

Category Theory as a Unifying Language for Science

Concrete examples from mathematics, computing, and natural sciences

Irus Grinis

Vilnius University, Faculty of Mathematics and Informatics, Institute of Computer Science

06/22/2026

Outline

- 1 The Basics
- 2 Mathematics
- 3 Structural Patterns
- 4 Computer Science
- 5 The Natural Sciences
- 6 Systems and Behavior
- 7 The 3rd Level

Main idea

Category theory studies

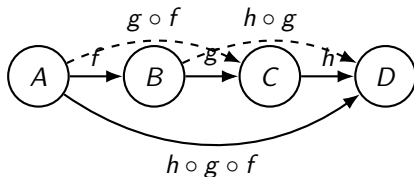
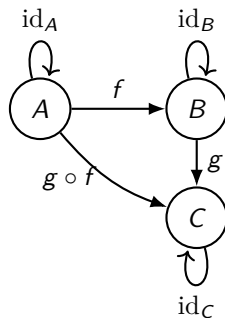
- objects;
- morphisms (arrows);
- identity morphisms;
- composition of morphisms;
- laws of composition.

Basic laws

$$(h \circ g) \circ f = h \circ (g \circ f)$$

$$f \circ \text{id}_A = f, \quad \text{id}_B \circ f = f$$

The same diagrammatic language appears in many fields. See Awodey [1]



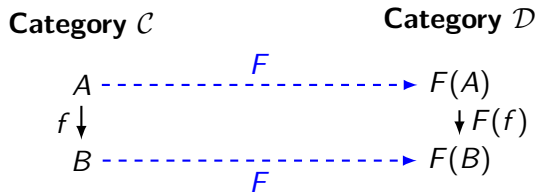
Dictionaries Between Categories: Functors

If categories are structural worlds, functors are structure-preserving translations between them.

A **Functor** $F : \mathcal{C} \rightarrow \mathcal{D}$ maps:

- **Objects** $A \in \mathcal{C}$ to objects $F(A) \in \mathcal{D}$.
- **Morphisms** $f : A \rightarrow B$ to morphisms $F(f) : F(A) \rightarrow F(B)$.

Crucial property: It preserves connectivity. $F(id_A) = id_{F(A)}$ and $F(g \circ f) = F(g) \circ F(f)$.



Endofunctors: A special case where the source and target categories are exactly the same ($F : \mathcal{C} \rightarrow \mathcal{C}$). We will use these extensively later to model behaviors and dynamic systems![1]

From Sets to Categories

Let's build intuition with discrete structures.

- **A Single Object (Monoids):**

Take a finite set X with a binary operation $X \times X \rightarrow X$. If we add associativity and an identity element, we get a monoid. *Categorical view: A monoid is just a category with exactly one object!*

- **Dots and Arrows (Directed Graphs):**

If we want multiple objects, we can draw a directed multigraph. Vertices act as **objects**, and edges act as **generating morphisms**.

- **The Free Category (Paths):**

To turn our graph into a category, we need composition. We simply define **morphisms** as *paths* (sequences of connected edges). Composition is just concatenating paths. See e.g. Awodey [1].

Example 1: algebra and topology

Algebra

Groups, rings, vector spaces are different structures, but each has:

objects + structure-preserving maps.

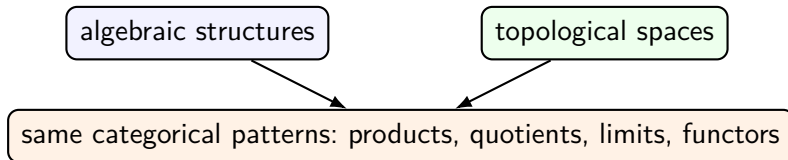
Examples: group homomorphisms, linear maps.

Topology

Topological spaces and continuous maps form another category:

$$X \xrightarrow{\text{continuous}} Y.$$

The categorical view lets us compare algebraic and spatial constructions. See Munkres [5].



Example 1a: from a circle to integers

Topological object

Consider the circle:

$$S^1 = \{z \in \mathbb{C} \mid |z| = 1\}.$$

Loops on the circle can wind around it:

$$0, 1, 2, -1, \dots$$

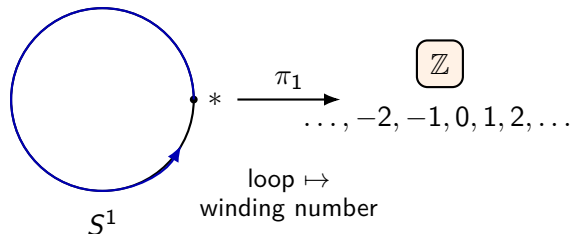
Algebraic shadow

The fundamental group records this as:

$$\pi_1(S^1) \cong \mathbb{Z}.$$

A loop is represented by its **winding number**.

See Munkres [5].



Categorical message

A functor turns geometric information into algebraic information:

$$\pi_1 : \mathbf{Top}_* \longrightarrow \mathbf{Grp}.$$

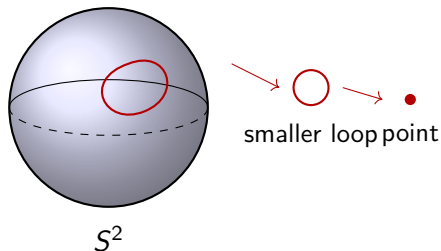
Example 1b: the sphere has no 1-dimensional hole

Topological object

Consider the sphere:

$$S^2 = \{(x, y, z) \in \mathbb{R}^3 \mid x^2 + y^2 + z^2 = 1\}.$$

A loop on the sphere can move freely on the surface.



Algebraic shadow

The fundamental group is trivial:

$$\pi_1(S^2) \cong 0.$$

So every loop is homotopic to the constant loop.

See Munkres [5]

Categorical message

The same functor distinguishes spaces:

$$\pi_1 : \mathbf{Top}_* \longrightarrow \mathbf{Grp}$$

$$S^1 \mapsto \mathbb{Z}, \quad S^2 \mapsto 0.$$

Example 1c: the torus has two independent loops

Topological object

Consider the torus:

$$T^2 = S^1 \times S^1.$$

A loop on the torus can wind in two independent directions.

