A tour of String Diagrams and Monoidal Categories

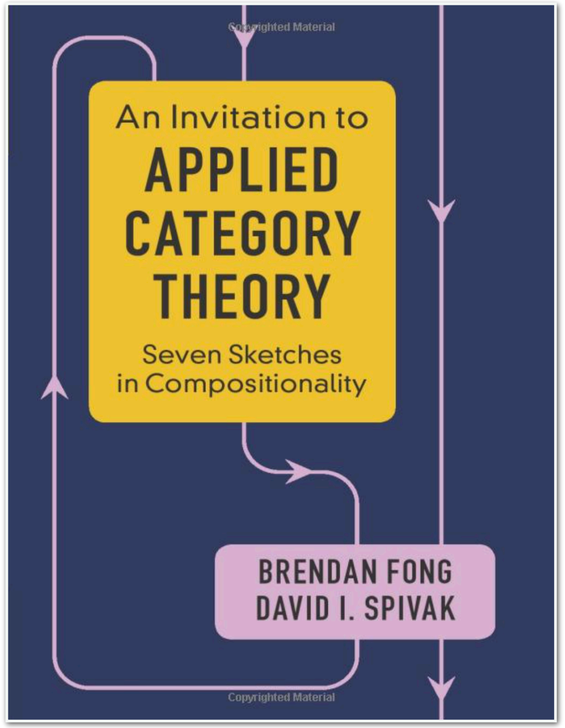
Juan Pablo Romero Méndez
@1jpablo1

Scale by the Bay
November 2020

Goals

- Show why Monoidal Categories are a good modeling tool for certain class of domains
- Show how Monoidal Categories can be represented graphically via String Diagrams
- Given the time constraint I'll go rather quickly but don't hesitate to ask questions after the presentation!

A few references



Physics, Topology, Logic and Computation:
A Rosetta Stone

John C. Baez
Department of Mathematics, University of California
Riverside, California 92521, USA

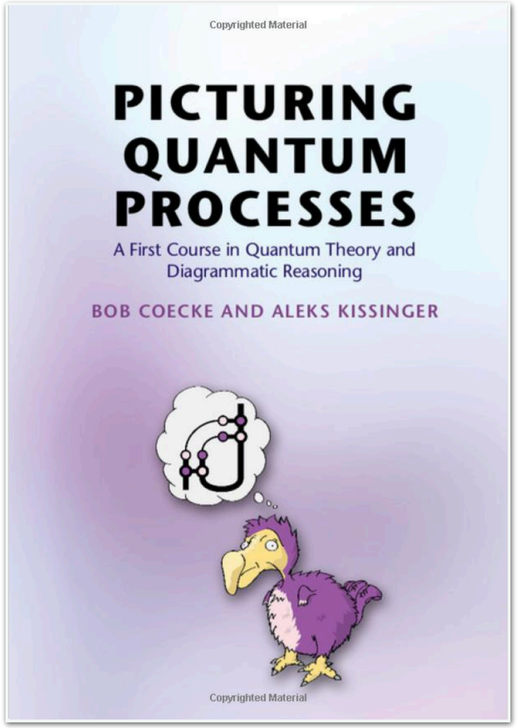
Mike Stay
Computer Science Department, University of Auckland
and
Google, 1600 Amphitheatre Pkwy
Mountain View, California 94043, USA

email: baez@math.ucr.edu, stay@google.com

March 2, 2009

Abstract

In physics, Feynman diagrams are used to reason about quantum processes. In the 1980s, it became clear that underlying these diagrams is a powerful analogy between quantum physics and topology. Namely, a linear operator behaves very much like a 'cobordism': a manifold representing spacetime, going between two manifolds representing space. This led to a burst of work on topological quantum field theory and 'quantum topology'. But this was just the beginning: similar diagrams can be used to reason about logic, where they represent proofs, and computation, where they represent programs. With the rise of interest in quantum cryptography and quantum computation, it became clear that there is extensive network of analogies between physics, topology, logic and computation. In this expository paper, we make some of these analogies precise using the concept of 'closed symmetric monoidal category'. We assume no prior knowledge of category theory, proof theory or computer science.



Pictures of Processes

Automated Graph Rewriting for Monoidal Categories and Applications to Quantum Computing

Aleks Kissinger
St. Catherine's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Michaelmas 2011

