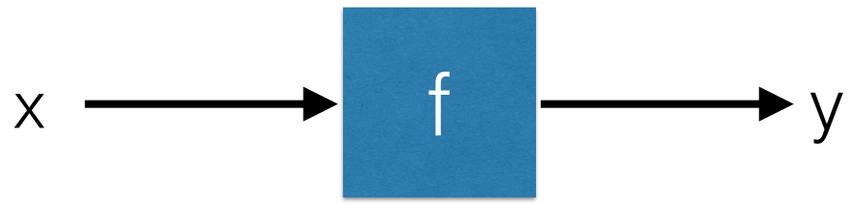


1. So we start with functions in the pure mathematical sense, and move on to functions as used in computing.
2. A function connects members of two different sets. Not going to go into the mathematical definition that we've all gone through in school and long forgotten, but that should do for now.

Functions

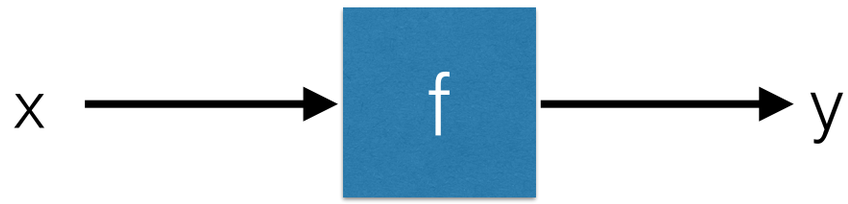


Functions



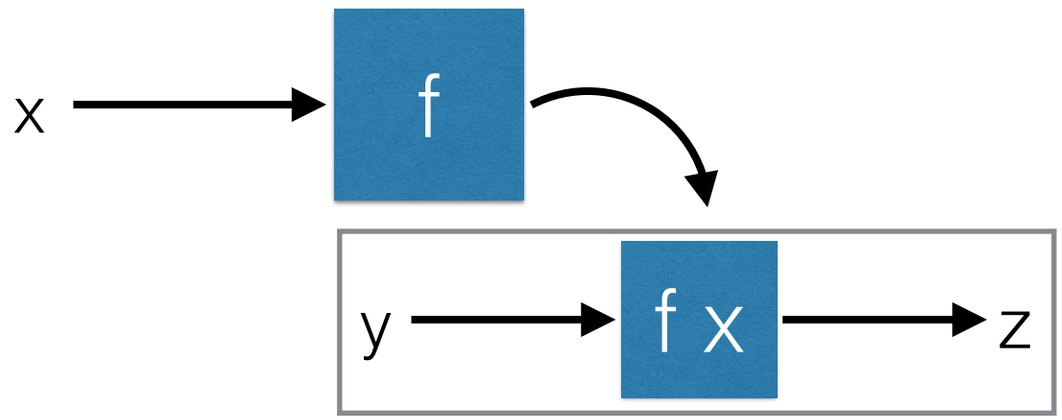

```
x :: X
y :: Y
f :: X → Y

y = f x
```

