A Theory of Functional Programming

LambdUp December 6, 2017

Eric Normand

PurelyFunctional.tv

Newton's Laws of Motion

1. Inertia

$$\sum F = 0 \Leftrightarrow \frac{dv}{dt} = 0$$

2. Acceleration

$$\sum F = ma$$

3. Action-reaction

$$F_a = -F_b$$

Force

Mass

Distance

Time

Aristotelian Physics

(excerpt)

Ideal speed

Natural place

Natural motion

Unnatural motion

For the video and transcript of this presentation, click here:

https://lispcast.com/lambdup-2017-theory-functional-programming/

What is Functional Programming?

Why use Functional Programming?

paradigm

a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

Merriam-Webster

philosophical or theoretical framework, world view

theories, laws,
generalizations

basic assumptions, ways of thinking, methodology

What is Functional Programming?

Why use Functional Programming?

Goals of my Theory

- Explain what it is we (functional programmers) actually do
  - in terms we can all understand
- Explain why it has advantages over other paradigms
  - to people who haven’t done FP
- Avoid focusing on features
- Give explanatory and predictive power
- Self-described functional programmers should agree

My Theory of FP

Actions

Data

Calculations

Actions

the process of doing something, typically to achieve an aim

- Typically called *Effects* or *Side-effects*

- Depend on *when you run them* or *how many times you run them*

- Examples

  - Sending a message over the network

  - Writing to file system — other programs can see the change

  - Changing or reading mutable state

Data

factual information used as a basis for reasoning, discussion, or calculation

- Inert
- Serializable
- Requiring interpretation
- Examples
  - Numbers
  - Bytes
  - Strings
  - Collections

Calculations

computation from inputs to outputs

- Mathematical functions
- Eternal — outside of time
- Referentially transparent
- Examples
  - List concatenation
  - Summing numbers

Contrast with OOP

OOP

Objects

References

Messages

Implementation

Haskell

- Data — built-in types and defined types
- Calculations — functions
- Actions — IO type

Implementation

Clojure

- Data — built-in types
- Calculations — pure functions
- Actions — impure functions

Further down the rabbit hole

- Everything “First-class”
  - Data
  - Calculations
  - Actions
- Minimum necessary to program functionally in a language

Domains are separate

Data

Data + Data => Data

Examples

- Addition
- Concatenation

Calculations

Calc + Calc => Calc

Actions

- Contagious!
  - Calculation + Action => Action
  - Data + Action => Action
- Examples
  - Print the square of a number — square => print!
  - Parse the input as a number — read! => parse

Calculations

- Algebraic manipulation
- Turing complete
  - implies the Halting problem
- Opaque
  - What is this code going to do?
  - Only way to know is to run it

Data

- Can represent something else
- Structure
  - Known Big-O complexities

Refactorings

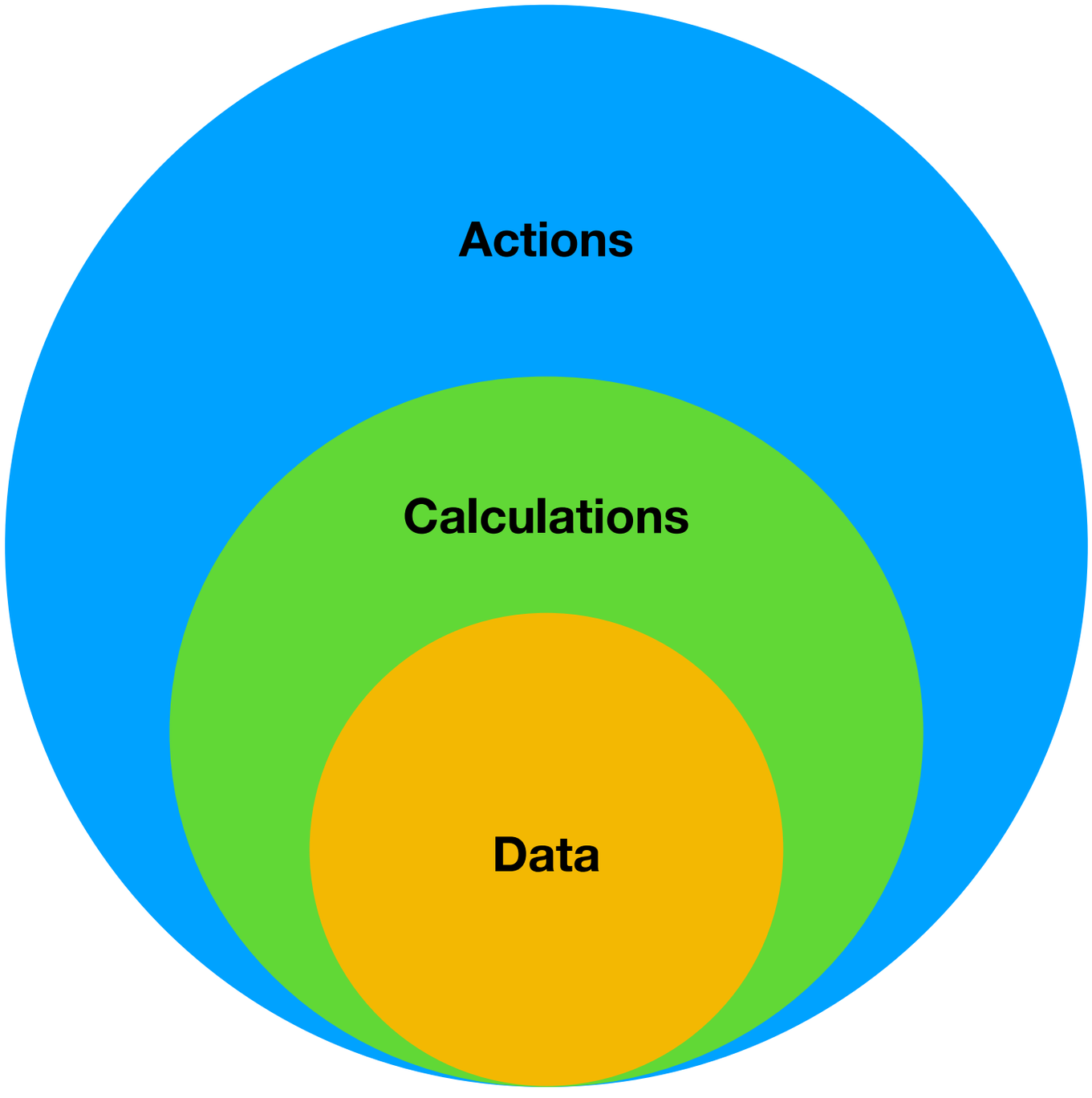
Actions

- Action => Action + Calculation
- Action => Action + Data
- Action => Action + Action

Calculations

- Calculation => Calculation + Data
- Calculation => Calculation + Calculation

Actions are universal



What counts as an Action?

Calculations	Actions
Timeless	**Bound in time**
Pure function	**Read/write to disk**
Pure function takes 24 hours to compute	**Read/write to temp file as buffer**

Actions

how many times they run

always matters - 0≠1≠more

launching a missile

sending an email

idempotent - 0≠1=more

setting public flag to true

free of side-effects - 0=1=more

GET request

reading mutable state

Actions

when they run

transactional read

guaranteed to be consistent

transactional+serialized writes

Order matters, but at least it’s some order

exactly once reads

Communicating Sequential Processes

Eric Normand

LispCast

Follow Eric on:

Eric Normand

@EricNormand

lispcast.com

eric@lispcast.com