Part I:
String Diagrams as Processes

Processes

```
sort: List[A] ⇒ List[A]
```

$$M = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

$$f(x, y) = x^2 + y^2$$

INGREDIENTS

2 large eggs

Kosher salt

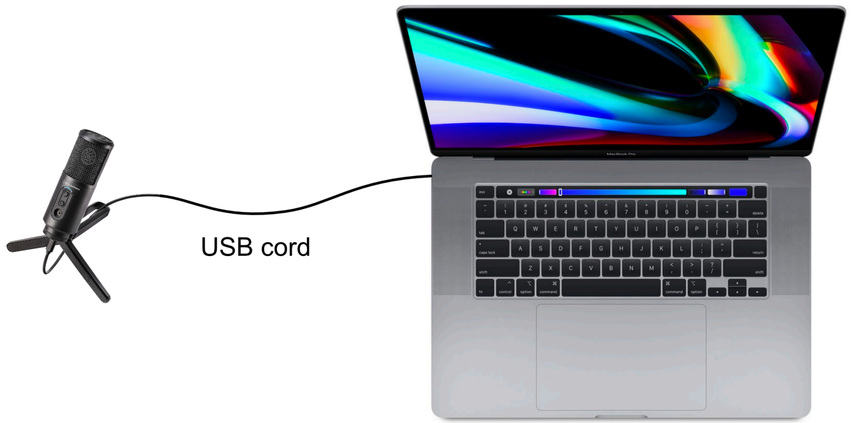
Freshly ground black pepper

Pinch red pepper flakes

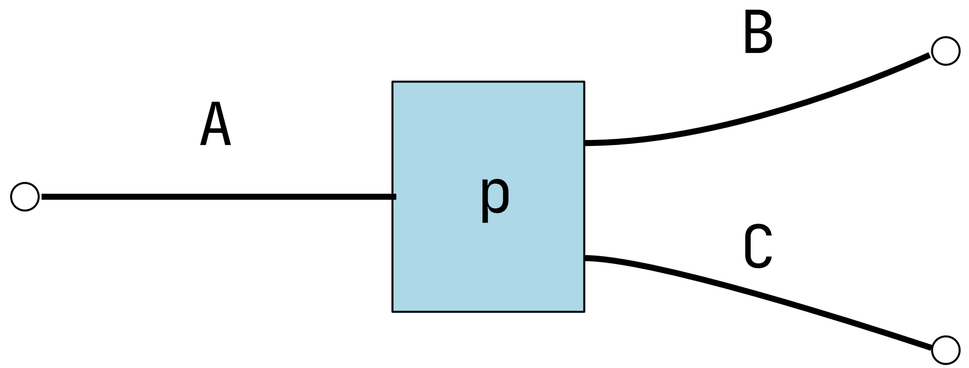
2 tbsp. butter

1/4 c. shredded cheddar

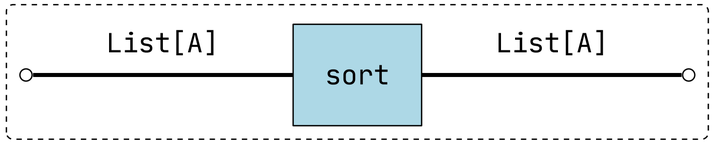
2 tbsp. freshly chopped chives

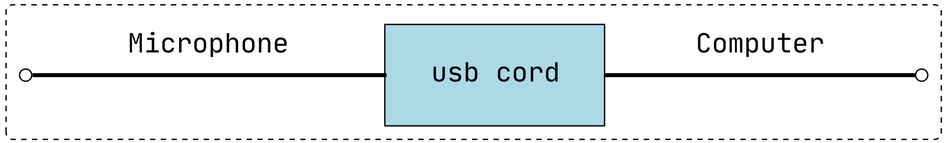


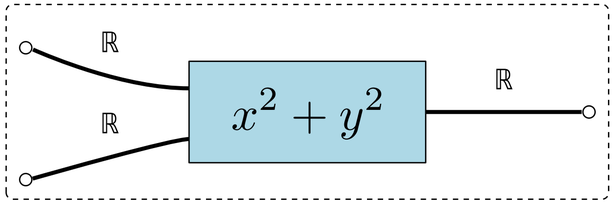
A process

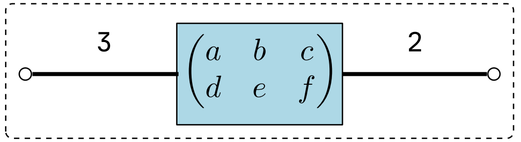


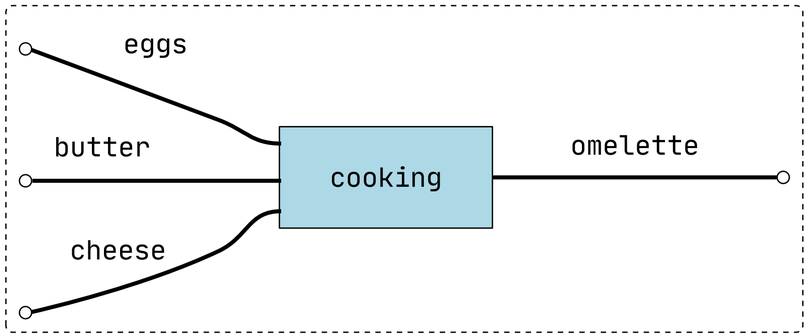
Processes as diagrams



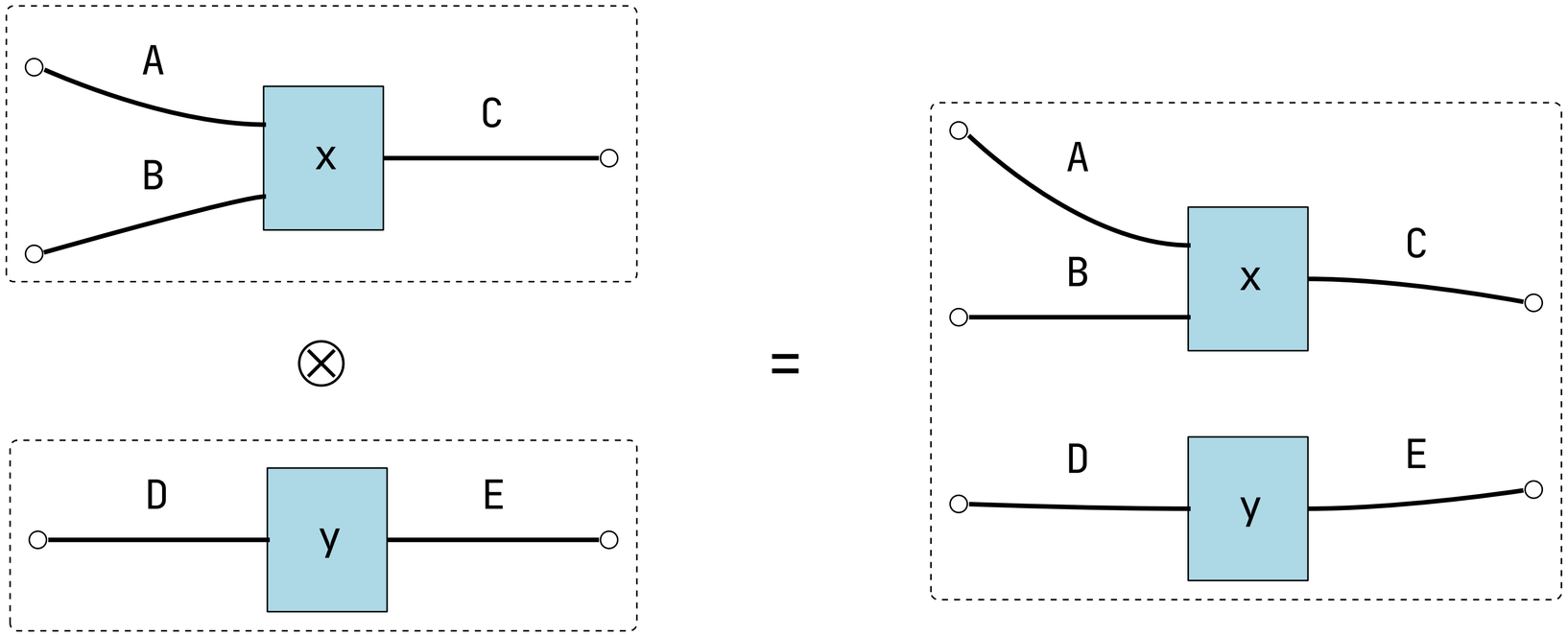