Functions

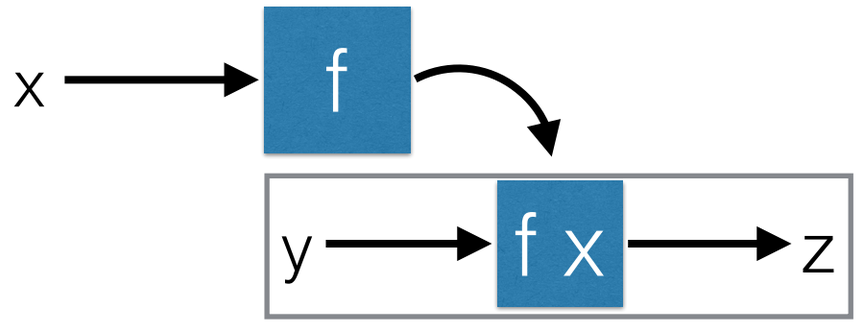



```
x :: X
f :: X → Y
f x :: Y
```

Functions that make functions

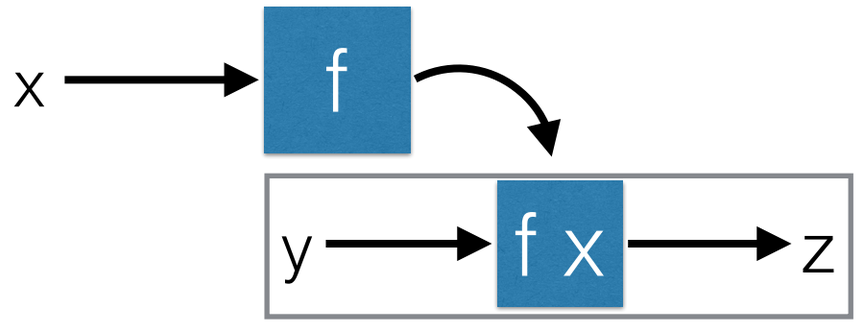


Functions that make functions




```
  x :: X
  y :: Y
  f :: X → (Y → Z)
f x :: Y → Z
f x y :: Z
```

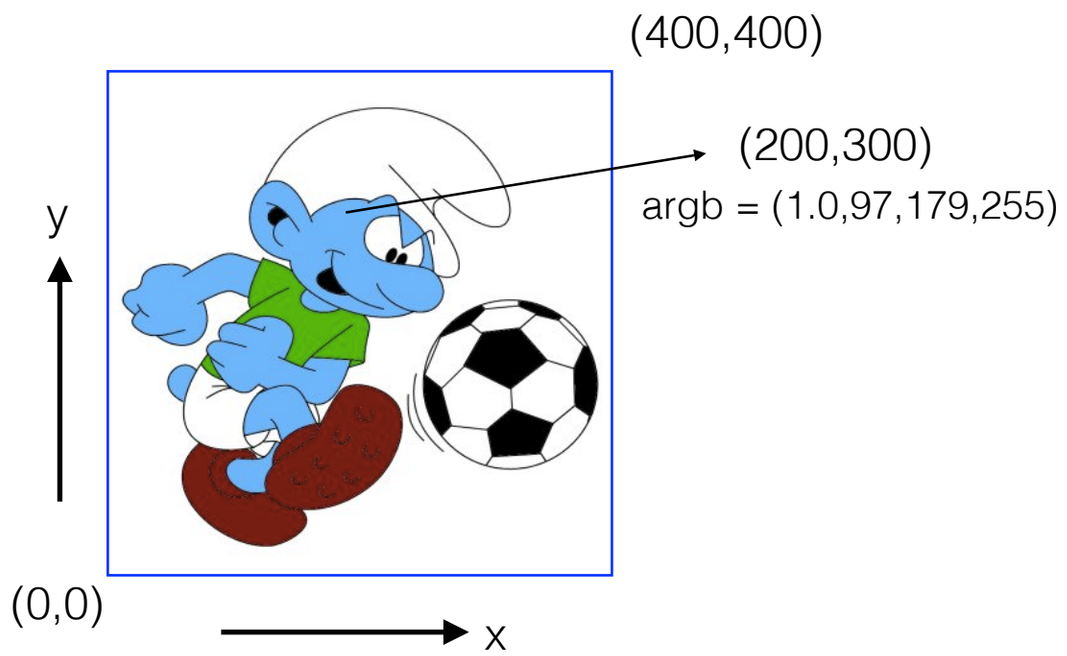
Functions that make functions




```
  x :: X
  y :: Y
  f :: X → Y → Z
f x :: Y → Z
f x y :: Z
```

Application:
Image processing

What is an image?



What is an image?

```
type Image = (Float,Float) → Color

shift :: Image → (Float,Float) → Image

shift img (dx,dy) =
    λ (x,y) →
        img (x - dx, y - dy)
```

What is an image?

```
type Image = (Float,Float) → Color

invert :: Image → Image

invert img = λ(x,y) →
    let (a,r,g,b) = img x y
    in (a,255-r,255-g,255-b)
```

What is an image?

```
type Image = (Float,Float) → Color

compose :: Image → Image → Image
compose im1 im2 = λ(x,y) →
    let (a1,r1,g1,b1) = im1 x y
        (a2,r2,g2,b2) = im2 x y
    in
        (a1 + a2,
         (a1 * r1 + a2 * r2) / (a1 + a2),
         (a1 * g1 + a2 * g2) / (a1 + a2),
         (a1 * b1 + a2 * b2) / (a1 + a2))
```

