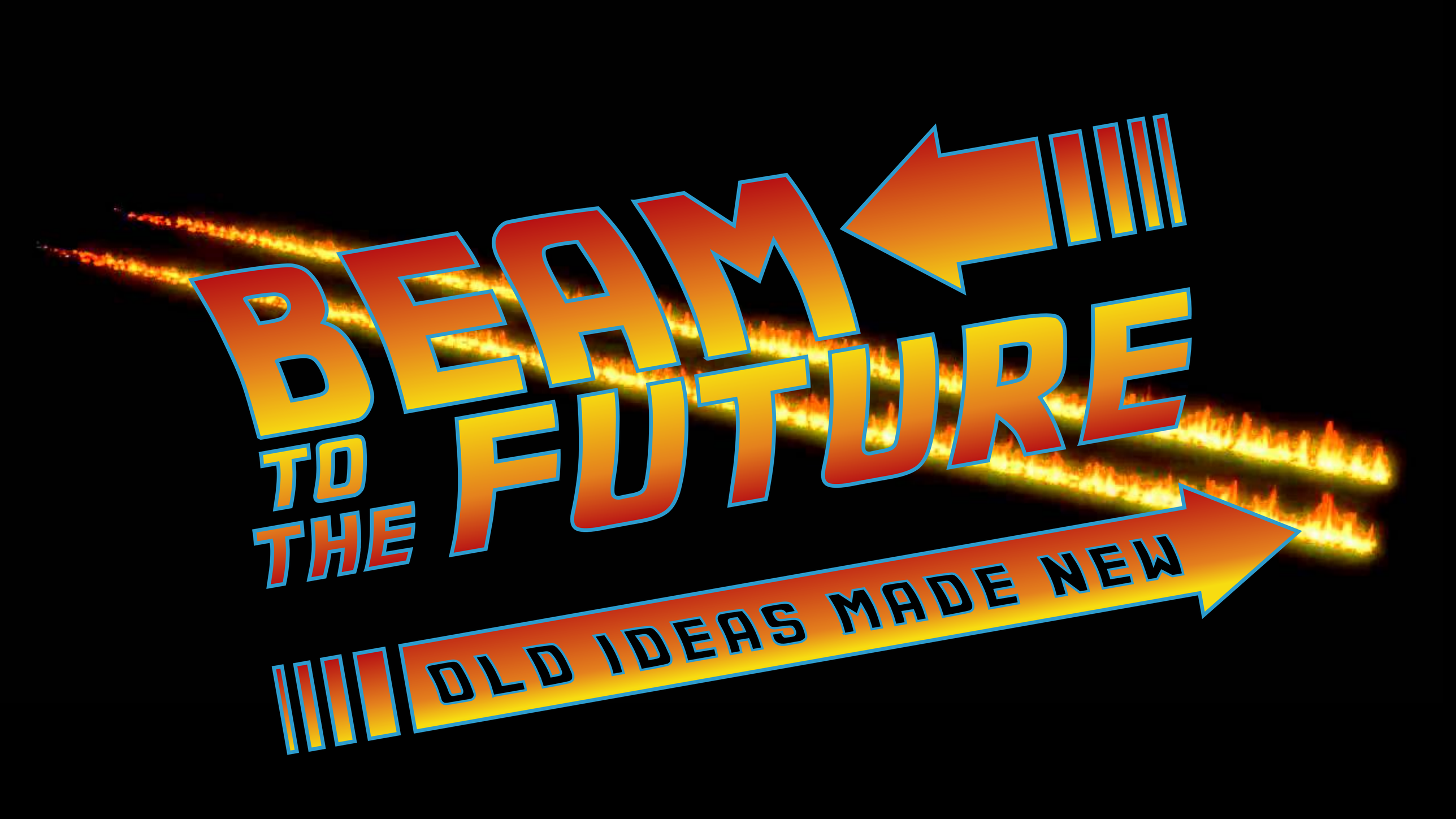

BEAM
TO THE FUTURE
OLD IDEAS MADE NEW

Sometimes in order to keep **moving forward**, not only must you take one step at a time, but you must be willing to **look back** occasionally and evaluate your past, no matter how painful it is. Looking back lets you know whether or not you are headed in the **right direction**.

– G.K. ADAMS

Epigram 53

So many good ideas are never heard from again
once they embark in a voyage on the semantic gulf

– ALAN J. PERLIS, EPIGRAMS ON PROGRAMMING (1982)

META

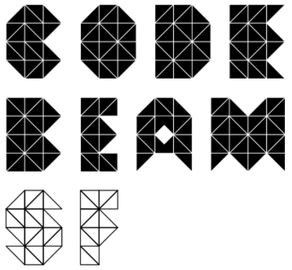
META

BROOKLYN ZELENKA, @expede

- Cofounder/CTO at Fission
  - https://fission.codes
  - Web native apps (“post-serverless")
  - Goal: make back-end & DevOps obsolete 😈
- PLT & VMs
- Prev. Ethereum Core Developer
- Founded VanFP & VanBEAM
- Primary author of Witchcraft Suite, Exceptional, &c

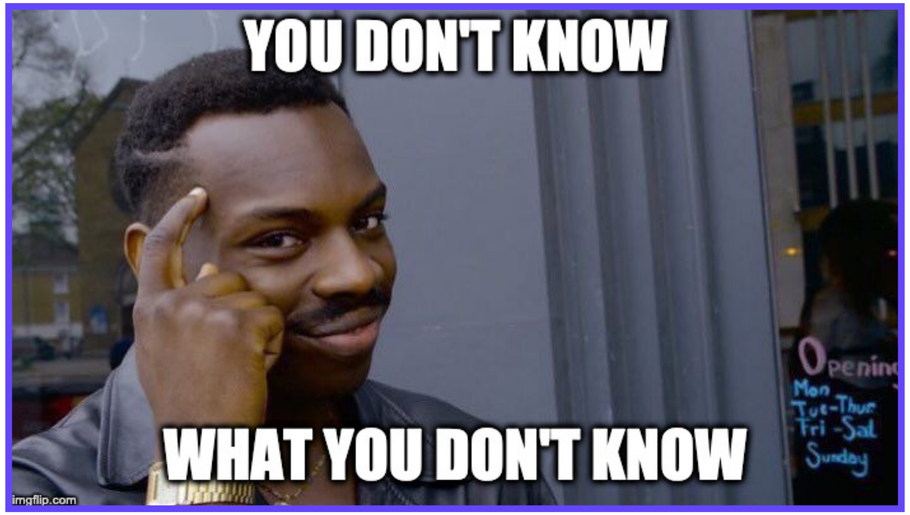
fission

DISCOVER THE **FUTURE** OF ERLANG AND ELIXIR



THE BEAM DOES **SO MUCH** RIGHT 👏

HOW DO WE MOVE FORWARD **AS AN ECOSYSTEM**, **CROSS** LANGUAGE, **CROSS** PARADIGM?



META
LOCAL MAXIMA

1. Breaking out of linear thinking / von Neumann 🌀
2. New types of modularity (for the BEAM) 🔌
3. Composable languages 🗣️

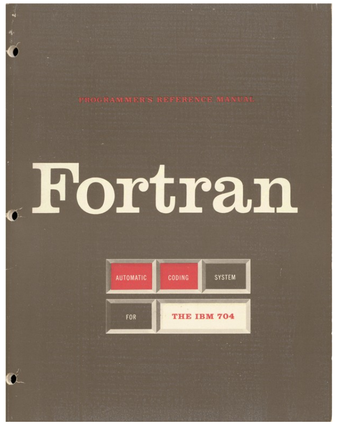
IN THE BEGINNING…

Epigram 28

Around computers it is difficult to find the
correct unit of time to measure progress.
Some cathedrals **took a century to complete**.
Can you imagine the grandeur and scope of a
program that would take as long?