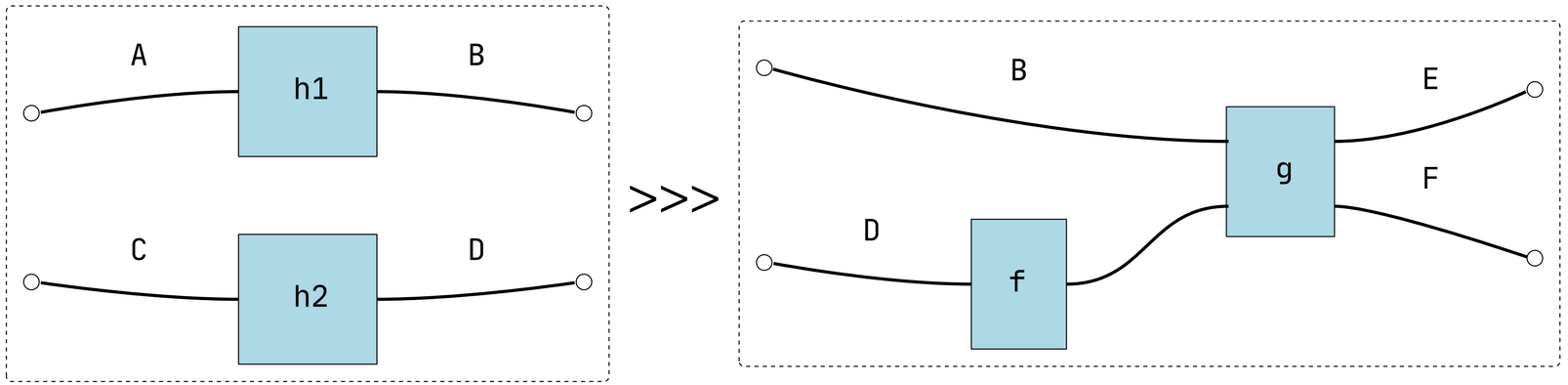




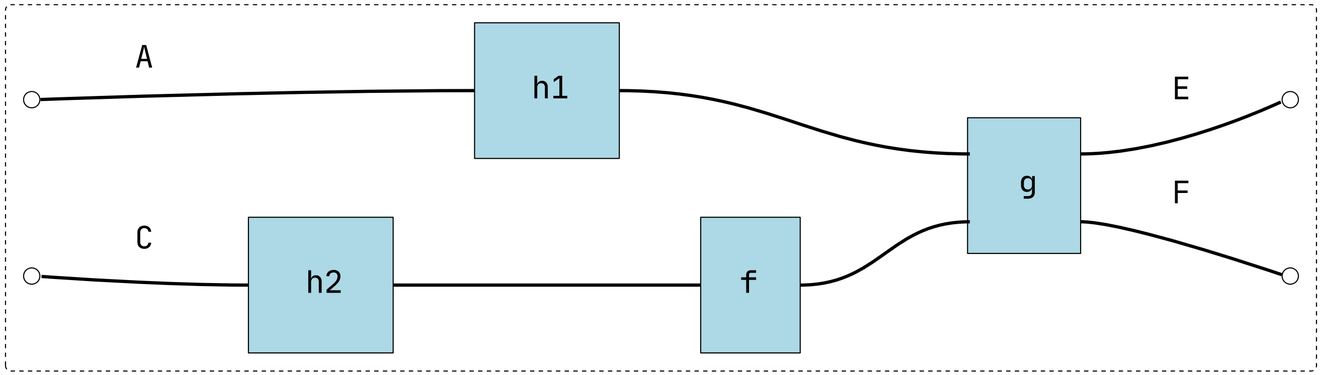
Parallel composition



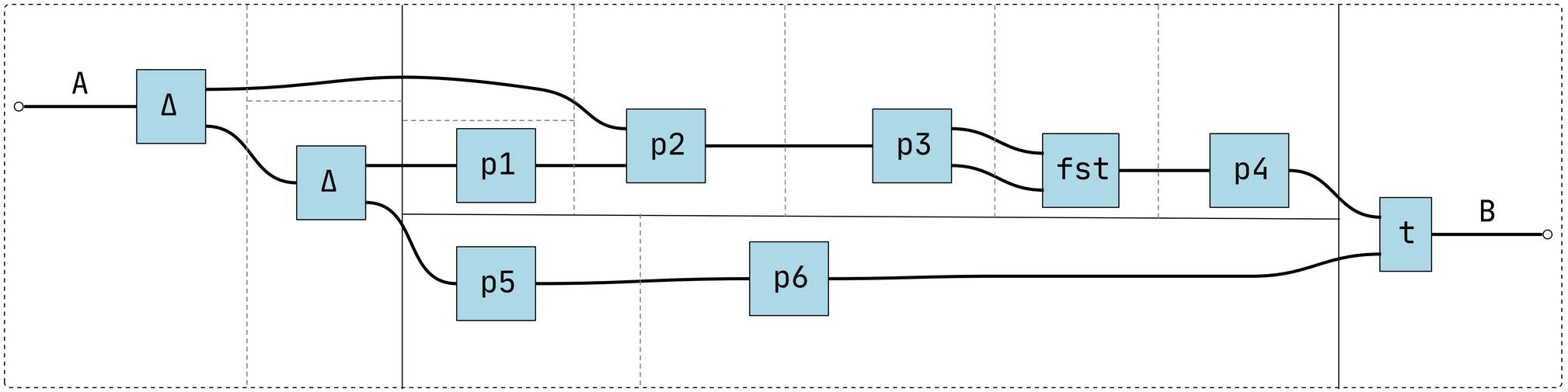
Sequential composition



=



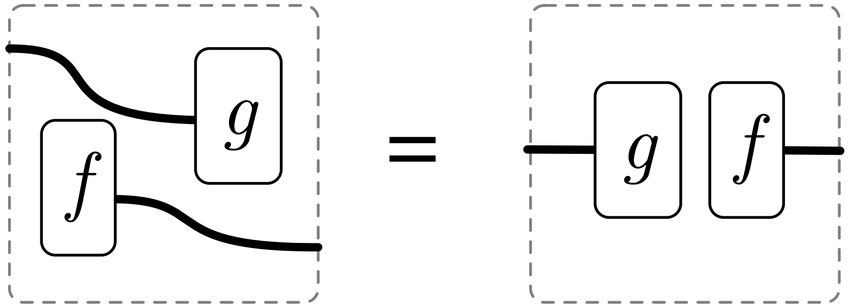
Complex diagrams

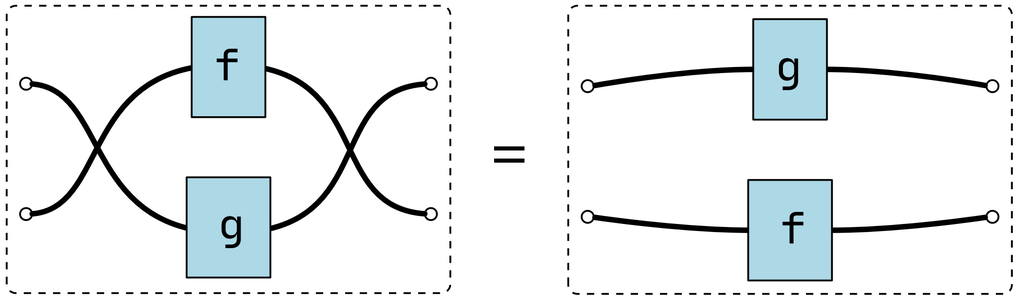


Process theories

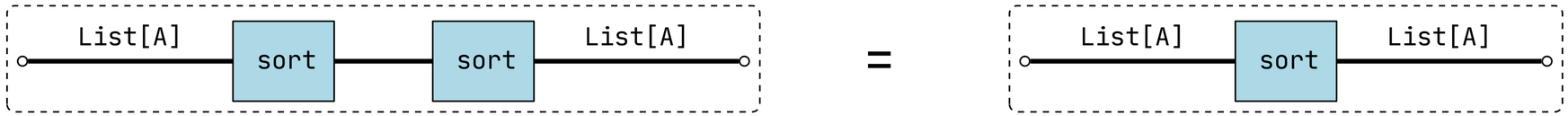
- A "process theory" is an interpretation of a diagram in terms of a concrete class of processes.
- In particular it provides an interpretation of **wires**, **boxes**, and **composition** operations of diagrams.
- Examples of process theories:
  - Functions and sets
  - linear maps and vector spaces
  - Matrices and natural numbers