Closure property

```
type Image = (Float,Float) → Color

shift :: Image → (Float,Float) → Image

compose :: Image → Image → Image

invert :: Image → Image
```

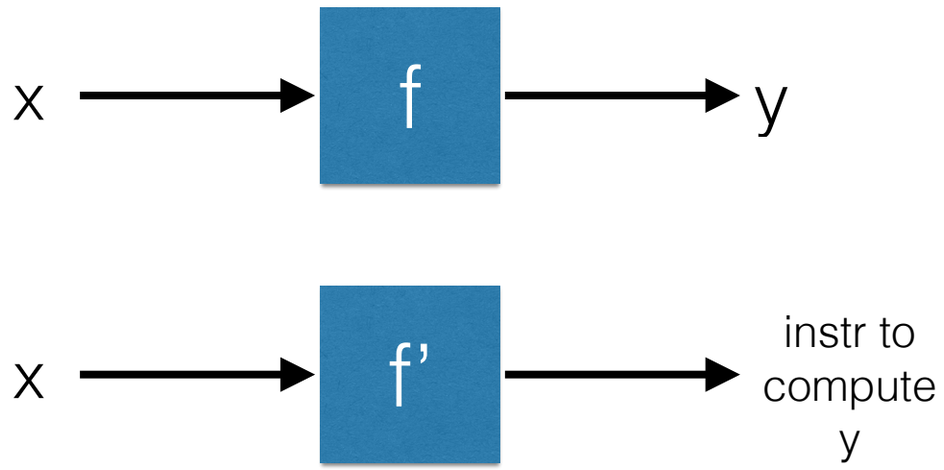
Application:
DB tables

DBTable and join

```
type DBTable key value = key ⟶ value

join :: DBTable k1 v1
        ⟶ DBTable v1 v2
        ⟶ DBTable k1 v2


join tab1 tab2 = λ k ⟶ tab2 (tab1 k)
```

1. Model the domain using the simple function tools to ensure correctness.
2. Then change the implementation to suit considerations of efficiency.
3. In this case, operators like “join” can produce instructions for DB lookup instead of actually performing the lookup

Generating specifications

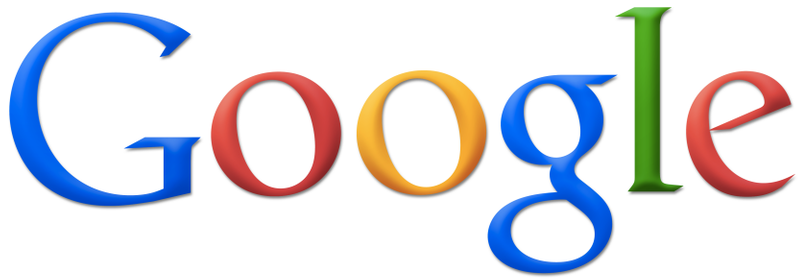


1. We dealt with functions as boxes that do things to produce values.
2. The values produced are up to us. One common pattern that results in good separation of concern and efficiency is to have these functions generate instructions for another subsystem.
3. Because “instructions” compose well too.

Some examples

- Compositing functions generate and compose shaders which are programs that are passed on to GPU (in games)
- DB interfacing functions generating SQL
- JIT compilers generating instructions for CPUs from byte code
- “Display Postscript” in MacOSX - instructions to draw on the screen, handled by PS driver.
- Apache Spark RDDs collect operations, optimize them all and perform them in parallel, and for “resilience”.

Application: MapReduce



MapReduce

```
map :: (x → y) → Coll x → Coll y

reduce :: (x → y → y) → y → Coll x → y

type Coll x = Leaf x | Branch (Coll x) (Coll x)

map f (Leaf x) = Leaf (f x)

map f (Branch l r) =
    Branch (map f l) (map f r)
```

1. To compute total words in a list of strings …
2. map each string to the number of words in it to get a list of numbers
3. sum all the numbers

MapReduce

```
map :: (x → y) → Coll x → Coll y

reduce :: (x → y → y) → y → Coll x → y

type Coll x = Leaf x | Branch (Coll x) (Coll x)

reduce f y0 (Leaf x) = f x y0

reduce f y0 (Branch l r) =
    reduce f (reduce f y0 l) r
```