– ALAN J. PERLIS, EPIGRAMS ON PROGRAMMING (1982)

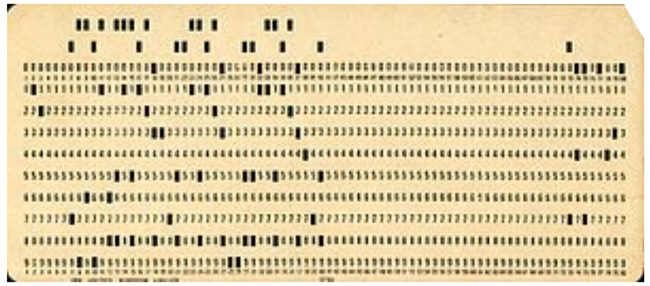
IN THE BEGINNING… 🌴🦕

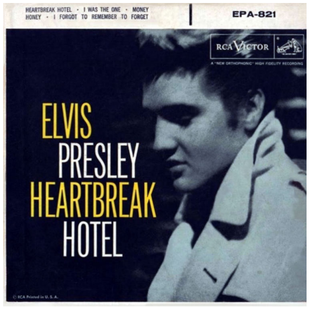


29 years

34 years

CODE BEAM SF





PARADIGM REDSHIFT

It's really difficult to distinguish a **new paradigm** from a **really bad** idea
[…] The **new shiny object** is part of the old paradigm

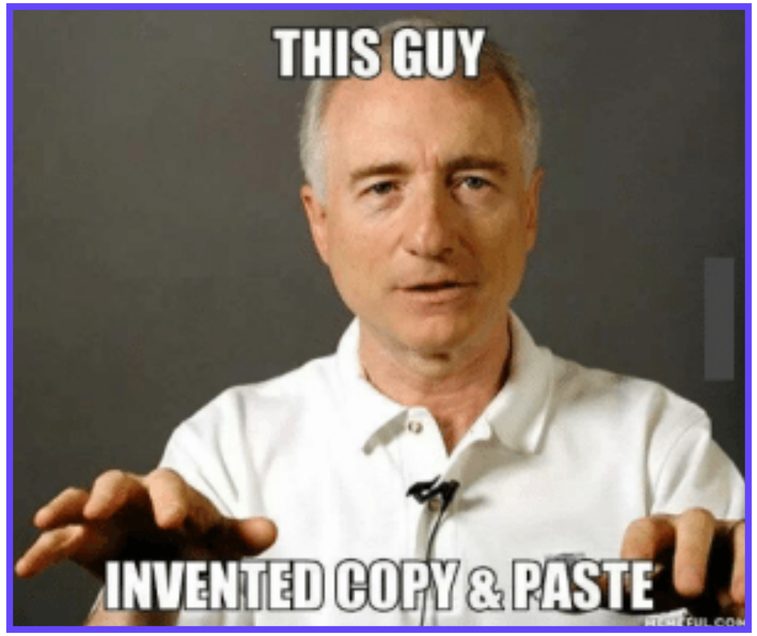
– DOUGLAS CROCKFORD, THE POWER OF THE PARADIGM (2018)


WORKS IN MY CONTAINER
OPS PROBLEM NOW
imgflip.com


IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN

LAW OF CONSERVATION OF COMPLEXITY

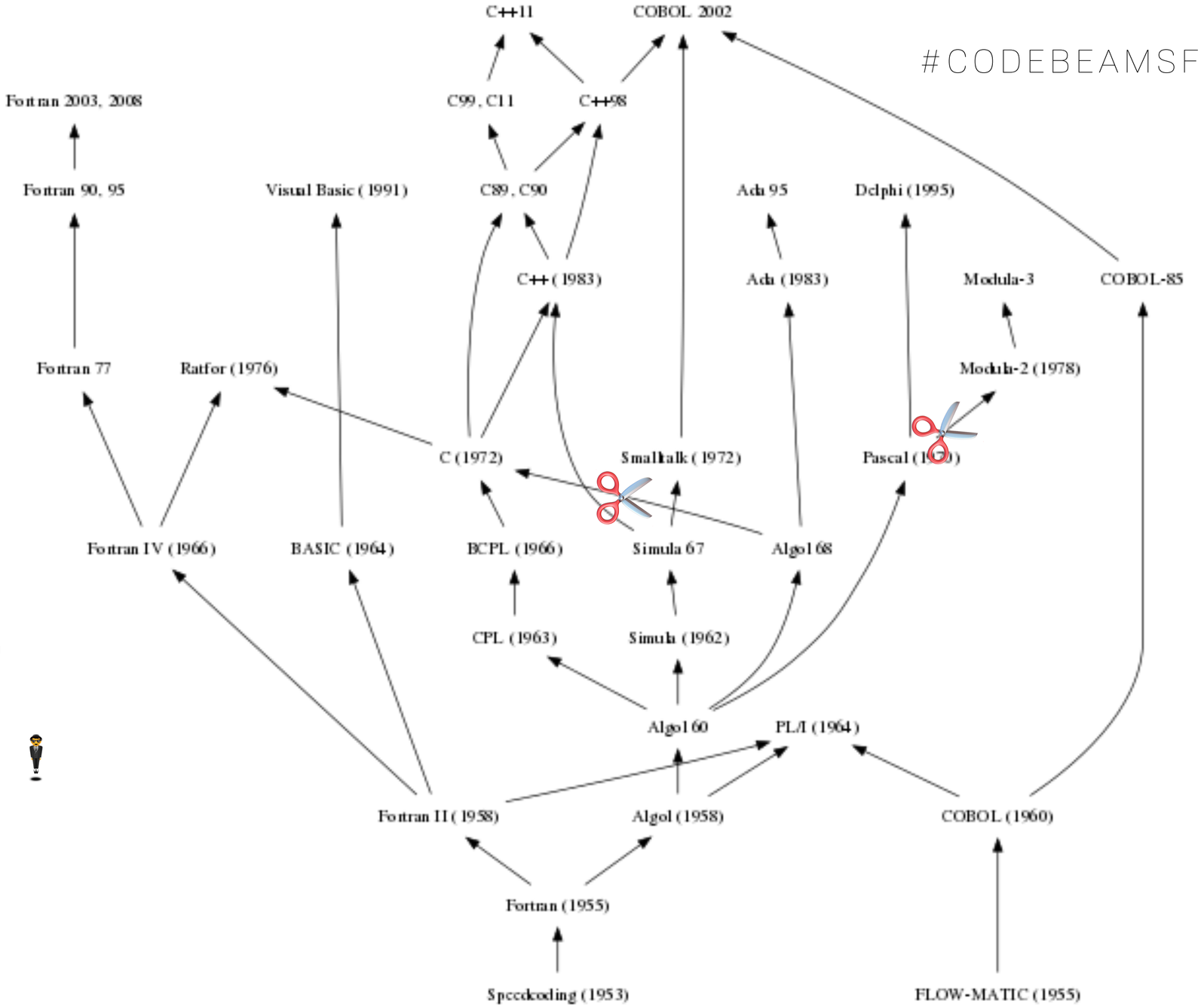
Every application has an inherent amount of complexity
that **cannot be removed** or hidden,
but **only moved** from place to place