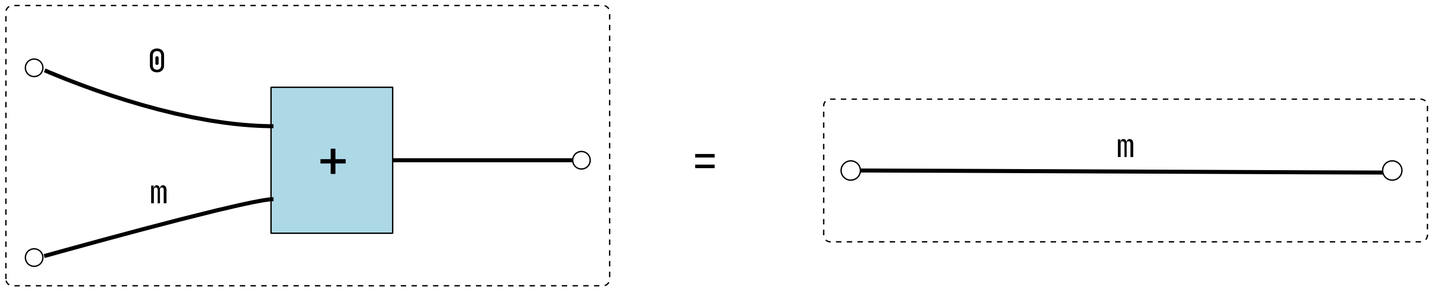
Diagram equations





Process equations



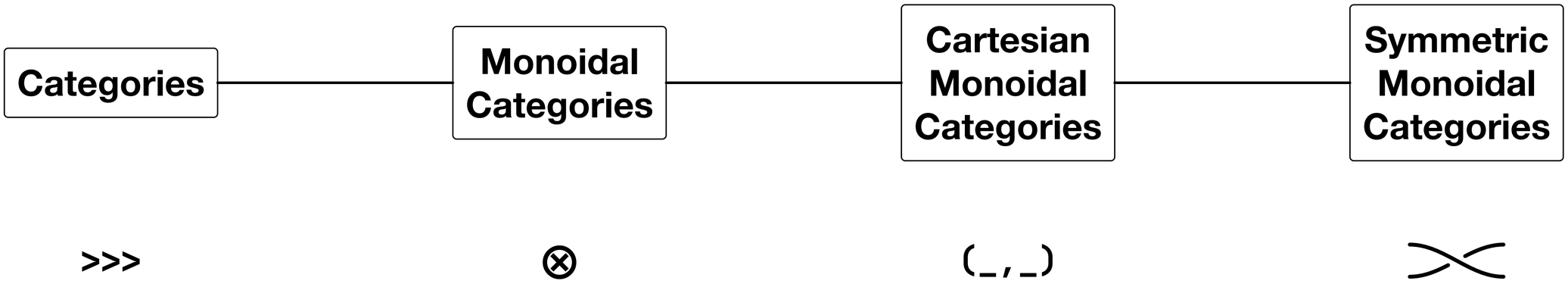


Part II:
String Diagrams as Monoidal Categories

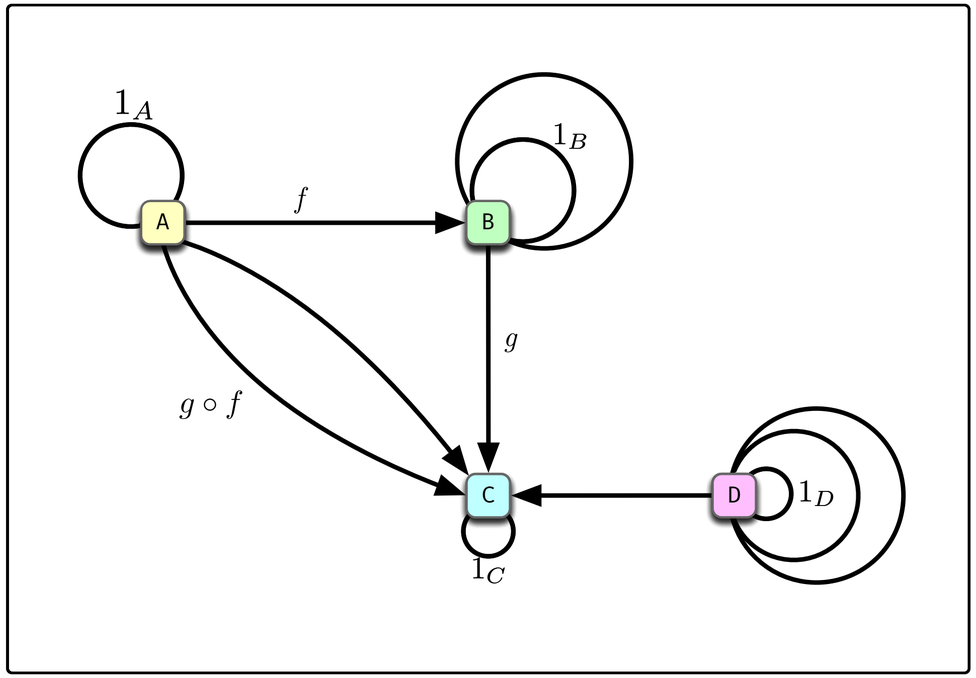
Formalization

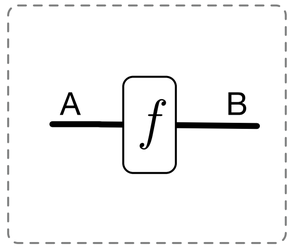
This diagrammatic language can be formalized by using category theory, specifically **monoidal categories**.

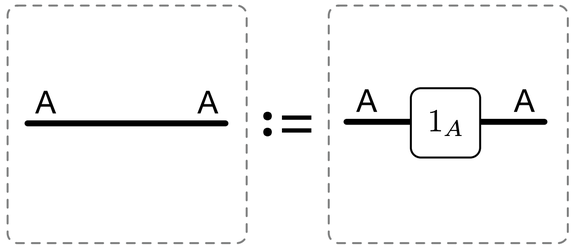
Plan

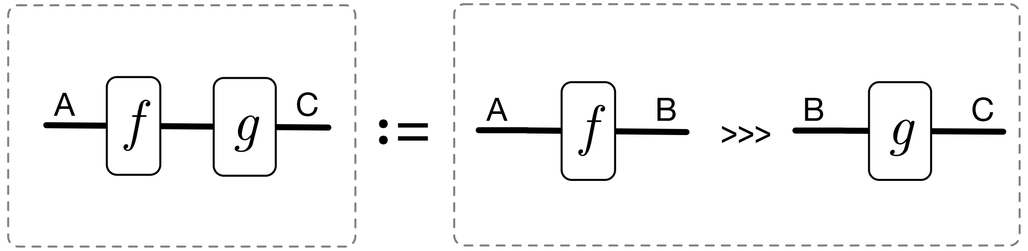

Categories
Monoidal Categories
Cartesian Monoidal Categories
Symmetric Monoidal Categories
>>>
⊗
(_,_)