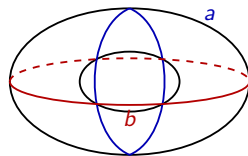
Algebraic shadow

Its fundamental group is

$$\pi_1(T^2) \cong \mathbb{Z} \times \mathbb{Z}.$$

A loop corresponds to a pair of winding numbers:

$$(m, n) \in \mathbb{Z} \times \mathbb{Z}.$$



Torus T^2

Categorical message

The same functor detects a richer structure:

$$\pi_1(T^2) \cong \mathbb{Z} \times \mathbb{Z}.$$

So topology is translated into algebra.

Example 1d: from algebra to topology

Algebraic input

Take a polynomial over $\mathbb{R}$:

$$p(x, y) = x^2 + y^2 - 1.$$

Zero set

Define:

$$V(p) = \{(x, y) \in \mathbb{R}^2 \mid p(x, y) = 0\}.$$

For this polynomial:

$$V(p) = S^1.$$

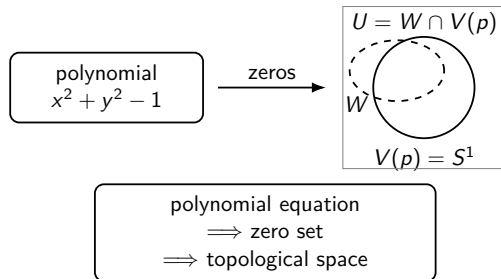
Topology

We give $V(p)$ the subspace topology from $\mathbb{R}^2$:

$$U \subseteq V(p) \text{ is open} \iff U = W \cap V(p)$$

for some open $W \subseteq \mathbb{R}^2$.

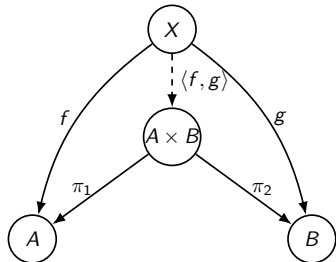
See Atiyah [6].



Universal construction: products — combining information

Universal idea

A product is not only a pair of objects. It is an object $A \times B$ that universally combines information from A and B .



Universal property

For every object X and arrows

$$f : X \rightarrow A, \quad g : X \rightarrow B,$$

there exists a unique arrow

$$\langle f, g \rangle : X \rightarrow A \times B$$

such that

$$\pi_1 \circ \langle f, g \rangle = f, \quad \pi_2 \circ \langle f, g \rangle = g.$$

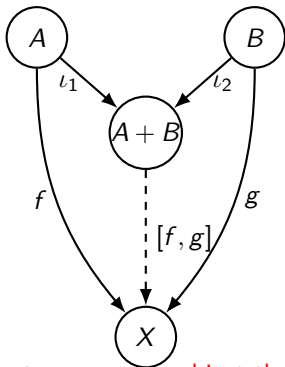
Product means: one object that contains exactly the information needed to map to both parts.

See Awodey [1].

Another universal construction: coproducts — combining alternatives

Dual universal idea

A coproduct is an object $A + B$ that universally combines alternatives: values may come from A or from B .



Universal property

For every object X and arrows

$$f : A \rightarrow X, \quad g : B \rightarrow X,$$

there exists a unique arrow

$$[f, g] : A + B \rightarrow X$$

such that

$$[f, g] \circ \iota_1 = f, \quad [f, g] \circ \iota_2 = g.$$

Coproduct means: one object that contains exactly the alternatives needed to define a map out of both parts. See Awodey [1].

Example 2a: Applications of Products and Coproducts

Products $A \times B$: *both*

- **Sets:** ordered pairs (a, b)
- **Programming:** records, structs, tuples
- **Logic:** conjunction $P \wedge Q$
- **Databases:** joining compatible info
- **Machine Learning:** feature vectors
- **Geometry:** a cylinder ($\text{Circle} \times \text{Line}$)
- **Probability:** joint random variables
- **Systems:** parallel components

Coproducts $A + B$: *either*

- **Sets:** tagged alternatives $A \sqcup B$
- **Programming:** variants, enums, unions
- **Logic:** disjunction $P \vee Q$
- **Databases:** alternative record types
- **Machine Learning:** classification classes
- **Geometry:** composite shapes ($\text{Triangle} + \text{Square}$)
- **Probability:** mixture / case distinction
- **Systems:** branching behaviours

Categorical Message:

Products systematically **collect information**; coproducts systematically **represent choices**.

Example 2b: product and coproduct types in C

Product type: struct

Contains both parts

```
typedef struct {  
    int x;  
    int y;  
} Point;
```

$$\text{Point} \cong \text{int} \times \text{int}$$

A value of type Point contains *x and y*.

$$p \mapsto p.x, \quad p \mapsto p.y$$

These are projections.

Coproduct-like type: tagged union

Contains one chosen alternative

```
typedef enum { INT_VALUE, DOUBLE_VALUE } Tag;  
  
typedef struct {  
    Tag tag;  
    union { int i; double d; } data;  
} Number;
```

$$\text{Number} \approx \text{int} + \text{double}$$

A value of type Number is either an int or a double; the tag says which one.

Categorical message

Products model data with several components. Coproducts model data with alternatives.

$$A \times B : \text{both } A \text{ and } B, \quad A + B : \text{either } A \text{ or } B.$$

Example 2c: coproducts as choice types

A common programming situation

A computation may return one of two kinds of results:

`Result = Success + Error.`

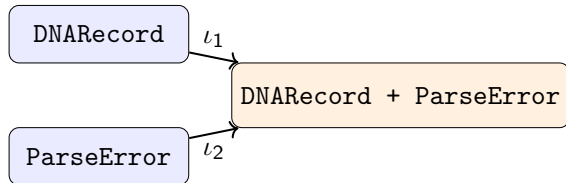
For example, parsing a DNA sequence may produce:

`DNARecord + ParseError.`

Coproduct intuition

The coproduct $A + B$ contains values coming from either A or B :

$$A \xrightarrow{\iota_1} A + B \quad B \xrightarrow{\iota_2} A + B.$$



one type containing
two alternatives

Categorical message

A coproduct models a controlled choice:

either a valid result, or an error.