– LARRY TESLER

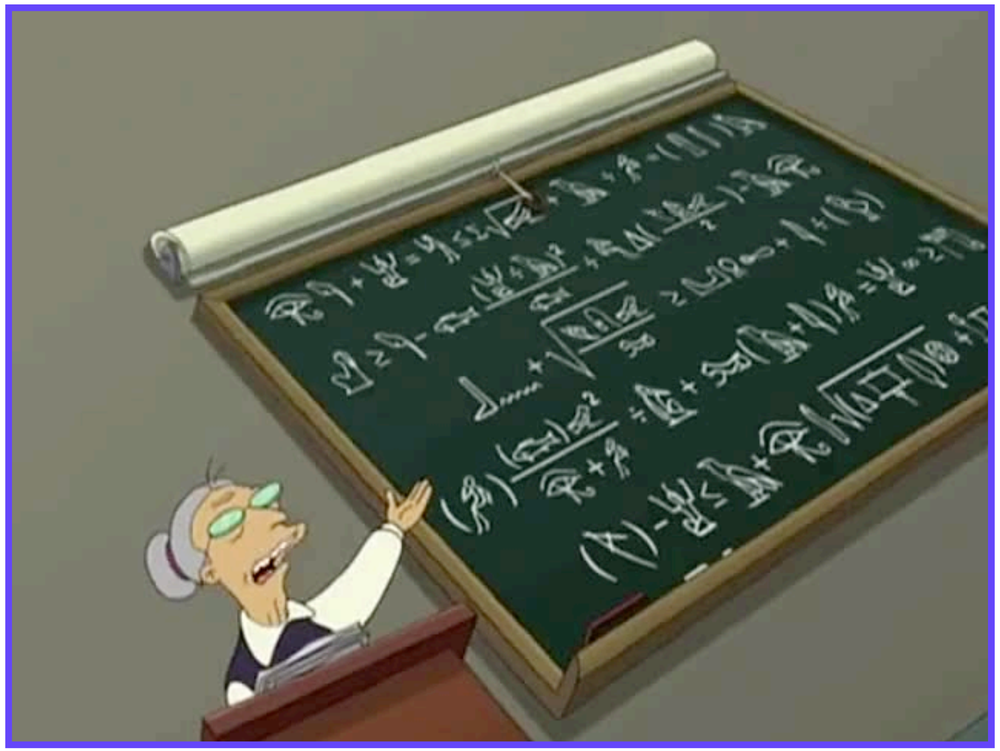
PARADIGM REDSHIFT
THE “WHAT IF” TREE

1. Array-based 📈
2. Applicative Model 🤹
3. SML Module System 🔌
4. Natural & Biz Language 🕴

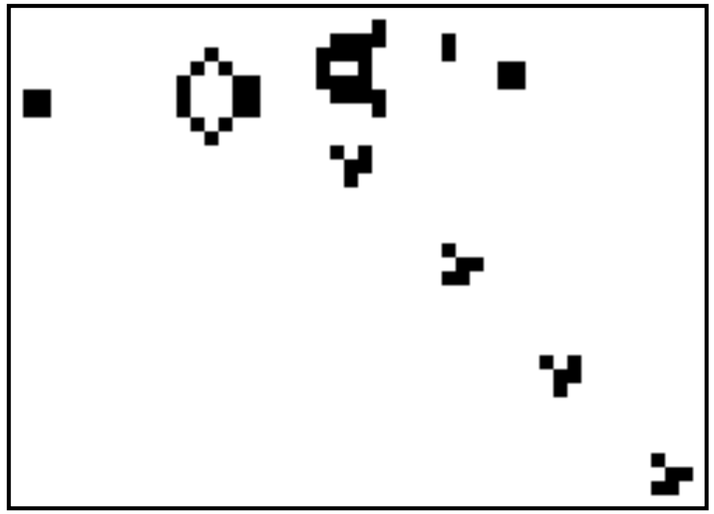


LET'S START WITH SOMETHING ALIEN

🛸👽

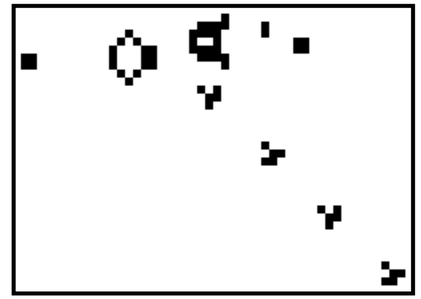


```
{↑1 ⍵ ∨.∧3 4=+/,¯1 0 1∘.⊖¯1 0 1∘.⌽⍵}
```