1. Categories

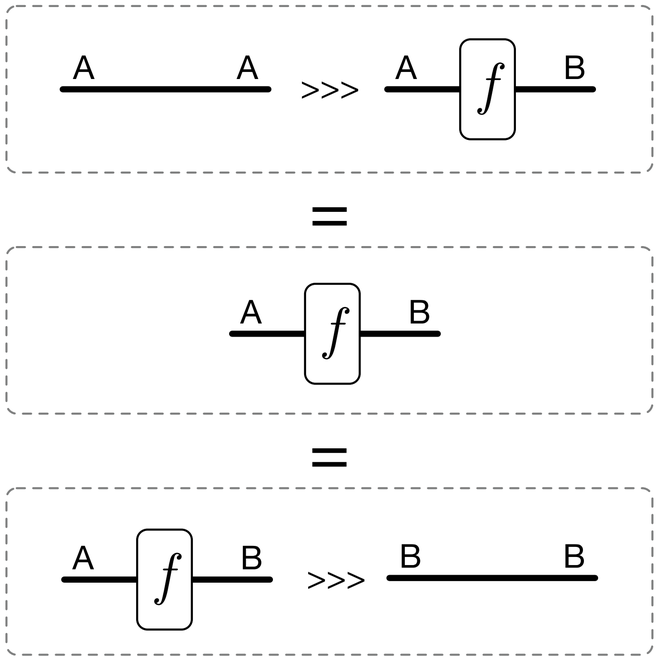


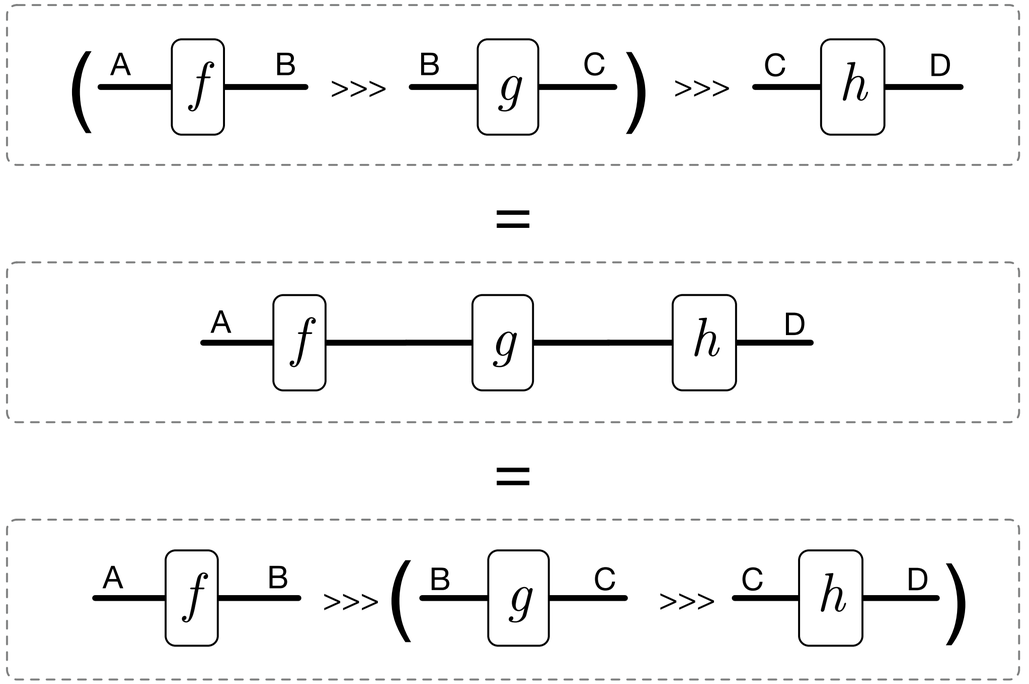






Categories: equations





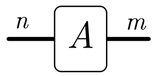
Example: Matrices

- objects: natural numbers
- arrows $n \to m$:
  matrices $m \times n$
- composition:
  matrix multiplication
- identity 1_n:
  identity matrix I_n

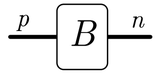
input (n)

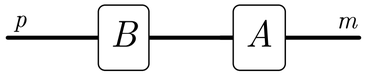
$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

output (m)

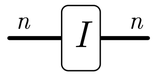


$$B = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{np} \end{pmatrix}$$





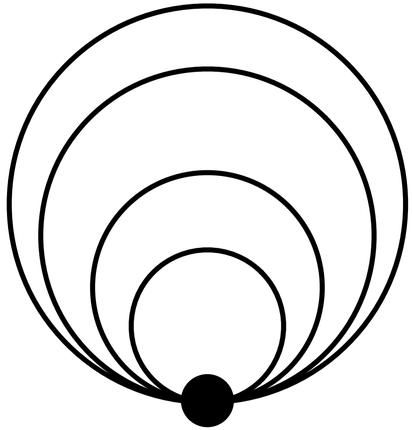
$$A \cdot B = \begin{pmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mp} \end{pmatrix}$$



$$I_n = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

2a. Strict Monoidal Categories

A category with a monoid structure on the objects and arrows

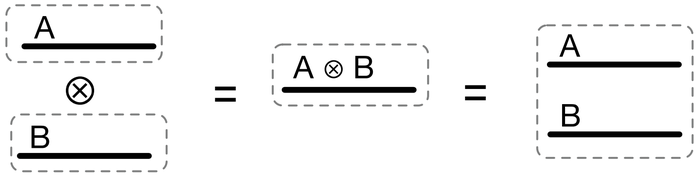


Strict Monoidal Categories

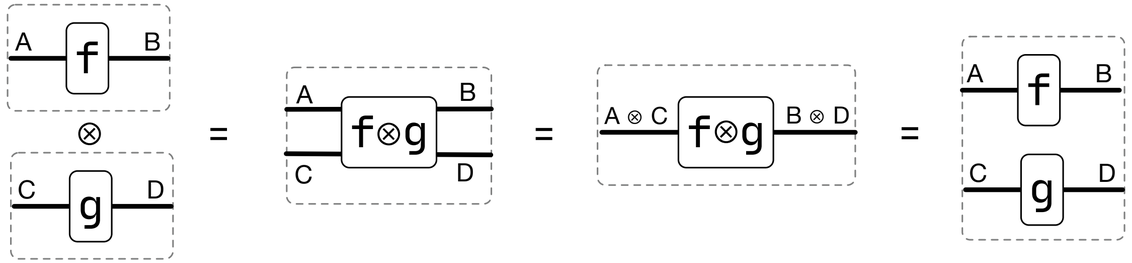
- Parallel composition: an *associative* binary operation called the ***tensor*** product