Example 2d: Products and Coproducts in Other Sciences

Products $A \times B$: *both*

- **Classical Physics:** phase spaces
(Position $\times$ Momentum)
- **Chemistry:** required co-reactants in a network
- **Biology:** paired homologous chromosomes
- **Engineering:** multi-sensor fusion
(Sensor 1 $\times$ Sensor 2)
- **Functional Prog.:** tuple types (a, b)

Coproducts $A + B$: *either*

- **Classical Physics:** disjoint physical boundaries or subsystems
- **Chemistry:** alternative, branching reaction pathways
- **Biology:** divergent evolutionary lineages
- **Engineering:** fault-tolerant backups
(Primary + Backup)
- **Functional Prog.:** sum types like
Either a b

Categorical Message:

Whether tracking particles, reactions, or abstract algebraic data types, Products enforce **concurrent existence**, while Coproducts model **divergent possibilities**.

Example 3: computer science

Types and programs

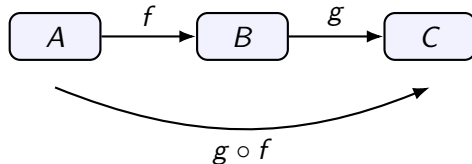
A simple categorical interpretation:

types as objects, programs as morphisms.

Composition is ordinary program
composition.

Why useful?

- modular semantics;
- compositional reasoning;
- type constructors as functors;
- effects via monads.



Message

Correct composition of programs mirrors
correct composition of morphisms.

Example 3 (cont.). The Hidden Categories in Computer Science

Before evaluating code, we must distinguish the text from its meaning.

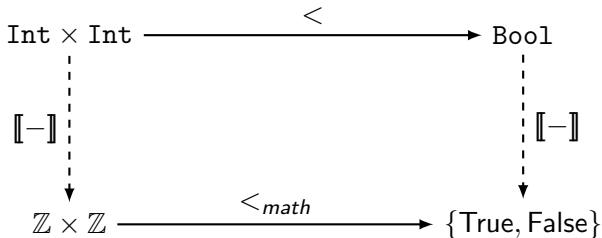
- **The Category of Syntax ($\mathcal{C}_{syn}$):**
 - **Objects:** Types in our programming language (e.g., `Int`, `Bool`).
 - **Morphisms:** Operations and language constructs (e.g., `<` is a morphism from `Int × Int` → `Bool`).
 - **Composition:** Nesting expressions to build syntax trees.
- **The Category of Semantics ($\mathcal{C}_{sem}$):**
 - **Objects:** Mathematical sets (e.g., $\mathbb{Z}$ for integers, $\{\text{True}, \text{False}\}$ for booleans).
 - **Morphisms:** Actual mathematical functions mapping between these sets.

Evaluation is a Functor

How do we formally connect syntax to semantics?

We define a Functor $F : \mathcal{C}_{syn} \rightarrow \mathcal{C}_{sem}$ (traditionally denoted using semantic brackets $\llbracket - \rrbracket$).

This functor maps our code to its mathematical reality while **strictly preserving structure**.



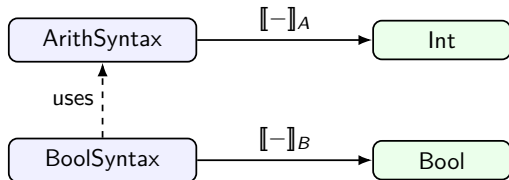
The Categorical Guarantee:

Because $\llbracket - \rrbracket$ is a functor, evaluating a complex syntax tree is guaranteed to be equivalent to mapping its parts and composing them in $\mathcal{C}_{sem}$.

Example 3a: modular semantics as composition

A useful viewpoint

A programming language can be assembled from fragments, and its semantics can be assembled from the semantics of these fragments.



Language = ArithSyntax + BoolSyntax + Commands

$$\llbracket - \rrbracket_{\text{Language}} = \llbracket - \rrbracket_A + \llbracket - \rrbracket_B + \llbracket - \rrbracket_C$$

Slogan: meanings of large systems are assembled from meanings of smaller systems.

Example 3a: modular semantics of a Python expression

What is modular semantics?

The meaning of a complex expression is built from the meanings of its smaller parts. Each syntactic construct has its own semantic rule.

Expression

$$(a < 3) + (b == 2)$$

Assume:

$$\rho(a) = 2, \quad \rho(b) = 5.$$

Semantic modules

$$\llbracket a \rrbracket_\rho = \rho(a), \quad \llbracket b \rrbracket_\rho = \rho(b), \quad \llbracket n \rrbracket_\rho = n,$$

$$\llbracket e_1 < e_2 \rrbracket_\rho = \llbracket e_1 \rrbracket_\rho < \llbracket e_2 \rrbracket_\rho,$$

$$\llbracket e_1 == e_2 \rrbracket_\rho = \llbracket e_1 \rrbracket_\rho == \llbracket e_2 \rrbracket_\rho.$$

The expression is evaluated compositionally: part by part, rule by rule.

Example 3a: assembling the meaning

Python allows Booleans to be used as integers in arithmetic contexts:

$$\text{num}(\text{True}) = 1, \quad \text{num}(\text{False}) = 0.$$

So addition in this expression is interpreted as

$$\llbracket e_1 + e_2 \rrbracket_\rho = \text{num}(\llbracket e_1 \rrbracket_\rho) + \text{num}(\llbracket e_2 \rrbracket_\rho).$$

Now compute:

$$\begin{aligned} \llbracket a < 3 \rrbracket_\rho &= \llbracket a \rrbracket_\rho < \llbracket 3 \rrbracket_\rho = 2 < 3 = \text{True}, \\ \llbracket b == 2 \rrbracket_\rho &= \llbracket b \rrbracket_\rho = \llbracket 2 \rrbracket_\rho = 5 = 2 = \text{False}. \end{aligned}$$

Therefore:

$$\begin{aligned} \llbracket (a < 3) + (b == 2) \rrbracket_\rho &= \text{num}(\llbracket a < 3 \rrbracket_\rho) + \text{num}(\llbracket b == 2 \rrbracket_\rho) \\ &= \text{num}(\text{True}) + \text{num}(\text{False}) \\ &= 1 + 0 = 1. \end{aligned}$$

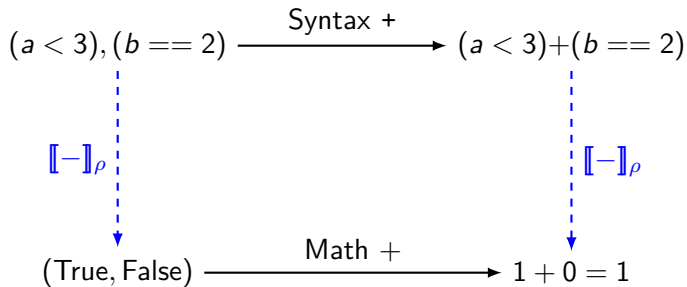
Modular semantics

The whole meaning is assembled from smaller semantic modules: variables, constants, comparisons, Booleans, conversion, and addition.