MapReduce

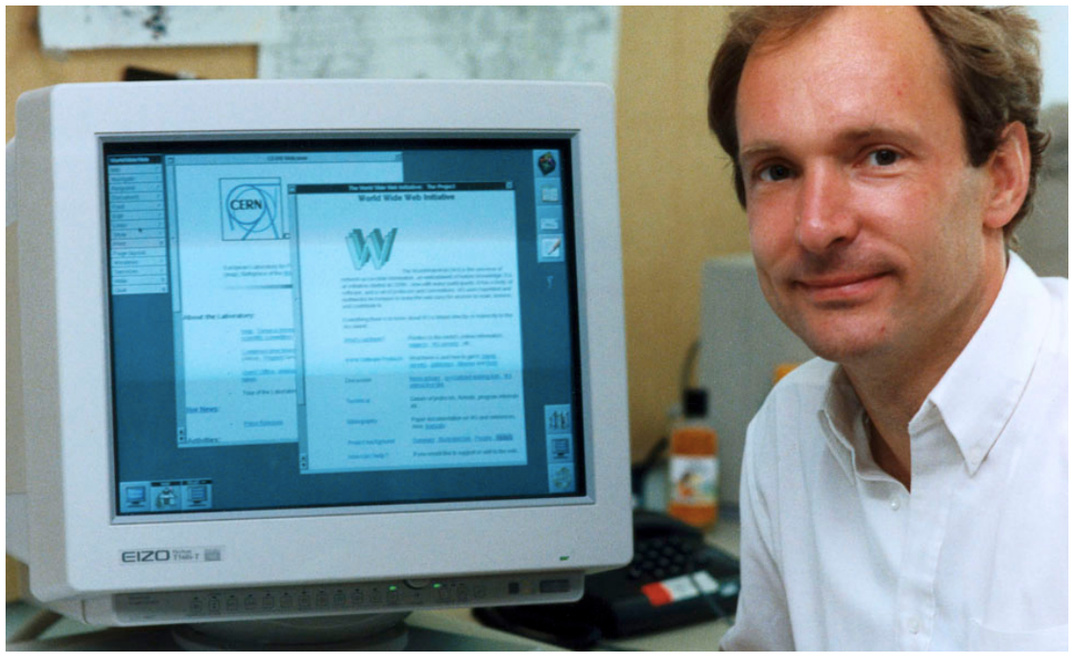
```
map :: (x → y) → Coll x → Coll y

reduce :: (x → y → y) → y → Coll x → y

type Coll x = Leaf x | Branch (Coll x) (Coll x)

reduce f y0 (Leaf x) = f x y0

reduce f y0 (Branch l r) =
    combine (reduce f y0 l) (reduce f y0 r)
```

MapReduce

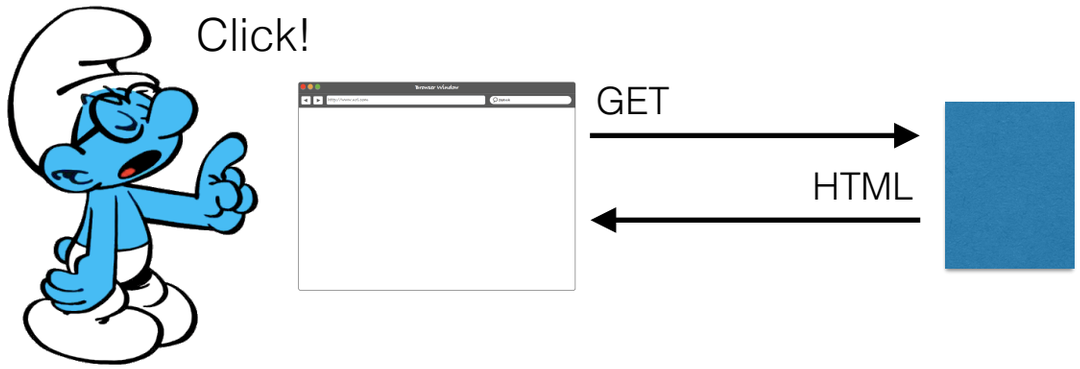
```
mapreduce :: (x ⟶ y)
           ⟶ (y ⟶ z ⟶ z)
           ⟶ z
           ⟶ Coll x
           ⟶ z

mapreduce mf rf z0 (Leaf x) = rf (mf x) z0

mapreduce mf rf z0 (Branch l r) =
    combine (mapreduce mf rf z0 l)
            (mapreduce mf rf z0 r)
```