$$\otimes : C \times C \to C$$

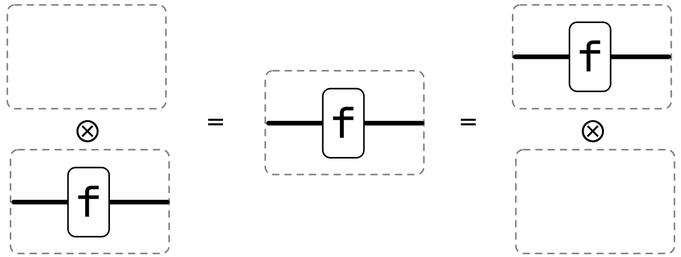
- That operates on objects



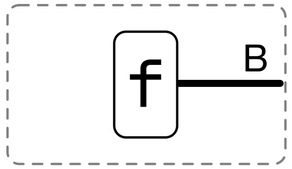
- and on arrows



- A neutral or unit object I, with identity $1_I : I \to I$

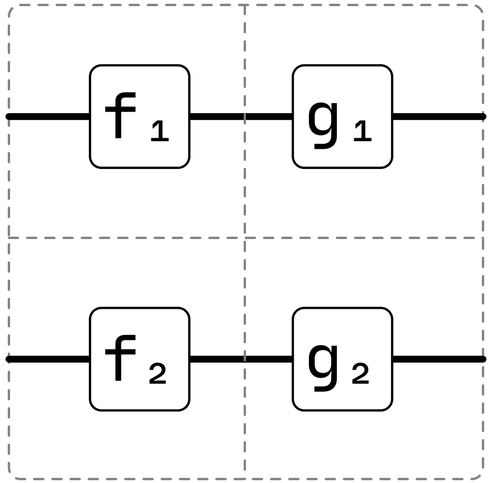


- for example, a morphism $f : I \to B$



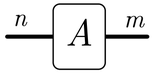
One more law

$$(f_1 \ggg g_1) \otimes (f_2 \ggg g_2) = (f_1 \otimes f_2) \ggg (g_1 \otimes g_2)$$

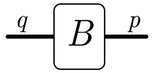


Matrices again

- Matrices form a monoidal category if we use the Kronecker product as parallel composition of arrows
- On objects it operates as multiplication of natural numbers
- The number 0 is the unit I



$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$



$$B = \begin{pmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{p1} & \cdots & b_{qp} \end{pmatrix}$$

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

$$I = ()$$

2b. Monoidal categories

- In many cases it's not literally true that:

  $$(X \otimes Y) \otimes Z = X \otimes (Y \otimes Z)$$

  or

  $$I \otimes X = X = X \otimes I$$

- For example in Scala

  ```
  ((x, y), z) ≠ (x, (y, z))
  ```

- Instead we're forced to require isomorphisms:

- **associator**:

  $$a_{X,Y,Z} : (X \otimes Y) \otimes Z \xrightarrow{\sim} X \otimes (Y \otimes Z)$$

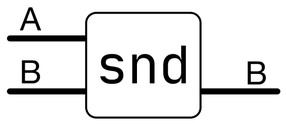
- left and right **unitors**:

  $$l_X : I \otimes X \xrightarrow{\sim} X$$
  $$r_X : X \otimes I \xrightarrow{\sim} X$$

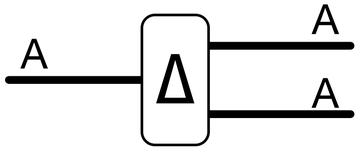
- *(satisfying certain laws)*

3. Cartesian Monoidal Categories

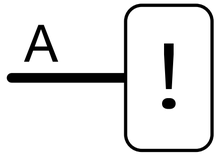
- We can extract the first and second components:



- we can duplicate information:

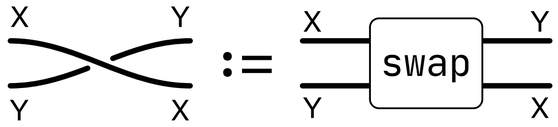


- and we can discard information:

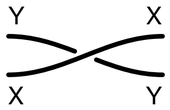


4. Symmetric Monoidal Categories

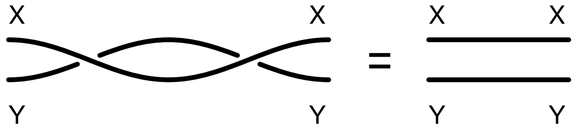
- we have a **swap** operation



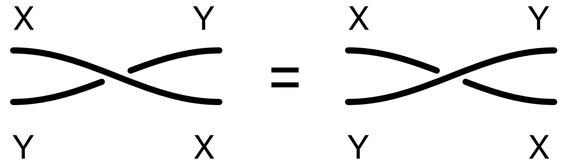
- with inverse depicted:



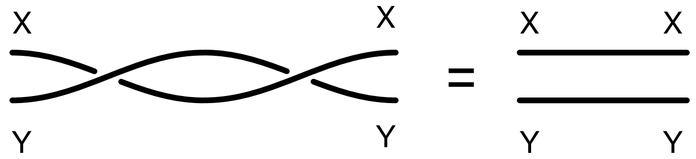
- this representation is chosen so that:



- furthermore, the swap satisfies



- i.e.



Circuit diagrams

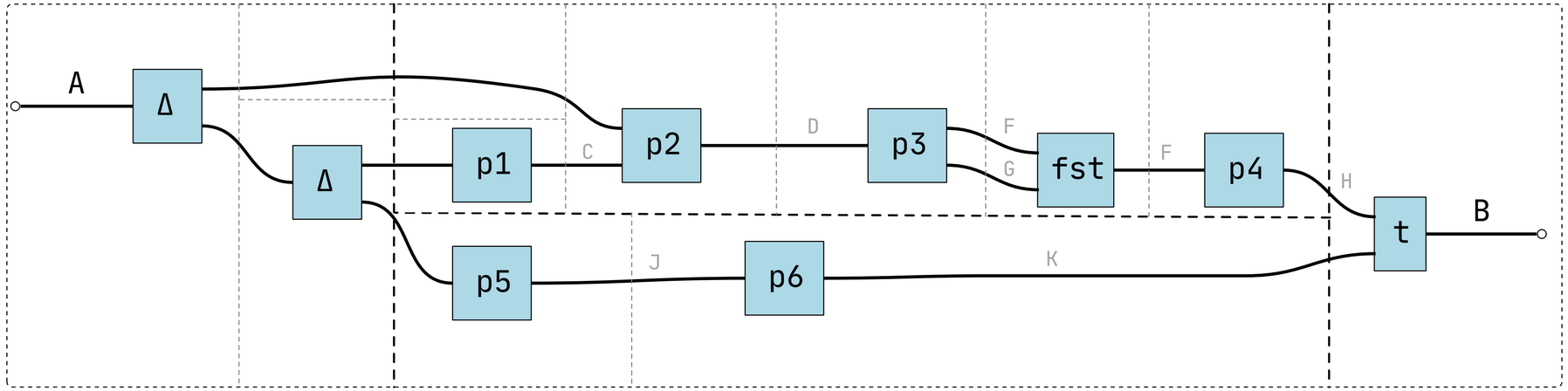
- The diagrams that we have discussed so far contain no loops.
- To be more precise they are called "circuit diagrams"
- Feedback loops can be introduced via **compact closed categories** (which won't be discussed here)

Part III:
String Diagrams in Scala

Airflow-like process manager

```
val top = (id ++ p1) >>> p2  >>> p3 >>> fst >>> p4
```

```
val initial =
  Δ >>> (id ++ Δ) >>> assocL
```