Functor's Play: Evaluation as a Commutative Diagram

Let's watch the Functor $\llbracket - \rrbracket_\rho$ play with our Python expression.

Given $\rho(a) = 2$ and $\rho(b) = 5$, the diagram must commute.



The Commutative Square:

We can assemble the syntax string and evaluate the whole (top $\rightarrow$ right $\rightarrow$ bottom), OR evaluate the parts and sum the numbers mathematically (left $\rightarrow$ bottom $\rightarrow$ right). Both paths yield 1.

Example 3b: compositional reasoning

Expression

$$(a < 3) + (b == 2), \quad \rho(a) = 2, \quad \rho(b) = 5.$$

Instead of reasoning about the whole expression at once, split it:

$$e_1 = (a < 3), \quad e_2 = (b == 2).$$

$$\llbracket e_1 \rrbracket_\rho = 2 < 3 = \text{True},$$

$$\llbracket e_2 \rrbracket_\rho = 5 = 2 = \text{False}.$$

Now combine the local results:

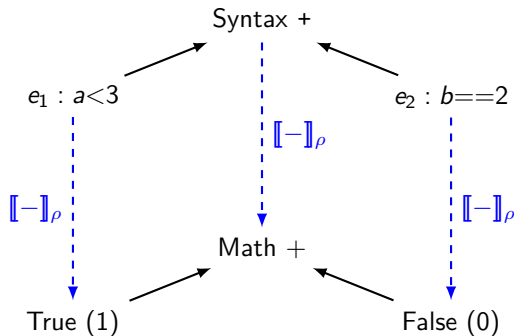
$$\begin{aligned} \llbracket e_1 + e_2 \rrbracket_\rho &= \text{num}(\text{True}) + \text{num}(\text{False}) \\ &= 1 + 0 = 1. \end{aligned}$$

Slogan

Reason locally, then compose the conclusions.

Functor's Play: Local Reasoning on Trees

Let's watch the Functor apply local reasoning to our expression tree. Compositional reasoning means we do not need to hold the entire program in our head. We apply the functor $\llbracket - \rrbracket_\rho$ to isolated branches.



Bottom-Up Evaluation:

Because the functor preserves structure, we safely "snip" e_1 and e_2 , map them to their semantic values locally, and let the root node add the results: $1 + 0 = 1$.

Example 3c: type constructors as functors

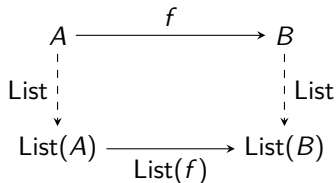
From functions to lifted functions

Given a function

$$f : A \rightarrow B,$$

the list type constructor gives a new function

$$\text{List}(f) : \text{List}(A) \rightarrow \text{List}(B).$$



Interpretation

The type constructor changes the objects, and `map` changes the arrows.

Example 3d: effects via monads

Problem

Ordinary functions compose directly:

$$f : A \rightarrow B, \quad g : B \rightarrow C \quad \Rightarrow \quad g \circ f : A \rightarrow C.$$

But programs may fail, read input, write logs, use state, etc.

Effectful functions

For possible failure:

$$f : A \rightarrow \text{Maybe}(B), \quad g : B \rightarrow \text{Maybe}(C).$$

Now ordinary composition does not fit directly.

$$A \xrightarrow{f} \text{Maybe}(B) \quad B \xrightarrow{g} \text{Maybe}(C)$$

Monad idea

A monad provides a controlled way to compose effectful functions:

$$\text{bind} : \text{Maybe}(B) \times (B \rightarrow \text{Maybe}(C)) \rightarrow \text{Maybe}(C).$$

```
safe(s):  
    x = parse_int(s)      # Maybe(Int)  
    y = reciprocal(x)     # Maybe(Float)  
    return y
```

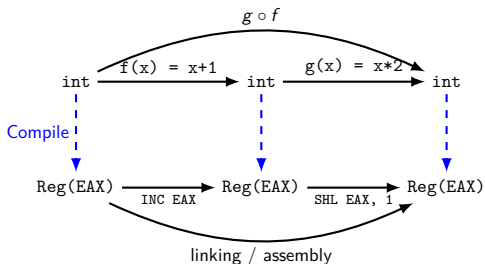
Slogan

A monad lets us compose functions whose results carry an effect.

Example 3e: Compilation as a Functor

Problem: How do we know a compiler is correct? A compiler translates high-level code into machine code. Categorically, this is a mapping between two entirely different structural worlds.

- **Source Category:** High-level types (objects) and functions (morphisms).
- **Target Category:** Machine hardware states like registers and memory layouts (objects), and assembly instruction sequences (morphisms).



Categorical Message: In a suitable categorical model, compiler correctness can be expressed as preservation of composition and meaning.

Physics: how can a physical law become categorical?

Categorical viewpoint

A physical theory often describes:

states and processes transforming states.

So we may form a category:

objects = physical states or systems, morphisms = physical processes.

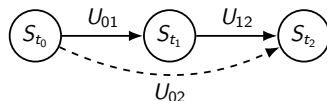
Example: motion

Newtonian evolution sends an initial state to a later state:

$$(q(t_0), v(t_0)) \longrightarrow (q(t_1), v(t_1)).$$

Time evolution composes:

$$U_{t_2, t_3} \circ U_{t_1, t_2} = U_{t_1, t_3}.$$



$$U_{12} \circ U_{01} = U_{02}$$

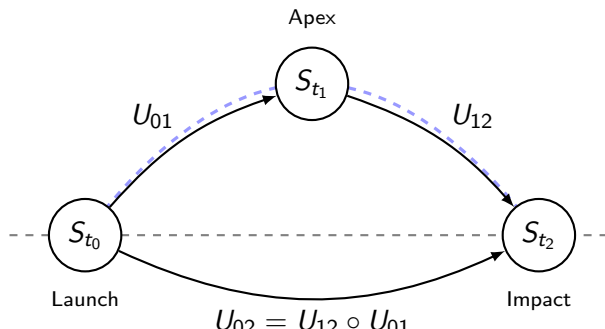
A physical law becomes categorical when it tells us how processes compose. See Fong, Spivak, 2019 [?]