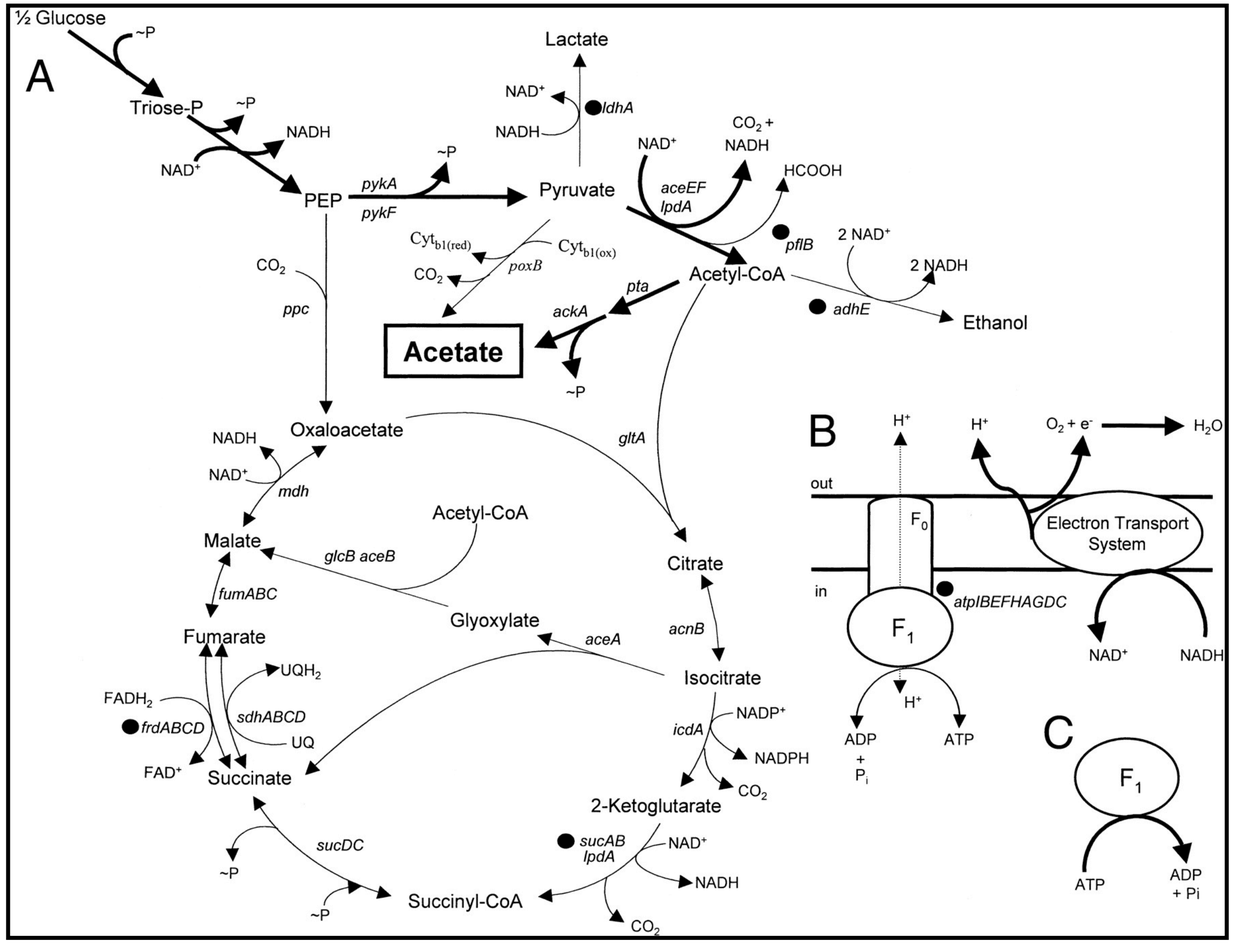


Attribution: wikipedia user LucasVB

WHAT CAN WE LEARN FROM APL?
CELLULAR AUTOMATA & ACTORS AS ORGANISMS 🌸

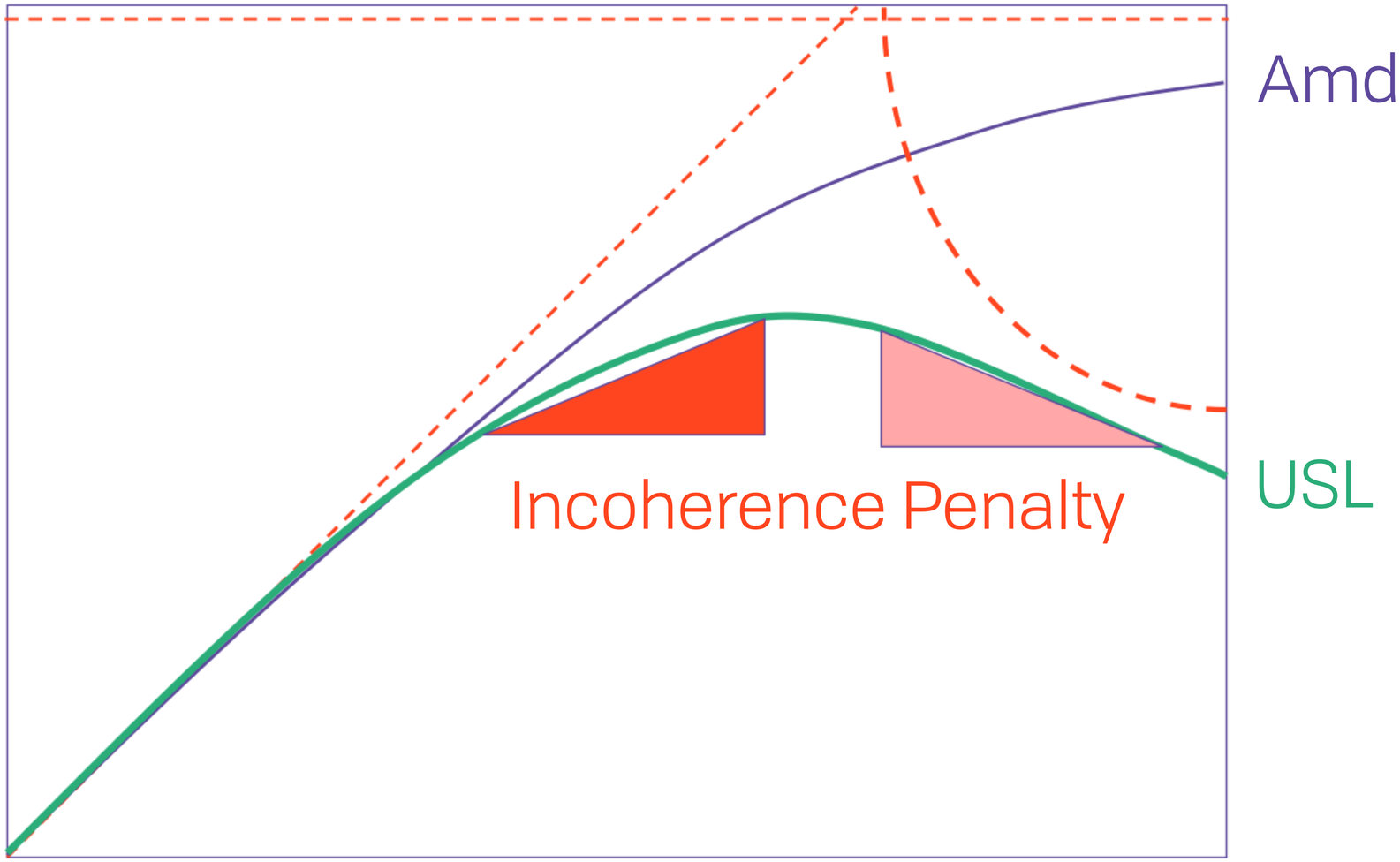


- Each step is very simple
- Reasoning about dynamic organisms is hard
- Emergent behaviour 😱
- VM in your brain to reason at a higher level
- We can abstract some of this away
  - Broadway
  - Arrows



WHAT CAN WE LEARN FROM APL?

UNIVERSAL SCALING LAW



http://www.perfdynamics.com/Manifesto/USLscalability.html

WHAT CAN WE LEARN FROM APL?

DOALL / AUTOMATIC PARALLELISM / CYCLIC MULTITHREADING 🔄

```
import Witchcraft.Functor

0..10_000
|> Enum.to_list()
|> async_map(fn x ->
  Process.sleep(500)
  x * 10
end)
#=> [0, 10, ...] ~1s
```

1. Shared-nothing architecture
2. Good for embarrassingly parallel problems
3. Macro could do a LOT more with this at compile-time
4. Impurity and granular control mean that we don’t get this by default (with good reason)

OTP → TOP

TABLE ORIENTED PROGRAMMING

TABLE-ORIENTED PROGRAMMING
NAIVE TABLES

```
@type t :: User{
  name   :: String.t(),
  handle :: String.t(),
  city   :: String.t()
}
```

Name	Handle	City
Brooklyn	expede	Vancouver
Boris	bmann	Vancouver
Steven	icidasset	Ghent

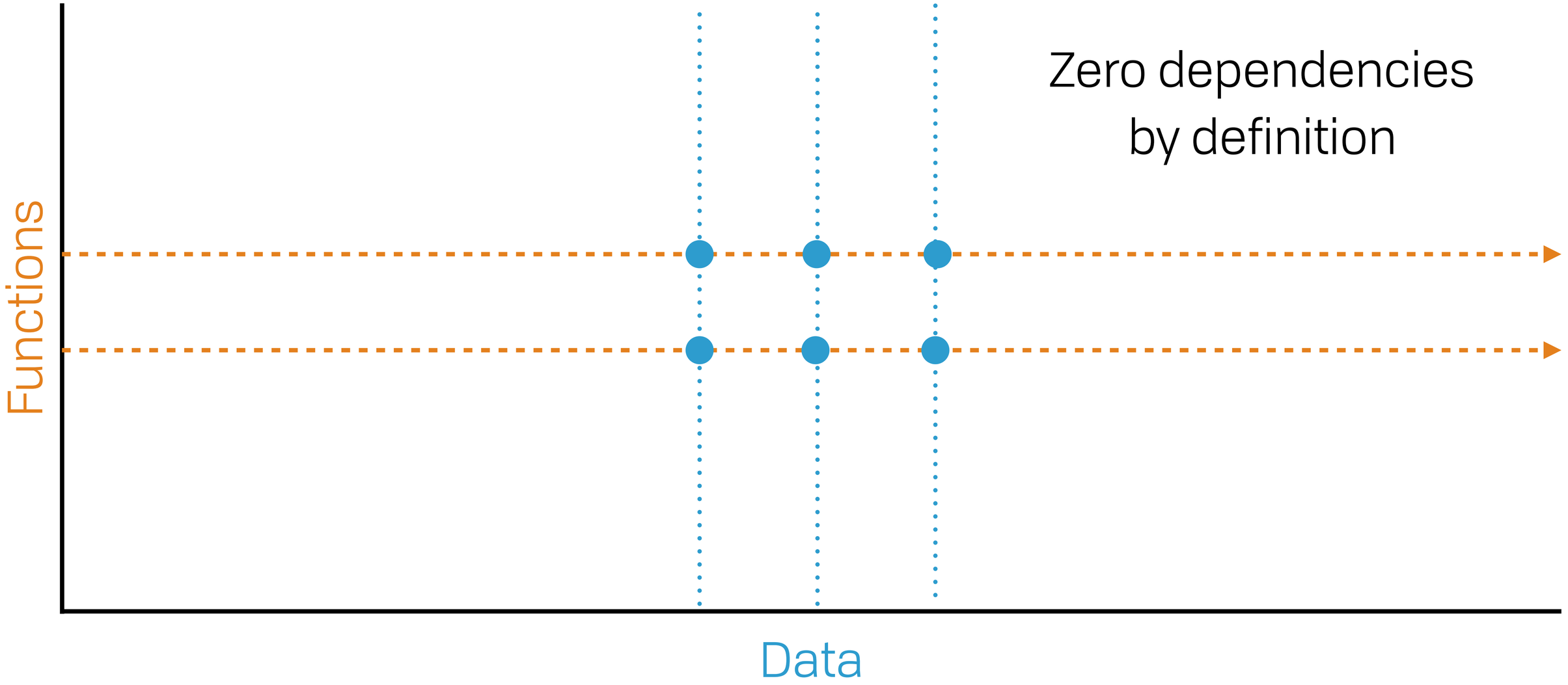
```
[
  %User{name: "Brooklyn", handle: "expede",    city: "Vancouver"},
  %User{name: "Boris",    handle: "bmann",     city: "Vancouver"},
  %User{name: "Steven",   handle: "icidasset", city: "Ghent"}
]
```

PARADIGM REDSHIFT

#CODEBEAMSF

TABLE-ORIENTED PROGRAMMING

PERFECTLY PARALLEL CONTROL TABLES



PARADIGM REDSHIFT

TABLE-ORIENTED PROGRAMMING

THE NTH-DIMENSION 🚀

#CODEBEAMSF

```
[&+/2, &*/2]
|> provide([1, 2, 3])
|> provide([4, 5, 6])
```

```
[
  5, 6, 7,
  6, 7, 8,
  7, 8, 9,
  4, 5, 6,
  8, 10, 12,
  12, 15, 18
]
```

PARADIGM REDSHIFT
TABLE-ORIENTED PROGRAMMING

COMPOSITION & MODULARITY

Epigram 6

Symmetry is a complexity-reducing concept; seek it everywhere.