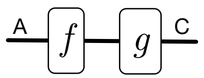

```
val bottom = p5 >>> p6
```

```
val program = initial >>> (top ++ bottom) >>> t
```

Tagless Process DSL

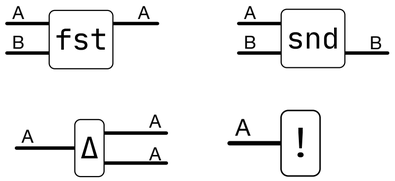
Category Structure

```
trait ProcessOpsDSL[Process[_, _]] {

  type ⇝[A, B] = Process[A, B]
  type ⇜[B, A] = Process[A, B]

  def id[A]: A ⇝ A

  extension [A, B, C] (f: A ⇝ B)
    @alpha("andThen")
    def >>> (g: B ⇝ C): A ⇝ C

  extension [A, B, C] (g: B ⇝ C)
    @alpha("compose")
    def ∘ (f: A ⇝ B): A ⇝ C = f >>> g
```



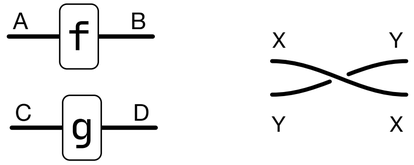
Cartesian Structure

```
def fst[A, B]: (A, B) ⇝ A
def snd[A, B]: (A, B) ⇝ B

extension [A, B, C] (f: A ⇝ B)
  @alpha("mergeInput")
  def &&& (g: A ⇝ C): A ⇝ (B, C)

@alpha("duplicate")
def Δ[A]: A ⇝ (A, A) =
  id[A] &&& id[A]

// terminal object and monoidal unit
type I = EmptyTuple

def discard[A]: A ⇝ I
```



Monoidal Structure

```
def assocR[X, Y, Z] : ((X, Y), Z) ⇝ (X, (Y, Z))
def assocL[X, Y, Z] : ((X, Y), Z) ⇜ (X, (Y, Z))

def injR[X]: X ⇝ (I, X)
def injL[X]: X ⇝ (X, I)


// tensor on objects:   A ⊗ B = (A, B)
// tensor on arrows: f ⊗ g = f ++ g

extension [A1, A2, B1, B2] (f: A1 ⇝ B1)
  @alpha("combine")
  def ++ (g: A2 ⇝ B2): (A1, A2) ⇝ (B1, B2)


def swap[X, Y]       : (X, Y) ⇝ (Y, X)
def swapInverse[X, Y]: (X, Y) ⇜ (Y, X)
```



Laws (the fine print)

Category Laws

```
@Law
 def associativity[A, B, C, D](
   f: A ⇝ B,
   g: B ⇝ C,
   h: C ⇝ D
 ) =
   h ∘ (g ∘ f) ⟷ (h ∘ g) ∘ f

 @Law
 def identityL[A, B](f: A ⇝ B) =
   (id[B] ∘ f) ⟷ f

 @Law
 def identityR[A, B](f: A ⇝ B) =
   f ∘ id[A] ⟷ f
```

Tensor Laws

```
@Law
def tensorCompositionLaw[
  A1, A2, B1, B2, C1, C2
](
  f1: A1 ⇝ B1, f2: A2 ⇝ B2,
  g1: B1 ⇝ C1, g2: B2 ⇝ C2,
) =
   (g1 ∘ f1) ++ (g2 ∘ f2) ⟷
   (g1 ++ g2) ∘ (f1 ++ f2)


@Law
def tensorIdentityLaw[A, B] =
  id[A] ++ id[B] ⟷ id[(A, B)]
```

Monoidal Laws

```
@Law
def triangleEquation[X, Y] =
  assocR[X, I, Y] >>> (id[X] ++ snd[I, Y]) ⟷
  (fst[I, Y] ++ id[Y])

@Law
def pentagonEquation[W, X, Y, Z] =
  (assocR[W, X, Y] ++ id[Z]) >>>
    assocR[W, (X, Y), Z] >>>
    (id[W] ++ assocR[X, Y, Z]) ⟷
  (assocR[(W, X), Y, Z] >>> assocR[W, X, (Y, Z)])
```

Symmetric Monoidal Laws

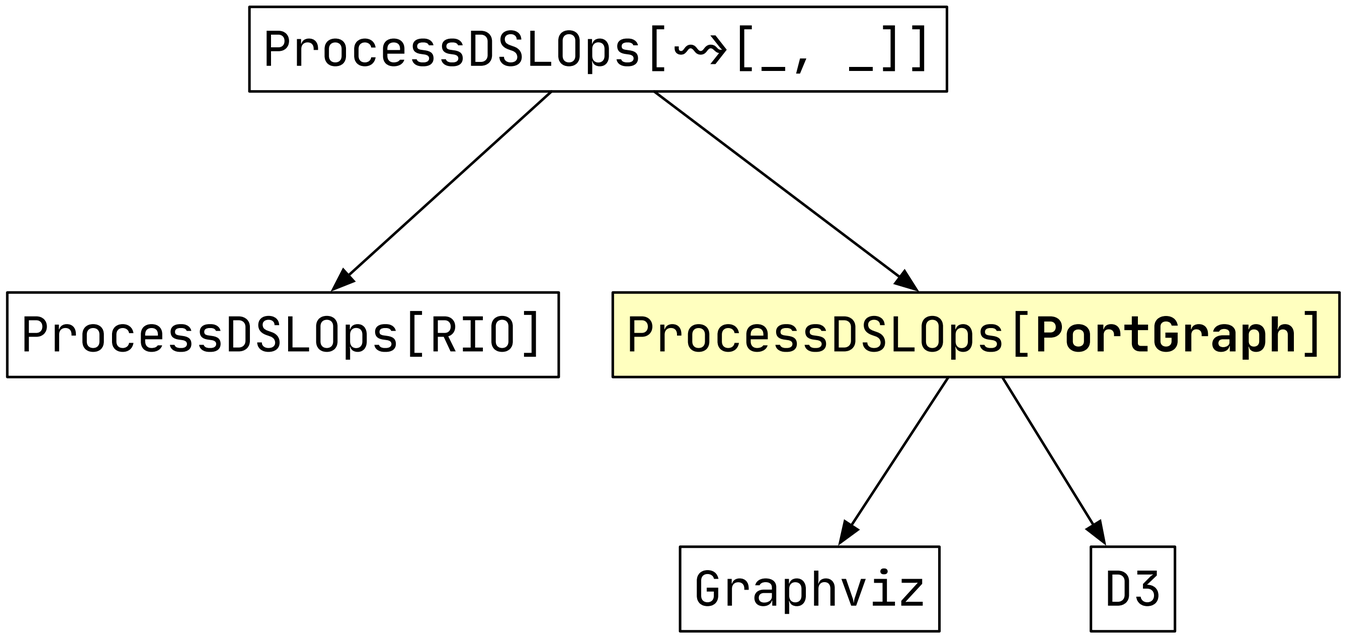
```
@Law
def hexagonEquation1[X, Y, Z] =
  (assocL[X, Y, Z] >>> (swap[X, Y] ++ id[Z]) >>>
    assocR[Y, X, Z] >>> (id[Y] ++ swap[X, Z]) >>>
    assocL[Y, Z, X]) ⟷
    swap[X, (Y, Z)]

@Law
def hexagonEquation2[X, Y, Z] =
  (assocR[X, Y, Z] >>> (id[X] ++ swap[Y, Z]) >>>
    assocL[X, Z, Y] >>> (swap[X, Z] ++ id[Y]) >>>
    assocR[Z, X, Y]) ⟷
    swap[(X, Y), Z]

@Law
def symmetry[X, Y] =
   swap[X, Y] ⟷
   swapInverse[Y, X]
```