Example: The Categorical Cannonball

Let's make time evolution concrete with a classic physics problem.

Imagine firing a cannonball. We can model its flight as a category:

- **Objects (States):** The exact condition of the cannonball at time t .
 $S_t = (\text{position } (x, y), \text{velocity } (v_x, v_y))$
- **Morphisms (Processes):** The laws of kinematics (gravity acting over time) evolving the ball from state S_{t_0} to S_{t_1} . Let's call this U_{01} .



Physics: time-independent systems

Idea

Not every physical model must describe time evolution. Sometimes we study possible configurations of a system:

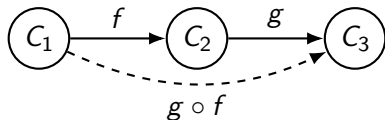
points, positions, fields, shapes, equilibria.

Categorical model

We can form a category where:

objects = configurations,

morphisms = allowed transformations between configurations.



Physics: 1D, 2D, and 3D systems

Dimension	Physical example	Objects	Morphisms
1D	spring, wire, mass on a line	positions, intervals, configurations	allowed displacements, deformations
2D	membrane, field on a plane	points, regions, field states	continuous maps, symmetries, boundary changes
3D	solid body, fluid, electric or magnetic fields	volumes, shapes, field configurations	rotations, flows, transformations

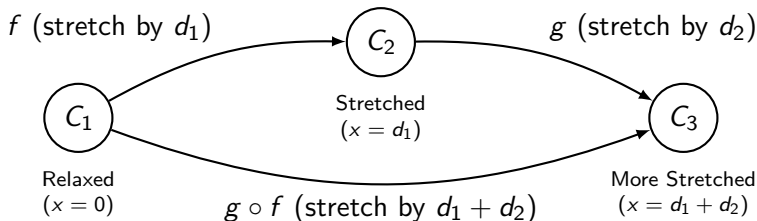
Objects describe possible configurations; morphisms describe allowed changes between them.

Example: The Categorical Spring

Let's make a static 1D system concrete.

Imagine a simple stretchable spring. Instead of tracking time, we track **configurations** and **deformations**.

- **Objects (Configurations):** The static states of the spring (e.g., relaxed, stretched, compressed). Let's call them C_i .
- **Morphisms (Deformations):** The physical action of applying a displacement Δx that changes one configuration into another.



Example 4: physics, chemistry, and reaction networks

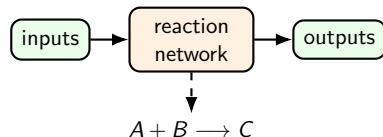
Open systems

Many scientific systems have inputs, internal dynamics, and outputs.

- electric circuits compose along terminals;
- chemical reactions compose through shared species;
- physical processes compose through interfaces.

Category theory helps separate:

interface + internal mechanism.
See Baez, 2011 [4].



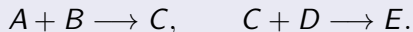
Compositional modeling

Large networks can be assembled from smaller reaction modules.

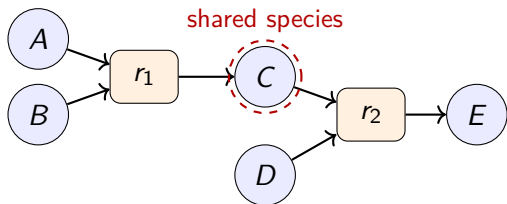
Example 5a: chemical reactions compose through shared species

Small chemical network

Two reactions can be connected by a species that is produced by one reaction and consumed by another:

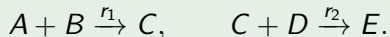


The species C is the **interface** between the two reactions.



Categorical reading

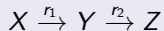
A reaction is not only an equation. It can be viewed as a small system with inputs and outputs:



Composition glues systems along the common boundary C .

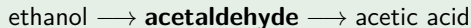
Example 5b: reactions compose through shared species

One recurring pattern

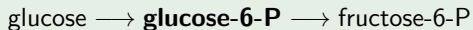


The intermediate species Y acts as an **interface**.

Organic chemistry



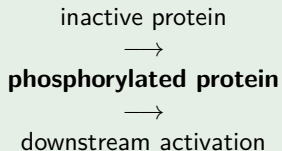
Glycolysis



Enzyme reaction



Signaling cascade

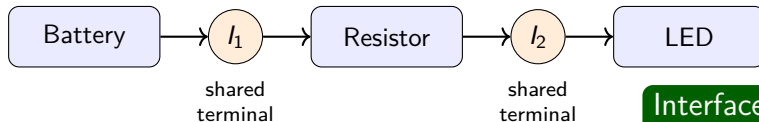


Categorical message: larger reaction networks are built by gluing smaller processes along shared species.

Example 6a: physical processes compose through interfaces

Physical systems are often open systems

A component is not isolated: it has **interfaces** through which it exchanges energy, matter, signals, or forces with other components.



$$\text{Battery} \xrightarrow{I_1} \text{Resistor} \xrightarrow{I_2} \text{LED}$$

Interface condition

When two physical components are connected, some quantities must agree at the interface:

$$V_{\text{out}} = V_{\text{in}}, \quad I_{\text{out}} = I_{\text{in}}.$$

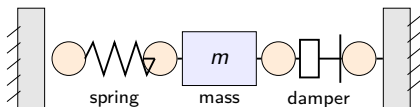
Categorical reading

Each component is an open system with input and output ports. Composition means gluing compatible ports together.

Example 6b: mechanical systems compose through interfaces

Mechanical components as open systems

A spring, a mass, and a damper can each be viewed as a small physical system with boundary points where motion and force are exchanged.



spring $\circ$ mass $\circ$ damper

Interface condition

At a shared mechanical boundary, connected components must agree on motion:

$$x_{\text{left}} = x_{\text{right}}, \quad v_{\text{left}} = v_{\text{right}}.$$

Forces are transmitted through the same boundary.

Categorical message:

composition is not only logical or algebraic; it also describes how physical machines are built.

Categorical reading

The whole mechanical system is assembled by gluing smaller open systems along compatible boundary variables.

Example 6c: what is the category?

Open systems form categories

For many physical examples, the categorical pattern is:

objects = interfaces, morphisms = open systems.