Application:
Web sites

Tim Berners-Lee edition



Tim Berners-Lee edition

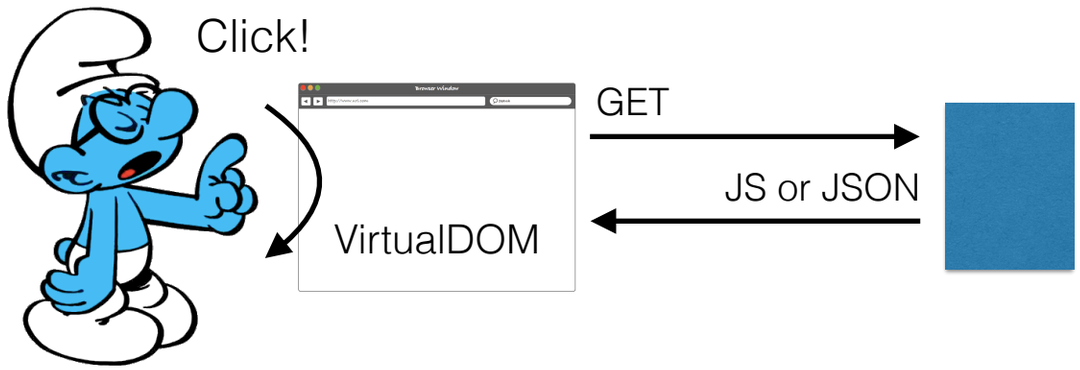



```
app :: Input ⟶ Html
```

AJAX edition

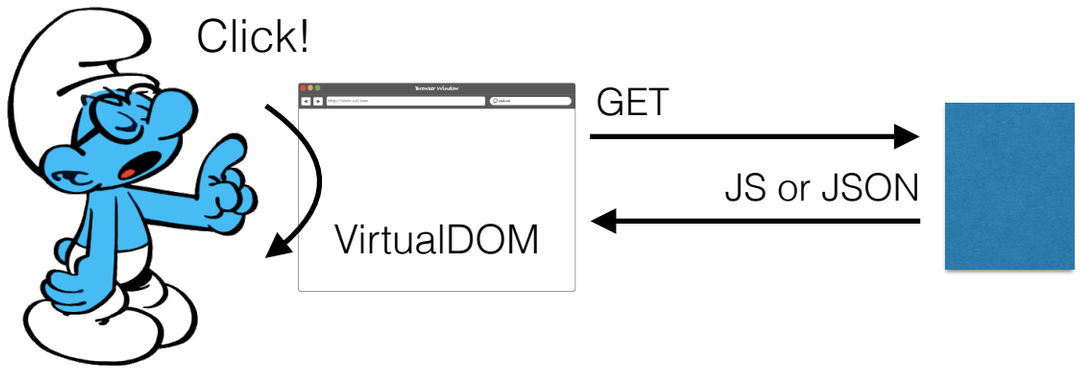

```
app :: Input ⟶ ???
```

ReactJS edition




```
app :: Input ⟶ VirtualDOM
```

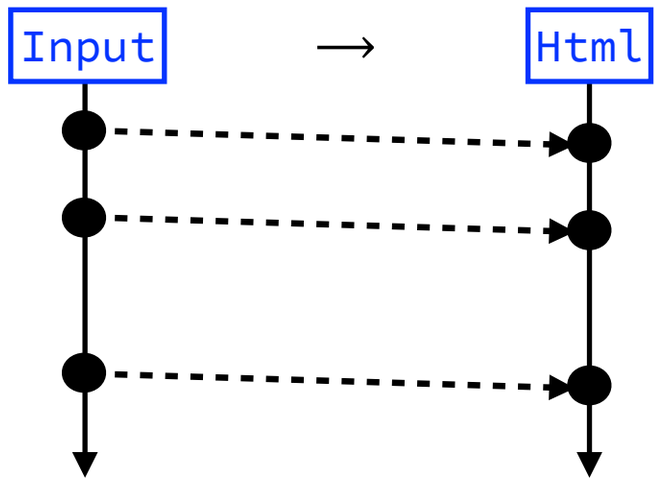
ReactJS edition




```
app :: Input ⟶ Html
```