ZIO interpreter

```scala
given RIOProcessOps as ProcessDSLOps[RIO]:

  def id[A] = RIO.identity[A]

  @alpha("andThen")
  def [A, B, C] (f: RIO[A, B]) >>> (g: RIO[B, C]) = f andThen g

  def fst[A, B]: RIO[(A, B), A] = RIO.first
  def snd[A, B]: RIO[(A, B), B] = RIO.second

  @alpha("mergeInput")
  def [A, B, C] (f: RIO[A, B]) &&& (g: RIO[A, C]): RIO[A, (B, C)] = f &&& g

  def discard[A]: RIO[A, I] = RIO.succeed(EmptyTuple)

  def assocR[X, Y, Z] : ((X, Y), Z) ⇝ (X, (Y, Z)) = RIO.fromFunction { case ((x, y), z) ⇒ (x, (y, z)) }
  def assocL[X, Y, Z] : ((X, Y), Z) ⇜ (X, (Y, Z)) = RIO.fromFunction { case (x, (y, z)) ⇒ ((x, y), z) }

  def injR[X]: X ⇝ (I, X) = RIO.fromFunction { a ⇒ (EmptyTuple, a) }
  def injL[X]: X ⇝ (X, I) = RIO.fromFunction { a ⇒ (a, EmptyTuple) }

  @alpha("combine")
  def [A1, A2, B1, B2] (f: A1 ⇝ B1) ++ (g: A2 ⇝ B2): (A1, A2) ⇝ (B1, B2) =
    val left  = fst >>> f
    val right = snd >>> g
    (left zipWithPar right) ((x, y) ⇒ (x, y))

  def swap       [X, Y]: (X, Y) ⇝ (Y, X) = RIO.swap
  def swapInverse[X, Y]: (X, Y) ⇜ (Y, X) = RIO.swap
```

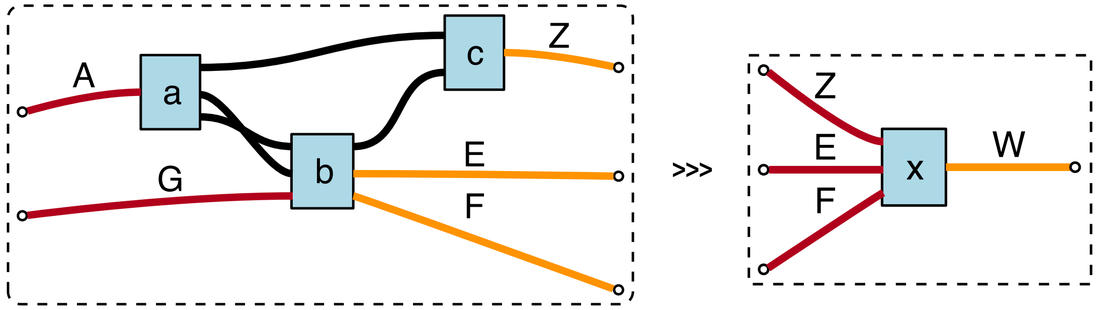
```
RIO[A,B]
≈
A ⇒ Either[Throwable, B]
```

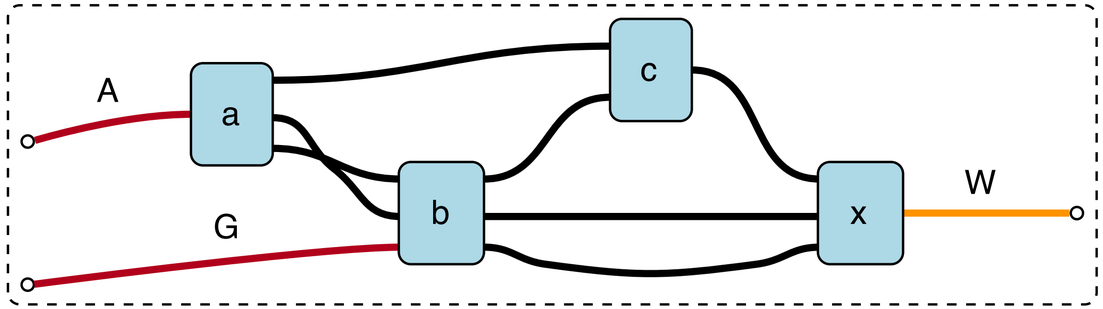
Process DSL


ProcessDSLOps[⇝[_, _]]
ProcessDSLOps[RIO]
ProcessDSLOps[PortGraph]
Graphviz
D3

Category of Port Graphs

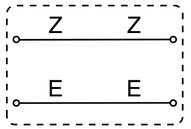
```scala
val graph1 =
  PortGraph(
    boxes = SeqMap(
      a → (List("A"), List("B","C","D")),
      b → (List("D","C","G"), List("Y","E","F")),
      c → (List("B","Y"), List("Z")),
    ),
    incoming = List((a, 0), (b, 2)),
    inner = List(
      (a, 0) → (c, 0),
      (a, 1) → (b, 1),
      (a, 2) → (b, 0),
      (b, 0) → (c, 1),
    ),
    outgoing = List((c, 0), (b, 1), (b, 2))
  )

val graph2 = singleBox("x", List("Z", "E", "F"), List("W"))

val combined = graph1 andThen graph2
```






```scala
def render[B, P](pg: PortGraph[B, P]): graphviz.model.Graph
  = ???
```

Identity:



the end

Bibliography

- Brendan Fong David I. Spivak . Seven Sketches in Compositionality
- John C. Baez, Mike Stay. Physics, Topology, Logic and Computation: A Rosetta Stone
- Bob Coecke, Aleks Kissinger. Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning.
- Aleks Kissinger. Pictures of Processes

Extras

Tagless Process DSL

Constructor

```
trait ProcessDSL[⇝[_, _]]:
  def httpCall[A: BodySerializer, B: Decoder](
    name: String, uri: Uri
  ): A ⇝ B
```

ZIO instance

```
given RIOProcessDSL as ProcessDSL[RIO]:
  val backend = HttpURLConnectionBackend()
  import sttp.client3._
  import sttp.client3.circe._

  def httpCall[A: BodySerializer, B: Decoder](
    name: String, uri: Uri
  ): RIO[A, B] =
    RIO.fromFunctionM[A, B] { (a: A) ⇒
      RIO.fromEither {
        basicRequest
          .post(uri)
          .body(a)
          .response(asJson[B])
          .send(backend)
          .body
          .left.map(new Exception(_))
      }
    }
```