Electrical circuits

Ob = finite sets of terminals

$\text{Hom}(A, B)$ = circuits from A to B

$$A \xrightarrow{C_1} B \quad B \xrightarrow{C_2} D$$

compose by connecting the shared terminals B :

$$A \xrightarrow{C_2 \circ C_1} D.$$

Mechanical systems

Ob = boundary ports

$\text{Hom}(A, B)$ = subsystems from A to B

Ports may carry variables:

$$x, \quad v, \quad F.$$

Composition glues compatible boundaries:

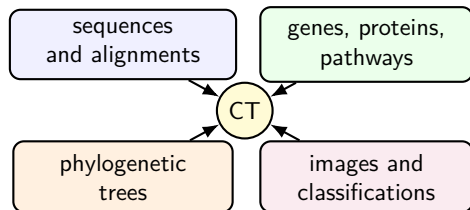
Example 7: biology and bioinformatics

Biological data are relational

- sequences and alignments;
- genes, proteins, pathways;
- phylogenetic trees;
- images and classifications.

Categorical thinking suggests asking:

*What are the objects, what are the maps,
and what must be preserved?*



Example 7a: Sequences and Alignments

Biological Object: DNA, RNA, or protein sequences.

In bioinformatics, we don't just ask if two sequences are identical; we want to know how one might transform into the other.

- **Objects:** Finite sequences of nucleotides or amino acids.
- **Morphisms:** Alignments (matching, inserting, or deleting characters).

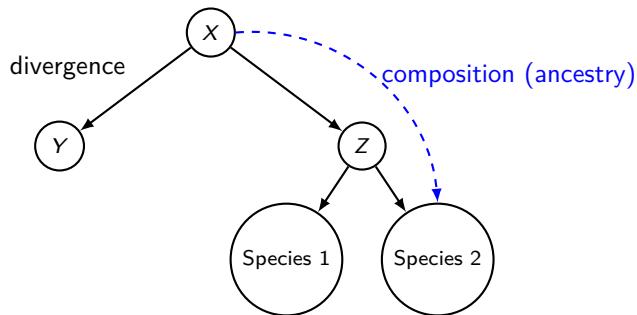


Categorical Message: Edit paths or alignments can be treated as morphisms between sequences.

Example 7b: Phylogenetic Trees as Categories

Biological Object: Evolutionary ancestry. A phylogenetic tree represents the evolutionary relationships among various biological species.

- **Objects:** Species or specific taxonomic units (living or extinct).
- **Morphisms:** An arrow $A \rightarrow B$ exists if A is an ancestor of B .

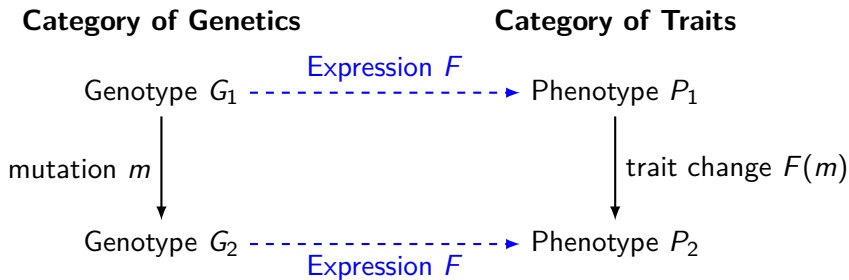


Categorical Message: Because ancestry is transitive (if X is an ancestor of Z , and Z is an ancestor of Species 2, then X is an ancestor of Species 2), the tree is a category.

Example 7c: The Genotype-Phenotype Functor

Biological Process: Gene expression and traits.

Biology bridges two distinct "worlds": the underlying genetic code and the observable physical characteristics. We can model this as a functor.



Categorical Message: In a simplified model, genotype-to-phenotype translation can be viewed functorially. In richer models, one may need relations, spans, profunctors, or stochastic maps.

Example 8: systems with behavior

Coalgebraic viewpoint

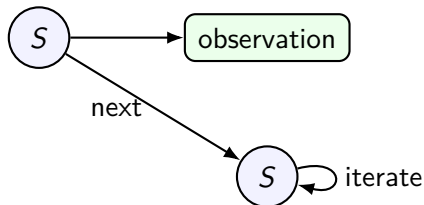
For a dynamic system, we often observe behavior step by step:

state $\longrightarrow$ observation + next state.

This fits models such as:

- automata and transition systems;
- streams and time series;
- interacting biological or computational components.

See Jacobs, 2012 [9]



Message

Algebras build structure; coalgebras describe behavior.

The Categorical Definition: Algebras and Coalgebras

Let $\mathcal{C}$ be a category and $F : \mathcal{C} \rightarrow \mathcal{C}$ an endofunctor.

F -Algebra

An F -algebra is a pair (X, α) consisting of:

- A **carrier** object $X \in \text{Ob}(\mathcal{C})$.
- A **structure map** $\alpha \in \text{Hom}_{\mathcal{C}}(F(X), X)$.

$$\begin{array}{c} F(X) \\ \downarrow \alpha \text{ (eval)} \\ X \end{array}$$

Constructive view: Evaluates components of shape F into a concrete state X .

F -Coalgebra

An F -coalgebra is a pair (X, β) consisting of:

- A **state space** object $X \in \text{Ob}(\mathcal{C})$.
- A **transition map** $\beta \in \text{Hom}_{\mathcal{C}}(X, F(X))$.

$$\begin{array}{c} X \\ \downarrow \beta \text{ (unfold)} \\ F(X) \end{array}$$

Observational view: Unfolds a hidden state X into an observable shape $F(X)$.

The Structure of Behavior

The functor F precisely defines the interface of the system.

By changing the "shape" of $F(X)$, we define entirely different dynamical systems:

- **Infinite Streams:** Let $F(X) = A \times X$.

An F -coalgebra is a map $X \rightarrow A \times X$.

Meaning: Given a state, output a value in A (the head) and transition to a new state X (the tail).

- **Deterministic Automata:** Let $F(X) = \text{Bool} \times X^\Sigma$.

An F -coalgebra is a map $X \rightarrow \text{Bool} \times X^\Sigma$.

Meaning: Given a state, observe if it is accepting (Bool) and provide a function mapping any input Σ to the next state (X^Σ).

A Concrete Example in **Set**

Let's instantiate this in the simplest category: **Set**.

- **Category $\mathcal{C}$:** **Set** (Objects are sets, morphisms are ordinary functions).
- **Endofunctor F :** $F(X) = \mathbb{R} \times X$ (Takes a set and pairs it with a real number).