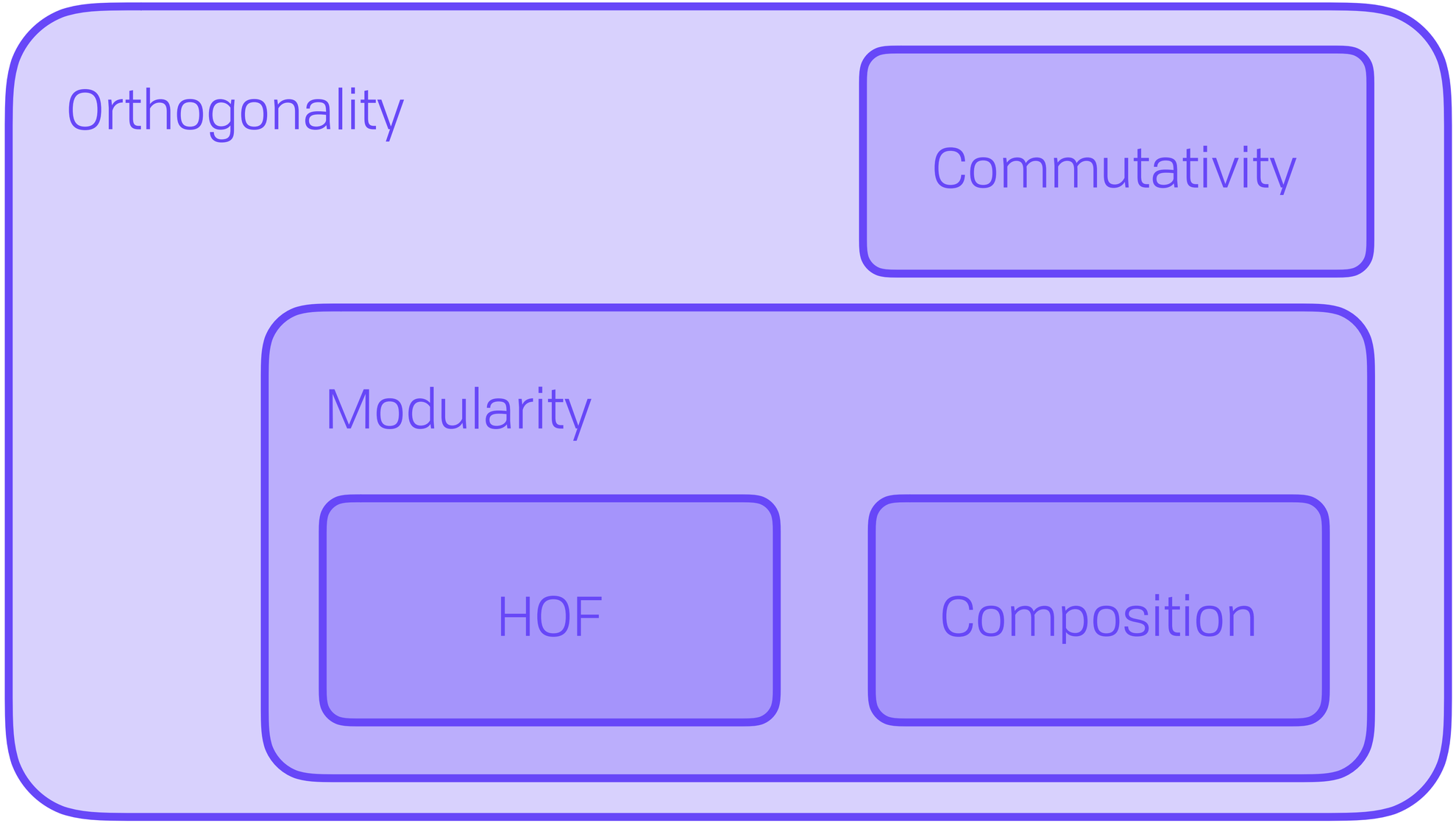
Epigram 105

You can't communicate complexity, only an awareness of it.

– ALAN J. PERLIS, EPIGRAMS ON PROGRAMMING, 1982

WHAT DO YOU MEAN BY “COMPOSABLE”?

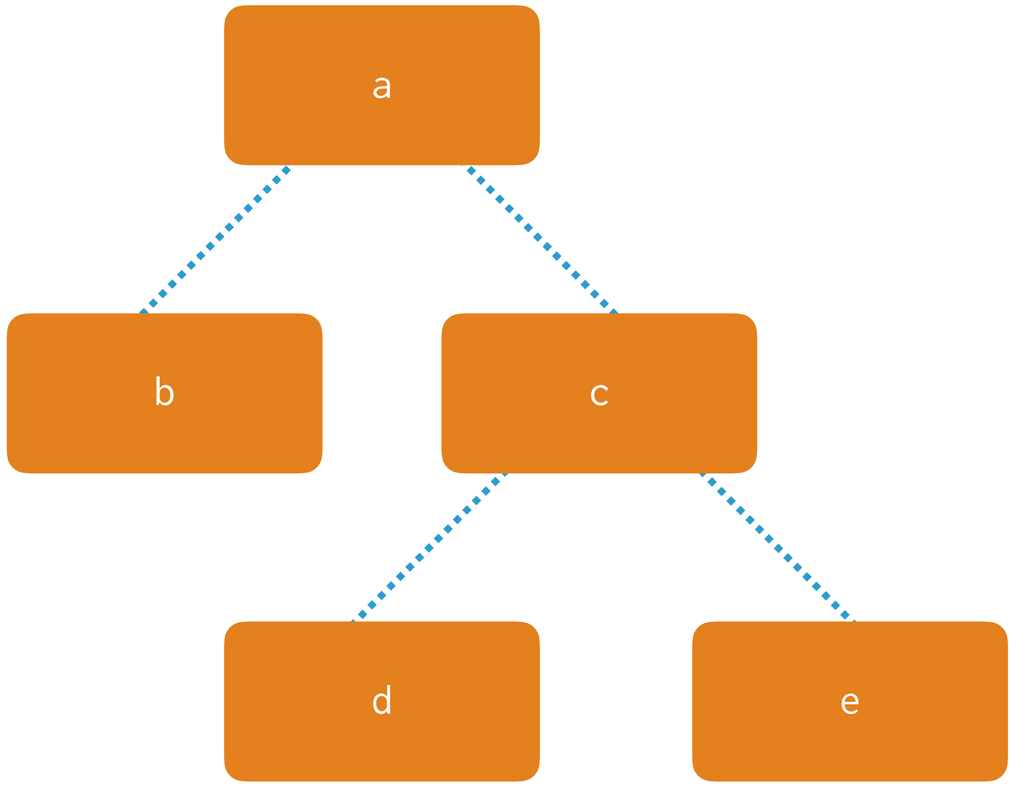
WHAT DO YOU MEAN BY “COMPOSITION”?



WHAT DO YOU MEAN BY “COMPOSITION”?