ReactJS edition

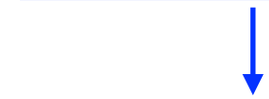
```
type Input = Click (Int,Int) | Key Int
```



ReactJS edition

```
app :: Input → Html
```

```
app :: Signal Input → Signal Html
```



```
app :: SigProc Input Html
```

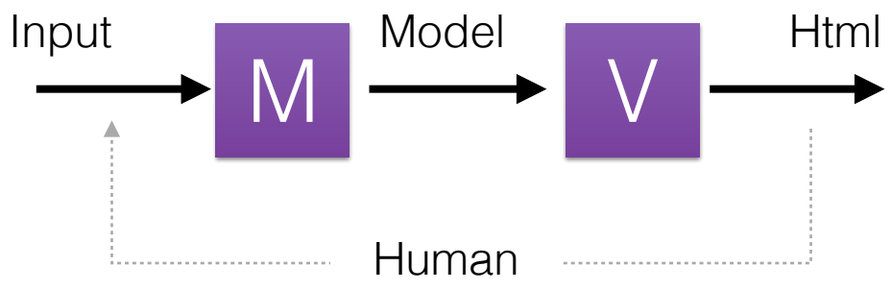
So what?

Reactive edition

```
app :: SigProc Input Html


model :: SigProc Input Model
 view :: SigProc Model Html
    app = model >>> view
```

Reactive edition

```
app :: SigProc Input Html
```



Reactive edition

```
app :: SigProc Input Html


model :: SigProc Input (Model,ViewState)
 view :: SigProc (Model,ViewState) Html
          app = model >>> view
```

Reactive edition

```
app :: SigProc Input Html
```

```
model :: SigProc Input Model
view :: SigProc (Model,ViewState) Html
```

How to produce ViewState?

Reactive edition

```
app :: SigProc Input Html
```

```
model :: SigProc Input Model
controller :: SigProc (Input,Model) ViewState
view :: SigProc (Model,ViewState) Html
```

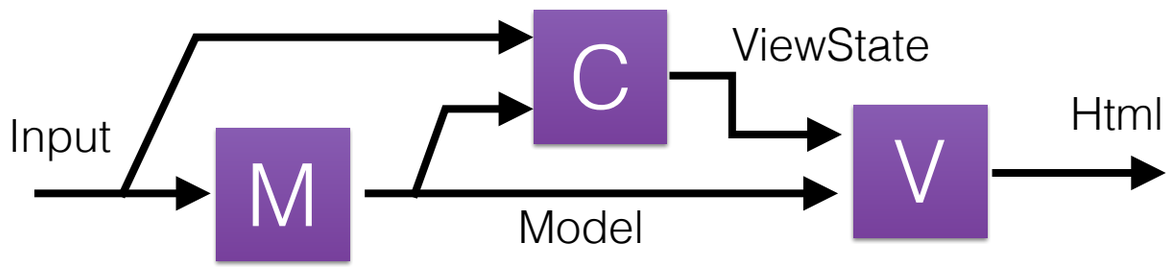
1. Ok. Somebody sat in a toilet, had an epiphany and came up with the model-view-controller way of approaching user interaction systems.
2. How do we sit on the shoulder of this person (while not on the toilet) and see any further?

Reactive edition

```
app :: SigProc Input Html

model :: SigProc Input Model

director :: SigProc (Input,Model) Direction

animator :: SigProc (DT,Direction) ViewState

view :: SigProc (Model,ViewState) Html
```

Reactive edition

```
app :: SigProc Input Html
```



Why does this work?

- All aspects of system are explicit. No unaccounted for “side effects”.
- Functions can compute anything - including functions, processes, or instructions to graphics subsystems.
- Composable system aspects are mapped to composable functions.
- Mathematical guarantee of correctness if you stick to the algebra.

1. No surreptitious AJAX requests when rendering a model. No x = x + 1.
2. If your code type checks in Haskell, it will fairly likely run as you expect it.
3.

Questions?