F -Algebra (Construction)

Let our object be $X = \mathbb{R}$.

We need a function $F(X) \rightarrow X$.

$$eval : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

$$eval(a, b) = a + b$$

Meaning: We take a pair of numbers and construct a single, evaluated sum.

F -Coalgebra (Observation)

Let our object be $X = \mathbb{R}^{\mathbb{N}}$ (infinite streams).

We need a function $X \rightarrow F(X)$.

$$unfold : \mathbb{R}^{\mathbb{N}} \rightarrow \mathbb{R} \times \mathbb{R}^{\mathbb{N}}$$

$$unfold(S) = (head(S), tail(S))$$

Meaning: We take a hidden stream, observe its first element, and transition to the rest.

A Geometric Example: Affine Transformations

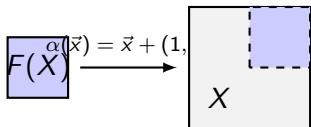
Let's step away from Sets and look at shapes in $\mathbb{R}^2$.

- **Category $\mathcal{C}$:** Objects are subsets of $\mathbb{R}^2$. The morphisms in $\text{Hom}(X, Y)$ are exactly the affine transformations f such that $f(X) \subseteq Y$. (If no such f exists, the Hom-set is simply empty!)
- **Endofunctor F :** The "Half-Scale" functor. $F(X) = \frac{1}{2}X$ (shrinks the shape by half). For morphisms, $F(f)(\vec{x}) = \frac{1}{2}f(2\vec{x})$.

F -Algebra (Construction)

A map $\alpha : F(X) \rightarrow X$.

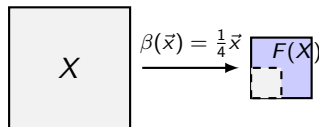
We take the half-sized shape and embed it into the full shape.



F -Coalgebra (Observation)

A map $\beta : X \rightarrow F(X)$.

We take the full shape and compress it to fit inside its half-sized version.



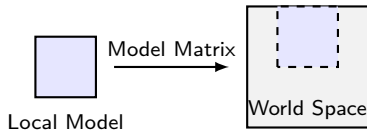
Example: Algebras and Coalgebras in Computer Graphics

The 3D rendering pipeline perfectly mirrors this categorical duality.

F-Algebra: Scene Construction

Building up (Local $\rightarrow$ World)

We use **Model Matrices** to take local 3D models and embed them into a global World Space.

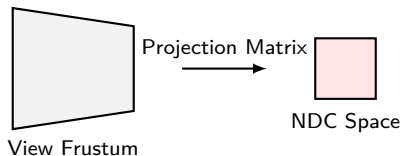


Categorical view: $\alpha : F(X) \rightarrow X$. We map local pieces into the whole environment.

F-Coalgebra: The Viewing Pipeline

Observation (World $\rightarrow$ Screen)

We use **Projection Matrices** to take the Camera's View Frustum and compress it into Normalized Device Coordinates (NDC).



Categorical view: $\beta : X \rightarrow F(X)$. We map the vast world down into an observable box.

The Category of F -Algebras

Algebras form their own category, giving us the power to compare constructions.

- **Objects:** Pairs (X, α) where $\alpha : F(X) \rightarrow X$.
- **Morphisms:** A structure-preserving map $f : X \rightarrow Y$ between algebras (X, α) and (Y, β) .

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \alpha \downarrow & & \downarrow \beta \\ X & \xrightarrow{f} & Y \end{array}$$

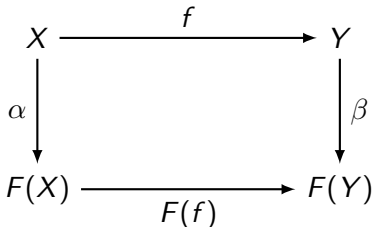
The Commutative Square:

This diagram guarantees that mapping the components and then evaluating $(\beta \circ F(f))$ is identical to evaluating first and then mapping the result $(f \circ \alpha)$. The **initial object** in this category represents all finite data structures!

The Category of F -Coalgebras

Coalgebras also form their own category, giving us the power to compare systems.

- **Objects:** Pairs (X, α) where $\alpha : X \rightarrow F(X)$.
- **Morphisms:** A structure-preserving map $f : X \rightarrow Y$ between coalgebras (X, α) and (Y, β) .



The Commutative Square:

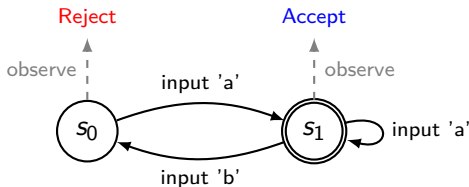
This diagram is the exact mathematical foundation of **bisimulation**. Furthermore, the final object in this category represents the universe of all possible infinite behaviors!

Example 8a: Automata as Coalgebras

System: Finite State Machines / Automata.

An automaton is a classic coalgebra. Instead of looking at how the machine was built, we only care about how it behaves when we interact with it.

- **State Space (S):** The hidden internal states.
- **Observation:** Is the current state "Accepting" or "Rejecting"?
- **Transition:** Given an input character, what is the next state?

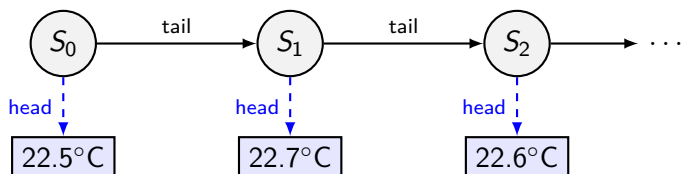


Categorical Message: The coalgebraic map is $S \rightarrow \{Accept, Reject\} \times S^\Sigma$. It perfectly describes one step of behavior without revealing the entire machine at once.

Example 8b: Infinite Streams

System: Time series, sensors, or lazy data structures.

How do you mathematically model an infinite stream of data (like a temperature sensor reading every second)? You cannot build it algebraically because it never ends. You must unfold it coalgebraically.