FOCUS

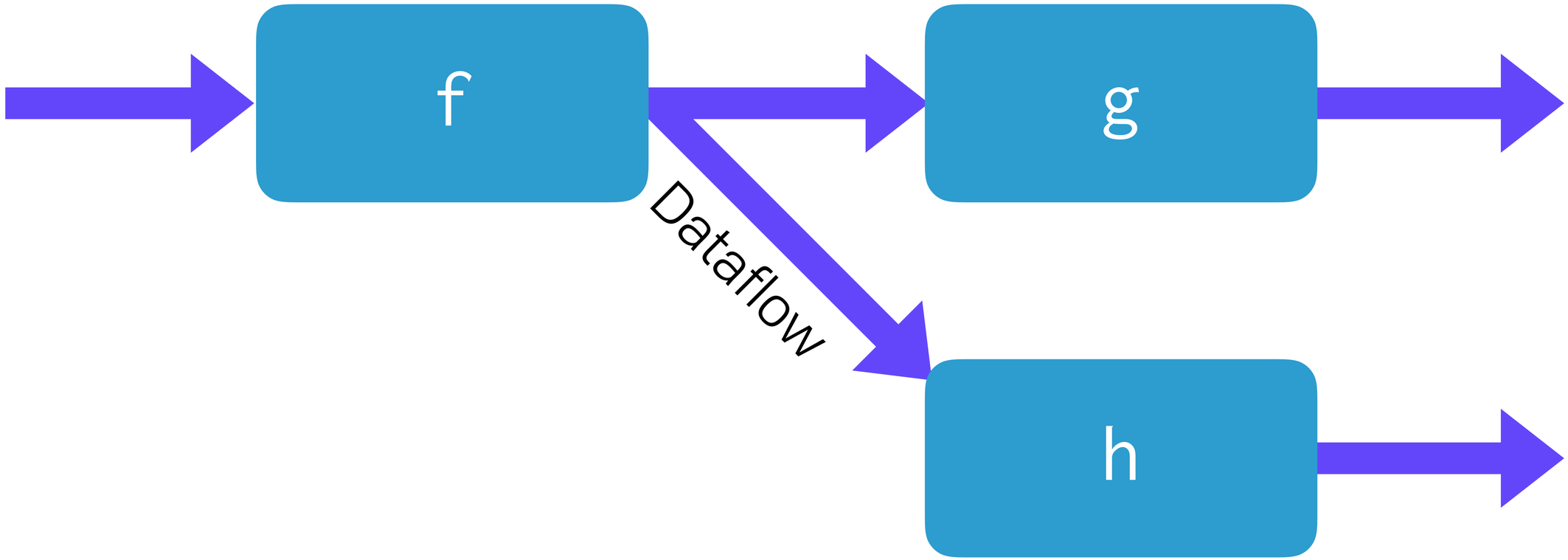
Composition of the **data** dimension



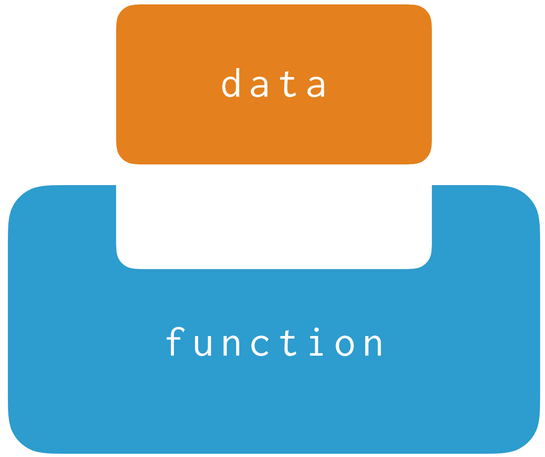
WHAT DO YOU MEAN BY “COMPOSITION”?

FOCUS

Composition of the **function** dimension



Composition of **capabilities**
(protocols and HOFs)



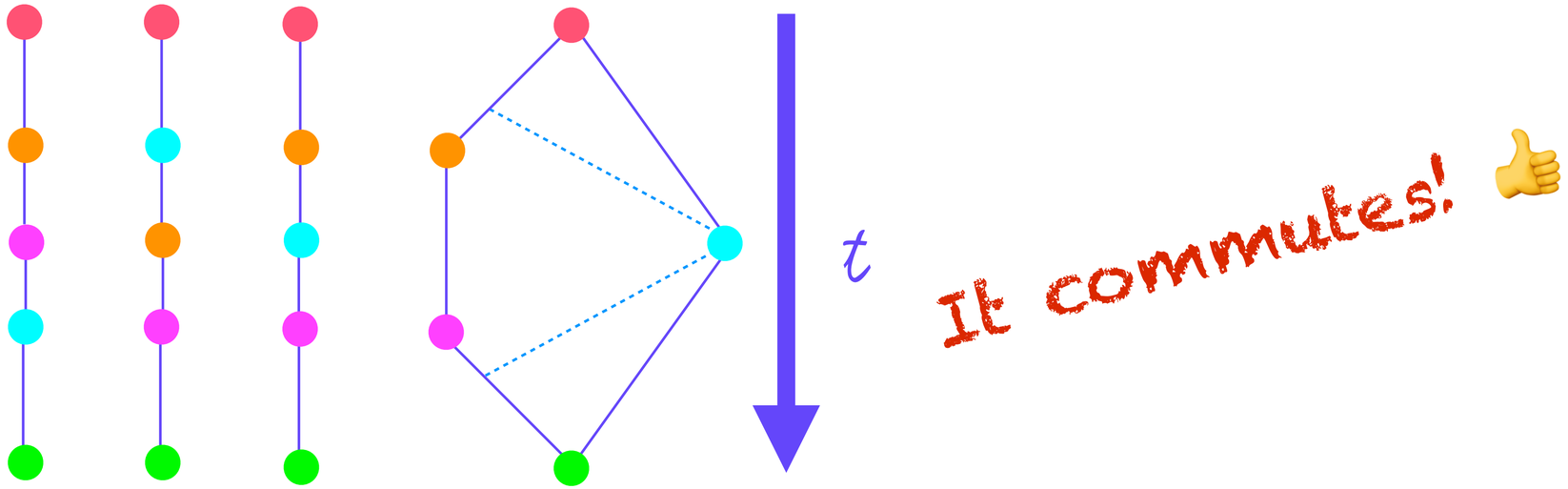
🔁

WHAT DO YOU MEAN BY “COMPOSITION”?

EXECUTION ORTHOGONALITY

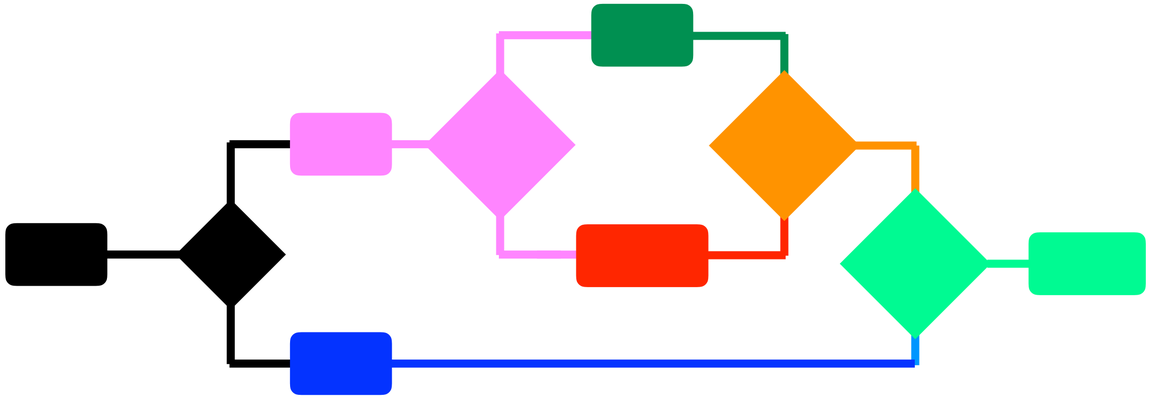
A program can be **developed on a sequential platform,**
even if it is meant to run on a parallel platform, because the behaviour is not affected by whether we execute it using a sequential or parallel dynamics.

– ROBERT HARPER, PRACTICAL FOUNDATIONS FOR PROGRAMMING LANGUAGES, 2012



WHAT DO YOU MEAN BY “COMPOSITION”?

EXPLICIT DATA FLOW



```
arrow_diagram =
  fanout(fn x -> x / 5 end, fn y -> y + 1 end
                          <~> fanout(&inspect/1, fn z -> z end)
                          <~> unsplit(fn(a, b) -> "#{b}#{a}" end))
   | <~> unsplit(&String.at(&2, round(&1)) end)
```

HOW MODULAR ARE MODULES?

(AND LIBRARIES)

```
@modulespec Todo.Verical :: (Query.m(), Schema.m(), JSON.m() -> Vertical.m())
defmodule Todo.Vertical(queryMod \\ Ecto.Query, schemaMod \\ Ecto.Schema, jsonMod \\ Poison.Encoder) do
  use Phoenix.Vertical

  use queryMod
  use schemaMod

  schema "users" do
    field :name,     :string
    field :complete, :bool, default: false
  end

  def view_one(todo), do: json.encode(todo)
end
```

“hot-swappable dependencies”

HOW MODULAR ARE MODULES?

~~HACKING~~ EXTENDING THE MODULE SYSTEM 😉

```elixir
defmodule TypeClass.Dependency do
  defmacro __using__(_) do
    quote do
      import unquote(__MODULE__)
      unquote(__MODULE__).set_up()
    end
  end

  defmacro set_up do
    quote do
      import TypeClass.Utility.Attribute
      register(:extend, accumulate: true)
    end
  end

  defmacro extend(parent_class) do
    quote do
      require unquote(parent_class)
      @extend unquote(parent_class)
    end
  end
```

```elixir
  defmacro extend(parent_class, alias: true) do
    quote do
      extend unquote(parent_class)
      alias unquote(parent_class)
    end
  end

  defmacro extend(parent_class, alias: alias_as) do
    quote do
      extend unquote(parent_class)
      alias unquote(parent_class), as: unquote(alias_as)
    end
  end

  defmacro run do
    quote do
      def __dependencies__, do: @extend
    end
  end
end
```

HIGHER ORDER MODULES

💖

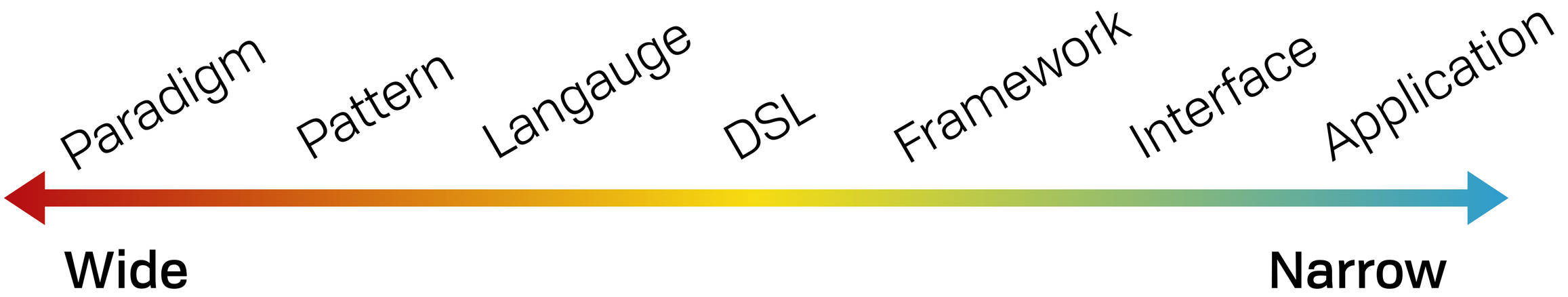
BEHAVIOURS

DECLARATIVE EMBEDDED DSLS

I have regarded it as the highest goal of programming language design to enable **good ideas** to be **elegantly expressed**

– TONY HOARE, TURING AWARD LECTURE, 1980

DECLARATIVE EMBEDDED DSLS
ISN'T EVERYTHING A DSL?



ISN'T EVERYTHING A DSL?

COUNTEREXAMPLE

```
[1,2,3]
|> hypothetical_returns_exception()
~> Enum.map(fn x -> x + 1 end)
~> Enum.sum()
>>> fn x -> x + 1 end.()
```

THIS IS JUST ELIXIR!

DECLARATIVE EMBEDDED DSLS
ALGEBRAIC DATA TYPE EDSL

```
defmodule Maybe do
  defsum do
    defdata Nothing :: none()
    defdata Just    :: any()
  end
end

Maybe.new()
#=> %Maybe.Nothing{}
```

```
defmodule Light do
  defsum do
    defdata Red    :: none()
    defdata Yellow :: none()
    defdata Green  :: none()
  end

  def from_number(1), do: %Light.Red{}
  def from_number(2), do: %Light.Yellow{}
  def from_number(3), do: %Light.Green{}
end
```

```
defmodule Player do
  defdata do
    name       :: String.t()
    hit_points :: non_neg_integer()
    experience :: non_neg_integer()
  end

  @spec attack(t(), t()) :: {t(), t()}
  def attack(%{experience: xp} = player, %{hit_points: hp} = target) do
    {
      %{player | experience: xp + 50},
      %{target | hit_points: hp - 10}
    }
  end
end
```

BUSINESS LANGUAGE FOR BUSINESS TIME

WHAT CAN WE LEARN FROM COBOL?

(YES. REALLY. *COBOL*.)

– Toggl, How to Kill at the Dragon in 9 Programming Languages

WHAT CAN WE LEARN FROM COBOL?

```
PROCEDURE DIVISION.
*Run the code as performed paragraphs
      PERFORM GET-DATA
      PERFORM CALC-DATA
      PERFORM SHOW-DATA
      PERFORM FINISH-UP
      GOBACK.

*A performed paragraph to get user input
GET-DATA.
      MOVE SPACE TO WS-USER WS-FULL-NAME
      DISPLAY "What is your first name?"
      ACCEPT WS-FIRST-NAME OF WS-USER
      DISPLAY "What is your last name?"
      ACCEPT WS-LAST-NAME OF WS-USER
      DISPLAY "What is your age?"
      ACCEPT WS-AGE OF WS-USER
      STRING WS-FIRST-NAME OF WS-USER DELIMITED BY SPACE
      SPACE DELIMITED BY SIZE
            WS-LAST-NAME OF WS-USER DELIMITED BY SPACE
            SPACE DELIMITED BY SIZE
            INTO WS-FULL-NAME
            ON OVERFLOW
            DISPLAY "SORRY, YOUR DATA WAS TRUNCATED"
     END-STRING.

*A performed paragraph for doing calculation
CALC-DATA.
* Sample addition statement
       ADD  WS-AGE-DELTA WS-AGE OF WS-USER TO WS-NEW-AGE.

*A performed paragraph to display output
SHOW-DATA.
       DISPLAY "Welcome " WS-FULL-NAME " In ten years you will be: "
        WS-NEW-AGE.

*A performed paragraph to end the program
FINISH-UP.
      DISPLAY "Strike any key to continue".
      ACCEPT WS-CLOSE
      DISPLAY "Good bye".
END PROGRAM RESEL-WORLD.
```

DECLARATIVE EMBEDDED DSLS
WHAT CAN WE LEARN FROM COBOL?
WHEN NO ONE WANTS TO GO TO JAIL 👩‍⚖️

- Needs to be readable by lawyers
  - i.e. who can't read code
- Formal methods / static analysis
  - Including the compiler, of course
- An unholy union of COBOL and Prolog

```
TOKEN cross_border
  IS ERC777
  IS ERC902

  CONFORM
    bc_securities_commission
    AND usa_sec
    AND korea_exchange

  ALLOW issue
    AFTER  BLOCK 9000000
    BEFORE BLOCK 10000000

  RESTRICT holder
    CHECK allowed IN ERC902 AT 0xaa9cf224d798123fde793fc41c72d3bb8c1c9a09

  ISSUANCE
    START 0
    MAX   10_000
```

UPSIDE

1. *Great* for communicating with domain experts
2. We know how these DSLs work (e.g. they form an algebra)
3. They can be correct-by-construction
4. Check for various properties
    1. Compile- or run-time
5. A language that *exactly* fits your needs (DDD)