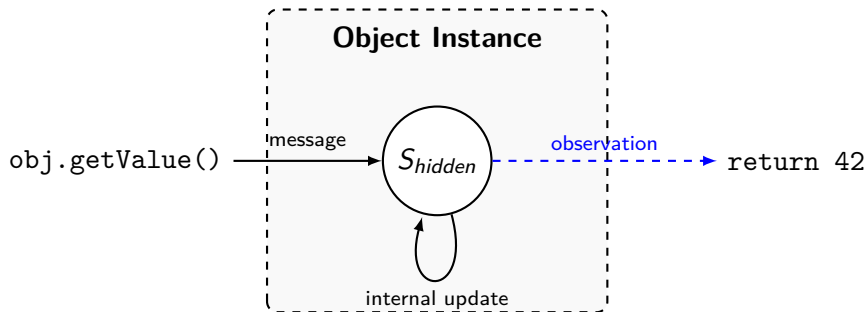
Categorical Message:

A stream is a coalgebra $S \rightarrow A \times S$. At any moment, we can observe the current value (head) and transition to the rest of the stream (tail).

Example 8c: Objects are Coalgebras

System: Object-Oriented Architecture.

In object-oriented programming, objects encapsulate private internal state. We cannot access the state directly; we can only interact with it via public methods.



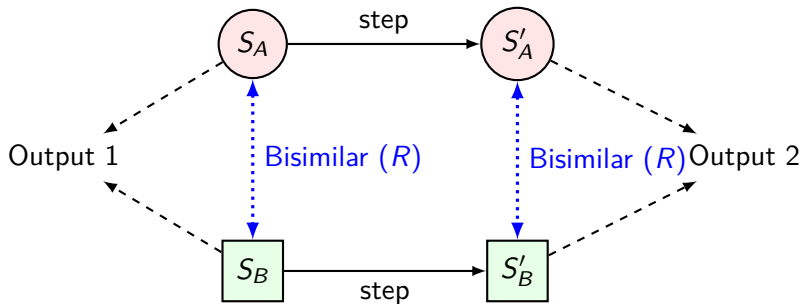
Categorical Message:

Classes define coalgebras. Calling a method on an object is exactly applying the coalgebraic map: it yields an observable return value and secretly transitions the object to a new internal state.

Example 8d: Bisimulation

System: Verifying refactored code or alternative hardware.

If we have two different systems (e.g., an old slow algorithm and a new fast one), how do we prove they are "the same"? Coalgebra uses **bisimulation**.



Categorical Message: Two states are bisimilar if they produce the same observations now, and transition to states that are also bisimilar. If we cannot observe a difference, they are coalgebraically equivalent!

The Third Pillar: Natural Transformations

If functors are dictionaries between categories, natural transformations are translations between dictionaries. Given two functors $F, G : \mathcal{C} \rightarrow \mathcal{D}$, a **natural transformation** $\alpha : F \Rightarrow G$ provides a morphism $\alpha_X : F(X) \rightarrow G(X)$ in $\mathcal{D}$ for every object X in $\mathcal{C}$.

The Naturality Square:

It must preserve the internal arrows. For any $f : X \rightarrow Y$, this diagram must commute:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ & \searrow \alpha \Rightarrow & \\ & & \begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ \alpha_X \downarrow & & \downarrow \alpha_Y \\ G(X) & \xrightarrow{G(f)} & G(Y) \end{array} \end{array}$$

Categorical Message:

First apply F then translate ($\alpha_Y \circ F(f)$) $\equiv$ First translate then apply G ($G(f) \circ \alpha_X$). The mapping is "natural" because it doesn't disrupt the flow of the system.

Examples of Natural Transformations

Natural transformations ($\alpha : F \Rightarrow G$) appear when changing perspectives uniformly.

1. Computer Science (Polymorphism)

- **Functors** F, G : $F = \text{List}$, $G = \text{Maybe}$ (represents an optional value, either present or empty).
- **Objects** X, Y : Types (e.g., $X = \text{Int}$, $Y = \text{String}$).
- **Morphism** f : A function like `intToString: Int -> String`.
- **Nat. Trans.** α_X : The function `head: List<X> -> Maybe<X>`.
- *Naturality*: `head ∘ List.map(f) ≡ Maybe.map(f) ∘ head`.

2. Linear Algebra (Double Duals)

- **Functors** F, G : Identity $F(V) = V$, and Double Dual $G(V) = V^{**}$.
- **Objects** X, Y : Vector spaces V and W .
- **Morphism** f : A linear transformation $T : V \rightarrow W$.
- **Nat. Trans.** α_V : The canonical embedding mapping $v \in V$ into V^{**} .
- *Naturality*: Embedding V then mapping to W^{**} is identical to mapping $V \rightarrow W$ then embedding into W^{**} .

3. Engineering (Analog-to-Digital Conversion)

- **Functors** F, G : $F = \text{Continuous signal space}$, $G = \text{Discrete sampled space}$.
- **Objects** X, Y : Physical systems (e.g., $X = \text{raw audio}$, $Y = \text{filtered audio}$).
- **Morphism** f : A physical process (e.g., an analog low-pass filter).
- **Nat. Trans.** α_X : The Sampling operation (an A/D converter).
- *Naturality*: Filtering the analog signal then sampling it is identical to sampling it first then applying a digital filter.

Conclusion

Category theory is a language of composition

It helps us describe many different systems by asking the same basic questions:

What are the objects? What are the morphisms? How do they compose?

- **Functors** compare different worlds while preserving structure.
- **Universal constructions** describe canonical ways to combine information.
- **Coalgebras** describe systems through their observable behaviour.
- **Natural transformations** compare different translations between worlds.

From structures to processes,
from local parts to global systems.

Thank you!

References I

-  S. Awodey, *Category Theory*, Oxford University Press, 2nd edition, 2010.
-  T. Leinster, *Basic Category Theory*, Cambridge University Press, 2014. Also available on arXiv.
-  B. Fong and D. I. Spivak, *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*, Cambridge University Press, 2019.
-  J. C. Baez and M. Stay, *Physics, Topology, Logic and Computation: A Rosetta Stone*, New Structures for Physics, Lecture Notes in Physics 813, 2011.
-  J. R. Munkres, *Topology*, Pearson, 2nd edition.
-  M. F. Atiyah and I. G. Macdonald, *Introduction to Commutative Algebra*, Addison-Wesley, 1969.
-  B. C. Pierce, *Types and Programming Languages*, MIT Press, 2002.

References II

-  G. Winskel, *The Formal Semantics of Programming Languages*, MIT Press, 1993.
-  B. Jacobs, *Introduction to Coalgebra: Towards Mathematics of States and Observation*, Cambridge University Press, 2012.
-  J. Rutten, *Universal Coalgebra: A Theory of Systems*, 2000.