BUT WE HAVE A PROBLEM 😱

HINT: IT'S INFLEXIBILITY

SHALLOW EMBEDDING

THE QUICK AND DIRTY WAY

SHALLOW EMBEDDING

```elixir
defmodule Algae.Tree.BinarySearch do
  alias __MODULE__, as: BST

  defsum do
    defdata Empty :: none()

    defdata Node do
      node  :: any()
      left  :: BST.t() \\ Empty.new()
      right :: BST.t() \\ Empty.new()
    end
  end
end
```

```elixir
BST.Node.new(
  42,
  BST.Node.new(77),
  BST.Node.new(
    1234,
    BST.Node.new(98),
    BST.Node.new(32)
  )
)
```

```elixir
%Algae.Tree.BinarySearch.Node{
  node: 42,
  left: %Algae.Tree.BinarySearch.Node{
    node:  77,
    left:  %Algae.Tree.BinarySearch.Empty{},
    right: %Algae.Tree.BinarySearch.Empty{}
  },
  right: %Algae.Tree.BinarySearch.Node{
    node:  1234,
    left:  %Algae.Tree.BinarySearch.Node{
      node:  98,
      left:  %Algae.Tree.BinarySearch.Empty{},
      right: %Algae.Tree.BinarySearch.Empty{}
    },
    right: %Algae.Tree.BinarySearch.Node{
      node:  32,
      left:  %Algae.Tree.BinarySearch.Empty{},
      right: %Algae.Tree.BinarySearch.Empty{}
    }
  }
}
```

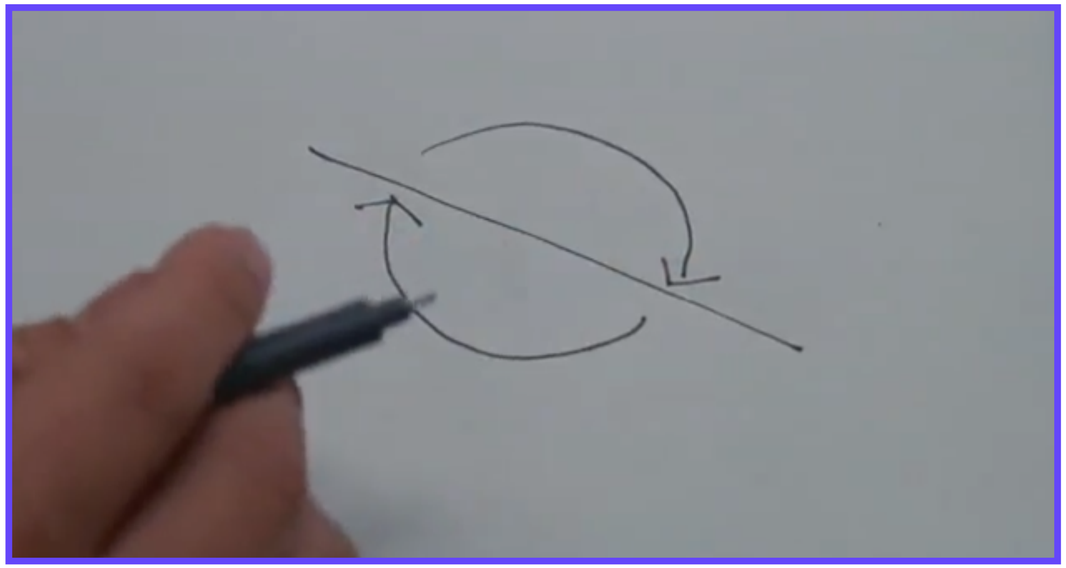
- Just use the built-in AST
- What it can represent is limited
- e.g. Ecto, Algae

DEEP EMBEDDING

"BETTER AST"

DEEP EMBEDDING
THREE STEPS

1. Build a game plan
2. Transform (optional)
3. Tear down



DEEP EMBEDDING

“BUILD A PLAN” EXAMPLE

```
%Unsplit{
  split: %Split{
```

```
|> split do
  fn x -> x / 5 end

  fn y -> y + 1 end |> split(do
    &inspect/1
    fn z -> z end
  end)
  |> unsplit(fn (a, b) -> "#{b}#{a}" end)
end
|> unsplit(&String.at(&2, round(&1)))
```

```
> "#{right}#{left}" end
)
}
|> Witchcraft.Functor.map(&IO.inspect/1)
```

DEEP EMBEDDING
TRADEOFFS

- More work to write (write your own AST)
- Way more powerful (full control)
- Precisely the vocabulary that you need — exact surface area
- Can check more things about the meaning of your code
- Logic-as-data is MUCH simpler to debug than running functions
- Time travelling debugging!
- Unlike protocols, you're not locked into one canonical implementation

DEEP EMBEDDING

COMBINE POWERFUL, MODULAR, REUSABLE DSLS!

```
with_npc :dog do
  go_north
  wait_seconds 2

  set_caps true
  print "woof"
  set_caps false

  go_west
  wait_seconds 1
end
```

DEEP EMBEDDING

DESUGAR

```
%AST{
  cmd: %GoNoth{},
  next: %AST{
    cmd: %Wait{seconds: 2}
    next: # ...
  }
}
```

```
[
  %GoNoth{},
  %Wait{seconds: 2},
  %SetCaplock{is_cap: true},
  %Print{text: "woof"},
  %SetCaplock{is_cap: true},
  %GoWest{},
  %Wait{seconds: 3}
]
```

DEEP EMBEDDING

HUH, KINDA THIS FEELS LIKE GENSERVER

```
def handle_text(%Print{msg: msg}, _), do: IO.puts(msg)

def handle_text(%SetBlink{should_blink: should_blink}, _) do
  if should_blink, do:
    IO.ANSI.blink_rapid()
  else
    IO.ANSI.blink_off()
  end
end

def handle_text(%SetColour{red: r, green: g, blue: b}, _), do: IO.ANSI.color(r, g, b)

def handle_text(%CAPSLOCK{set_caps: is_caps}, state_agent), do
  Agent.update(state_agent, fn state ->
  IO.ANSI.cursor(@agent.current_line -
end

def handle_text(not_text_cmd, _), do: n
```

```
def handle_gameplay(%BuyItem{item: item}, _), do: ...
def handle_gameplay(%ApplyPowerup{item: item}, _), do: ...
def handle_gameplay(not_gameplay, _), do: not_gameplay
```

```
def handle_movement(%GoNorth{}, _), do: ...
def handle_movement(%GoSouth{}, _), do: ...
def handle_movement(%GoEast{}, _), do: ...
def handle_movement(%GoWest{}, _), do: ...
def handle_movement(not_movement, _), do: not_movement
```

DEEP EMBEDDING

WITH THEIR POWERS COMBINED!

```
def interpreter(cmd, agent) do
  cmd
  |> handle_gameplay(agent)
  |> handle_movement(agent)
  |> handle_text(agent)
end
```

DEEP EMBEDDING
ONE LAST LINE

```
with_npc :dog do
  go_north
  wait_seconds 2

  set_caps true
  print "woof"
  set_caps false

  go_west
  wait_seconds 1
end
|> Enum.map(fn cmd -> interpreter(cmd, agent) end)
```

DEEP EMBEDDING
MORE FLEXIBLE THAN PROTOCOLS

1. Protocols require canonicity
2. Libraries of well-defined mini-languages, even without interpreter
3. Different in tests and prod (trivial to mock)

LET'S MAKE NEW MISTAKES! 💥✨

5 PROBLEMS FOR THE NEXT 30 YEARS OF BEAM

5 PROBLEMS FOR THE NEXT 30 YEARS OF BEAM

1. Wasm — client & edge BEAM
2. Lower the barrier to entry (low code)
3. Correctness tools (i.e. better static & dynamic analysis, formal methods)
4. Automatic dynamic parallel evaluation
5. Mobile agents (incl. dynamic FaaS)

PARTING THOUGHT 

Epigram 101

Dealing with **failure is easy**:
work hard to improve.

Success is also easy to handle:
you've solved the wrong problem.

Work hard to improve.

– ALAN J. PERLIS, EPIGRAMS ON PROGRAMMING (1982)

🇺🇸 THANK YOU, SAN FRANCISCO 🎉

brooklyn@fission.codes
github.com/expede
@expede